Enabling Functional Tests of Asynchronous Circuits Using a Test Processor Solution

Von der Fakultät für Mathematik, Naturwissenschaften und Informatik
der Brandenburgischen Technischen Universität Cottbus-Senftenberg

zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
(Dr.-Ing.)

genehmigte Dissertation

vorgelegt von

Diplom-Informatiker

Steffen Zeidler

geboren am 29. August 1980 in Potsdam

Gutachter: Prof. Dr.-Ing. Rolf Kraemer

Gutachter: Prof. Dr.-Ing. Heinrich Theodor Vierhaus

Gutachter: A.o. Univ.-Prof. Dr. Andreas Steininger

Tag der mündlichen Prüfung: 12. Dezember 2013

"The road to success is always under construction."

— Anonymous

To my sweet, little daughter 

Contents

Acknowledgements

First of all, I would like to thank all my colleagues from IHP, who supported me in writing this thesis. Especially, I would like to acknowledge my colleague and friend Christoph Wolf, who always was open for my questions no matter what these were about. Christoph, you helped me with fruitful discussions and enabled me to write this thesis, although my tasks piled on your desk. It was a pleasure to work and to learn from you. Thank you for the great time and everything.

Many, many thanks go also to Miloš Krstić, who helped me with every single hint concerning how to write a PhD thesis. Miloš, without your expertise this work would never have been finished, not to mention the great responds I got. Thank you.

I also have to thank Marcus Ehrig for carefully proofreading this work. The thesis would be much more clumsy at certain places without your critical eye. I also enjoyed the discussions with you, Marcus. You also helped me a lot with your knowledge about FPGA-designs.

I would also like to express my gratitude to Eckhard Grass. Thank you Eckhard for improving my English.

My special thanks go to Prof. Rolf Kraemer, who gave me the chance to write the thesis and who was very patient with me. Even after repeated postponement, he did not lose the confidence that I will finally finish the thesis. Thank you, Rolf.

Of course, I also would like to thank Prof. Andreas Steininger and Prof. Vierhaus for spending their time to review the thesis.

Last but not least, I would like to thank my family for encouraging me all the time. My thanks are especially dedicated to my girlfriend Juliane. Without her support, especially in the very last phase of the writing, the thesis would not have been finished in time. Thanks to my little daughter Lena-Sophie. So often, she made me smile when I was discouraged. Of course, I would like to thank my parents for their support, e.g., by taking care of Lena when Juliane and I were not able to do that. Thanks, Mutz und Paps.

Abstract

During the last years, the asynchronous design style has been rediscovered as a potential solution to upcoming design issues in deep-submicron technologies. However, besides the lack of commercial tools supporting this design style, one major challenge is the test of asynchronous designs. Especially their event-driven behavior leads to problems during test. Basically, the timing of asynchronous circuits is determined by gate and wire delays that are sensitive to variations of environmental parameters (process, voltage and temperature). This leads to uncertainties in the timing of the responses. Consequently, standard commercial test systems cannot be used, because such systems read the responses at specific cycles and, therefore, could reject fault-free devices.

Furthermore, available hardware testers are, in principle, not designed to react to signal events from the design-under-test as it is necessary to establish asynchronous communication via handshake signalling. As a result, even simple functional tests that only apply stimuli and read the responses of the design-under-test cannot be realized without preparatory measures.

This work addresses these issues and proposes a concept to enable functional tests of asynchronous designs. The concept is based on a special test processor that provides generic interfaces used to establish asynchronous handshake communication with a device-under-test. By this, elastic functional tests can be realized that overcome the static timing of conventional tests and emulate the real operating environment of the design. Apart from the generic test processor architecture, an essential part of the concept deals with the establishment of the processor as a stand alone or embedded test equipment. A workflow is provided that describes how the device-under-test can be embedded into the test processor environment for performing the tests. Besides the interconnection between the asynchronous design and the test processor, this especially includes the generation of programs that realize the functional tests of the design. A methodology is introduced that generates the desired programs for the processor from a standard functional simulation of the design-under-test.

Based on the generic concept, a framework including both a test processor implementation and the realization of the program generation is delivered. In order to evaluate the entire concept, this framework has been applied to functionally test an asynchronous arithmetic-logic-unit. In combination with additional experiments, conducted to determine the required resources, it has been shown that the introduced concept is a suitable approach to test asynchronous designs.

Zusammenfassung

Aufgrund von Problemen bei der Integration komplexer Systeme in nano-skalierten Technologien zeichnet sich in den letzten Jahren der Trend zum asynchronen Entwurf integrierter Schaltungen ab. Allerdings wird dieser Paradigmenwechsel neben dem Mangel an Entwurfswerkzeugen insbesondere durch Probleme beim Testen gehemmt. Diese Probleme ergeben sich aus der Ereignis-getriebenen Funktionsweise von asynchronen Schaltungen, deren Zeitverhalten durch Leitungs- und Gatterverzögerungen bestimmt wird. Da diese Verzögerungen von Umgebungsparametern wie Temperatur und Versorgungsspannung, aber auch von Prozessvariationen abhängen, führt dies zu Unbestimmtheit im Antwortverhalten eines asynchronen Prüflings. Da jedoch kommerzielle Testsysteme die Ausgaben eines Prüflings zu festgelegten Zeitpunkten erwarten, kann diese Unbestimmtheit dazu führen, dass ein asynchroner Prüfling als fehlerhaft deklariert wird, obwohl dieser richtige Ausgaben – allerdings zu unerwarteten Zeitpunkten – liefert.

Hinzukommt, dass kommerzielle Hardwaretester nicht dazu konzipiert sind, auf vom Prüfling erzeugte Signalereignisse zu reagieren. Diese Fähigkeit ist jedoch Grundvoraussetzung für die Kommunikation mit einem asynchronen Prüfling mittels so genannter Handshake-Verfahren. Als Folge können selbst einfache Funktionaltests, die die reale Umgebung des Prüflings emulieren sollen, mit Standardtestern nicht durchgeführt werden.

Diese Arbeit greift diese Problematik auf und liefert ein Konzept, das Funktionaltests für asynchrone Schaltungen ermöglicht. Dieses Konzept beruht auf einem speziellen Testprozessor, welcher generische Schnittstellen zur asynchronen Handshake-basierten Kommunikation mit dem Prüfling bereitstellt. Dadurch werden *elastische* Tests ermöglicht, die das statische Zeitverhalten konventioneller Tests vermeiden und folglich die reale Betriebsumgebung eines asynchronen Designs emulieren können. Neben der generischen Architektur des Testprozessors umfasst das Konzept auch eine Beschreibung, wie der Prozessor als Testwerkzeug etabliert werden kann. Zum einen muss dazu der Prüfling in die Testprozessorumgebung eingebettet werden. Dazu muss insbesondere eine Verbindung zwischen den Schnittstellen des Prüflings und des Prozessors hergestellt werden. Zum anderen werden für die Realisierung der durchzuführenden Funktionaltests Programme für den Testprozessor benötigt. Zu diesem Zweck wird eine Methodik vorgestellt, die aus einer Standardfunktionalsimulation ein entsprechendes Testprozessorprogramm generiert.

Aufbauend auf dem generischen Konzept wird ein Framework beschrieben, das eine Implementierung des Prozessors sowie der Programmgenerierung beinhaltet. Zur

Evaluierung des Konzepts wurde dieses Framework für den Funktionaltest eines asynchronen Designs angewendet. Zusammen mit weiteren Experimenten, die die benötigten Ressourcen bestimmen, konnte dadurch erfolgreich gezeigt werden, dass das Konzept ein geeignetes Verfahren für den Funktionaltest asynchroner Schaltungen darstellt.

Acronyms

ABMM Asynchronous Burst Mode Machine.

ACL Asynchronous Control Logic.

ADL Architecture Description Language.

ALU Arithmetical Logical Unit.

ATE Automated Test Environment/Equipment.

BIST Built-In Self-Test.

BNF Bacus-Nauer-Form.

BRAM Block RAM.

CAD Computer-Aided-Design.

CSP Communicating Sequential Processes.

DfT Design-for-Testability.

DI Delay Insensitive.

DP Data Port.

DUT Device Under Test.

EDA Electronic Design Automation.

EMI Electro-Magnetic Interference.

ETE External Test Equipment.

FIFO First-In First-Out.

FPGA Field Programmable Gate Array.

FSM Final State Machine.

GALS Globally-Asynchronous Locally-Synchronous.

GCD Greatest Common Divisor.

HDL Hardware Description Language.

HP Handshake Port.

IC Integrated Circuit.

IOB Input/Output Buffer.

IP Intellectual Property.

ITRS International Technology Roadmap for Semiconductors.

LFSR Linear-Feedback Shift-Register.

LISA Language for Instruction Set Architectures.

LUT Lookup Table.

MAA Memory Access Arbiter.

MAC Memory Access Controller.

MISR Multi-Input Signature-Register.

MTBF Mean Time Between Failure.

MUTEX Mutual Exclusion.

NBTI Negative Bias Temperature Instability.

NoC Network on Chip.

NoTePAD Novel TEst Processor for Asynchronous Devices.

PVT Process, Voltage and Temperature.

QDI Quasi Delay Insensitive.

RAM Random Access Memory.

RISC Reduced Instruction Set Computer.

ROM Read Only Memory.

RTL Register Transfer Level.

SB Synchronous Block.

SDF Standard Delay Format.

SI Speed Independent.

SoC System-on-Chip.

ST Self-Timed.

STG Signal Transition Graph.

TAP Test Adapter Port.

TP Test Processor.

TPG Test Pattern Generator.

TRA Test Response Analyzer.

UUT Unit Under Test.

VCD Value Change Dump.

VHDL Very High Speed Integrated Circuit Hardware Description Language.

VLSI Very Large Scale Integration.

"To make an apple pie from scratch,
you must first invent the universe."

— Carl Sagan

Chapter 1

Introduction

1.1 Motivation

For the last decades, the design of Integrated Circuits (ICs) has benefitted from the assumption that systems process their data in discrete time steps. This synchronous design paradigm has drastically eased the design flow of ICs as it simplifies the handling of critical hazards and signal races by the introduction of the clock period. Accordingly, CAD tools as well as test methods have been especially optimized to the synchronous design world. As a consequence of the ease of the design methodology and the availability of tools, designers still prefer the utilization of the synchronous design methodology in order to create complex digital systems. However, the ongoing shrink of feature sizes has several influences on the design of digital systems. With today's nano-scale technologies the circuits are more susceptible to process, voltage and temperature (PVT) variations. Therefore, the designs reach the limits of the physical capabilities of ICs. Furthermore, gate and wire delays are getting the order of the clock period resulting in challenges to meet the timing closure of the design. Thus, the clock distribution in complex synchronous systems becomes a critical point. Furthermore, the clock tree itself consumes a large amount of the total power of a synchronous design. In order to solve this, power management techniques and methods to partition a design into multiple clock domains are applied to ensure efficient operation. But this again leads to even more complex design processes.

To this end, alternative ways of designing digital systems have to be explored. At this point, the utilization of the asynchronous design methodology seems to be a promising solution to tackle the upcoming challenges. This paradigm is *a priori* able to compensate delays in the communication, offers modularity and has advantages in security aspects.

Especially, the aspect of modularity is of interest with respect to the design of complex systems. The International Technology Roadmap for Semiconductors (ITRS) expects an increased utilization of the asynchronous design paradigms in future ICs in order to counteract problems with the clock distribution [ITRS 2012]. The general concept of these systems is to avoid the use of a global clock signal. Instead, the data is exchanged by means of some convention, i.e., a protocol, organizing the communication between a sender and a receiver. Hence, an asynchronous circuit is considered to be a composition of modules, each having interfaces for sending and receiving data. The protocol used to organize the communication depends on the architecture and the demands of the application. The most widely used family of asynchronous circuits utilizes asynchronous handshake signaling for the communication between modules.

However, since the synchronous paradigm dominates the IC design world and although lots of research has been carried out in the field of asynchronous system design and test, there are still two issues that hinder the utilization of this promising design methodology. First, there is a lack of commercial tools supporting the asynchronous design flow, and second, there still exist various problems with respect to testing. Even the simplest test in the synchronous design world, i.e., functional test, is an issue for asynchronous devices. This leads to the classical chicken-egg problem. Without having tools and test methods the asynchronous design paradigm will not be applied and vice versa.

In general, testing is an essential part of the semiconductor life cycle. Typically, the costs of testing an IC is approximately 50% of the total cost of a chip. Apart from well established test methods mainly applied in production test, further test strategies are required for debugging during the prototype phase of a design. In this context, functional tests of the developed design play a major role. However, functional tests of asynchronous designs are still a major issue, since the event-driven behavior of such circuits leads to problems with existing hardware testers. This work addresses this issue and provides an approach to perform functional tests of asynchronous designs.

1.2 Contribution to the State-of-the-Art

Currently, there is a general belief that asynchronous circuits are very difficult to test. This actually still applies to some extent, although numerous publications address the test of asynchronous circuits. One big issue is the execution of functional tests. These tests are frequently applied during system prototyping. Furthermore, functional testing regains more and more importance in System-on-Chip (SoC) design. Often these systems comprise Intellectual Property (IP) cores whose internal structure is unknown

and must not be changed to prevent violations of IP rights. Thus, commonly applied Design-for-Testability (DfT) techniques, such as scan test, cannot be applied. Therefore, performing functional tests is often the only possibility to validate such components.

The problem with functional tests of asynchronous designs arises from the event-driven behavior. Usually, asynchronous designs utilize bidirectional transfer protocols, e.g., based on completion indication, to organize the data exchange with their environment. This has some implications to the test of such designs. The utilized test system has to react to signal events generated by the device under test (DUT). However, today's big iron test systems, such as the Advantest V93000 [Advantest 2013], are strictly oriented towards synchronous designs that provide their responses at specific clock cycles. These testers are not designed to react to events from the DUT. Thus, a handshake-based data exchange cannot be realized.

In order to cope with this issue, one may think about the application of alternative test techniques also for functional tests. One possible solution is to utilize scan test. Such an approach was, e.g., proposed in [Gürkaynak 2002] to individually test the synchronous blocks and their interfaces in a globally-asynchronous locally-synchronous (GALS) system. Although this technique was especially designed for GALS systems, it can also be adapted to fully asynchronous systems. Nevertheless, the scan technique has one major drawback that is even more critical when performing complex functional tests. In order to perform one single test iteration, the stimuli and the responses have to be scanned in and out serially. This enormously reduces the throughput during test and, therefore, makes the emulation of real operating conditions impossible.

Another technique suitable for functional tests of asynchronous circuits is Built-In Self-Test (BIST). The advantage of BIST, intended for simple functional tests, is that it is not intrusive. Thus, it does not require any change to the internal structure of the DUT as it is required for scan. Therefore, it enables black box testing and is very suitable for integration into asynchronous high-performance circuits as, e.g., applied for the RAPPID asynchronous instruction length decoder [Roncken 2000]. Furthermore, in order to cover various test scenarios, the components of a BIST can be hierarchically organized. Such a technique has been successfully applied in a GALS baseband processor for the WLAN standard IEEE 802.11a [Krstić 2005a]. Unfortunately, BIST also has some drawbacks with respect to functional tests. A major one is related to the weak diagnostic capability of the approach, since often only pass/fail information or a signature gained from the compression of the captured responses is delivered. Furthermore, the pattern generators are typically realized by pseudo-random number generators, such as Linear-Feedback Shift-Registers (LFSRs). Although these components can be implemented such that they are configurable, the generated sequence of patterns remains

static. Complex sequences of non-random patterns are also hard to implement. Indeed, a BIST can be extended with patterns from a built-in memory, but this considerably increases the hardware overhead and complexity of the test logic.

A further approach to realize functional tests could be the utilization of buffers such as asynchronous FIFOs as, e.g., proposed in [Wolf 2011]. These buffers could be added to the interfaces of the DUT in order to compensate the variations of the timing. However, the integration of FIFOs into the DUT could result in considerable hardware overhead. Therefore, as proposed in [Wolf 2011], these FIFOs could also be implemented by a Field Programmable Gate Array (FPGA) that is part of the test equipment. Then, input buffers are filled and output buffers are flushed by the test equipment in bursts under consideration of the worst case timing of the DUT. This solution is very effective and simple to implement for pipeline architectures. But, since there is no control at the level of single tokens, this solution might not be feasible for complex devices having multiple interfaces with complicated relationships between the data transfers.

Therefore, a new test approach is required that provides similar capabilities with respect to functional tests as a standard tester while offering mechanisms to enable asynchronous communication. Thus, the scheme needs to have the following capabilities:

- *Elastic test capability* — Obviously, due to the afore mentioned issues with asynchronous circuits, the provided solution needs to resolve the static timing behavior of traditional test systems. Instead of aligning the data change to clock cycles, an event-driven behavior is required in order to realize handshake based data transfers.

- *Configurability* — In order to provide a generic solution capable of testing different devices, the desired scheme should provide a programmable pin configuration as common hardware testers do. Furthermore, in order to test various types of asynchronous devices, different handshake protocols have to be supported.

- *Flexibility with respect to changes of patterns* — In order to perform various test scenarios it is required to change the test patterns. Therefore, the solution has to provide mechanisms to exchange the patterns during a testflow.

- *Full controllability of the dataflow* — To support complex test scenarios, it might be necessary to have the full control of the data exchanges with the DUT. Therefore, the aimed test approach has to be equipped with mechanisms to specify at which interface data shall be exchanged.

➢ *High-performance* — Although handshake interface circuits are designed to tolerate timing variations in the execution of transfers, the test approach has to provide high data rates in order to emulate the real operating environment of the DUT.

Based on the identified requirements, this work proposes a methodology for performing elastic functional tests which are fully event-driven. The core of the concept is a test processor solution which is equipped with special ports supporting various types of handshake protocols. With the help of these ports, asynchronous communication channels can be established between the DUT and the desired test processor. These channels are intended to apply and receive patterns with an elastic timing behavior. Besides the processor, a methodology is required to generate programs implementing the targeted functional tests. Therefore, a workflow is proposed describing all necessary steps to gain an elastic test from a standard logic simulation.

1.3 Publications Related to this Work

In the context of this work three papers have been published on national and international conferences. In [Zeidler 2011] the general concept of the targeted test processor and a first implementation is proposed. The provided solution describes a simple 16-bit Reduced Instruction Set Computer (RISC) processor equipped with special handshake ports that are combined with data ports to enable asynchronous communication. The association of the ports with asynchronous channels is described in software. Therefore, this approach is very flexible, but performance degradations have to be accepted. Due to several limitations of this initial test processor design, an extended 32-bit processor has been proposed in [Zeidler 2012a]. This implementation has been improved with respect to data transmission rates by combining several required steps into single processor instructions. Finally, in [Zeidler 2012b] a flow is proposed to generate a test program from a logic simulation. Consequently, these publications cover all aspects of the solution provided in this work. However, this work draws the entire picture of the approach by combining and complementing these aspects with further improvements of the concept.

1.4 Overview of the Work

Chapter 2 introduces the principles of asynchronous circuit design and its main characteristics. This includes the principle of asynchronous handshake signalling, the classification of asynchronous circuits based on timing models, issues in the design of asynchronous circuits and typical circuit architectures. Afterwards, the basics of testing asynchro-

nous designs are presented. Thereto, several techniques, which potentially can be used for functional tests, are introduced. This chapter builds the foundation for understanding the issues related to the test of these designs.

Chapter 3 starts with the detailed discussion of the problem of performing functional tests of asynchronous designs. It describes why conventional testers cannot be used in this context. Based on this, possible solutions are discussed. This includes previous works dealing with functional tests of asynchronous designs as well as further ideas that are basically applicable. These approaches are analyzed with respect to their pros and cons. This finally explains the demand for a novel functional test scheme for asynchronous designs.

The general concept of such a scheme is introduced in Chapter 4. This chapter proposes a generic Test Processor (TP) architecture adapted to the requirements of asynchronous devices. This processor provides interfaces for asynchronous channels which are the base for the communication with the DUT during functional tests. For this test processor, programs are required that realize the desired tests. To this end, a methodology is presented that generates a program from a standard functional simulation of the DUT. The crux of the idea is the generation of a *transfer protocol* during simulation which describes the interactions between the TP and the DUT. Afterwards, this protocol needs to be translated to a processor specific program.

An implementation of the test processor concept is presented in Chapter 5. The provided solution, called NoTePAD (Novel Test Processor for Asynchronous Devices), is a special processor which is highly optimized for applying stimuli and receiving responses.

Chapter 6 discusses the realization of the methodology for generating test programs. It is shown how the *transfer protocol* is generated from a logic simulation. After that, the translation of the protocol to a program for the NoTePAD architecture is delivered.

The evaluation of the entire concept is illustrated in Chapter 7. Here, the entire framework is applied to an asynchronous arithmetical logical unit (ALU). The selected DUT was combined with the TP to form a demonstrator design which was integrated into an FPGA. Afterwards, the application of the flow generating the program is discussed. The gained test program was simulated and compared with the initial functional simulation of the DUT. Additionally, the chapter illustrates further properties of the solution, such as the hardware requirements and test time.

Finally, Chapter 8 concludes the work by summarizing the achievements and the major limitations. Based on this, an outlook about further enhancements and other future activities in the context of this work is given.

"Not everything that can be
counted counts, and not everything
that counts can be counted."

— Albert Einstein

Chapter 2

Basics of Asynchronous Circuits and Their Testing

2.1 Asynchronous Circuits

This section gives an overview about the main target circuits, i.e., asynchronous handshake circuits. Therefore, the main characteristics, advantages and disadvantages of such systems are presented.

2.1.1 Concept and History

The concept of the asynchronous design methodology goes back to the automaton theory in the 1950's and 60's. At that time D.E. Muller was a pioneer on the design of asynchronous circuits. He invented one of the most important components for asynchronous circuits — the *C*-element, i.e., a logical gate with hysteresis. Furthermore, he proposed a pipeline architecture that is the backbone of almost all asynchronous circuits. As a result of his work the ILLIAC II was developed at the University of Illinois in 1962 which was the first fully asynchronous processor. As opposed to a synchronous implementation, this processor did not require a global clock signal for the coordination of the data exchange between its components. Instead, such an asynchronous circuit implementation utilizes local control signals triggering the sequential (storage) elements to update their states. C.L. Seitz described this way of system timing as *self-timed* [Seitz 1980]. In general, *self-timed* circuits are composed of modules that are connected via bundles of wires, so called *channels*. Thereby, a *sender* provides data that is read from the channel

by a *receiver*. To coordinate the data exchange between these modules, the channels utilize the concept of indication. This means that data is exchanged when indicated rather than at repetitive points in time. This principle was employed by I.E. Sutherland in [Sutherland 1989]. He presented an asynchronous pipeline architecture called *Micropipelines* composed of modules strictly separated into data and control logic.

Based on Sutherland's *Micropipeline* architecture the Amulet asynchronous ARM processor series was developed at the University of Manchester. The implementation of the ARM6 compatible Amulet1 [Furber 1994b, Woods 1997] was a feasibility study to demonstrate that complex systems can be implemented using the asynchronous design methodology. Furthermore, this study was intended to illustrate that asynchronous implementations can keep up with their synchronous counterparts with the goal to encourage the industry to use the asynchronous design methodology. Subsequent implementations, the Amulet2e [Furber 1997, Furber 1999] and the Amulet3i [Furber 1998, Furber 2000, Garside 2000], extended the capabilities of the Amulet1 and made the processor compatible with the ARM7 and ARM9 instruction sets, respectively. Furthermore, in the time of the millennium change, the company Fulcrum Microsystems presented a couple of ultra-high performance asynchronous crossbar switches. The one presented in [Lines 2004] reached a cross-section bandwidth of 780 Gbit/s. The company was taken over by Intel which now produces high-speed ethernet switches based on the technology of Fulcrum. Latest projects include microcontrollers, e.g., TAM16 [Tiempo 2008], HT-80C51 [Solutions 2004], multi-core processor implementations, e.g., SEAforth 40C18 [IntellaSys 2008], and cryptoprocessor cores, e.g., Tiempo DES/3DES, Tiempo AES, and Tiempo RSA/ECC [Tiempo 2013].

In parallel to the evolution of fully asynchronous designs the paradigm of GALS systems was introduced by the PhD thesis [Chapiro 1984] of D.M. Chapiro in 1984. These circuits are a tradeoff between the synchronous and the asynchronous design styles. Systems that follow this concept are composed of separated synchronous modules operating at potentially independent clock speeds. The modules, referred to as synchronous blocks (SBs), are interconnected via asynchronous channels. Over the years, several techniques have been proposed to implement the communication between the separate synchronous blocks. Basically three major approaches are established [Krstić 2007]: synchronizer-based interconnects, channels based on asynchronous FIFOs, and the pausible clock scheme presented by K.Y. Yun and P.R. Donoghue in [Yun 1996]. The GALS design methodology has been applied in various chip designs [Lines 2004, Fan 2010, Krstić 2011, Plana 2011].

2.1.2 Synchronous versus Asynchronous Designs

Most traditional industrial designs are synchronous and there are some good reasons for that. The assumption that computation is done in time slots and the according data is transmitted at repetitive points in time eases the generation of digital designs and has led to the fast evolution of semiconductor devices for the last decades.

However, the growing complexity makes the design and development of VLSI circuits more difficult and designers have to face several challenges. Especially the distribution of a global clock signal becomes a rising problem. To ensure the clock signal to simultaneously arrive at all sequential elements, the clock tree has to be balanced very precisely. This becomes the critical point in the design of complex synchronous systems and results in increased design time and additional circuit complexity.

Another aspect, which is strictly coupled with a complex clock tree, is the power consumption. The sequential elements of synchronous circuits are continuously triggered to store data even when this data does not change. Thus, a huge amount of energy is wasted by useless switching of the clock tree buffers. In complex synchronous designs, up to 50% of the total power consumption is dissipated by the clock tree. Indeed, clock gates can be introduced at different granularity levels in order to avoid switching activity of inactive parts. For example, one can think about disabling entire functional blocks. More fine granular clock gating methods are also possible [Bhutada 2007]. However, this complicates the design of a synchronous system even more.

A promising solution to such upcoming challenges in designing complex VLSI circuits in nano-scaled technologies might be the asynchronous design methodology. Unlike synchronous systems, asynchronous circuits do not presume that all signals are sampled at the same time. The concept of exchanging data only when indicated omits the overhead for distributing a clock signal. This leads to modules that are enabled only when data is available. Consequently, transistor switches are minimized resulting in a potentially reduced power consumption. Therefore, asynchronous designs are suitable for low power systems. Furthermore, due to asynchronous switching activities of the transistors the electromagnetic interference (EMI) is reduced. In synchronous designs the simultaneous switching of transistors causes considerable EMI. This may have negative impacts on other components and could cause system malfunctions. Moreover, aging effects like Negative Bias Temperature Instability (NBTI) permanently affect circuits by changing the threshold voltages of transistors. This increases signal delays which may cause a synchronous circuit to malfunction. Asynchronous circuits are more robust to such effects due to their self-timed properties.

Apart from the commonly known properties, e.g., summarized in [Sparsø 2001],

some more beneficial characteristics have been identified during the recent years. In summary, these characteristics include the following aspects:

- Robustness to changes of the environment (supply voltage, temperature)

  Changes of external parameters (temperature, voltage) of an asynchronous system affect the timing, but not the correct execution of the operation [Fang 2005].

- Robustness to process variations

  In nano-scaled technologies, large process variations can critically affect the threshold voltages of the transistors which leads to increased propagation delay of the logic cells [Cortadella 2010]. This delay gets the order of up to 40% of the clock period of synchronous circuits. Due to their self-timed properties, asynchronous circuits are more robust to such effects.

- Deep subthreshold design

  Today's mobile wireless application demand ultra-low power systems. It has been shown that even systems can be realized that operate with subthreshold voltages of approximately 150 mV [Wang 2006]. With the application of the asynchronous design style, it is expected that the supply voltages can be further decreased while the circuit still operates robustly.

- Low power dissipation

  Asynchronous circuit modules consume power only if it is necessary, i.e., when valid data is applied to the inputs of the module. That minimizes transistor switchings and results in lower power consumption [Furber 1994a, Furber 1997, Nielson 1997].

- Operation at maximum speed

  The operation speed of asynchronous modules is determined by their gate and wire delays, but not by the worst-case latency of the entire system. Therefore, the modules work at their maximum operation speed for their specific inputs. In total, this is close to the average timing.

➢ Reduced electromagnetic radiation

In synchronous circuits the most amount of electromagnetic noise is induced by simultaneous switching of cells triggered by the global clock signal. This is avoided in asynchronous circuits due to the asynchronous update of the state of the individual modules.

➢ Possibility of modular designs

Due to the data exchange based on the concept of indication, the modules of an asynchronous system can be developed independently. This also implies high reusability of the modules.

2.1.3 Asynchronous Handshake Protocols

Typically, asynchronous circuits utilize a convention, i.e., a *protocol*, that coordinates the flow of the data transmission. The most widely used type of such conventions is the class of *handshake protocols*. Although the concrete implementation may vary, the basic principle behind these *protocols* is the same. The *initiator* of a transfer issues a *request event* to start the data transmission. Afterwards, the *responder* indicates the receipt of the data by generating an *acknowledgment event*. The convention defines which of these *protocol events* is generated by the *sender* and which is generated by the *receiver*. Due to this bidirectional nature, these *event-driven* data transmission conventions are called *handshake protocols*, also known as *handshake signalling*.

Figure 2.1 shows the general schematic of this approach. The sender and the receiver are connected via *asynchronous communication channels* comprising control and data signals. As shown in the figure, the asynchronous modules themselves consist of asynchronous control logic (ACL), sequential elements (i.e., registers) and optionally

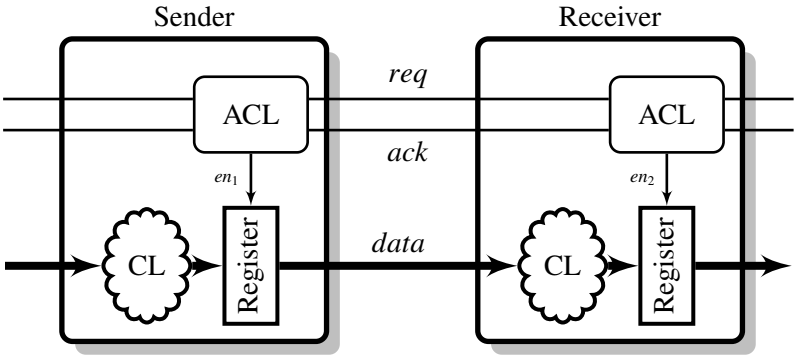


Figure 2.1: Asynchronous handshake circuits

some computational elements implemented by combinational logic that is depicted by the clouds left to the registers. The control logic generates local clock or register enable signals triggering the respective storage elements of the module to store their currently applied input. Depending on the protocol, the request signal is either issued by the sender or by the receiver. The same applies to the acknowledgment signal. For this reason, the handshake control signals in Figure 2.1 are not directed as opposed to the data signals.

As commonly known, handshake protocols can be categorized with respect to three main criteria (see e.g., [Sparsø 2001]): the number of phases, the data encoding and the initiator of the handshake.

Number of Phases This characteristic is related to the number of phases required for the completion of a handshake. Two major types need to be distinguished: *2-phase* and *4-phase* protocols. These types have different properties regarding their power and time consumption as well as the meaning of signal levels and transitions.

In case of 4-phase protocols, the protocol events are encoded within the levels of the handshake signals as shown in Figure 2.2. Therefore, 4-phase handshake protocols are said to be *level-based*. Such protocols are simple to implement. Typically, a request event is issued by setting the respective wire to logical-1. However, also inverted logic might be possible. Accordingly, the responder induces the end of the transfer by setting the acknowledgment signal to logical-1. Finally, the handshake signals have to be set to their initial values starting with the request signal. This is called to be the *return-to-zero* phase of a 4-phase protocol. Furthermore, one has to consider the time window when the data is valid. In this context, one can distinguish between the *early*, the *late* and the *broad* data validity schemes which are illustrated in Figure 2.3.

In comparison to that, *transition-based* 2-phase handshake protocols encode the protocol events into signal transitions. Instead of interpreting the logic level values of signals, these *transition signalling* protocols assign meanings to the edges of the

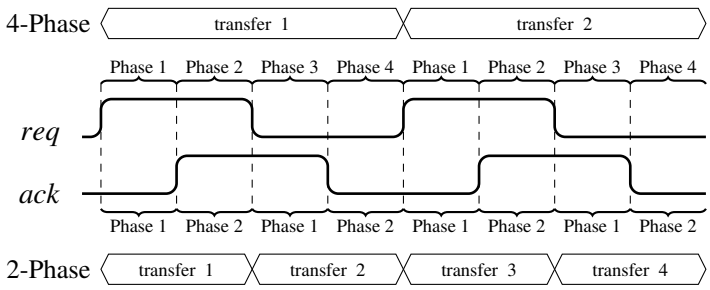


Figure 2.2: Asynchronous handshake protocols

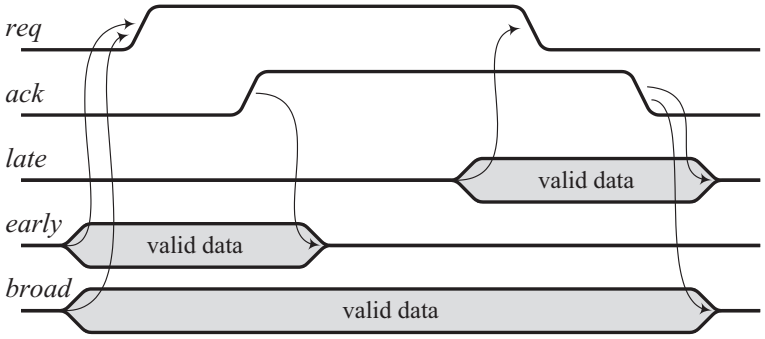


Figure 2.3: Data validity schemes

control signals. Thus, any transition, either rising or falling, has the same meaning [Sutherland 1989]. As a result, 2-phase protocols have no return-to-zero phase. They are more power and time efficient, because the required number of signal switches is reduced to the minimum. However, the implementation of the controllers and/or registers of such circuits was very complex, until the introduction of the MOUSETRAP handshake controller [Singh 2007].

Data Encoding Another criterion considers the encoding of the data. A handshake-protocol with one wire per data bit is called *single-rail protocol*. To coordinate the data transmission, channels that utilize these protocols have explicit request and acknowledgment signals. In this type of protocols, the handshake signal, which is driven by the sender, has to be delayed by the propagation time of the data from the sender to the receiver. This so-called *matched delay* ensures that the data has settled before the protocol event is recognized by the receiver. Due to this strict relationship between the data and the corresponding handshake signal, single-rail handshake protocols are also called *bundled-data* protocols. Figure 2.4 illustrates such a bundled-data protocol with a matched delay on the request line.

As opposed to single-rail protocols, *dual-rail handshake protocols* use two wires

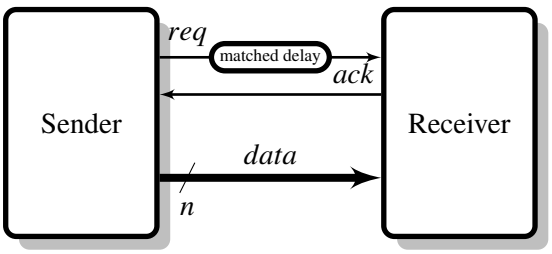
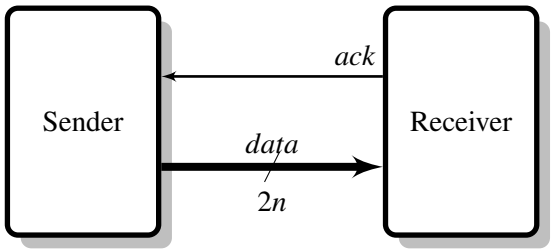


Figure 2.4: Bundled data protocol

	d_f	d_t
NULL	0	0
valid-0	1	0
valid-1	0	1
invalid	1	1

(a) Dual-rail encoding



(b) Schematic

Figure 2.5: Dual-rail handshake protocol

$(d_f d_t)$ to encode a data bit. As given in Table 2.5a, a valid logic-1 is encoded with $(d_f d_t) = (01)$ and a valid logic-0 with $(d_f d_t) = (10)$. If both wires are set to logical-0 then no data is present. This corresponds to the so called *null-value*. The last case, if both signals are logical-1, is a forbidden and invalid value. As shown in Figure 2.5b, dual-rail asynchronous channels have $2 \times n$ wires for n data bits to transmit, but no explicit handshake signal from the sender to the receiver. Instead, the respective protocol event is encoded within the data lines. It is issued when all data bits are valid. Implementations of these protocols comprise dedicated circuitry used for completion detection. For a single data bit, this can be achieved by a two input OR-gate whose inputs are connected to d_f and d_t. For more than one data bit the output of these OR-gates have to be synchronized using a special cell, i.e., the C-element as shown in Figure 2.6.

Besides these two commonly used encodings, there are also protocols with more complex data encodings, e.g., m-of-n-codes. Such protocols encode the data within n bits and depending on the implementation, a valid codeword has m logic-1s or logic-0s, either. If a valid codeword is detected the respective protocol event is issued. Obviously, dual-rail protocols are special forms of n-of-m protocols, where $n = 2$ and $m = 1$.

Initiator and Responder of a Protocol The last criterion concerns the initiator of the handshake. Protocols in which the sender initiates the transmission are called *push*-protocols. In such protocols the sender is the active party and issues the request. The receipt of the data is then acknowledged by the receiver that is the passive, responding party of the protocol. Figure 2.7a illustrates a channel using the push convention. The initiator is indicated by the dot. The opposite is called *pull*-protocol. There, the receiver is the active party and initiates the data transfer by issuing the request. This is shown in Figure 2.7b.

Solution Space of Asynchronous Handshake Protocols Several specific protocol implementations have been proposed in the past each having different properties.

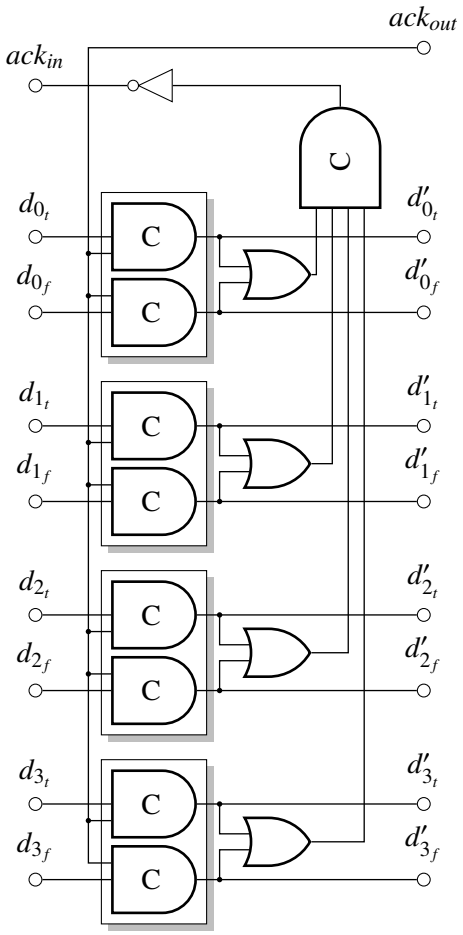


Figure 2.6: 4-bit dual-rail register with completion detection

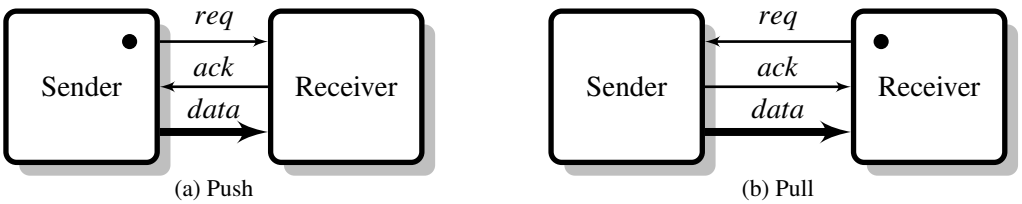


Figure 2.7: Push vs. pull protocols

Finally, as defined in [Sparsø 2001], most common protocol implementations are combinations of the afore mentioned criteria. Therefore, the solution space of these protocols can be described by the cross product shown in Equation 2.1.

$$\Sigma = \{\text{2-phase, 4-phase}\} \times \{\text{single-rail, dual-rail, } m\text{-of-}n\text{-code, } \dots\} \times \{\text{push, pull}\} \quad (2.1)$$

2.1.4 Asynchronous Channels

Based on the defined protocols and the requirements for implementing communication channels, an abstract model for asynchronous channels can be defined. This model includes the minimal set of resources to cover the most common types of asynchronous communication channels and the protocol information required for coordinating the data transmission. As previously identified, an asynchronous channel is a bunch of wires including a set of control signals and a set of data signals. Typically, each of these sets comprise at least one signal. However, it is also possible that the set of data signals is empty. This is the case for control channels used for synchronization purposes only. Thus, an asynchronous channel can be defined in the following way:

Definition 1. *Let $H \subseteq \{req, ack\}$ be a non-empty set of control signals that coordinate the data transfer, where req designates the request or data valid signal provided by the initiator and ack designates the acknowledgment signal delivered by the responder. Corresponding to that, let $v_0 : H \rightarrow \{0, 1\}$ be a function that maps the control signals to digital values representing the initial values of the handshake signals. Furthermore, let $p \in \Sigma$ (see Equation 2.1) be the protocol type that describes the flow of interaction between a sender and a receiver. Finally, let D be a set of data signals. Using these components an asynchronous channel is a quadruple $C = (H, D, p, v_0)$.*

2.1.5 Classification Based on Delay Models

As summarized in [Sparsø 2001], asynchronous circuits can also be classified depending on assumptions concerning their gate and wire delays. Thereby, the assumptions define preconditions under which the asynchronous circuit is supposed to work correctly. Four types can be distinguished: *self-timed* (ST), *speed independent* (SI), *quasi delay insensitive* (QDI) and *delay insensitive* (DI) circuits.

An asynchronous circuit which works correctly with arbitrary positive bounded gate and wire delays is said to be delay insensitive. In consideration of the classical example shown in Figure 2.8, this means that all delays $\delta_A, \delta_B, \delta_C, \delta_1, \delta_2$ and δ_3 can be arbitrary, but the circuit works correctly. Such circuits are robust to any variations of environmental parameters not exceeding threshold values.

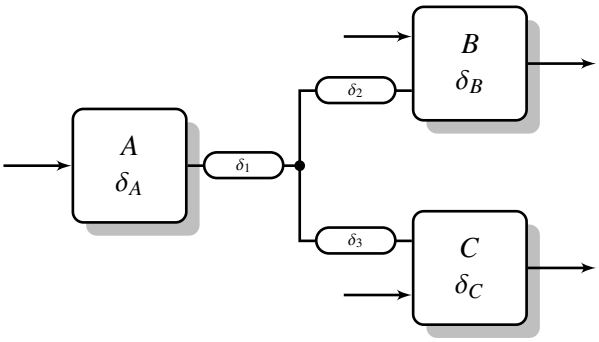


Figure 2.8: Circuit fragment with delays in logical gates and wires [Sparsø 2001]

In QDI circuits, the delays of gates and wires are positive, as well. The difference to DI circuits is that some forks of the circuit are assumed to be isochronic, i.e., the delays of the fork branches are assumed to be equal. For the given example, this means that δ_2 is equal to δ_3.

SI circuits consider only gate delays as being positive bounded. Delays in wires are assumed to be zero, i.e., $\delta_1 = \delta_2 = \delta_3 = 0$. However, the delays of wires can be lumped into the gates, because the gate delays are arbitrary. This means that all forks are assumed to be isochronic.

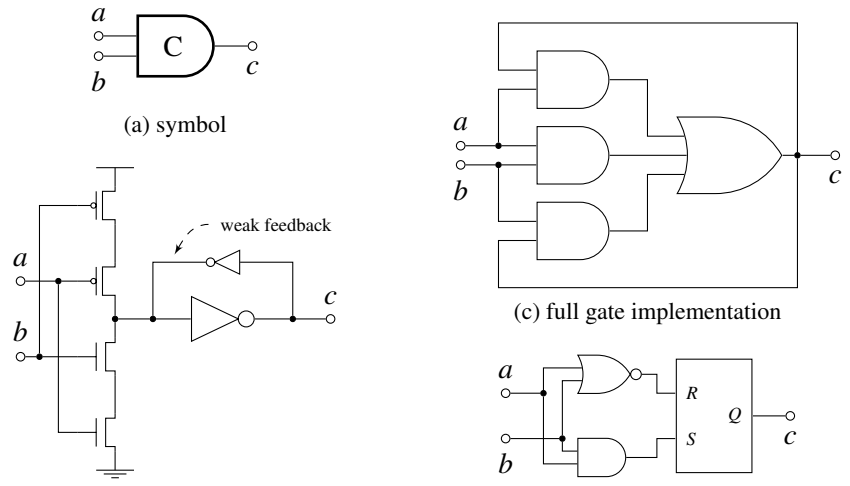
The class of ST circuits has more elaborate timing assumptions under which they work correctly. For example, an asynchronous circuit using a bundled-data protocol is self-timed, since the delay on the request line has to match the delay of the respective data signals.

2.1.6 Elementary Components

Typically, asynchronous circuits comprise special non-standard logic components. The following subsections introduce the most important cells, and shows their functionality and structure.

2.1.6.1 Muller C-element

The *C*-element, invented by D.E. Muller, is one of these basic logic components. It is used to synchronize the logical values of its inputs. Therefore, it sets the output to the value of the inputs if these have the same value. Otherwise, the output remains at its current value. As shown in Figure 2.9, there are several ways to implement this functionality. One of the three common transistor-level implementations is shown in 2.9b. This implementation was introduced by A.J. Martin [Martin 1989]. It uses two cross-coupled inverters forming a latch to store the current output value. In this implementation a weak



(a) symbol

(b) dynamic transistor implementation

(c) full gate implementation

(d) latch implementation

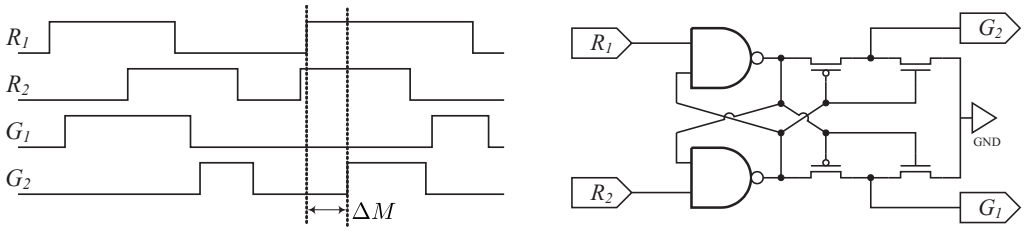
Figure 2.9: The Muller *C*-element

inverter is used as a keeper. This ensures that the pull-up and the pull-down logic networks can overwrite the value stored in the latch. Besides this one, there are also other transistor-level implementations. An overview about these implementations and their characteristics is given in [Shams 1996]. Furthermore, the *C*-element can also be realized such that some inputs influence only the output value on either a rising or falling transition. These *C*-elements are said to be *asymmetric*.

However, the utilization of a transistor-level versions of the *C*-element requires that the cell is included in a cell library. If the cell is not available, one can implement the *C*-element using logical gates, as shown in Figure 2.9c, or using a combination of logical gates and a latch or a flip-flop as shown in Figure 2.9d. However, these versions have to be carefully integrated in order to prevent glitches due to critical signal races. Moreover, such implementations require more cell area and consume more power than transistor-level implementations.

2.1.6.2 MUTEX element

Mutual exclusion (MUTEX) elements are used to realize arbitration logic [Seitz 1980]. This is, e.g., necessary for resolving possible conflicts when accessing a shared resource. From the functionality point of view, the MUTEX forwards the first activity at one of its two inputs to the corresponding output and inhibits the propagation of activities on the other signal as long as the first signal was not deactivated. Figure 2.10a illustrates this behavior.



(a) Behavior (b) Gate implementation

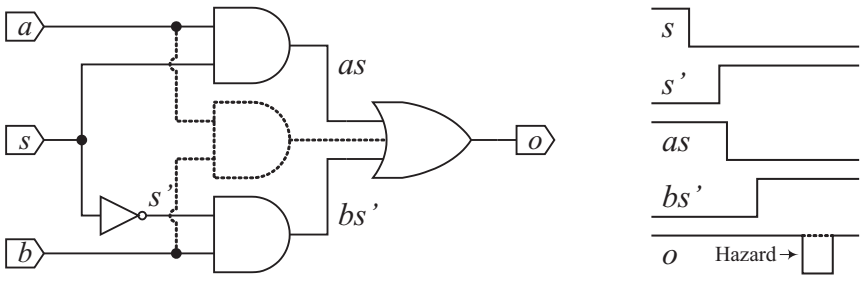
Figure 2.10: The MUTEX-element

As shown in Figure 2.10b, these elements are typically realized using a reset-set-latch structure. In order to suppress metastability of the outputs, the MUTEX-element comprises a metastability filter connected with the outputs of the latch. Usually, incoming signal transitions do not occur at the same time. Therefore, the latch has enough time to stabilize. However, if both input signals change from logical-0 to logical-1 in a time window shorter than the settling time of the latch, then the entire latch goes into a metastable state. As shown in [Anderson 1991], the time window ΔM of this metastable state cannot be determined.

2.1.7 Design Issues of Asynchronous Circuits

2.1.7.1 Race Conditions and Hazards

Race conditions are effects which may influence the output of an integrated circuit due to the sequence and/or timing of uncontrollable signal transitions. As a result of a race condition a circuit may produce incorrect signal transitions at its outputs. These transitions are called *hazards*. As an example, consider the multiplexer implemented with logical gates as shown in Figure 2.11a. According to this, Figure 2.11b illustrates the



(a) A multiplexer (b) Timing diagram

Figure 2.11: Hazard effect

timing diagram of the multiplexer assuming both signals a and b to be logical-1 while the control signal s switches from logical-1 to logical-0. For now, assume that the dashed gate is not present. Due to the delay of the inverter, the output signal bs' of the lower AND-gate rises later than the falling edge of the upper signal as reaches the OR-gate. This causes a short logical-0 pulse, i.e., a hazard, on the output o.

Such an effect can cause an asynchronous circuit to malfunction, since these circuits are potentially sensitive to every signal transition. To prevent such behavior, asynchronous circuits (or at least their components that are exposed to race conditions) can be equipped with redundant logic. In this particular example, an additional AND-gate can be introduced causing the output to be stable. Obviously, this logical gate is redundant from the logical point of view, but it inhibits the hazards.

2.1.7.2 Metastability

Metastability occurs in sequential elements, such as flip-flops, when at least two inputs change in a specific time window causing internal feedback signals to compete with each other where each of the transitions of one of the internal signals results in a change of the other one [Chaney 1973]. This is, for example, the case when setup and hold times of a flip-flop are violated. In this case the quasi-simultaneous change of the input and the clock signal causes internal nodes of the flip-flop to oscillate for a potentially unbounded amount of time. Theoretically, this metastable state can persist indefinitely. However, due to external parameters (temperature, induced noise, different wire delay) one of the competing signals dominate the other and the sequential cell settles in a stable state [Couranz 1975]. Such conditions can be found in almost all sequential elements having internal (combinational) feedbacks. The ability to leave a metastable state at a time t is defined by the probability

$$P(\text{stable}_{\triangle t}) = 1 - e^{-\frac{t}{\tau}} \tag{2.2}$$

where τ is the *settling time* of the sequential element. Both the size of the critical time window, in which the sequential cell becomes metastable, and the settling time of the cell can be determined by simulations and experiments.

Unfortunately, until now no method exist to fully avoid metastability. This is a problem for synchronous circuits that has an asynchronous input. If such a signal switches in the critical time window, then this transition may cause metastability of a sequential cell. Therefore, metastability is basically a problem of synchronous designs, where an asynchronous signal has to be migrated into the clock domain of a synchronous reading block. Asynchronous circuits prevent a priori the effects of metastability due to

their special design. One way to hide the metastability of a cell is the introduction of a metastability filter as applied in the MUTEX-element. A further simple countermeasure, which significantly reduces the probability of the system to malfunction, is to resynchronize asynchronous inputs into the clock domain of the reader of the signal [McCluskey 1986]. This can be achieved by the utilization of so-called *synchronizers*, which consist of consecutively connected flip-flops or latches triggered by the reader's clock signal. Thus, an asynchronous signal may violate the setup and hold time of the first flip-flop. However, a second flip-flop, as it is used in the two-flop synchronizer shown in Figure 2.12, considerably reduces the probability that the output of the synchronizer is metastable, too. In order to further reduce the Mean Time Between Failure (MTBF), the sequential cells of the synchronizers are specially adapted, such that the settling time τ is as small as possible [Semiat 2003, Zhou 2006, Zhang 2010].

Several implementations of such synchronizers have been proposed, each having different properties regarding synchronization latency, area overhead, metastability filtration and MTBF [Dike 1999, Kinniment 2002]. A comprehensive overview of the most common synchronizers and their pitfalls is presented in [Ginosar 2003]. Although synchronizers are very effective, their integration results in performance loss, since additional clock cycles are required to pass the signals through the extra register stages.

2.1.7.3 Non-determinism

In asynchronous circuits, one typical cell, which is exposed to metastability, is the MUTEX-element. Since the result of a metastable state is sensitive to external factors, such MUTEX-elements behave unpredictably. Thus, a MUTEX-element may arbitrarily select one of its inputs and forward the activity to the corresponding output rather than the other one. Furthermore, due to the lack of a global clock signal, the behavior of an asynchronous circuit is influenced by PVT variations. This strengthens the unpredictability of the circuit. As a result, the next state of an asynchronous circuit is not only determined by its current inputs and its state. This is known as *non-determinism*. However, one can distinguish between two types of non-determinism in such a system.

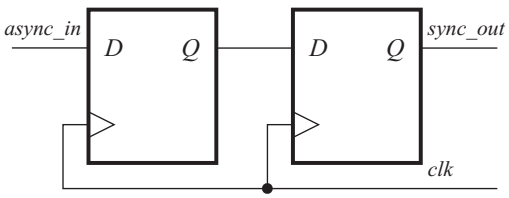


Figure 2.12: A two-flop synchronizer

On the one hand, an asynchronous circuit might be *functionally non-deterministic*. This might be the result of arbitration logic controlling the access to a shared resource. Eventually, the order of the responses of the shared resource becomes unpredictable. From an external point of view, the responses of one output channel may occur in a different order. To summarize this more formally:

Definition 2 (Functional determinism and non-determinism)**.** *A circuit is called to be functionally deterministic if the sequences of values of all output signals are well defined. Correspondingly, a circuit is called to be functionally non-deterministic if the sequence of values of at least one output signal cannot exactly be determined.*

On the other hand, non-determinism may only affect the timing of a circuit. Basically, due to the lack of the global clock signal, the timing of an asynchronous circuit is determined by the delays of gates and wires. These delays are susceptible to PVT variations. Although the timing variations of the gates and wires are proportionally small, their accumulated value might be large enough to affect the overall timing with respect to an external timing resource (e.g., system or tester clock). Thus, the overall system becomes *timing non-deterministic*. To conclude this:

Definition 3 (Timing non-determinism)**.** *A circuit is called to be timing non-deterministic if the point in time of at least one output signal cannot exactly be determined.*

2.1.7.4 Deadlocks

Apart from the afore mentioned difficulties, the design of asynchronous systems is an exhausting task, since the event-driven data exchange is more complex. This is due to the fact that the modules are triggered by events generated by the environment or by other modules. A tricky situation happens when a set of processes (modules of an asynchronous system) wait for an event of a process that is also in this set. Then, these processes are blocked and may not evince further activity. This situation is called *deadlock* [Tanenbaum 2007]. The cause of a deadlock can be versatile. The most prominent causes are design errors resulting from unconsidered behavior of a system. But also faults may lead to a deadlock of a system. Furthermore, deadlocks can be the result of unexpected interaction of a system with its environment. This in turn might be caused by non-deterministic behavior of a system. With respect to this work consider the following example:

Suppose a functionally non-deterministic DUT shall be tested with a test system expecting full deterministic behavior of the DUT. Obviously, this is an unfortunate situation, but assume that the test engineer did not know about the non-deterministic behavior.

The DUT has one input channel and delivers a response on one of its output channels o_1 and o_2, either. Furthermore, assume that the DUT has an arbiter controlling which of the channels shall deliver the response. Suppose that the test system is only able to listen to one channel at a time. Now, consider the case that the tester expects a response at o_1, but due to the non-deterministic behavior, the response is delivered at o_2. Since the test system will not apply further stimuli unless it had received a response, the entire system is in a deadlock.

Such conceptual errors and the resulting deadlock situations should be identified in an early stage of the system design. Therefore, formal validation techniques can be applied to check for system deadlocks. However, current techniques are not able to handle large designs. As a result, designs have to be manually validated using extensive simulations or by decomposing the system in subsystems that can be automatically validated.

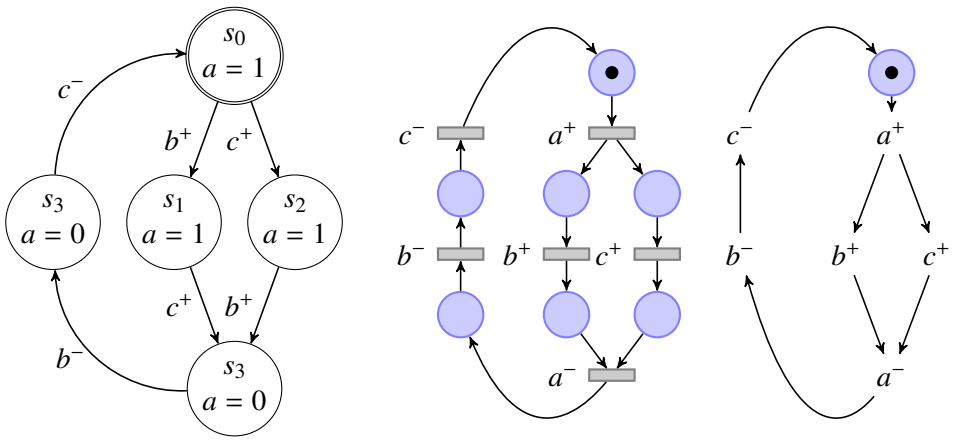
2.1.8 Modelling and Design of Asynchronous Circuits

A couple of modelling and design techniques have been proposed over the years. This section introduces the most important approaches.

2.1.8.1 Modelling Asynchronous Circuits

Asynchronous circuits are typically modelled using graphs that express the causal relations of the circuit activities. As well as in the synchronous world, often graph-based description of finite state machines (FSMs) are used to model asynchronous circuits. Moreover, there are also other types of graphs that offer mechanisms to describe concurrency of the activities of an asynchronous circuit. The most important ones are *Petri-Nets* and *Signal Transition Graphs (STGs)*. Figure 2.13 illustrates an example circuit which is modelled using an asynchronous FSM, a Petri-Net and an STG.

Asynchronous finite state machines are a mathematical model of a class of automatons. These automatons can be described by graphs that have a set S of nodes, called *states*, and a set of edges. The graph describes two functions: the *state transition function* δ and the *output function* μ. The *state transition function* $\delta : S \times I \rightarrow S$ defines the next state of the FSM depending on the currently applied input of an input alphabet I and the current state of the circuit. The *output function* specifies the output of the FSM. With respect to this, one has to distinguish between two types of state machines: Mealy and Moore-FSMs. In a Mealy-FSM, the *output function* is defined as $\mu : S \times I \rightarrow O$. It maps the current state and inputs to an element of the output alphabet O. To illustrate this, the edges are labelled with the applied input and the corresponding output. This



(a) FSM (b) Petri-Net (c) STG

Figure 2.13: Graph-based descriptions of an asynchronous circuit

is different in a Moore-FSM. There, the output of the FSM is determined only by its current state, i.e., μ is defined as $\mu : S \rightarrow O$. Therefore, the nodes of the graph include the specification of the output in this state.

Petri-Nets were invented by C.A. Petri in the 1960's. These nets are directed graphs with two disjoint sets of nodes – the set T of *transitions* and the set P of *places* – connected with a set E of *edges*. The edges have weights defined by a function $W : E \rightarrow \mathbb{N}$. Each edge $e \in E$, which starts from a transition node, ends in a place node, and vice versa. Correspondingly, a transition has *input* and *output places*, whereas a place has *input* and *output transitions*. In its basic version, each place can be filled with an arbitrary number of *tokens*. A *marking* of a Petri-Net is a specific occupancy of places with tokens. Starting from a specific marking m, a transition t transfers a Petri-Net to a different marking m', such that each edge $e_i \in E$ ending in t consumes $W(e_i)$ tokens from the respective input place. Furthermore, the transition causes each edge $e_j \in E$ starting from t to produce $W(e_j)$ tokens at the respective output place. The transformation caused by the transition t is called *firing* of t and is denoted by $m \xrightarrow{t} m'$. The places in combination with tokens and the weights of the edges are the preconditions for the transitions to fire.

The transitions and places of a Petri-Net can be interpreted at different levels of abstractions. In the context of modelling circuits, the transitions typically designate signal switches and a marking of a Petri-Net can be seen as the state of a circuit. However, at higher level of abstraction, transitions might also be interpreted as data transfers between modules depicted by the places. In this case, the tokens designate the data exchanged between the modules.

Signal Transition Graphs are a special variation of Petri-Nets. These graphs are *free-choice* Petri-Nets that interpret the transitions as rising and falling edges of the signals of a circuit. Rising edges, also called *positive transitions* ($0 \rightarrow 1$), of a signal s are denoted by s^+. Falling edges, i.e., *negative transitions* ($1 \rightarrow 0$), are indicated with s^-. The *free-choice* property means that for every two transitions t_1 and t_2, which share the same place p, p is the only input place for both t_1 and t_2. STGs are typically simplified, such that they do not contain places with only one output transition. The edges, which would lead to such a place p, are directly connected with the output transition reachable from p. The only exception is the set of places which are required to define the initial marking of the STG.

2.1.8.2 Design Approaches

Various modelling languages and tools have been developed for the design of asynchronous circuits. One of the first approaches was the language CSP (Communicating Sequential Processes) proposed by C.A.R Hoare [Hoare 1978]. It was developed for the description of concurrent processes at a high level of abstraction. By this, it overcame limitations of existing languages with respect to synchronization and communication of parallel execution units. CSP is a member of a large class of languages to describe concurrent systems. Other prominent examples are Tangram/Haste [van Berkel 1991, van Berkel 1993], Tast [Dinh Duc 2002], Balsa [Edwards 2000, Edwards 2002], and Tiempo ACC [Zhou 2011, Tiempo 2012], which are intended specifically for the design of asynchronous VLSI circuits.

Tangram was developed during a research project of the Philips Laboratories in the 1990's. The developments ended up in a spin-off company, called Handshake Solutions. After that, the language was renamed to Haste. The language is very similar to Pascal. However, it is extended by constructs to express concurrency and communication as well as hardware-specific constructs. The EDA environment of Tangram/Haste provides a tool chain, called TiDE, that covers the entire frontend design flow for handshake circuits.

Tast is another CSP like language. It was developed at the TIMA Laboratory in the period around the millennium. The respective CAD tools comprise a compiler that allows the generation of several outputs including behavioral VHDL models for simulation, and gate level models according to the Micropipeline and QDI styles.

A very similar language is Balsa which was developed at the University of Manchester. Similar to Tangram/Haste, the design environment of Balsa comprises the language itself and respective tools to synthesize and simulate handshake circuits. Both,

TANGRAM/HASTE and BALSA originally use a control-driven design style for the synthesized circuits. This means that an explicit control network coordinates the circuit activities. The modules of such a circuit have at least one control channel, which is driven by the control network. Typically, this control network is slower than the data is processed. This leads to performance limitations compared to data-driven architectures which omit the control network. Instead, the activities of a data-driven circuit are directly coordinated by the data tokens propagating through the circuit. With the PhD thesis of S.M. Taylor [Taylor 2007], such a data-driven approach has been integrated into the BALSA framework. Therefore, the language has been extended by further constructs. A similar data-driven approach has been introduced with the alternative description language BISCOTTI [Jin 2009].

As opposed to the afore mentioned approaches, TIEMPO ACC (Asynchronous Circuit Compiler) is a tool to generate asynchronous circuits from a model written in a standard hardware description language. The input of ACC is a SystemVerilog model which is synthesized into a gate-level netlist in Verilog format.

Another class of tools addresses the design of asynchronous self-timed controllers. Important examples are PETRIFY [Cortadella 1996], 3D [Yun 1992a, Yun 1994], and MINIMALIST [Fuhrer 1999]. These tools typically utilize STGs for modelling the circuits. With respect to this, a fundamental work was presented in PhD thesis of by T.-A. Chu [Chu 1987]. He proposed a technique for synthesizing VLSI circuits from graph-theoretic specifications.

PETRIFY was developed by a group of researchers around J. Cortadella in the late 1990's. It is used to synthesize asynchronous FSMs from STGs. A key issue of the tool is the creation of hazard-free input-output mode circuits. Similar tools are 3D and MINIMALIST. However, unlike PETRIFY, these tools are used to generate Asynchronous Burst Mode Machines (ABMMs) from FSM specifications. Such ABMMs are introduced later in Section 2.1.9.2.

Finally, other techniques are based on the transformation of a synchronous to an asynchronous circuit [Cortadella 2004, Branover 2004, Jährig 2004]. These *desynchronization* techniques replace the clock tree of a synchronous circuit by an asynchronous handshake network. In order to automate this process, respective tools were developed that analyze the synchronous circuit with respect to communicating modules. By this, a graph is created that is the base for the generation of the respective bundled-data circuit.

2.1.9 Typical Architectures of Asynchronous Circuits

Several architectures of asynchronous circuits have been proposed in the past. These can be divided into two groups: data path and control path architectures. Apart from fully asynchronous circuits, another system architecture has become popular during the last 20 years: GALS systems. These systems utilize asynchronous design paradigms in order to establish different clock domains. This promising system architecture often uses asynchronous handshaking. Therefore, it is worth to introduce it together with the most important fully asynchronous circuit architectures.

2.1.9.1 Asynchronous Data Path Architectures

Several data path architectures have been proposed for the design of complex asynchronous systems. These architectures are basically composed of individual asynchronously communicating modules. Each of these modules consists of a register bank, control logic coordinating the update of the registers, and optional combinational logic. The most important architectures are described in the continuation.

Muller-Pipeline The first regular asynchronous circuit architecture was the *Muller pipeline* invented by D.E. Muller. As shown in Figure 2.14, this structure uses the 4-phase single-rail handshake protocol and consists of consecutively connected pipeline stages. Each stage is composed of a register implemented by latches and simple ACL comprising one *C*-element and an inverter.

The advantage of this pipeline is its regular structure. It operates as a ripple-through-FIFO. The operation of this circuit is the base for most common asynchronous circuits. Initially, all control signals are logical-0. When the sender issues data on the left port of the pipeline, the corresponding request is set to logical-1 (req_{in}^{+}). Then, the output of the first *C*-element becomes logical-1 causing the first latch to be transparent such

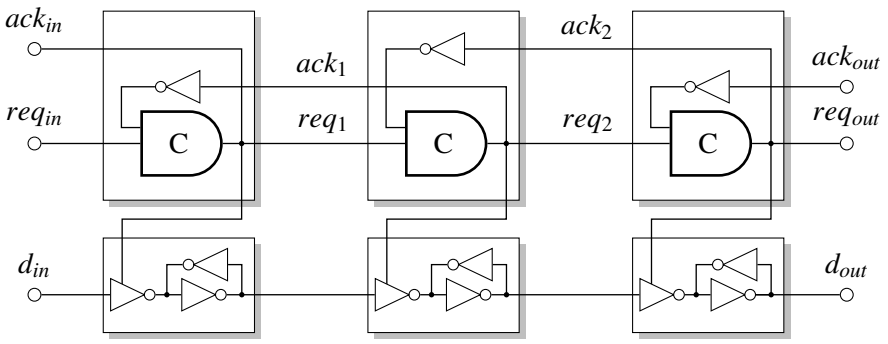


Figure 2.14: 4-phase Muller pipeline

that the data enters the first register stage. Accordingly, req_1 and ack_{in} become logical-1 as well, where ack_{in}^{+} indicates that the data has been received. Then, the sender can set req_{in} to logical-0. In parallel, req_1^{+} indicates that the first stage issues data to the second stage. This causes the second C-element to set its output to logical-1. Consequently, the second latch is transparent and the data enters the second stage. In this way, the data ripples through the entire pipeline. However, in case one stage is full, then its preceding stage is blocked since the *ack* between these stages is still logical-1. The data can be transmitted only if this acknowledgment signal becomes logical-0. This example perfectly illustrates the self-timed nature of the circuit.

Several different variations of the Muller pipeline have been proposed in the past. A popular variation is the 4-phase dual-rail pipeline as shown in Figure 2.15. In contrast to the single-rail implementation this pipeline architecture is *delay-insensitive*. The pipeline registers are implemented via two C-elements per data bit. The OR-gate realizes the completion detection.

Micropipelines In 1989 I.E. Sutherland presented an asynchronous pipeline architecture that is based on the 2-phase transition signalling protocol. The pipeline uses special *capture-pass* storage elements each composed of two parallel latches. Thus, these elements can store two data bits simultaneously. Thereby, the *capture*-signal defines which of the two latches shall store the value currently applied at the data input. The outputs of the latches are connected to a switch controlled by the *pass*-signal. This switch selects the latch to connect with the output of the cell.

Using this kind of latches in combination with its interconnection as shown in Figure 2.16, the pipeline works as follows: The sender on the left port of the pipeline issues data and performs a signal transition at req_{in}. This forces the data at the input of the register to enter the lower latch. In parallel, the transitions ack_{in}^{+} and req_1^{+} fire concurrently, which indicates that the transfer has been completed and that the data can be transferred to the

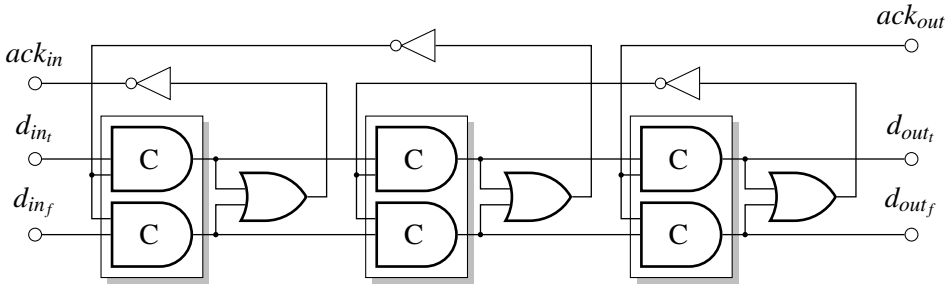


Figure 2.15: Dual-rail Muller pipeline

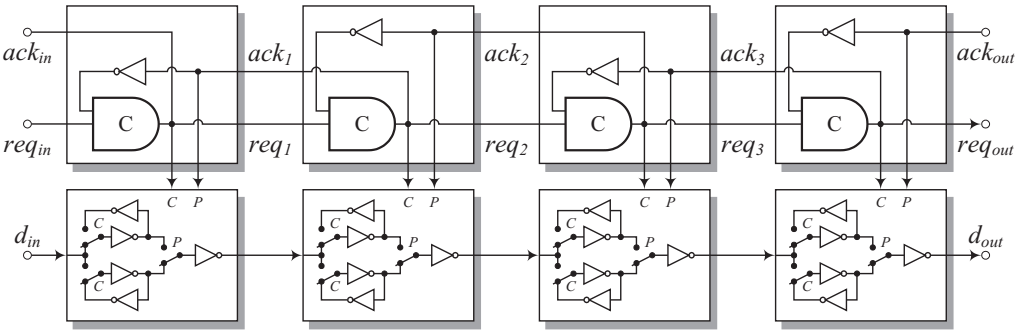


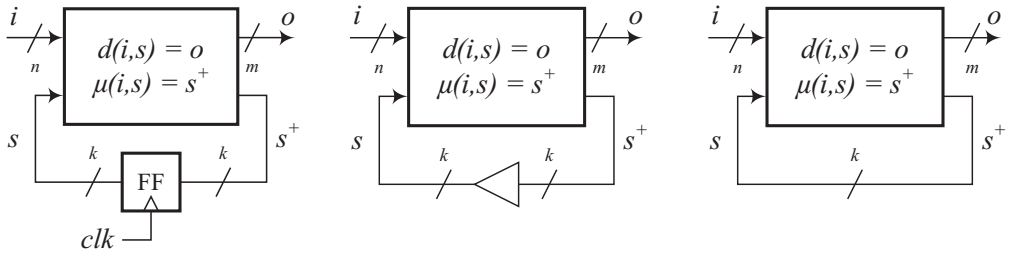
Figure 2.16: 2-phase Micropipeline

second stage. If, at this point, the sender on the right side issues new data before the first token has reached the second stage, then the new data token is stored in the upper latch of the first register stage. When the request transition req_1^+ has reached the second stage, it will force the respective C-element to switch. Thus, req_2^+, ack_1^+ and the capture signal of the second register fire. Thus, the data is stored in the lower latches of the second register. Due to the firing of ack_1^+ the switches of the first register will connect the upper latches with the output of the register. Hence, the new data token is applied to the inputs of the second register stage. In this way data tokens are rippling through the pipeline, such that two consecutive tokens are alternately stored in the upper and the lower latch of the respective storage elements.

2.1.9.2 Control Path Architectures

Asynchronous control path architectures typically implement control logic required for the coordination of the activities of asynchronous designs. Simple examples are the register control logic blocks of the Muller- or Micropipeline. However, these circuit blocks may also implement quite complex state machines. Asynchronous control path architectures are often described using traces of causal relations between the transitions of their input and output signals. Petri-Nets and STGs are the most commonly used techniques for specifying such traces [Sparsø 2001]. In the following, the basics of these circuit architectures are outlined.

Asynchronous Finite State Machines As well as synchronous FSMs, asynchronous FSMs are circuits with hysteresis, thus, their outputs depend on the current inputs and the state of the circuit. However, unlike synchronous FSMs, shown in Figure 2.17a, asynchronous circuits have combinational feedback loops without sequential cells [Miller 1965]. Therefore, the change of the circuit state can happen at any time rather



(a) synchronous (b) asynchronous (fundamental mode) (c) asynchronous (input-output mode)

Figure 2.17: Finite state machines

than at repetitive points in time. However, there are often assumptions on the application of inputs and the generation of outputs of an asynchronous FSM. These assumptions are based on the *fundamental mode* or the *input-output mode* principle, either [Sparsø 2001].

In *fundamental mode* only one input is allowed to change. Furthermore, the time between input changes has to be greater than the settling time of the circuit. Therefore, the restriction on the environment is an absolute minimal time between input signal changes. This ensures the circuit to stabilize and, therefore, prevents inconsistent states. As shown in Figure 2.17b, asynchronous fundamental mode FSMs assume delays on the feedback lines which store the current state.

In *input-output mode* the environment can change the inputs, although the circuit has not stabilized after a previous input change. The restrictions on the environment are causal relations between transitions on the input and output signals. These relations are typically described using Petri-Nets or STGs. The model of an asynchronous input-output-mode circuit is shown in Figure 2.17c.

Asynchronous Burst Mode Machines are a class of Huffman circuits and a special type of asynchronous finite state machines. They originate from the automaton theory in the 1950's and 60's [Huffman 1955, McCluskey 1963, Unger 1983]. ABMMs extend the concept of fundamental mode circuits by allowing sequences of input and output changes. These signal changes are aligned in so-called *bursts*, i.e., a set of one or multiple hazard-free signal transitions without any predefined order. The circuit receives input bursts and produces output bursts without going through any transient state. However, also these circuits have some restrictions. Input bursts need to be complete and are applied after a complete output burst. This has some implications to the set of valid inputs. An input burst cannot be a subset of another input burst. Furthermore, each state can only be accessed with a unique set of input values [Gill 2005].

ABMMs are mainly used to implement asynchronous controllers that operate with low delay [Yun 1992b, Yun 1994, Rutten 1997, Fuhrer 1999]. A technique for the automatic-synthesis for such asynchronous burst-mode controllers was presented by S.M. Nowick [Nowick 1995].

2.1.9.3 Globally-Asynchronous Locally-Synchronous Circuits

A tradeoff between the synchronous and the asynchronous design style is provided by the GALS system paradigm. It was presented in the PhD thesis [Chapiro 1984] of D.M. Chapiro in 1984. GALS systems benefit from the advantages of both the synchronous and the asynchronous design methodologies in order to cope with the afore mentioned issues related to a global clock signal while still maintaining at least parts of the synchronous design flow.

The general idea behind this approach is to decompose a system into several interconnected synchronous blocks (SBs). These blocks are triggered by individual clock signals that basically do not have to correlate to each other. For the communication, the SBs are interconnected via asynchronous channels that compensate the differences of the clock signals to ensure safe data transmission. Typically, the SBs are embedded into asynchronous wrappers that interface the SBs to the asynchronous channels. These wrappers may provide multiple input and output ports implementing the interface logic for the channels. Figure 2.18 illustrates such a GALS system with several interconnected SBs.

Over the years, several GALS system architectures have been proposed. A comprehensive overview of the most important architectures is given in [Krstić 2007]. The main difference between these architectures is the way of interconnecting the SBs, thus, the realization of the asynchronous channels. Basically, one can think about three tech-

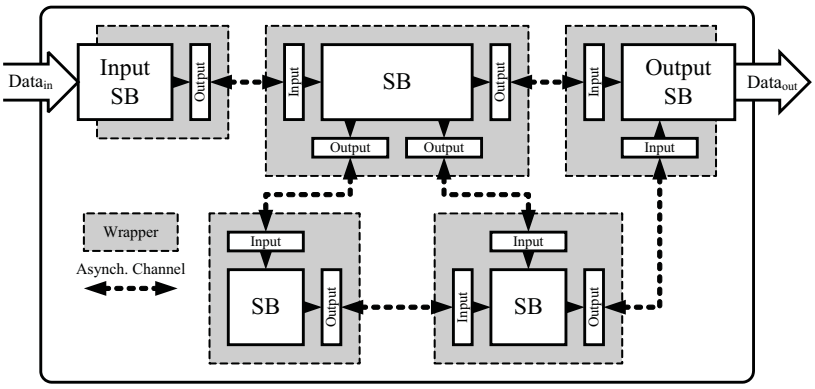


Figure 2.18: A GALS system

niques for a safe data transfer. First, the asynchronously arriving signals are migrated into the clock domain of the block reading these signals. The second technique is to compensate the differences of the clock signals of the communication partners. The last possibility is to synchronize the clocks during the data transfer.

As summarized in [Krstić 2007], three main GALS interconnect schemes have been established based on these theoretical approaches: synchronizer-based GALS systems, GALS based on FIFO interconnects and GALS using a stoppable or so called pausible clock. The next sections provide a short overview about these GALS architecture types.

Synchronizer Based Interconnects The application of synchronizers is a well-known technique of interconnecting modules operating with different clock frequencies. Synchronizer-based interconnects rely on the concept of migrating asynchronous signals into the clock domain of the reading block. To this end, synchronizers are integrated onto the handshake signals of the communication channel as shown in Figure 2.19. The data signals remain unaffected with this approach.

Synchronizer-based interconnects are very effective with respect to their costs. However, the synchronizers increase the communication latency. This is the major disadvantage of the approach. For example, in case of using a two-flop synchronizer solution, a 4-phase handshake requires at least six clock cycles to complete instead of four cycles. Therefore, this approach might not be feasible for high-speed communication channels.

FIFO Based Interconnects With this scheme, the communication partners are connected via asynchronous FIFOs providing an input and an output port with independent clocks. These FIFOs compensate the differences between the clock phases of the communicating parties. Thereby, the compensation capability is closely coupled to the depth

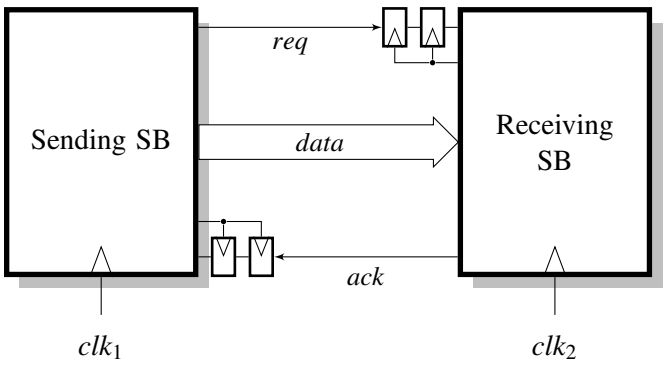


Figure 2.19: An asynchronous communication channel based on synchronizers

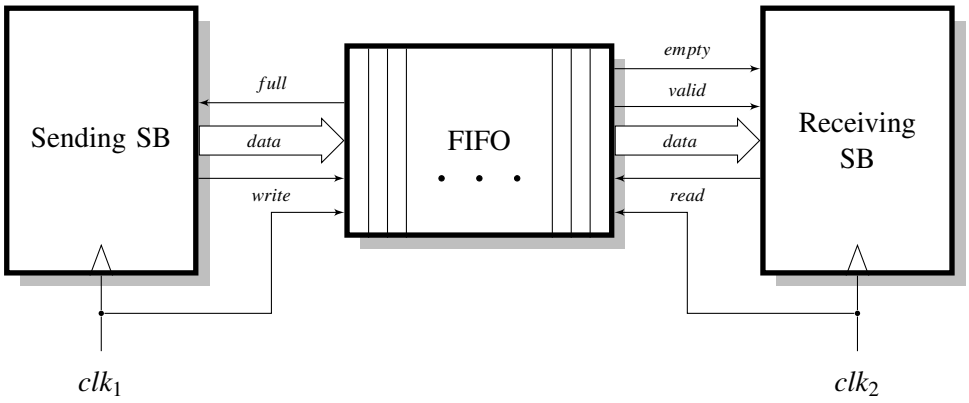


Figure 2.20: A FIFO-based point-to-point communication channel

of the FIFO. Due to the nature of the FIFOs, this scheme can only be applied to point-to-point communication channels as shown in Figure 2.20.

However, there are also more sophisticated approaches based on the utilization of FIFOs. Typically, asynchronous Network on Chip (NoC) architectures integrate FIFOs into routers and network adapters in order to buffer the packets and compensate the different timings of the system components. For example, in [Beigne 2006] an asynchronous NoC based GALS architecture is proposed that uses FIFOs for the synchronization of the SBs and the NoC.

The major advantage of the FIFO interconnect scheme is its simple implementation and general applicability to all kinds of interconnections, i.e., asynchronous-synchronous, synchronous-synchronous, asynchronous-asynchronous connections. Additionally, this scheme does not affect the operation of the SBs. On the other hand, the main disadvantage lies in the significant area overhead required for the FIFO. Furthermore, depending on the realization of the FIFOs, they often introduce additional latency. Although FIFOs with high throughput and low latency can be realized, as, e.g., shown in [Chelcea 2000], the performance loss might still be critical for some applications.

Pausible Clocking The concept of GALS based on pausible clocking was proposed by K.Y. Yun in [Yun 1996]. It has been successfully applied in several GALS systems [Yun 1999, Muttersbach 2000, Krstić 2005b]. With this interconnect scheme, the SBs are clocked by local ring oscillators whose clock generation can be stopped during a data transfer. The general idea is to stop the generation of the local clock signal if the asynchronous handshake signals arrive in the setup and hold time window of the reading block. To this end, each SB is embedded into a wrapper comprising the clock generator as well as input and output ports. The input and output ports implement the

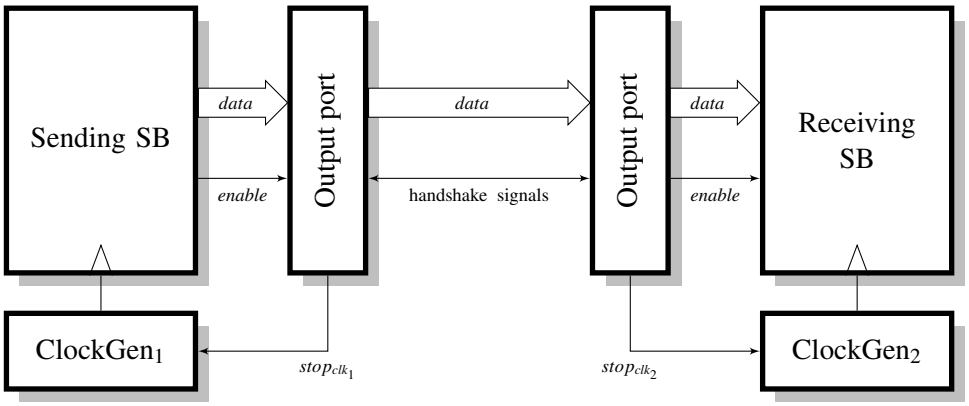


Figure 2.21: Sender and receiver of a pausible clock communication channel

asynchronous interface logic and are connected to the clock generator. When a protocol event is generated by one of the communicating partners, then the port, which receives this event, sends a request to the clock generator to pause the clock. If this request occurs in the setup and hold time window of the reading interface, then it is rejected until the end of the window. In this case, the protocol event is recognized one cycle later. If the request occurs right before the window, then the clock generation is stopped for a short time. This lets the asynchronous signal settle in the reading interface. Afterwards, the clock generation is continued. In all other cases, the ring oscillator directly acknowledges the request from the port. This means that the asynchronous signals can be safely stored. Figure 2.21 shows the general scheme of the pausible clock approach.

2.2 Testing of Asynchronous Circuits

In parallel to the evolution of design methods, various techniques have been developed for the test of asynchronous circuits. Before continuing with the discussion about other works related to the subject of this thesis, the basics of testing are outlined. This includes basic terms, such as *fault models*, and techniques that can be applied to test the functionality of an asynchronous circuit.

2.2.1 Fault Models

Fault models abstract from real physical defects or environmental influences that result in malfunctions of a circuit. They are applied to simplify the test generation. To be more precise, a test can only be generated for a specific fault model. The model defines the characteristics of the covered faults. Fault models can be defined at different levels of abstraction. One commonly distinguishes between transistor level, logic level, and system level faults and respective models. Furthermore, fault models can be classified in certain ways. The most important models in the context of this work are described below.

2.2.1.1 Functional Faults

Functional faults are defined in conjunction with a functional model of the DUT. Thus, a functional fault may change the truth table of a component or inhibit an operation [Abramovici 1990]. Typically, functional faults can be modelled using fault variables. Using these variables, a system S, which has a functional fault f, can be remodelled to S_f such that it reflects the operation of S in the fault-free case. In the faulty case, S_f shows the effect of f [Menon 1978]. For example, consider a functional fault that affects the operation of an ALU to perform an addition instead of a subtraction. This can be modelled in the following way:

$$c = \begin{cases} a + b & \text{if } f \text{ is present} \\ a - b & \text{otherwise.} \end{cases}$$

Unfortunately, this technique requires explicit definition of the faults of interest. Since the solution space of altering a function of a circuit is unbounded, it is not possible to automate the process generating a test that fully tests for all possible functional faults [Abramovici 1990].

2.2.1.2 Structural Faults

Faults defined in conjunction with a structural model, e.g., a gate netlist, are referred to as structural faults. These faults assume the components of a circuit to be fault-free. Only the interconnections of the components are affected. Typically, such faults are shorts and opens that can be modelled at different levels of abstraction [Abramovici 1990].

The stuck-at fault model is a simple model that supposes at least one wire of a circuit to have a constant value, thus, logical-0 or logical-1 either. It is the most commonly used fault model in IC testing. One can distinguish between the *single* and the *multiple* stuck-at fault model. Thereby, the complexity of generating a test increases exponentially with the number of faults assumed to be simultaneously present in the circuit.

A stuck-at fault within the data signals of an asynchronous circuit can directly be tested using the same *test sequence* as its synchronous counterpart. However, in asynchronous circuits the effects of stuck-at faults can be even more critical. For example, a stuck-at fault within the handshake signals can result in a deadlock of the entire circuit [Martin 1991].

Stuck-opens, shorts and bridging faults are more fine granular fault models. These fault models are more realistic, since they model real defects of semiconductor devices at the transistor level. A *stuck-open fault* assumes a break of a certain wire. Typically, such faults on unidirectional wires appear as stuck-at faults. *Shorts* are interconnections between two wires. One can distinguish between a short of a signal wire with power or ground, and shorts between two signal wires. Similar to opens, shorts between a signal wire and power or ground cause the signal to be constant, thus, they appear as stuck-at faults. Shorts between signal wires are called *bridging faults* and result in a new function of the affected lines. Based on the resulting logical function, one distinguishes between AND and OR bridging faults [Abramovici 1990].

Although these models perfectly reflect defects in ICs, they are often not used. The reason for this is the complexity of testing such faults, which typically exceeds the limits in test time and costs. Fortunately, most of these faults are covered by fault models at a higher level of abstraction, e.g., the stuck-at fault model. Nevertheless, it has been shown that tests for fault models at logic level are insufficient to detect dynamic faults and to adequately characterize the function of dynamic logic [Shi 2006]. For this reason, various publications deal with the detection of transistor level faults in synchronous, e.g., [Hawkins 1994], and in asynchronous circuits, e.g., [Roncken 1996, Shi 2006].

Delay faults A further commonly used model is the delay fault model. These faults affect only the timing of a path but not the logical function. The test for delay fault is essential for synchronous, but also for asynchronous self-timed circuits. The correct operation of these circuits depends on specific timing constraints. A delay fault may violate such a constraint which causes the circuit to malfunction. For this reason, delay faults are often the subject of investigations in testing asynchronous circuits [Khoche 1994, Hulgaard 1994, Petlin 1995b, Kishinevsky 1997, King 2004]. Typically, delay faults can be detected by applying a *test sequence* for a respective stuck-at fault [Abramovici 1990].

2.2.2 Standard Test Methods

A large variety of techniques have been proposed to test IC. These techniques differ in their properties. An important property is the *fault coverage*, which designates the ratio between detected faults and all possible faults with respect to a specific fault model. This implies that the set of faults of a model is countable which is typical for structural fault models. Further properties are the runtime and speed of the test, test time, controllability and observability, as well as intrusiveness in performance and area of the DUT.

The runtime defines when the test is executed. Basically, one distinguishes between tests that are executed when the DUT is already integrated into the target system (*on-line*), or prior to its integration (*off-line tests*). The speed of the test determines whether a test runs at the operation speed (*at-speed* testing) or not. At-speed testing allows tests under normal operating conditions of the design. The test time is the time required for executing the test.

Furthermore, a test technique can be intrusive or non-intrusive to a design. Non-intrusive means that no measures have to be integrated into the internal structure of a design to perform the test. In contrast to that, an intrusive approach introduces special properties, e.g., by adding extra hardware, to a design that enables the test technique to be applied. This basically results in area overhead, but also the performance can be affected. Typically, intrusive test approaches are applied to observe and control internal nodes of a circuit during test.

The procedure of adding intrusive test approaches to a design is called DfT. The majority of DfT-techniques were developed specifically for application in synchronous circuits. However, several of these approaches have been adopted for testing asynchronous designs.

The following sections summarize the most important techniques as well as the requirements and adjustments to test asynchronous circuits.

2.2.2.1 Functional Test

A classical non-intrusive technique to determine whether a circuit works correctly is to perform functional tests. The *test patterns* of such a test include the stimuli and the according responses which are derived from a functional simulation of an ideal model of the DUT. During a functional test, input stimuli are applied by the Automated Test Environment/Equipment (ATE) to the DUT which, in response to that, produces signatures at its outputs. These signatures are sampled and compared with the golden signatures from the ideal model. If the signatures are equal, then the design is assumed to be fault-free. More about functional tests and the issues with respect to asynchronous circuits can be found later in Chapter 3.

2.2.2.2 Self-Checking Capabilities of Asynchronous Circuits

Asynchronous handshake circuits have the beneficial property to stop their operation in the presence of a stuck-at-fault within the ACL [Hulgaard 1994]. To illustrate this, Figure 2.22 shows two control logic blocks of a Micropipeline with a stuck-at-0 fault at the acknowledgment signal ack_2. Initially, the outputs of all C-elements are logical-0. Now, a full handshake at the left interface shall be performed, i.e., req_1 is set to logical-1 which causes the output of the first C-element to be set to logical-1 as well. In response to req_2^+, the output of the second C-element is also set to logical-1. But due to the stuck-at-0 fault at ack_2, the first C-element will not reset its output when req_1 is set to logical-0. Hence, the circuit is in a deadlock. The same behavior can be shown for any stuck-at fault on the handshake signals. To detect such deadlocks, an external timer can be used. If the timer exceeds a defined threshold, the circuit is assumed to be deadlocked. Thus, a test for all these stuck-at faults can be performed by executing a sequence of handshakes that activates all control paths of the circuit.

As illustrated by this example, the asynchronous control logic is a critical point. A fault in this circuitry may corrupt the entire system. Even though it is basically easy to determine that a fault is present, it might be impossible to infer its location. Furthermore,

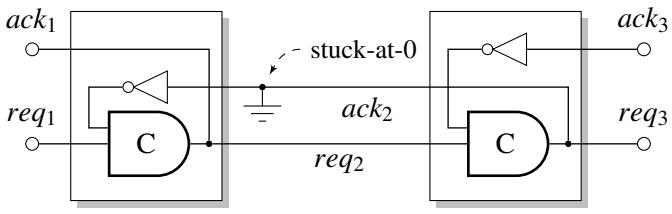
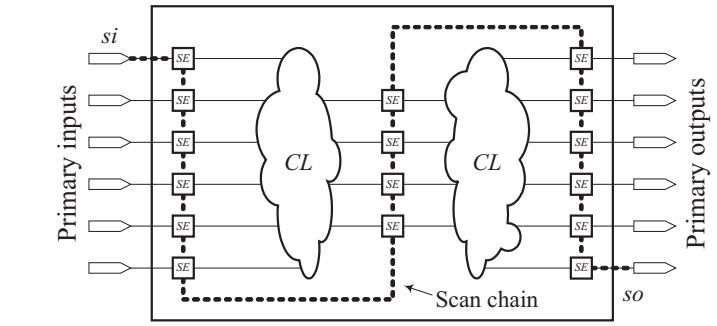


Figure 2.22: Two ACL units with a stuck-at-0 fault at ack_2

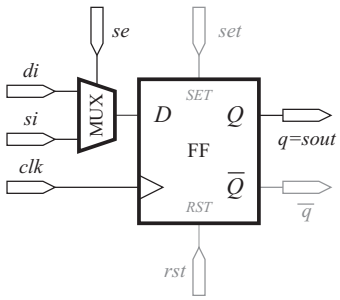
besides stuck-at faults on the handshake signals, other faults can corrupt the functionality of the handshake logic. For example, dynamic faults may cause glitches under some circumstances. Such faults are extremely hard to discover. For this reason, special handshake checkers have been provided in [Shang 2006, Zeidler 2010] that increase the testability of the handshake logic. These protocol checkers observe the handshake signals during normal operation in order to detect faults that violate the handshake protocol.

2.2.2.3 Scan-Test

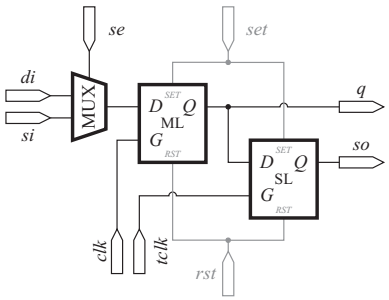
Scan test is a powerful DfT technology which can be used to detect various types of faults, such as stuck-at faults, delay faults, as well as more realistic faults, such as stuck-open and bridging faults. The general idea of this technique is to connect the sequential elements of a device to one or more shift registers, called *scan chains*. Figure 2.23a illustrates this. To achieve this, the sequential elements of a design are replaced by their scannable counterparts. These scan elements (SE) have one additional data input (scan-in/SI) and one control input (scan-enable/SE). This is shown in the Figures 2.23b and 2.23c, which show scannable versions of a flip-flop and a latch, respectively. The scan-



(a) Design with scan



(b) Scannable flip-flop



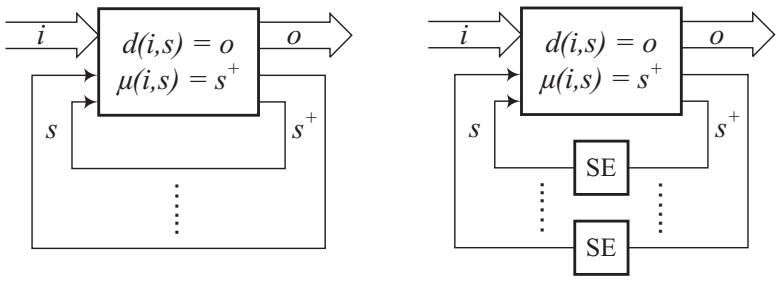
(c) Scannable latch

Figure 2.23: Scan technique

input is used to connect the sequential cells to scan chains, whereas the control input defines whether the cell operates in normal or in scan mode. In normal mode each scan element memorizes the value applied to their normal data input. Otherwise, the value of the previous cell in the chain is stored. In this way either all (*full-scan*) or a subset of all (*partial-scan*) sequential cells can be made scannable. A full-scan offers high fault coverage, but also results in huge hardware overhead. Partial-scan techniques sacrifice fault coverage and/or test time in order to save hardware costs. In order to perform one test iteration, stimuli are shifted into the chain. Then, the circuit is switched into normal operation mode for a defined number of iterations. Finally, the values stored in the sequential elements are scanned out and compared with the expected signatures.

There are several issues with the integration of the scan technique into asynchronous designs. One issue is that asynchronous circuits may have combinational feedback loops without sequential elements as shown in Figure 2.24a (cf. [Abramovici 1990]). This includes the feedback loops of data signals as well as the ones included in the control logic. These feedback loops have to be broken by inserting scan elements or by modification of present sequential cells, such as the *C*-element [Khoche 1995]. Figure 2.24b illustrates this. In normal mode these elements are transparent in order to maintain the original behavior of the circuit [te Beest 2002]. In scan mode, the test patterns are shifted in and out. In a further test mode, the scan elements drive the values of the feedback loops.

Another issue is related to the clock signal. The scan technique presupposes a common clock signal at least for all sequential elements of one chain. In a GALS design, this is manageable by setting constraints which force that each scan chain includes the sequential elements of only one synchronous block. In case of fully asynchronous designs, additional means are required to synchronize the sequential elements in scan mode. One way is the integration of a test clock signal. However, this results in significant hardware overhead for the required clock tree. Additionally, a synchronous clock tree imposes



(a) without scan (b) with scan

Figure 2.24: Asynchronous sequential circuit with and without scan elements

timing and power considerations that should have been avoided when applying the asynchronous design methodology. Other approaches propose the extension of the handshake control logic and registers to support scan [Petlin 1995a, Petlin 1995b, te Beest 2002, Schöber 2001]. In this case, the handshake logic is adapted such that it can also be used to shift the patterns in and out of the scan chains.

2.2.2.4 Built-In Self-Test

BIST techniques integrate test capabilities directly into the chip. This has the advantage that tests can be performed at the operation speed of the DUT. Furthermore, it allows the execution of tests when the DUT is already integrated within the target system. In general, a BIST consists of a Test Pattern Generator (TPG) and a Test Response Analyzer (TRA). The TPG generates stimuli for the unit-under-test (UUT), while the TRA receives the response and either generates a signature for further analysis or directly compares the response with an expected golden signature.

One can distinguish between structural BIST and functional BIST. Structural BIST uses structural information of the UUT and is often combined with scan-chains, i.e., the scan-chains are connected to the TPGs and the TRAs, respectively. Functional BIST techniques aim at testing the functionality of the UUT. With respect to this work, only functional BIST are considered in the continuation. To integrate such a functional BIST into an asynchronous circuit, the TPGs and the TRAs have to be equipped with handshake logic [Alves 1998]. Figure 2.25 shows the general structure of a BIST for handshake based asynchronous circuits.

However, a conventional functional BIST also has some drawbacks, e.g., the lack of diagnostic capabilities. This is due to the fact that the responses are typically compacted by the TRA. Depending on its realization the TRA either delivers pass/fail information or a signature generated from the responses of the UUT. The former option enables

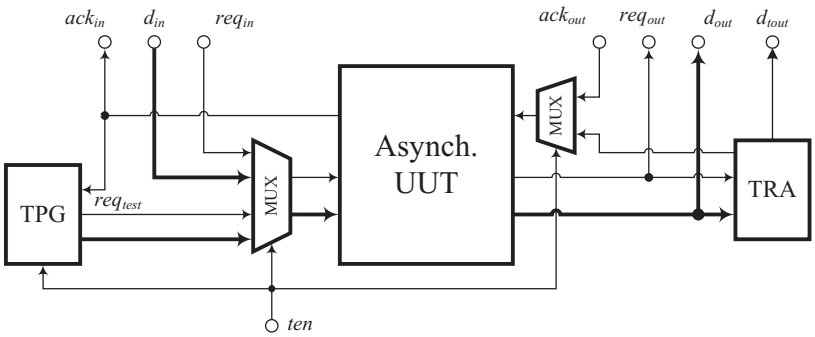


Figure 2.25: BIST for an asynchronous unit-under-test

a simple evaluation, but totally lacks of diagnostic information. For example, in case that the test fails, there is no information whether the fault is inside the TRA or the UUT. This is different for a BIST providing a signature which can be further analyzed. Nevertheless, even in this case, the diagnostic information from a BIST is typically very limited.

"You don't understand anything
unless you understand there are
at least three ways."

— Marvin Minsky

Chapter 3

The Challenge of Functional Tests of Asynchronous Designs

In this chapter, the issue of functional tests of asynchronous designs is discussed. After that, a couple of works are presented that address functional tests of asynchronous and GALS circuits. Furthermore, some ideas are introduced that are basically applicable.

3.1 Discussion of the Problem

Apart from the role in IC prototyping, functional tests regain more and more importance in complex system design. On the one hand, complex ICs, such as SoCs, often comprise IP cores bought from external providers. The purchasers usually do not have access to the internal structure of these cores. Even if that is the case, they are often not allowed to change the structure for testing purposes. In either case, the only possibility to validate the IP, is to perform functional tests. This may also apply to high-performance blocks. The integration of (intrusive) DfT techniques can critically affect the performance of a block. Non-intrusive functional tests are a possible solution for testing the design without sacrificing performance.

Although much effort has been spent into test methodologies of asynchronous designs, only a few approaches address directly functional tests of designs that utilize asynchronous design paradigms [Sparsø 2001, Kermani 2001]. With respect to this, there is one important open issue. In order to perform functional tests, equipment is required that provides the stimuli and receives the responses of the asynchronous DUT. Thus, to test asynchronous handshake circuits, the test equipment has to support handshake

protocols. Although commercial "big iron" hardware testers, e.g., ADVANTEST V93000 [Advantest 2013], are very powerful and provide much freedom with respect to configurability, they are conceptually designed for testing synchronous designs. Hence, they assume that the outputs of the DUT are aligned to a reference clock signal in a certain way. This imposes problems to the test of *event-driven* devices, such as asynchronous circuits.

At this point some clarification is required. With respect to test equipment, one can distinguish between *event-based* and *cycle-based* testers. *Event-based* test systems, such as the CERTIMAX, are basically able to generate edges and strobes at arbitrary points in time, where typically all pins of the system operate asynchronously, i.e., independently of the others. The input patterns for such testers are usually *event-based* as, e.g., the Value Change Dump (VCD) pattern [IEEE 1364-1995, IEEE 1364-2001]. As opposed to that, *cycle-based* testers, such as the V93000, require *cyclized* patterns, i.e., the drive edges and strobes are aligned to the tester clock.

However, neither *cycle-based* nor *event-based* testers are able to handle the *event-driven* behavior of asynchronous handshake circuits. They apply stimuli and expect responses at deterministic points in time. If a response of a DUT arrives not at the expected time determined by the pattern, the test is classified as being failed. The problem is that the responses delivered by an asynchronous DUT may not occur at predictable points in time due to *arbitrating processes* and the sensitivity of the timing behavior to *PVT variations*.

As a consequence of timing non-determinism, the arrival times of the simulated and actually measured output responses may vary. An example of such a scenario is illustrated in Figure 3.1. The figure shows the timing diagram of a data output that is aligned to a request signal driven by the DUT. The response of the DUT is expected to occur at the second tester clock cycle. Thus, the tester strobes the response at exactly this cycle. But, due to the uncertainty of the timing, the DUT actually delivers the output response one cycle later in the measurement. Thus, the tester samples the previous or an

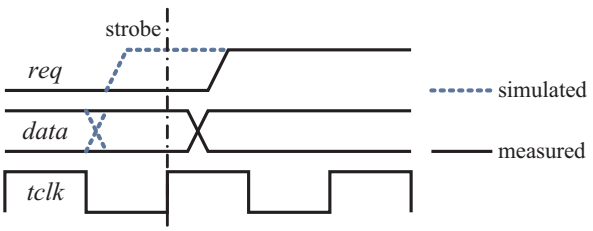


Figure 3.1: Timing variations of simulated and measured responses of an asynchronous device-under-test

intermediate data word. Consequently, the test fails.

Timing non-determinism would not be a problem for testing if hardware testers would be able to react to signal events generated by the DUT. Apparently, due to their orientation towards deterministic, synchronous designs, common test systems do not or only in a limited manner support such reaction capabilities. Thus, a real asynchronous, handshake-based communication between the tester and the DUT is impossible.

3.2 Alternative Solutions

In order to cope with timing non-determinism of an asynchronous DUT during test, several approaches are possible. This section gives an overview about the few works and ideas related to functional testing and handling of non-determinism of asynchronous circuits.

3.2.1 Assuming Worst-Case Behavior

The simplest approach to treat timing non-determinism is to assume the worst-case behavior of the DUT. Thus, for every single response, no matter whether it is a handshake signal or a data output, one has to adjust the corresponding action (applying new stimuli or strobing the response) to the latest expected time. If the response does not match the expected value, the DUT is supposed to be faulty. Such an approach is described by B.G. Kermani *et al.* in [Kermani 2001]. The patented method proposes that the DUT delivers each of its output responses for a long time window. Thereby, the tester ignores all outputs until the point in time at which the result is expected at the latest. Finally, the tester samples the result.

Although the general approach seems to be simple, its practical realization imposes some considerable overhead. For example, the model of the DUT is typically ideal and does not reflect the worst-case behavior. In this case, the model has to be adapted in order to generate patterns for tests under worst-case conditions. Furthermore, it is necessary to mask responses in the generated patterns that occur earlier than expected. This typically requires an additional post processing of the generated pattern.

A further problem arises from the fact that the DUT never operates at its maximum speed if the approach is applied to read all output responses. Obviously, it is not possible to test the best-case performance of a design under worst-case conditions. Therefore, this technique is often only applied to read the result of a BIST rather than complex output sequences.

3.2.2 Utilization of Scan

A further solution could be the utilization of scan chains. This is especially interesting if the design already has scan chains for other test purposes. Then, it is obviously possible to utilize the chains also for functional tests. If the design does not have scan-chains, e.g., due to possibly imposed performance losses, the utilization of the boundary scan technique could be an option. This technique adds scan cells only to the inputs and outputs of the considered UUT. Using such a scan approach, functional patterns are shifted into the scan chain(s) as usual. Then, the UUT is switched into normal operation mode and finally, the resulting responses are scanned out.

An example of a scan-based approach for performing functional tests was proposed for GALS systems in [Gürkaynak 2002]. The purpose of the scheme is to test the individual SBs and their interconnections. Therefore, scan chains are integrated into the input and output ports of the wrappers as shown in Figure 3.2. Moreover, each wrapper is extended by a so-called Test Extension Element (TEE) that controls the wrapper and its scan chains in test mode. These TEEs are controlled by a Centralized Test Controller (CTC) that coordinates the test activities of all blocks. Therefore, the CTC sends instructions and patterns to the TEEs which, in turn, send the test responses of the SBs to the CTC. Accordingly, the CTC also provides an interface to the tester that is used to up- and download the patterns. To prevent synchronization issues, all TEEs and the CTC are

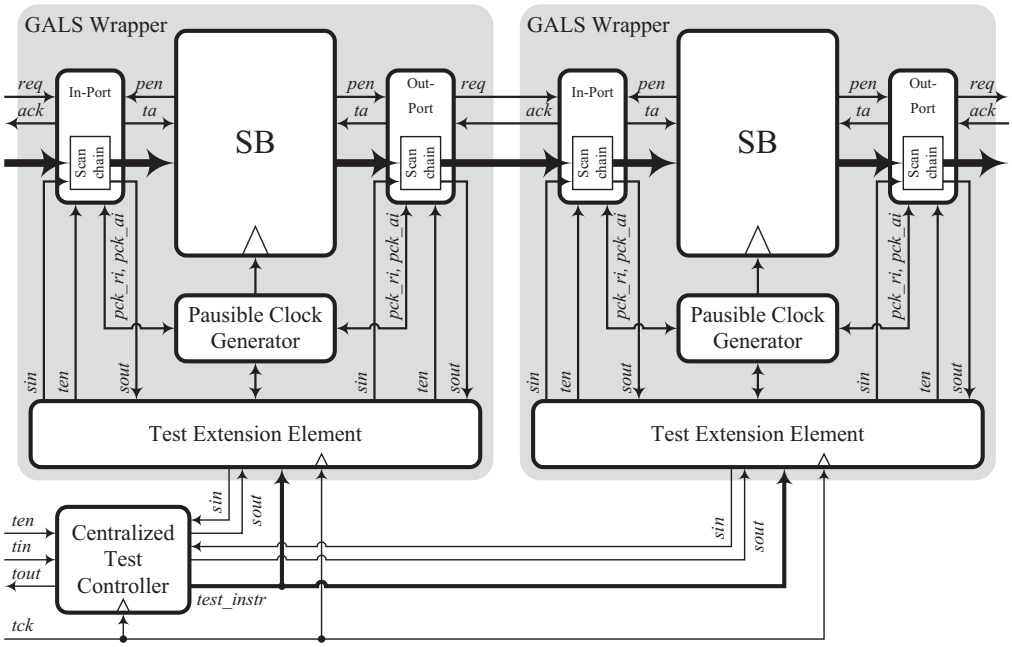


Figure 3.2: GALS system architecture with scan [Gürkaynak 2002]

triggered by a common global clock signal provided by the tester.

In principle, this concept can also be adapted to fully asynchronous designs. To keep the overhead low, one can identify relevant sequential elements of the design which should be equipped with scan capabilities, e.g., the registers at the inputs and outputs of a block. However, apart from the issues mentioned in Section 2.2.2.3, the entire approach has one major drawback. The shift process requires considerable time which inhibits tests with high data rates. Moreover, if the tester is used to compare the expected and the captured response signatures, then one has to adjust the patterns to the worst-case timing as well.

In summary, it is generally possible to utilize scan for functional tests. But there are several issues with respect to test time, required overhead, and performance evaluations of the DUT.

3.2.3 Utilization of Built-In Self-Test

Functional BIST is a common technique to evaluate the functionality of asynchronous circuits [Petlin 1997, Roncken 2000, Krstić 2005a]. Its major advantage is that it enables at-speed testing under real operation conditions of the UUT. Also it is not intrusive to the internal structure of the UUT, since only multiplexing logic is required to integrate the BIST components. Thus, BIST is basically a suitable technique to perform functional tests.

An example of a BIST for evaluating the functional correctness was presented by M. Krstić and E. Grass in [Krstić 2005a]. This technique was integrated into a GALS baseband processor. The transmitter and receiver part of the processor are divided into several SBs interconnected with asynchronous channels. The BIST was used to test both, the asynchronous channels and the SBs. To this end, the TPGs and TRAs are organized such that they can be combined in several ways to perform a variety of different tests of individual subsystems. Similar to the scan approach, a centralized test controller coordinates the TPGs and TRAs and provides an interface for the tester. Local and global tests can be performed by activating particular TPGs and TRAs. Furthermore, the transmitter and the receiver are connected together, such that stimuli at the input of the transmitter can propagate through the entire GALS architecture to the outputs of the receiver. By this, global tests can be performed that check the functionality of the entire design.

The major drawback of a BIST approach is its limited diagnostic information. A functional test using a commercial tester delivers cycle-accurate pass/fail information of each output bit provided by the DUT. This information is needed especially in system

prototyping to debug the system. As afore mentioned, a BIST only provides pass/fail information or a signature generated from the responses. Hence, it is often not possible to determine in which state of a test a malfunction occurred. Furthermore, as for the other mentioned approaches, it is also necessary to assume the worst-case timing when reading the result of the BIST. But in comparison to the other approaches, this has to be done only once per test rather than for each data exchange.

A further drawback is related to the stimuli generated by the TPGs. Typically, TPGs are realized by pseudo-random number generators, such as LFSRs. These components are simple to implement and require only little hardware resources. However, complex sequences of non-random patterns are hard to implement. Therefore, a memory block, e.g., a Read Only Memory (ROM), storing the stimuli can be integrated and combined with LFSRs in order to realize special pattern sequences. But the integration of a memory only for test purposes might be too expensive with respect to hardware overhead.

3.2.4 Utilization of Memories and FIFOs

A cognate approach is the utilization of external memories that store and buffer the patterns during test. This prevents the integration of memories into the DUT. Only an interface has to be provided which is already available in many designs. Such a scheme was described in [Sparsø 2001]. They used an external ROM to store a program testing the functionality of a fully asynchronous microcontroller of a smartcard. This program computed a signature which was written to another memory and analyzed after the test. However, this approach imposes the DUT to have memory controllers and respective interfaces which might not be desired in all cases.

A similar approach is based on the utilization of FIFOs in order to interface the DUT with the tester. By this, the timing uncertainties are compensated by the FIFOs. To realize this scheme, an FPGA can be used that is mounted on the load board of the test equipment. This FPGA implements the interface logic between the DUT and the FIFOs as well as the interface between the FIFOs and the tester. Thus, the test equipment is adapted to the interfaces of the DUT rather than the other way round. Accordingly, no further means have to be integrated into the DUT to perform the functional test. Figure 3.3 shows such a scheme. Depending on the handshake protocol, the types of the FIFOs have to be carefully selected. In the shown example, the channels operate according to a push-protocol. Therefore, a so-called fall-through FIFO is used for the input channel. This ensures that the data is already present at the output interface of the FIFO when the request is issued. In comparison to that, standard FIFOs are sufficient for output channels. During test execution, the input buffers are filled and the output buffers are

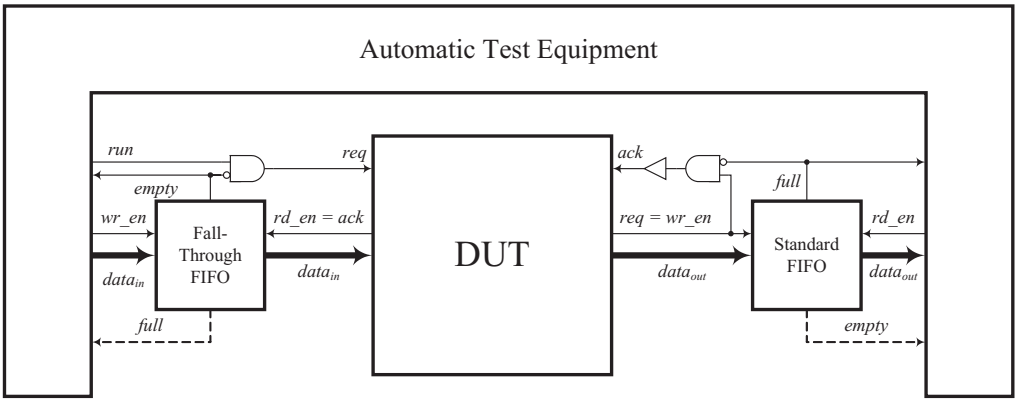


Figure 3.3: Integration of FIFOs to compensate timing non-determinism

emptied in bursts by the tester. Thus, under consideration of the average timing of the DUT, it should be possible to periodically fill and empty the FIFO buffers in bursts comprising half of the words fitting into the buffers.

The benefit of this scheme is its simple implementation. However, the FIFOs and the interface logic need to be adapted for every DUT. Furthermore, this test infrastructure does not allow a precise control of the data exchange. The data applied via the FIFOs are directly fed to DUT and vice versa.

3.2.5 Eliminating Non-deterministic Behavior

Another solution that directly addresses the problem of non-determinism of GALS systems with respect to test was proposed by M.W. Heath and I.G. Harris in [Heath 2003, Heath 2005]. Their approach, called *synchro-tokens*, introduces additional logic to the asynchronous wrapper surrounding the synchronous blocks. This logic implements nodes of a token ring integrated for each communication channel between two SBs. These nodes are equipped with two counters. One counter stores the number of local clock cycles in which a token is expected to arrive. The second counter stores the number of cycles in which the access to the input or output of the channel is granted. This allows multiple data words to be transmitted for each token exchange. If one of the two counters reaches zero, then the token ring node stops the local clock generation. Consequently, the data is exchanged at deterministic cycles of the local clocks which results in an overall deterministic behavior of the system.

Figure 3.4 shows the entire GALS architecture of this approach. The SBs are connected with each other via self-timed FIFOs. Furthermore, additional wires carry the token exchanged between the communicating SBs. One of these wires comprises an inverter to realize transition signalling between token ring nodes. For test purposes, an

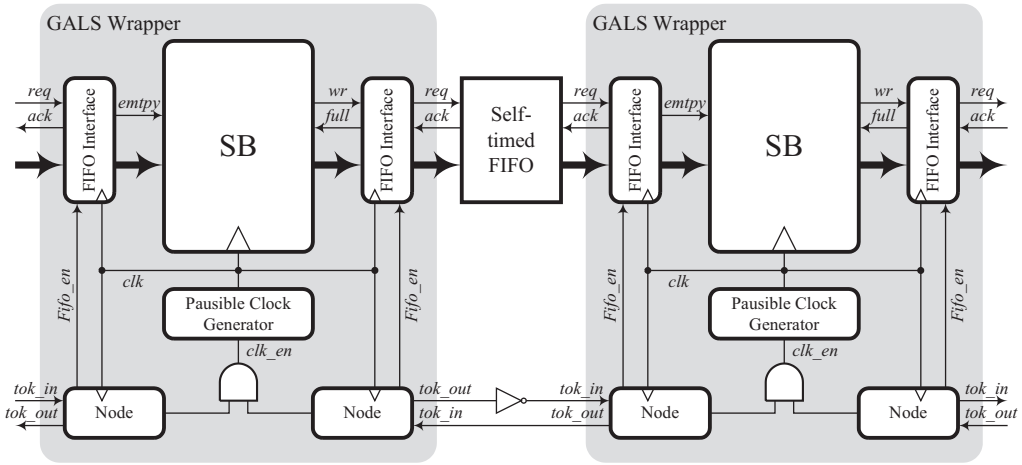


Figure 3.4: *Synchro-Tokens* GALS architecture [Heath 2004]

additional SB (not shown in the figure) is introduced that is connected to all SBs via separated token rings. By this, the block is able to coordinate the activity phases of the individual SBs. Furthermore, this block has a test access port (TAP) to setup the node counters and to provide access to internal nodes (e.g., via scan) during test.

However, this architecture is only applicable to GALS designs, since the number of asynchronous communication channels is strongly limited. An adaption of this scheme to fully asynchronous designs is unfeasible, not only by the fact that a pausible clock is required, but also due to the complexity of the additional hardware required for every channel. Additionally, the DUT has to be switched into a test mode to suppress the non-deterministic behavior. This hides the real behavior of the circuit and makes performance evaluations of the DUT difficult.

"If you have built castles in the air, your work need not be lost; that is where they should be. Now put the foundations under them."

— Henry David Thoreau

Chapter 4

Concept for Functional Tests of Asynchronous Circuits

This chapter introduces a concept for realizing functional tests of handshake circuits. The concept introduces an asynchronous abstraction layer that abstracts from single signal transitions of conventional tests. By this, test stimuli are applied to the DUT and its output responses are received via performing data transfers based on asynchronous communication channels. The abstraction layer is represented by a test processor that provides a generic interface to establish asynchronous communication channels. Besides the generic test processor architecture, a procedure is proposed that describes the generation of programs for the processor. These programs are derived from behavioral simulations of the DUT and implement the desired functional tests.

4.1 Model of the Device-Under-Test

Prior to the definition of the concept, assumptions have to be made that help to abstract from the actual realization of asynchronous designs. This is the prerequisite for providing a generic approach that is applicable to a large variety of asynchronous designs.

One basic assumption is related to the behavior of the considered asynchronous device. Functionally non-deterministic circuits are not addressed in this work. To test such circuits, a magic predictor would be required that foretells non-deterministic outputs. Such a predictor cannot be implemented by any deterministic finite state machine. Therefore, this work covers only the test of asynchronous circuits whose output se-

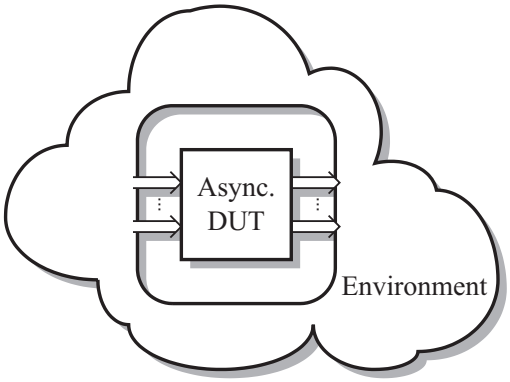


Figure 4.1: Abstract model of the DUT

quences are well-defined. Only the timing of the responses is allowed to vary. For this kind of circuits, the interaction with the environment is governed by events of communication protocols.

According to this, the patterns have to be exchanged via unidirectional asynchronous handshake channels that are structured as described in Section 2.1.4. Thus, stimuli are applied via the input channels of the DUT and responses are received via the output channels as shown in Figure 4.1. However, there is one exception to this assumption, i.e., special control signals. Sequential circuits typically have at least one of such signals: the reset signal. This also applies to asynchronous circuits. Therefore, the TP has to provide an interface such that those control signals can be realized.

4.2 Test Processor Concept

As discussed in Section 3.1 conventional hardware testers are not able to handle the non-deterministic timing behavior of an asynchronous DUT. Consequently, different test equipment is required that applies stimuli to the targeted non-deterministic DUT and receives its output responses in an elastic manner. Since it is assumed that data is exchanged via asynchronous channels, this equipment needs to provide handshake channel interfaces that are

- able to react to protocol events from the DUT and
- configurable to support various types of handshake protocols.

Additionally, the equipment should be programmable for the sake of flexibility. This leads to the demand of a special test processor that supports handshake signalling in

a programmable manner. As a direct consequence, a common test system is not required for performing the functional tests. Instead, a standard PC can be used to supply the patterns, and to receive and process the final test results delivered by the processor. However, the provided TP is designed only for functional tests. A common tester is, therefore, beneficial to complement the test, e.g., by performing parametric measurements, such as continuity-, leakage-, and operating current tests. In this case, the TP should be connected with the tester. Therefore, in the continuation of the work any kind of equipment connected to the TP for controlling and monitoring purposes is here and after referred to as external test equipment (ETE).

The utilization of a TP component for extending the available tester hardware is not a new concept. Several publications proposed TP solutions to implement programmable LFSRs [Ali 1996, Ali 2002, Kabir 2009]. In [Darus 1997] a low-cost TP was introduced to realize multiple polynomial LFSRs with programmable seeds that also supports scan chain testing. In [Altaf-Ul-Amin 1999] the design of a prototype TP is proposed that can be used for functional tests of digital ICs. Therefore, the processor is able to generate pseudo-random followed by deterministic test vectors. In the other direction the processor receives output responses of the DUT and provides means to compress them. In [Galke 2002] C. Galke *et al.* have presented a low-cost TP used to enable self-tests of system-on-chips (SoCs). The provided test processor has a RISC architecture and is equipped with special registers to realize configurable LFSRs and Multi-Input Signature-Registers (MISRs) for compaction, decompaction and filtering of patterns. In [Frost 2007] this concept was further extended by enabling the adaption of the TP to the demands of the SoC test. Also, mechanisms are presented to test logic blocks and bus structures using the TP.

However, all the afore mentioned TP solutions are intended for testing synchronous designs. Thus, the novelty of the concept provided in this work is the support of asynchronous handshake signalling. In particular, the novel test processor is equipped with configurable handshake interfaces in order to support a large variety of asynchronous circuits. Finally, as shown in Figure 4.2, the generic processor architecture is divided into three main components: a memory, the test processor core and a port component that provides the handshake interfaces. The TP core executes the program describing the configuration of the processor and the interactions with the DUT. The memory stores the program, the stimuli for the DUT as well as the responses and/or fault signatures captured during test execution. Furthermore, the TP provides a synchronous interface to the ETE. Using this interface, the ETE can control and monitor the TP, upload the test program and the related patterns, and receive the test results. Furthermore, the ETE has to be informed by the TP after test execution that the results can be downloaded. Thus,

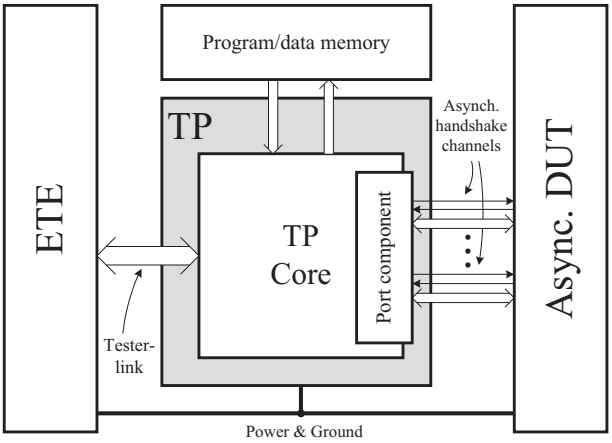


Figure 4.2: Concept of the test processor for asynchronous devices

a synchronization mechanism is required.

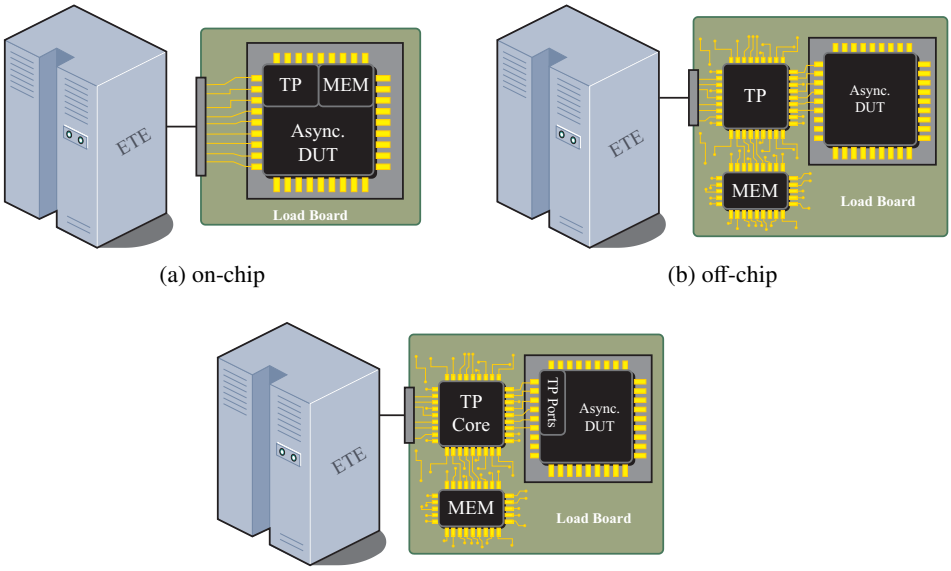
4.2.1 Implementation Schemes

In consideration of the general processor architecture, several implementation schemes can be identified. These schemes describe the locations where the individual components shall be physically placed. In general, one can implement the components of the TP inside or outside of the chip to be tested. Inside means that the components are placed within the DUT or added as test structure on the wafer. The latter aspect could be applied to test several devices within the same die using only one test processor. Alternatively, the TP can be fully integrated into the test equipment.

Apparently, there are three different implementation schemes of the concept:

- The simplest solution from the tester point of view is the complete integration of the TP into the DUT as given in Figure 4.3a. Such an approach was, e.g., applied in [Galke 2002, Frost 2007]. The major advantage of this scheme is that internal signals of the design can be accessed which leads to improved debugging capabilities. However, the overhead of integrating the processor into the DUT might be too large and, therefore, unsuitable for some designs. Nevertheless, the insertion of the TP into the design can be feasible under consideration of the possibility to perform self-tests when the design is already integrated into the target system. Furthermore, the processor can also be used for other tasks in the design, e.g., power management.

- As shown in Figure 4.3b, the second possibility is to fully place the processor outside of the chip. One can think about a stand-alone test system consisting of



(a) on-chip

(b) off-chip

(c) mixed

Figure 4.3: Implementation schemes

the TP and an interface board for the DUT. The integration of the TP into the load board of the tester equipment is also possible. This is a common way to complement the ATE with further capabilities, e.g., for realizing high-speed tests as it was done in [Keezer 2005, Majid 2010]. The benefit of this approach is that it does not impose any additional hardware overhead to the DUT. However, there is no possibility for the processor to access internal signals of the DUT.

➢ The last scheme combines both preceding approaches to a mixed implementation as presented in Figure 4.3c. Thereby, some components are placed outside of the chip while the DUT has to provide special capabilities. An example for such an architecture was proposed in [Majid 2005]. In this approach, the DUT has to provide BIST facilities, while the additional external test equipment receives the results of the BIST. In the context of the proposed test processor, one possible solution is to place the port component inside of the DUT. This enables the access to internal signals while keeping the overhead for this at a minimum. Nevertheless, a high-bandwidth interface between the port component and the TP core is required to feed the port components with data.

In practice, the selection of the implementation scheme depends on the device and the demands onto the test. However, the memory is the biggest issue for the on-chip implementation. If the DUT has a memory, then it should be possible to connect this

memory with the TP. Otherwise, the memory could be placed outside of the chip. For example, one can integrate the memory into the load board of the tester equipment. A more detailed list of the issues that affect the decision about the implementation scheme is given later in Section 4.3.1.

4.2.2 Definition of Interfaces

The most important aspect of the test processor is related to its interfaces. On the one hand, the processor has to provide a generic interface for the DUT. This interface has to comprise configurable ports in order to realize arbitrary asynchronous channels. On the other hand, the processor has to be connected with the ETE via a fixed and well-defined interface.

4.2.2.1 Interface to the DUT

In order to implement the abstract model of asynchronous channels described in Section 4.1, the TP needs the following types of ports:

- a set $\Omega = \{\omega_0, \ldots, \omega_{l-1}\}$ of data ports (DPs) for realizing the set of data signals D,
- a set $\Phi = \{\phi_0, \ldots, \phi_{m-1}\}$ of handshake ports (HPs) for realizing the handshake control signals H.

To transmit data in both directions from the TP to the DUT and vice versa, the data ports could be separated into inputs and outputs. However, to retain maximum flexibility, these ports shall support both input and output data transfers. Therefore, each port needs to comprise a set of inputs and an equal number of outputs. If the off-chip scheme is used for the implementation of the TP, one input and one output of a DP can be combined to a bidirectional pin of the TP chip. In this case, each port has to provide one additional signal defining the direction of the port. This signal can then be used to control the according I/O pad tristate buffers connected to the data inputs and outputs of the DP.

To implement the handshake ports, exactly one input `hi` and one output `ho` are required. These signals need to be connected with the respective handshake signals $H \subseteq \{req, ack\}$ of the DUT according to the protocol type $p \in S_C$ and the direction of the channel. For example, to interface a bundled-data input push channel of the DUT, the output `ho` needs to be connected with the request signal, and the input `hi` has to be connected with the acknowledgment signal of the channel interface of the DUT. In contrast to that, the input `hi` has to be connected with the request signal, and `ho` has to be connected with the acknowledgment signal, in order to establish an output push

channel of the DUT. Consequently, one can define a bijective mapping f_{HP} from the handshake signals of a channel C to the handshake control signals `hi` and `ho` as given in the following equations:

$$f_{HP}(req) = \begin{cases} \texttt{hi} & C \text{ is an } \textit{output push} \text{ or an } \textit{input pull} \text{ channel of the DUT} \\ \texttt{ho} & C \text{ is an } \textit{output pull} \text{ or an } \textit{input push} \text{ channel of the DUT} \end{cases} \tag{4.1}$$

$$f_{HP}(ack) = \begin{cases} \texttt{ho} & C \text{ is an } \textit{output push} \text{ or an } \textit{input pull} \text{ channel of the DUT} \\ \texttt{hi} & C \text{ is an } \textit{output pull} \text{ or an } \textit{input push} \text{ channel of the DUT} \end{cases} \tag{4.2}$$

Correspondingly, let f_{HP}^{-1} be the inverse function of f_{HP}.

In order to fully cover the components of the abstract channel model, the handshake ports also have to store the protocol information p and the initial values defined by the function v_0 (see Section 2.1.4). According to the defined mapping, the initial values $v_0(req)$ and $v_0(ack)$ of the handshake signals are associated with the signals of the ports. Thus, the initial value of `hi` is $v_0(f_{HP}^{-1}(\texttt{hi}))$ and the initial value of *ho* is $v_0(f_{HP}^{-1}(\texttt{ho}))$. Finally, an abstract asynchronous channel $C = (H, D, p, v_0)$ can be mapped to resources of the TP by combining one HP $\omega_i \in \Omega$ with a set of DPs $\Pi_i \subseteq \Phi$. Therefore, let k be the number of pins (inputs or outputs) provided by each DP. Then, the set Π has to be selected such that $|D| \leq k \cdot |\Pi|$.

Besides the channel interfaces, an asynchronous device may also have a few special in- and/or outputs that are not aligned to handshake events. As mentioned before, a prominent example is the reset signal, but also other control and/or configuration signals might be required, e.g., special statically programmable configuration signals. To fully embed the DUT into the TP environment, the TP has to provide interfaces for such control signal. These ports could be implemented by the data ports of the TP, but this actually depends on the implementation of the processor.

4.2.2.2 Interface to the ETE

The interface to the external test equipment has to fulfill two tasks. On the one hand, the test data has to be exchanged. Thus, the program and the patterns have to be uploaded to the TP. In the opposite direction the test results including the responses and/or fault signatures have to be downloaded for further evaluation. Therefore, the following ports are required:

- The input `ext_mem_req` indicates a read or write access request to the memory of the test processor from the external equipment.

- The input `ext_addr` is used for the specification of the memory address where data shall be stored or read from.
- The input `ext_din` is the port for uploading test program data from the ETE to the TP.
- Finally, the output `ext_dout` is used to download test results and other information from the TP to the ETE.

One the other hand, the TP has to be controlled and monitored by the ETE. With respect to this, the TP has to be initialized and it should also be possible to restart the test program. Furthermore, the TP and the ETE have to be synchronized in specific situations, e.g., prior to the download of the test results. A handshake mechanism is a proper scheme to realize this synchronization. To this end, the TP has to provide a completion indication signal and an acknowledgment input signal. The indication signal is the only one that the ETE has to observe. Therefore, the ETE can poll this signal continuously. If the ETE does not provide this, one can again assume the worst-case timing.

In summary, the required ports are:

- The input `rst` is used to reset the TP to its initial state.
- The input `rst_pc` is used to reset the program counter only. Other previously defined configurations are retained.
- The output `halt` indicates whether the TP is in halt state. This signal is part of the synchronization mechanism between the TP and ETE.
- The input `enable` is used to force the TP to continue with the program execution once it is in halt state. Thus, it is the second signal required for the synchronization.
- The output `fault` is a global fault indication signal. It indicates whether a fault has occurred during test. This output is optional.

4.2.3 Role of the Processor Core

An essential part of the concept is the processor core. This core is basically a microcontroller capable of executing programs that describe the configuration of the ports and coordinate the flow of the test. The latter aspect especially includes the control and observation of the dataflow. The observation includes especially the detection of deadlocks

during test which could, e.g., be caused by a defect. To detect such deadlocks, a timeout mechanism can applied that comprises a simple counter. This counter is decremented every cycle of an external clock source as long as a data transfer is incomplete. If the operation is finished, then the counter is reset. If this counter reaches zero while the transfer was not completed, then the processor core shall indicate a deadlock. For such cases, mechanisms to handle the unexpected behavior have to be provided. For example, a special program routine could be called that collects further information about the cause of the deadlock and that delivers this data to the ETE.

To execute programs, the processor core fetches and executes instructions from the memory and loads and stores patterns during test. Obviously, the core has to interface the memory storing the program and the respective test data. As a consequence of this, the core is also responsible for the coordination of the data exchange between the port component and the memory. It reads the stimuli from the memory and distributes them to the according data ports. In the other direction, the core forwards the sampled responses/fault signatures from the data ports and writes them to the memory.

The realization of this core depends on the implementation scheme of the entire processor. For example, consider the case that the DUT is a complex SoC comprising a microprocessor and some asynchronous components. In this case, it might be beneficial to extend the existing microprocessor to support the desired test functionalities. On the other hand, if the off-chip approach shall be applied, then an application specific processor can be used.

4.3 Workflow

Besides the test processor infrastructure itself, a procedure has to be defined that describes how the processor is integrated into a test flow in order to efficiently benefit from that infrastructure. On the one hand, this flow includes the embedding of the DUT into the test processor environment. On the other hand, test programs have to be generated that are executed by the TP and that describe the test of the DUT. The following sections describe the general approaches to tackle these issues.

4.3.1 Embedding the DUT into the Test Processor Infrastructure

The very first activity of the workflow is the analysis of the existing test infrastructure, the DUT as well as the demands of the test. Several questions have to be answered in order to decide about the way of implementation of the test processor. For this, one has to clarify the following aspects:

- ➢ *Is there more than one asynchronous design to test?* Testing various different designs favors the realization of the off-chip approach while the test of a single design most likely justifies the integration of the TP into the DUT.
- ➢ *Shall self-tests be performed?* Self-tests are obviously only possible with an on-chip implementation of the TP.
- ➢ *Shall internal signals of the design be accessed?* If internal channels need to be accessed either the on-chip or the combined approach has to be selected.
- ➢ *What are the maximum allowed costs?* Obviously, the integration of the TP into the DUT increases the costs of silicon area which have to be traded against the costs of an off-chip implementation.
- ➢ *Does the DUT already contain a processor?* An existing processor can be adapted such that it provides facilities to test other components. This decreases the costs of an on-chip implementation of the TP.

These questions need to be contemplate during the implementation phase of the DUT, since they affect the implementation scheme and, therefore, possibly the design of the DUT. Based on the answers to these questions, the implementation scheme has to be selected. In case of the on-chip or the mixed approach, it has to be further clarified how many asynchronous channels are needed, and which handshake protocol types have to be supported by the TP. As a result, the on-chip test processor can be adapted to exactly match the demands of the test. If the off-chip approach is selected, then the test processor should be implemented in a generic way in order to support a large variety of asynchronous designs. Thereafter, the TP and/or the DUT have to be adapted to the requirements identified. Finally, one will end up with the final TP and DUT designs.

The next essential activity is the interconnection of the test processor with the DUT. Thus, one has to define a configuration comprising a mapping of the ports of the TP to the ports of the DUT. This configuration is here and after referred to as *pin configuration*. Obviously, this mapping is a prerequisite for the physical interconnection of the TP and the DUT. Furthermore, it is required for the generation of the test program. To create this configuration, the handshake and data ports have to be grouped together to form asynchronous channels in the way described in Section 2.1.4. In order to automate this mapping, one can consider one channel of the DUT after the other and combine one handshake port with a set of data ports. After defining the *pin configuration*, the TP and the DUT can be physically interconnected. This activity strictly depends on the implementation scheme. For example, if the TP is implemented off-chip using an

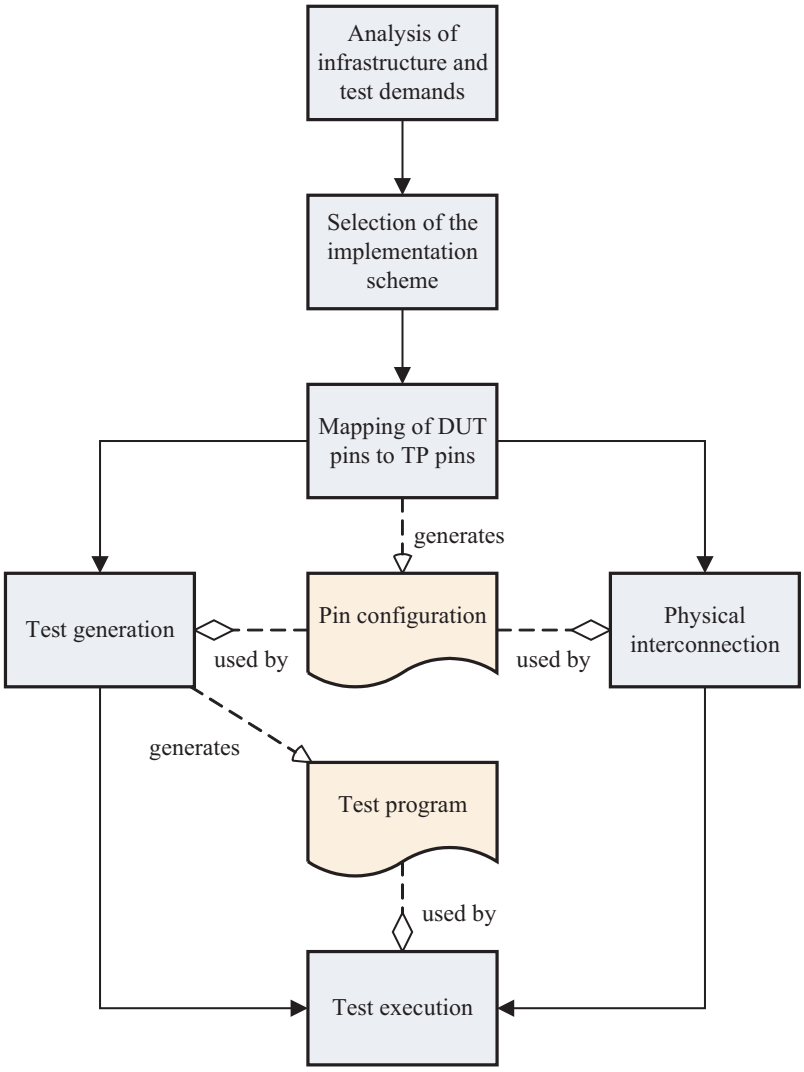


Figure 4.4: Integration of the test processor into the test flow

FPGA, which is mounted onto the load board, the pin configuration can be used to create a constraint file including pinning information for the FPGA. In case of the on-chip implementation, the pins of the DUT and the TP can be connected directly. Finally, the last task concerns the generation of the programs implementing the functional tests. This generation process is a comprehensive task which is discussed in the next section. In conclusion, Figure 4.4 outlines the steps to embed the DUT into the TP environment.

4.3.2 Generation of Tests

Obviously, the generation of a functional test for the DUT, which is embedded within the TP environment, is a matter of creating a program for the test processor. As mentioned,

this program consists of the sequence of actions which describes the flow for testing the DUT, and the corresponding data, i.e., the test patterns and other data required for the program execution. Thereby, the actions comprise the configuration of the TP, the test, and the upload of the test results to the ETE. During the configuration phase of the program, the ports of the TP have to be configured, for example, with respect to the desired handshake protocol. After that phase, the TP can run the test and execute the sequence of data transfers. Finally, the test results are uploaded. These results comprise the fault signatures and/or the captured responses from the DUT, and further information about the status of the test, e.g., if a deadlock was detected.

A straightforward way to create such a program is to write it manually. For example, one can think about writing an assembler program for the test processor core that is afterwards translated to binary machine code. However, the manual creation of such a program is very complicated and error-prone and, therefore, undesirable. A more sophisticated approach could be the utilization of a high-level language, such as C. Obviously, this requires a compiler and an additional library, which includes functions to access the special capabilities of the test processor. Nevertheless, writing a program manually is always prone to mistakes. For this reason, an approach should be sought, which enables an at least semi-automatic generation of programs for the TP. Such an approach is proposed in the following sections.

4.3.2.1 Test Program Generation Flow

The usual way of generating functional test patterns for a design is to perform logical simulations of an ideal model. To this end, a test bench has to be created that emulates the environment of the DUT. This test bench applies the input stimuli and typically evaluates the output responses. In order to generate the desired test patterns, the logic simulator is set up to record the interactions of the DUT with its virtual environment. Thus, the input stimuli and output responses are written to respective test pattern files during the simulation. Obviously, the major point of interest of this approach is that the simulator does the entire work. It computes the output sequences and creates the pattern files.

This highly automated approach lead to the idea of generating the program for the TP from the simulation of the asynchronous DUT. However, an essential issue is that the simulators typically generate patterns which exhibit a static timing. To dissolve this static timing, one has to extract the dataflow which is based on channel transfers rather than signal changes. For this, one can think about a post processing step after the generation of a standard pattern file. To accomplish this, one would need the *pin config-*

uration and the information about the association of the pins with the channels. Using this configuration in combination with the information about the handshake protocols used, it might be possible to extract the desired channel transfers from a standard pattern. Nevertheless, it is probably very difficult to extract relations between the transfers. For example, it is difficult to find out whether two consecutively occurring transfers have to be executed sequentially or in parallel.

A more sophisticated approach is to let the simulator directly create a special pattern file that describes the dataflow between the DUT and its environment. The base for this approach is the creation of a model for asynchronous channels and corresponding functions to perform data transfers. These functions, which are here and after referred to as *transfer procedures*, are used to apply stimuli and to read the responses of the DUT. The model and the functions have to be defined in a language that is supported by common logic simulators, for example VHDL, Verilog, SystemVerilog or SystemC. In order to increase the (re)usability, this model should be described in a separate module (e.g., a package) such that it can be used for different test projects. This module is here and after referred to as *channel simulation package*. To utilize the provided functionality, the test bench for the DUT has to import this package. Afterwards, the test bench has to declare instances of the provided *channel model* and perform the transfers using the corresponding *transfer procedures*.

Such an approach has been described in [Sparsø 2001], for example. Therein, a VHDL-package is described that provides models for asynchronous bundled-data push channels. As opposed to this implementation, the model proposed here shall be more general to cover more protocols. Therefore, the channel model additionally includes the protocol information. Corresponding to this information, the *transfer procedures* execute the appropriate handshake-based data transfers. As these procedures take care of the value assignments and the reactions to transitions of the involved signals, the details of the protocol implementation are completely hidden. The most important crux of the approach, however, is that these procedures write a description of the executed transfer into a file, i.e, the desired pattern file which is here and after referred to as *transfer protocol*.

To illustrate this better, consider the following. To perform the simulation, a test bench has to be created that declares the channels used for the interaction. Furthermore, the test bench opens the *transfer protocol* prior to any data transfer with the DUT. Now, to apply stimuli to the DUT, a transfer procedure `send()` applies the data to the DUT and performs the handshake corresponding to the protocol of the channel. Simultaneously, the method writes the information about the transfer into the *transfer protocol*. This description includes the involved channel, the direction and the transmitted data. In

response to that, the simulator computes the output responses of the ideal model of the DUT. To receive the response from an output channel, a `receive()` procedure has to be used. Accordingly, this *transfer procedure* also writes the description of the transfer into the *transfer protocol*. In this case, the description comprises the golden output signature computed by the simulator. Thus, after executing the complete logic simulation of the test bench the *transfer protocol* contains all channel transfers between the DUT and its environment.

Based on this *transfer protocol*, the program for the TP can be generated via a mapping from the transactions to respective instructions of the TP. This mapping can be implemented by a software tool that takes as input the *transfer protocol* and generates the program, for example in assembler code. Afterwards, the generated code can be translated to the desired binary program for the TP by utilizing the tool suite of the processor (assembler, linker). This binary program is downloaded to the TP via the respective interface to the ETE prior to the test execution. To this end, the binary program might be further translated into a format that the ETE can process. Thus, if the ETE is a standard tester, then it is beneficial to translate the binary code into a standard pattern format. Consequently, the program can be downloaded to the TP during the runtime of the testflow executed by the tester.

The resulting flow for the generation of the test program is shown in Figure 4.5.

4.3.2.2 Algorithm for Generating the Transfer Protocol

The major issue of the generation of the *transfer protocol* is the description of the transfers between the TP and the DUT. With respect to this, the question raises how asynchronous and potentially concurrent transfers are sequentialized such that they can be written into a file. Additionally, the generated sequence has to be relaxed from static timing, since the non-deterministic timing behavior of the DUT could affect the order of transfers observed during test compared to the order observed during simulation. Listing 4.1 shows two concurrent processes written in VHDL to illustrate this issue. Assume that, according to their names, one process writes to the input port and the other reads from the output port of an asynchronous FIFO. After writing one data token into the FIFO basically one of two events may happen next: either another word is written into the FIFO or the first data word is read from the output port. Obviously, the sequence of transfers written to the *transfer protocol* depends on the timing of the ideal model of the FIFO, whereas the measured sequence depends on the timing of the circuitry implementing the FIFO. Consequently, these sequences may differ.

In order to solve this issue, one needs to consider the possible causal relations of

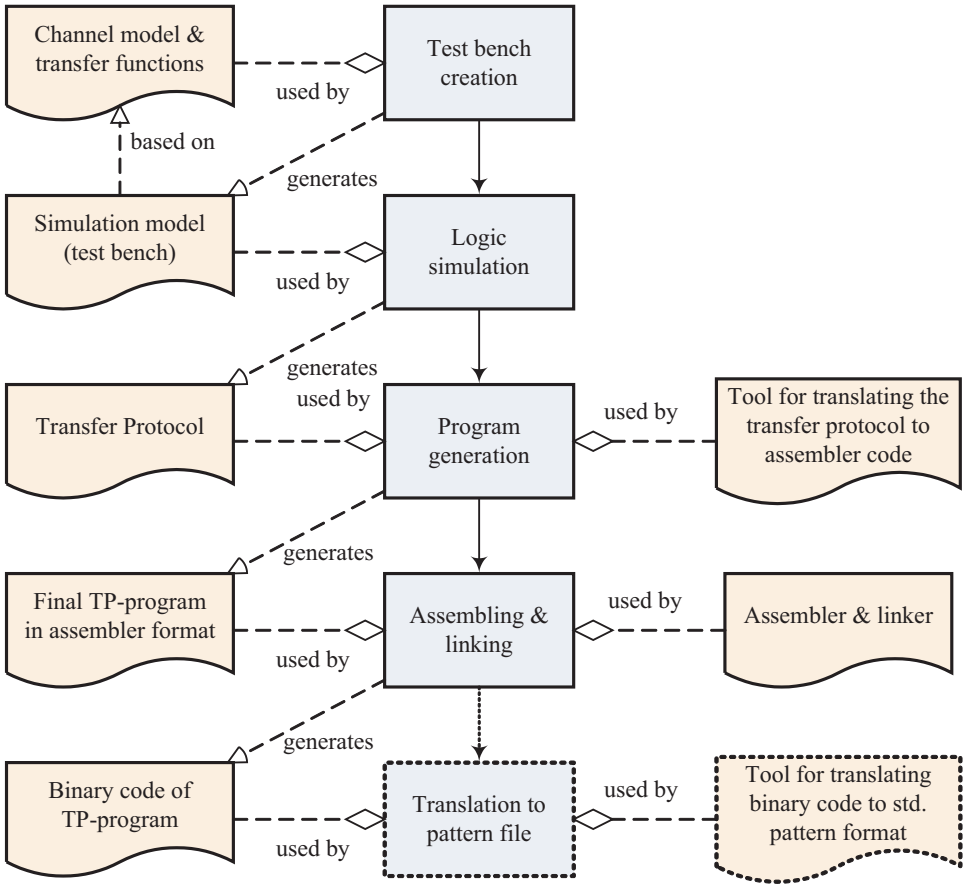


Figure 4.5: Flow for generating the test processor program

Listing 4.1: Two concurrent processes

```
...
sending : process
begin
    ...
    for i in 0 to 100 loop
        send( input_channel, i );
    end loop;
end process;

receiving : process
begin
    ...
    loop
        recv( output_channel );
    end loop;
end process;
...
```

the transfers. In [Seitz 1980] C.L. Seitz described two possible relations between the occurrences of signal transitions in a self-timed system. In the context of this work, channel transfer are considered here rather than signal transitions. The first relation between two transfers t_i and t_j is $t_i \leq t_j$ which means t_i occurs *before* t_j or, the other way round, t_j occurs *after* t_i. The second relation is $t_i \parallel t_j$ meaning t_i is *concurrent* and not ordered with t_j. Thereby, the relation $\leq$ is a partial ordering on the set of occurrences of channel transfers. This relation is reflexive ($t_i \leq t_i$), antisymmetric ($t_i \leq t_j \wedge t_j \leq t_i \Rightarrow t_i = t_j$), and transitive ($t_i \leq t_j \wedge t_j \leq t_k \Rightarrow t_i \leq t_k$). As further described by C.L. Seitz, the notion of *simultaneity* has no meaning and is disallowed due to the antisymmetric property of the relation $\leq$. It is additionally important to note that these relations describe causalities in the occurrences of transitions.

According to these relations, the *transfer protocol* needs to support mechanisms to express *concurrency* and *sequences* as it is also possible in common description languages for asynchronous circuits, such as Balsa and Tangram/Haste. Based on these two relations, a test can be defined as a *sequence* $G_1, G_2, G_3 \ldots$ of *groups of concurrent transfers* G_i, $i \in \mathbb{N}$, such that all transfers $t \in G_i$ happen before the transfers in G_{i+1}. One can compare these *concurrent transfer groups* with simultaneous value changes in standard pattern formats, such as the VCD format. But in contrast to that, the transfers of such a group are not required to fire simultaneously. Instead, they can fire at arbitrary moments in time and in any order.

To generate this *sequence of concurrent transfer groups* during logic simulation, one has to define the dividing lines between the groups. Therefore, it is assumed that any two transfers t_i, t_j can potentially be concurrent as long as they do not access the same resource. In case of the considered test processor scenario, such a resource is a channel. Accordingly, when a transfer t_j is encountered that accesses a channel which already has been accessed by a different transfer t_i, then t_j obviously occurs after t_i. Due to the transitivity of the sequence relation $\leq$, all transfers that fire after t_j also fire after t_i. Note that simultaneous transfers are basically not allowed. However, in theory a transfer t_k that is concurrent to a transfer t_j where $t_i \leq t_j$, does not necessarily occur after t_i. To resolve this, it is further assumed that the sequence of transfers during test shall have the same timing relations as the transfers in the simulation. Accordingly, the timing of the occurrences of the transfers is taken into account. Thus, any transfer t_k that temporally fires after a transfer t_j with $t_i \leq t_j$, is also assumed to occur after t_i. One can describe this formally as shown in Equation 4.3.

$$\Big(t_i \leq t_j\Big) \wedge \Big(time(t_j) < time(t_k)\Big) \Rightarrow t_i \leq t_k. \tag{4.3}$$

Under these considerations, an algorithm can be defined that generates the desired

sequence of concurrent transfer groups. Therefore, consider the following definitions

- Let S_C be the set of channels $C_1, \ldots, C_m$ between the DUT and the TP.
- Let T be the set of transfers on the channels $C_1, \ldots, C_m$.
- Let $f_C : T \rightarrow S_C$ be a function mapping a transfer to its corresponding channel.
- Let $G_i \subseteq T,\ 1 \leq i \leq n$ be the groups of concurrent transfers.
- Let $\sigma = (G_1, \ldots, G_n)$ be the sequence of concurrent groups.
- Let initially be $k = 1$ and $1 \leq k \leq n$.

Thereby, G_k designates the presently considered group of concurrent transfers. Initially, all groups $G_i,\ 1 \leq i \leq n$ are empty. Now, let t be the transfer that fires next in the simulation and, therefore, t shall be added to the *transfer protocol*. If t does not access a channel that has been accessed by any other transfer $t_x \in G_k$, then t can be added to G_k. Otherwise, t cannot be concurrent to the transfers in G_k and has to be added to the next group. Therefore, k is incremented and G_k designates the next group in the sequence. Finally, t is added to G_k.

Listing 4.2: Pseudo-code for generating the sequence of concurrent transfer groups

```
procedure log_transfer(t : T)
    variable b : boolean := false;
begin
  foreach t' ∈ G_k do
    if f_C(t) = f_C(t') then
      b := true;
      break;
    end if;
  done;

  if b = true then
    k := k + 1;
    G_k := ∅;
  end if;

  G_k := G_k ∪ {t};
end procedure;
```

To utilize this algorithm during simulation, the *transfer procedures* call the procedure `log_transfer()`. As will be shown in the continuation, this algorithm has one

limitation which has to be manually resolved by the user. However, to illustrate its applicability, it has to be shown that the relations of the transitions executed during test are the same as the ones observed in the simulation. Therefore, the relations of every two consecutive transfers in the simulation are considered, since the simulator also processes the transfers sequentially. Thus, let t_i, t_j be two transfers with $time(t_i) < time(t_j)$ and $\forall t_k \in T,\ time(t_j) < time(t_k)$, i.e., t_j fires next after t_i in the simulation. Now, one has to distinguish between several cases:

1. If $t_i \parallel t_j \wedge f_C(t_i) \neq f_C(t_j)$, then the algorithm adds t_i and t_j into the same group. Thus, t_i and t_j are concurrent in the *test sequence* as well.

2. The case where $t_i \parallel t_j \wedge f_C(t_i) = f_C(t_j)$ is invalid in consideration of functionally deterministic circuits.

3. If $t_i \leq t_j \wedge f_C(t_i) = f_C(t_j)$, the algorithm adds t_i and t_j into separate groups, say t_i is added to G_x and t_j is added to G_{x+1}. Thus, t_i and t_j are sequential in the *test sequence*.

4. Unfortunately, if $t_i \leq t_j \wedge f_C(t_i) \neq f_C(t_j)$, then the algorithm adds t_i and t_j into the same group which does not reflect the sequence relation between t_i and t_j. However, in practice the DUT itself often resolves this conflicting situation. For example, one may consider again the asynchronous FIFO. Due to the limitation of the algorithm, the generated sequence may start with a group including both a transfer at the input and one at the output interface. However, due to the behavior of the FIFO, first the input transfer will be performed and after that the transfer at the output. Nevertheless, a mechanism is required that allows the user to define sequential relations between transitions.

In order to overcome this limitation, a second method is required that waits for a set of transfers. Of course, the identification of a transfer might be ambiguous. Instead, it is easier to wait for the execution of transfers on a set of channels. A possible solution to determine whether a transfer on a specific channel was performed is to count the number of transitions on the channel. This is again only possible for functionally deterministic circuits, where the number and order of output responses are well defined. Accordingly, a function $f_N : S_C \rightarrow \mathbb{N}$ is defined that maps a channel $C_i \in S_C$ to the number of its transitions. Based on this, a second procedure can be defined that ensures the sequence relation between transfers on a specified set of channels and all transfers executed after its call. In case of the considered algorithm, the procedure simply has to increment the

Listing 4.3: Pseudo-code of the wait procedure

```
procedure wait_for_transfers(S'_C : S*_C)
    variable n_T : map from S_C to ℕ;
    variable b   : boolean;
begin
  foreach C ∈ S'_C do
    n_T(C) := f_N(C);
  done;

  do
    wait_for_activity(S_C);
    b := false;
    foreach C ∈ S'_C do
      if n_T(C) = f_N(C) then
        b := true;
        break;
      end if;
    done;
  while b = true;

  k := k + 1;
end procedure;
```

group index k. Listing 4.3 shows the resulting procedure, which is here and after referred to as *wait procedure*.

In the first loop, the current number of transfers on each of the channels $C \in S_C$ is stored. Afterwards, the second loop waits for any activity on any of the channels. Such a functionality is supported by any common hardware description language (HDL). Then, it is checked whether the numbers of transitions of all channels have changed. Note that the function f_N is concurrently adapted by the *transfer procedures* according to the current number of transfers on each channel.

To ensure that a transfer t occurs after the transfers on the specified channels, the *wait procedure* has to be called prior to the call of the *transfer procedure* performing t. Finally, after executing the entire simulation of the test bench, the sequence S contains all transfers aligned in *concurrent groups*. As mentioned, when executing the test, all transfers of a group G_i are executed before the transfers of G_{i+1}. Obviously, this may block transfers until all transfers that occur earlier in the simulation have been performed, even though one of the blocked transfers might be actually concurrent to some of the previously executed transfers. Similarly, it might happen that the execution of the transfer sequence is interrupted, although the transfer to wait for actually may also occur later. However, since the transfer sequence of the test shall be the same as the one of the

simulation, these situations are negligible.

4.3.2.3 Definition of the Transfer Protocol Format

The purpose of the *transfer protocol* is to describe the transactions of the DUT and its environment, i.e., the pattern for the functional test. As a prerequisite for this, information about the asynchronous channels used in the sequence is required. Accordingly, each channel accessed in the sequence has to be fully specified prior to its usage. This includes all the information that is part of the model defined in Section 2.1.4. Based on this information, the processor has to be configured before the actual *test sequence* is executed. Finally, as described above, the *test sequence* itself comprises *transfer statements* that are combined via *concurrency* and *sequence operators* expressing the relations between the transfers. In the following, the *transfer protocol* grammar is defined in the Bacus-Nauer-Form (BNF).

Before discussing the format in detail, the question shall be answered why a new format is defined rather than using an existing language, such as Balsa. The reason for this is the need for specific constructs which are not supported by any other language, but which are essential for the generation of a test processor program. An example for this is the construct to define properties of the channels used within the *test sequence*. However, the format of the *transfer protocol* has several similarities to existing languages. For example, the *channel transfer operators* '=>' and '<=' are very similar to the handshake operators in Balsa.

To separate the definition of the resources from the *test sequence*, the *transfer protocol* file is composed of a *declaration* and a *test sequence section*. These sections are introduced by the corresponding keywords `DECLARE` and `TEST`, and are terminated with the `END` keyword.

<*transfer protocol*> ::= <*declaration section*> <*test sequence section*>

In the declaration section all the resources used within the *test sequence section* have to be declared. As defined previously, a resource can either be a channel or a signal/bus.

<*declaration section*> ::= '`DECLARE`' <*declarations*> '`END`'
<*declarations*> ::= | <*channel declaration*> '`;`' <*declarations*>
| <*signal declaration*> '`;`' <*declarations*>

In order to address a channel in the *test sequence*, it is required to associate an identifier with a channel. Thus, a *channel declaration* comprises an identifier and the channel properties. These properties include the type (push or pull), the number of phases (two or four), the encoding (single-rail or dual-rail) and the initial values of the handshake

signals. Furthermore, it is required to specify whether a channel is an input or an output of the DUT. This information is essential for the configuration of the processor and for the mapping of the handshake signals to the input and output of a handshake port of the processor (see Equation 4.1 and 4.2). Therefore, the direction of the channel from the DUT point of view has to be defined. Thus, if an input channel of the DUT shall be defined the direction has to be set to `IN`. Besides these properties that concern the control part of the channel, the information about the data signals needs to be defined. This only includes the number of data signals. An explicit definition of the handshake and the data signals is not necessary, since these are hidden within the channel.

<*channel declaration*> ::= <*identifier*> '`:`'
<*direction*> '`CHANNEL`' '`(`' <*channel properties*> '`)`'
<*channel properties*> ::= <*type*> '`,`' <*phases*> '`,`' <*encoding*> '`,`'
<*req-init*> '`,`' <*ack-init*> '`,`' <*number*>

<*direction*> ::= '`IN`' | '`OUT`'
<*type*> ::= '`PUSH`' | '`PULL`'
<*phases*> ::= '`2P`' | '`4P`'
<*encoding*> ::= '`SR`' | '`DR`'
<*req-init*> ::= <*bit*>
<*ack-init*> ::= <*bit*>

The declaration of a signal resource is similar. A signal resource is declared by defining an identifier, the direction of the signal and the type, i.e., either a single signal or a bus. For a bus, one needs to define the bus range identifying the indices of the leftmost and the rightmost bus signal, respectively.

<*signal declaration*> ::= <*identifier*> '`:`' <*direction*> <*signal specification*>
<*signal specification*> ::= '`SIGNAL`' | '`BUS`' '`(`' <*number*> '`...`' <*number*> '`)`'

After declaring all resources the *test sequence* can be defined in a block embraced by the `TEST` and `END` keywords. The elements of this sequence are separated using the *sequence operator* '`;`'. Each of these elements can either be a group of *concurrent statements* or a *nonconcurrent statement*. Thereby, a *concurrent statement* is a *transfer statement* or a *signal operation*, either. To indicate the concurrency, the transfer statements of a concurrent group are separated by the *concurrency operator* '`,`'. A *Nonconcurrent statement* must not be part of a concurrent group. An example of this is the *wait statement* introduced later.

<*test sequence section*> ::= 'TEST' <*test sequence*> 'END'
<*test sequence*> ::= | <*concurrent statements*> ';' <*test sequence*>
| <*nonconcurrent statement*> ';' <*test sequence*>
<*concurrent statements*> ::= <*concurrent statement*>
| <*concurrent statements*> ',' <*concurrent statement*>
<*concurrent statement*> ::= <*transfer statement*> | <*signal operation*>

In general, a *transfer statement* is used to apply stimuli or to read responses from a channel. It comprises a channel identifier, the *transfer operator*, and a logic vector. The *transfer operator* indicates whether to write to, or to read from the channel. The logic vector describes the pattern. In case of an output channel, this pattern is the expected value. However, such a *transfer statement* can only be applied if the channel comprises data signals. Thus, a different construct is required to support control/synchronization channels which do not have any data signals. To this end, the SYNC keyword is introduced. In the same way as in BALSA, it forces a handshake on the specified channel without transferring data. This construct can also be applied to standard channels that have data signals. For example, if the response of an output channel is negligible, then the SYNC statement can be used to perform handshake synchronization only.

<*transfer statement*> ::= <*identifier*> <*transfer operator*> <*logic value*>
| 'SYNC' <*identifier*>
<*transfer operator*> ::= '<=' | '=>'

As opposed to transfer statements, *signal operations* assign or compare logical values to signal resources. Apart from that, a major difference to *transfer statements* is that signal operations are not flexible in timing. This means that they are executed at the moment of reaching them in the *test sequence*. This is clear for a signal assignment. For a comparison, this means that the *test sequence* is treated as failed if the signal resource does not correspond to the specified value. Obviously, such a comparison only makes sense at well-defined states of the DUT, e.g., after specific transfers. A cognate statement is the *signal wait statement* introduced farther below.

<*signal operation*> ::= <*identifier*> <*signal operator*> <*logic value*>
<*signal operator*> ::= ':=' | '=='

As mentioned, *nonconcurrent statements* affect the global state and behavior of the TP. Therefore, these statements must not occur in a concurrent group. In the current version of the *transfer protocol* format, these statements can either be a *timeout statement*, a *timeout routine statement*, a *wait statement* or a *signal wait statement*. *Timeout statements* and *timeout routine statements* are constructs dedicated to the detection of

deadlocks during the test. With the help of a *timeout statement* one can define the maximum time to wait for an event of the DUT. If the time required for an operation exceeds the specified timeout period, then a deadlock is assumed. In this case, a routine is called which can be specified with the help of a *timeout routine statement*. Therefore, the name of the routine has to be passed as argument to the statement. A *wait statement* is used to define an idle period which can be used to hold a certain state for a specified time. A similar construct is the *signal wait statement*. It also holds a certain state, but it uses a signal comparison as the condition to continue the execution of the *test sequence*.

<*nonconcurrent statement*> ::= <*timeout statement*> | <*wait statement*>
| <*signal wait statement*>
<*timeout statement*> ::= 'SET' 'TIMEOUT' <*time*>
<*timeout routine statement*> ::= 'SET' 'TIMEOUT_ROUTINE' <*identifier*>
<*wait statement*> ::= 'WAIT' 'FOR' <*time*>
<*signal wait statement*> ::= 'WAIT' 'UNTIL' <*compare statement*>

Finally, the supported simple data types of the *transfer protocol* are defined as follows:

<*identifier*> ::= <*alpha*> | <*identifier*> <*alpha*> | <*identifier*> <*digit*>
<*time*> ::= <*real*> <*time unit*>
<*real*> ::= <*number*> '.' <*number*>
<*number*> ::= <*digit*> | <*number*> <*digit*>
<*alpha*> ::= 'a' | ... | 'z' | 'A' | ... | 'Z'
<*logic value*> ::= <*bit*> | <*bit vector*>
<*bit*> ::= ''' <*logic*> '''
<*bit vector*> ::= '"' <*logic vector*> '"'
<*logic vector*> ::= <*binary*> | <*logic vector*> <*binary*>
<*logic*> ::= '0' | '1' | 'X'
<*digit*> ::= '0' | '1' | ... | '9'
<*time unit*> ::= 'ps' | 'ns' | 'us' | 'ms' | 's'

4.4 Summary of the Concept

To summarize, the concept comprises a test processor that provides generic interfaces including handshake control and data ports. The ports can be combined in order to establish asynchronous handshake channels. Thereby, the realization of the combination between the ports is intentionally not defined by the concept. Instead, this depends on the particular realization of the test processor. For example, in an on-chip implementation

the combination of the ports can be realized fully in hardware, since the interconnection to the components to be tested is fixed. Thus, there is no need for a programmable association between the handshake and the data ports. In comparison to that, the combination in an off-chip implementation should be configurable, such that it can be adapted to the interfaces of a large variety of designs to be tested.

In comparison to other functional test approaches, a test processor, as introduced here, fulfills all criteria for performing functional tests. It provides full flow control, supports fast and easy exchange of test patterns, overcomes worst-case timing assumptions due to real asynchronous communication. Furthermore, in case of an off-chip implementation, the processor can support various different designs by keeping the handshake ports and the combination of the ports to channels configurable. As will be shown in the next chapters, the processor concept is an appropriate test equipment for performing functional tests.

Besides the processor, the theoretical background to describe and generate patterns for elastic tests is defined. The key idea of the approach is that the pattern generation is based on standard behavioral simulations of the DUT.

"Nothing is particularly hard if
you divide it into small jobs."

— Henry Ford

Chapter 5

Test Processor Implementation

Based on the generic concept, a specialized test processor architecture, called NoTePAD (Novel Test Processor for Asynchronous Devices) is proposed in this chapter. The intention of this implementation is to provide a configurable, high-performance test equipment that enables elastic functional tests of asynchronous circuits. The introduced architecture is highly optimized for the two major tasks during functional tests: the application of stimuli, and receipt/comparison of responses. Consequently, the provided test processor solution is more feasible for an off-chip implementation. The development of the solution was basically driven by the ideas of preceding implementations presented in [Zeidler 2011, Zeidler 2012a]. Similar to the solutions for synchronous ICs, provided in [Galke 2002, Frost 2007], the intention of these approaches was to describe an on-chip test processor for testing asynchronous components of an SoC. The on-chip TP implementation especially makes sense if the SoC already comprises a processor. Then, the processor can be adapted, such that it can also be used for test purposes. Thus, readers interested in on-chip implementations of the concept are kindly referred to these publications.

5.1 Design Decisions

The two test processor implementations proposed previously fully comply to the generic concept. Although they are basically intended for integration into an SoC, the test processors can, of course, also be realized off-chip as a stand alone test systems. However, their RISC architecture imposes several performance limitations mainly caused by the following aspects:

- The handshake protocols are described in software. The handshake ports of the processors only provide mechanisms to ease the detection and the generation of protocol events. In order to generate or to react to a single event, the corresponding instruction has to be called. This implies that only one channel transfer can be processed at a time.

- The processor core has to transmit the data between the ports and the memory via explicit calls of instructions. This is one of the most critical bottlenecks of the architecture.

- The association between the handshake and data ports to form asynchronous channels is also implemented in software. To perform a data transfer, the ports associated with a channel are individually addressed in separate instructions. This extremely increases the program size and the time for a single data transfer.

To overcome these limitations, the novel architecture shall provide mechanisms to

- generate the sequence of handshake signal transitions completely in hardware,

- directly couple handshake ports with an arbitrary number of data ports,

- concurrently process an arbitrary number of transfers at different channels at a time.

A direct consequence of the realization of these mechanisms is that the number of instructions for performing one data transfer is reduced. To unify the execution of all types of asynchronous transfers, the HPs shall be implemented such that only a single instruction is required to perform any kind of asynchronous handshakes. Therefore, the handshake execution is implemented in hardware. After its initial configuration, a HP shall generate the sequence of handshake signal transitions depending on the protocol type and the direction of the transfer.

Based on these advanced requirements, a couple of decisions are taken that affect the design of the processor. One basic decision is related to the design methodology of the processor. As well as the previous RISC implementations, NoTePAD is a synchronous design. This eases the implementation, e.g., using an FPGA, and the test of the TP itself. Especially, the testability is dramatically improved in case the TP is implemented as an IC, since standard test approaches for synchronous designs, such as scan, can be applied.

A further design decision affects the protocol support. Protocols which encode the events into data signals, e.g., dual-rail protocols, require that these signals are safely

transferred to the receiver to prevent metastability. Thus, the synchronous implementation of the processor demands that the data input signals have to be migrated into the clock domain of the processor, e.g., using synchronizers. This considerably increases the interface logic. Furthermore, to detect the events of such protocols, it would be necessary to interconnect the data ports. For example, consider a dual-rail interface as previously shown in Figure 2.6. The completion detection requires the interconnection of pairs of data ports via OR-gates. Finally, the outputs of the OR-gates have to be connected via a *C*-element. A configurable implementation of this scheme would be extremely complicated. Therefore, NoTePAD will only support bundled-data protocols. By this, only the control input signals and the inputs of the handshake ports have to be equipped with a synchronizer as shown in Figure 5.1a. However, if the test processor is implemented using an FPGA, protocol converters can be used to interface the TP with the asynchronous DUT. These protocol converters have to be created for each channel that uses a non-single-rail encoding scheme. Afterwards, the converters have to be connected with the ports of the TP and finally integrated into the FPGA. Examples of such converters are given for the conversions between single and dual-rail interfaces in Appendix B. Moreover, with respect to 4-phase bundled-data protocols, the data validity scheme has to be defined. The most commonly used schemes are the *early* and the *broad* scheme. Therefore, NoTePAD will only support these schemes.

These decisions have some implications to the execution of the handshake signalling. Unfortunately, a synchronous implementation of the handshake procedure imposes additional latency. Although the HPs shall implement respective automatons for all different kinds of bundled-data protocols in hardware, the required number of cycles for a handshake is strictly coupled to the number of phases of the protocol and the synchronization latency, e.g., when using a two-flop synchronizers. For example, to perform a 4-phase handshake at least six cycles are required. Figure 5.1b illustrates this behavior for a 4-

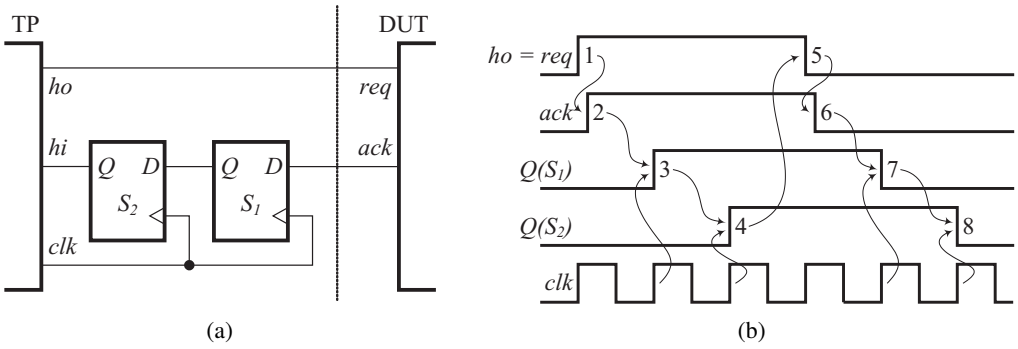


Figure 5.1: Handshake interface between the TP and the DUT

phase single-rail push handshake initiated by the TP. In fact, the handshake requires only 5 cycles, but one additional cycle is required before the next handshake can be initiated.

In step 1 the TP assigns the request whereupon the DUT responds by setting the respective acknowledgment in step 2. However, due to the two-flop synchronizer this response is no earlier than two cycles later visible to the TP in step 4. When the request was detected, the TP may initiate the return-to-zero phase one cycle later in step 5. Finally, the steps 6–8 exhibit the same behavior as the steps 2–4.

5.2 Hardware Implementation

The test processor is designed according to the criteria defined in Section 1.2 and the above mentioned design decisions. In order to optimize the architecture with respect to its performance, the processor is modelled at Register Transfer Level (RTL) rather than architectural level, e.g., using an architecture description language (ADL) such as LISA (Language for Instruction Set Architectures) [Zivojnovic 1996] as it was done with the previous TP approaches. This section discusses the overall architecture of NoTePAD and all its components.

5.2.1 Global Architecture of NoTePAD

The first step of the design of the processor architecture contemplates the elementary components. According to the concept, the processor comprises a core coordinating all activities of the processor and the port component that includes the handshake and data ports. In order to enable a direct coupling of the HPs with the DPs, the port component has to provide a mechanism for the interconnection of these ports. Therefore, a *port switch* is introduced which connects each HP with a set of DPs. To this end, the switch stores the channel information defining the association of the HPs with the DPs.

The idea followed with this interconnection is to let the handshake ports directly control the associated data ports. Thus, when a data transfer shall be performed at a certain channel, then the respective HP forces all associated DPs to apply new data or to read the responses of the DUT. In the opposite direction the DPs report their statuses to the *port switch* that bundles the information according to the channel configuration and sends completion information to the HPs.

As a second step, the tasks of the processor core are considered. According to the definition in Section 4.2.3, the core coordinates the activities of the processor and provides access to the memory for the DP, such that these can load and store patterns. These tasks are almost completely independent of each other. Therefore, the core is separated

into two independently operating modules: the *sequencer* and the *memory access controller*.

The *sequencer* is a simple microcontroller that fetches, decodes and executes the instructions from the program memory and controls all the other components of the processor. It sends commands to the handshake ports which, in turn, forward commands to the DPs via the *port switch*. For this to be applicable, the *port switch* has to be configured according to the *pin configuration*. This is one of the basic tasks of the *sequencer* prior to the test execution. Furthermore, the *sequencer* also provides the interface for the ETE as described in Section 4.2.2.

The *memory access controller* connects all DPs with the memory. Its major task is to coordinate the memory accesses initiated by the DPs. Behind this approach is the idea to let the DPs autonomously read and write data from and to the memory. By this, the central controlling unit, i.e., the *sequencer*, is relieved of managing the access to the data memory. This eliminates one essential bottleneck of the previously published test processor implementations. In order to read stimuli and to store test results, each DP can issue a read or write request to the *memory access controller*. Thus, the controller has to resolve access conflicts. Therefore, the controller arbitrates between the different requests and forwards them to the memory.

For the interconnection of the components, one has to take into account that the durations of the individual operations of NoTePAD, e.g., performing a data transfer with the DUT, are not fixed. Therefore, the global interconnection scheme between almost all components of NoTePAD is based on synchronous handshake mechanisms. This means that a receiving component provides a completion signal to the sending component. The sending component is only allowed to issue new data if the receiving component has indicated the completion of the last transaction. For example, for the control of the handshake ports, the *sequencer* sends one command to all of these ports and selects the ones that shall execute this instruction. Then, the *sequencer* waits until all selected handshake ports have completed their operation before issuing a new one. Therefore, each HP provides a completion indication signal to the *sequencer*. In a similar way, the HPs are connected with the DPs via the *port switch*. Each DP delivers a status signal to the *port switch*. According to the *pin configuration*, the switch combines these signals to one status signal for each HP.

Figure 5.2 illustrates the resulting control flow. The *sequencer* issues a command to the HPs which send a corresponding command to the *port switch*. The *port switch*, in turn, forwards the commands to the associated DPs which execute the command in the step denoted by the label 1. After the completion of the operation, each DP sends an indication signal to the *port switch*. The *port switch* then computes the completion

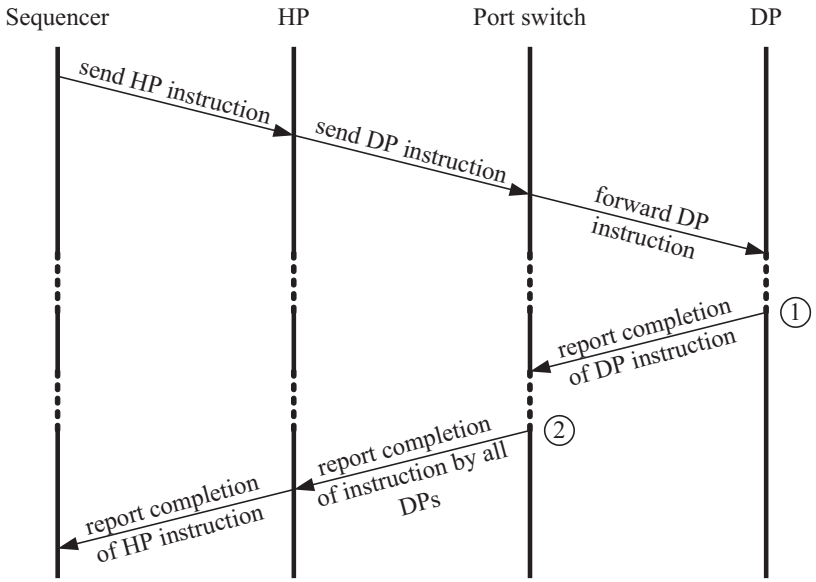


Figure 5.2: Control flow of NoTePAD

information from all DPs associated with a HP. If all associated DPs have completed the operation (labelled with 2), then the completion indication is sent to the HP. Finally, the HP reports the completion of the command to the *sequencer*.

Eventually, the global structure is derived from these basic architectural considerations. Figure 5.3 illustrates all components and their major interconnects. As shown in the figure, the processor is connected to two independent memory blocks. One stores the data including the test patterns and additional data required for the execution of the test, such as configurations for the ports. The other memory block stores the program for the *sequencer*. These blocks do not necessarily have to be separated. However, the memory should be connected via independent memory ports in order to enable the processor to simultaneously access the data and the program. Thus, one can also think about a dual-port Random Access Memory (RAM) implementation, where one port is connected to the *sequencer* and the other is connected to the *memory access controller*.

5.2.2 Design of the Data Ports

According to the concept, a data port is a component that provides a set of inputs and a set of outputs to exchange data with the DUT. Thus, a data port supports mechanisms for applying, receiving, and comparing data. For the implementation of the ports, these functionalities are considered separately.

As commonly known, three states have to be distinguished, when digital data is

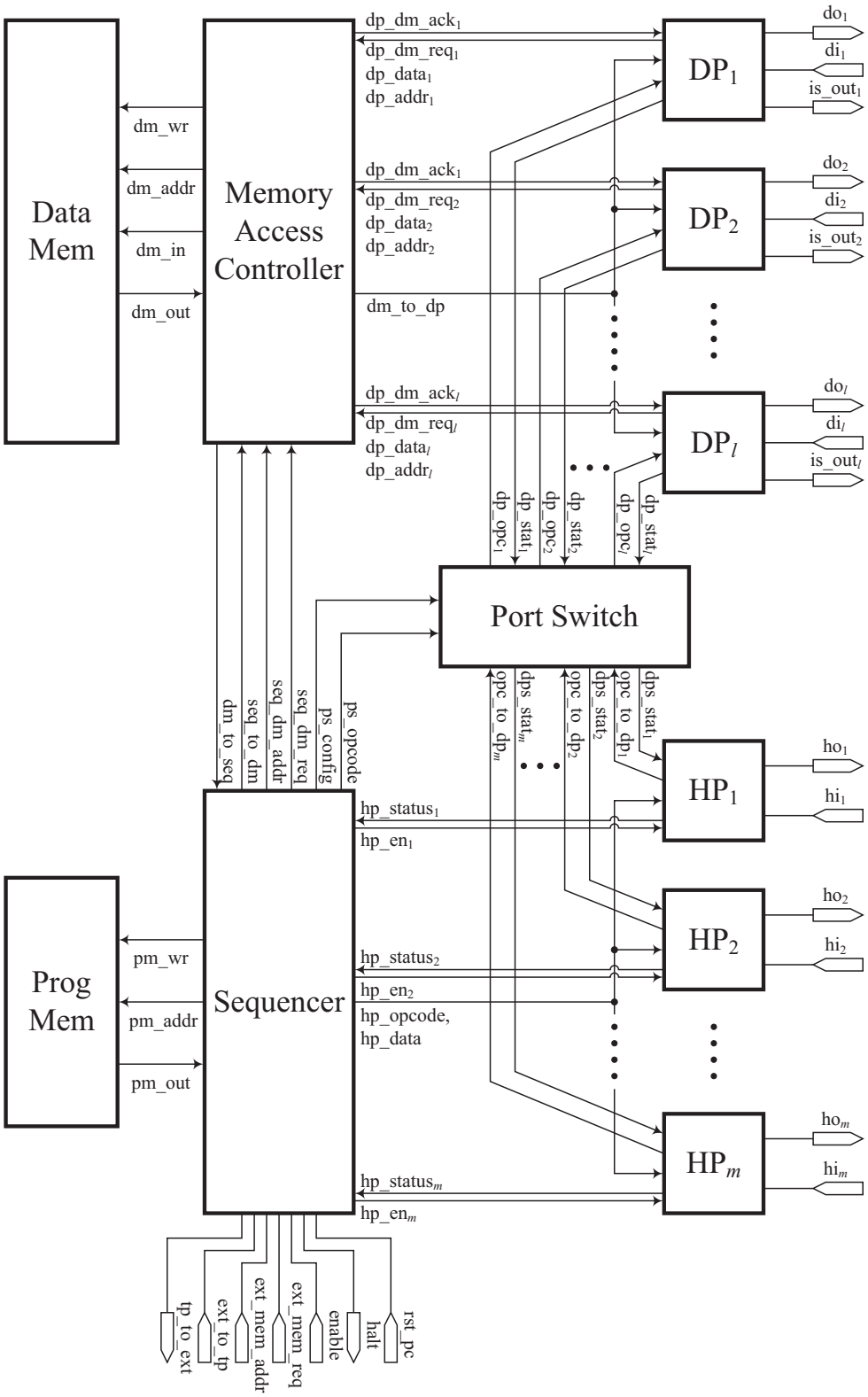


Figure 5.3: Architecture of NoTePAD

applied to a single output signal. These states can be described by a three-valued logic $L_{apply} = \{0, 1, Z\}$:

- $0/1$ designate the digital logical values applied to an output signal. These values represent the low and high voltage levels, respectively.
- Z indicates high impedance, thus, no data is applied. In this case, the output driver is switched off and the signal may obtain a value driven by another source.

In the opposite direction, there are also three states possible, when data is read from an input signal. These states can be described by the logic $L_{receive} = \{L, H, X\}$.

- L/H are the digital logical values read from an input source, representing low and high voltage levels, respectively. Note that these values are intentionally different to $0/1$, for a clear separation between drive and strobe operations.
- X indicates an unknown logical value. This means that a data bit is either logical-0 or logical-1.

Since the data ports shall support both operations, these logics are merged to $L_{test} = L_{apply} \cup L_{receive}$. To represent the members of the resulting logic with a digital logic, a triple (v_i, m_i, d_i), $v_i, m_i, d_i \in \{0, 1\}$ can be used, where

- v_i determines the logical value,
- m_i represents a mask determining whether an input is significant or not, and
- d_i is the drive information used to switch the output driver of the data bit b_i of a DP on or off.

Then, the logic L_{test} can be mapped to the following values of (v_i, m_i, d_i):

$$0 \mapsto (v_i, m_i, d_i) = (0, 1, 1) \qquad L \mapsto (v_i, m_i, d_i) = (0, 1, 0)$$
$$1 \mapsto (v_i, m_i, d_i) = (1, 1, 1) \qquad H \mapsto (v_i, m_i, d_i) = (1, 1, 0)$$
$$Z \mapsto (v_i, m_i, d_i) = (-, -, 0) \qquad X \mapsto (v_i, m_i, d_i) = (-, 0, 0)$$

where '$-$' represents a *don't care* value

This triple is the data required for every single bit of the data ports. To store this data, a scheme is applied that is similar to the solution of the test processor provided in [Galke 2002]. In this realization, the I/O ports use a combination of two registers: one stores the value $v = v_0 \dots v_{n-1}$ applied to the outputs or read from the inputs, and another register stores the mask $m = m_0 \dots m_{n-1}$. Thereby, n designates the bit width of the DPs, which is 16 in the implementation provided in [Galke 2002]. However, their approach

lacks of the drive information, since their test processor is intended for integration into an SoC. Thus, the DPs can directly be connected with in- and outputs of the components to be tested. Therefore, the drive information is not required.

In contrast to that, NoTePAD is a stand-alone test equipment that is intended to be realized outside of the DUT-chip. Its data I/O pins should be used for both input and output data transmissions. Therefore, a further register is required that stores the drive information. However, since each DP is associated with at most one unidirectional channel, it is not necessary to store drive information for every single pin of the DP. Instead, only a single flip-flop stores the global drive information for all bits of a data port.

A further issue of the DP concerns the number of inputs and outputs of the ports. As afore indicated, the DPs in previous test processor implementations comprise a relatively high number of inputs and outputs, i.e., 16 [Galke 2002, Zeidler 2011] and 32 [Zeidler 2012a] respectively. Basically, this number is derived from the width of the data bus of the processor. However, if channels shall be realized, whose data part is not a multiple of the bit width n of the DPs, then a considerable number of in- and outputs might be wasted. Thus, this number is too large for a generic solution that is sought by NoTePAD.

For this reason, a different scheme is applied. Basically, each data port also comprises a value register storing $v = v_0 \dots v_{n-1}$ and a mask register storing $m = m_0 \dots m_{n-1}$. However, the DPs have only $k < n$ in- and k outputs. In the continuation, these inputs and outputs are labelled with `di` and `do`, respectively. The crux of the implementation is that the registers are organized as shift registers. Thus, when applying new stimuli or performing a comparison operation the registers are shifted by k bits. The reason for storing n rather than k bits is related to the width of the bus. Thus, when reading patterns from the memory, the data word delivered by the memory comprises data for n/k transfer operations. Obviously, n has to be a multiple of k.

With this scheme, the bits $\widetilde{v} = v_0 \dots v_{k-1}$ are directly connected to the outputs $\texttt{do}_0 \dots \texttt{do}_{k-1}$ of the DP to apply stimuli. In order to implement the receive functionality, the same set of bits $\widetilde{v} = v_0 \dots v_{k-1}$ is compared with the k inputs of the DP. Therefore, the inputs $\texttt{di}_0 \dots \texttt{di}_{k-1}$ of the DP are bitwise connected with $v_0 \dots v_{k-1}$ via XOR-gates. The output of these XOR-gates are in turn connected with the k bits $\widetilde{m} = m_0 \dots m_{k-1}$ of the mask register via AND-gates. By this, all non-significant bits are set to logical-0. The resulting bits $(v_0 \oplus \texttt{di}_0)m_0 \dots (v_{k-1} \oplus \texttt{di}_{k-1})m_{k-1}$ are shifted into the value register. Hence, after each n/k shift operations the value register contains the desired fault signatures of the last n/k comparisons. This signature has to be written to the memory. However, since the new data can be loaded in parallel to the writing of the signature, the

number of shift operations is actually $n/k - 1$. Then, the result signature comprises the value $(v_0 \oplus \mathtt{di}_0)m_0 \ldots (v_{k-1} \oplus \mathtt{di}_{k-1})m_{k-1}$ of the outputs of the AND-gates and the value $v_k \ldots v_{n-1}$ stored in the shift register.

As mentioned previously, all DPs are indirectly connected with the memory via the memory access controller (MAC) for loading and storing data. As a basic prerequisite for that, each DP has to autonomously read and write the respective data from and to the memory. A key issue of this scheme is that the loading and the storing of the data of the shift registers become a bottleneck, since all DPs associated with a channel would issue a memory access request after each n/k data transfers. A simple solution to that issue is the introduction of FIFO buffers. Therefore, an *output FIFO* is used to buffer the data from the memory. Thus, depending on the use of the data port as an input or an output, this FIFO stores either input stimuli for the DUT or expected output signatures. The input interface of the FIFO is connected to the MAC, whereas its output interface is connected with the shift registers. Accordingly, an *input FIFO* is used to store the result signatures that are to be written to the memory. Therefore, its input interface is connected with the value register, and the output interface is connected with the MAC. This is illustrated later in Figure 5.5 showing the entire architecture of the ports.

The properties of the FIFO have to be carefully selected. Thereby, the width of the words stored in the FIFOs is strictly coupled to the width of the two shift registers. Since these registers have to be filled simultaneously with data, the bit width of the *output FIFO* is twice the bit width of these registers. In contrast to that, the width of the *input FIFO* is equal to the width of the value register. According to this, the data words read from the memory have also twice the bit width of the words that are written to the memory. The depth of the FIFOs depends on the total number of DPs. Typically, the depth should be kept at a minimum to reduce the hardware requirements. However, if the depth is too small, then the buffers may become empty during the test run, even though they were totally filled at the beginning of the test.

Using these buffers the memory requests of several DPs can be scheduled consecutively in time without sacrificing performance. A DP simply needs to issue a memory read access when the output FIFO is not full. Corresponding to that, a write request can be issued to store comparison results if the input FIFO is not empty.

In order to coordinate these requests, each DP is equipped with a *port controller*. Depending on the states of the FIFOs, it issues respective memory access requests which consist of the following information:

- an indicator for the request type (read or write),
- the address of the data to be read or written, and

- in case of a write request, the data to be written to the memory.

The acknowledgment for such a request is delivered by the MAC. In case of a read request, the acknowledgment is bundled along with the data read from the memory. This is different for a write request. In this case, the acknowledgment is directly asserted when the request is accepted by the MAC. Thus, there is no need to wait until the data reached the memory. In sum, the controller provides the following interface signals to the MAC:

- The output `memreq` encodes whether a memory request is issued and the type of the request.
- The input `memrdack` is the acknowledgment signal indicating the completion of a memory read request.
- The input `memwrack` is the acknowledgment signal indicating the completion of a memory write request.
- The output `memaddr` defines the address of the data to be read from or written to the memory.
- The input `din` is the input for data read from the memory.
- The output `dout` is the output for data to be written to the memory.

Considering these requests, there is one open issue. The addresses of the data have to be determined and managed. According to that, a major task of the *port controller* is the management of these addresses. Therefore, it contains a register for storing the address of the data to read, and one register which stores the memory address for the comparison results. In order to simplify this address management, each DP has an array of read data in the memory that is associated with it. Furthermore, in case a DP is used as an input port, an array of data within the memory is reserved, where the DP can store its comparison results. Thus, when a memory request has been completed, the value of the respective address register has to be simply incremented in order to point to the next data. To ensure that these address registers do not exceed the boundaries of the arrays, the controller has an additional register, i.e., the *data word counter*. This register stores the number of data words associated with the port. It is decremented each time a new data word has been loaded from the memory. Obviously, prior to the test operation all these registers have to be initialized with the start addresses of the respective data arrays and the number of data words. Therefore, the controller uses a unique identification number to determine the addresses of this data within the memory. For example, the

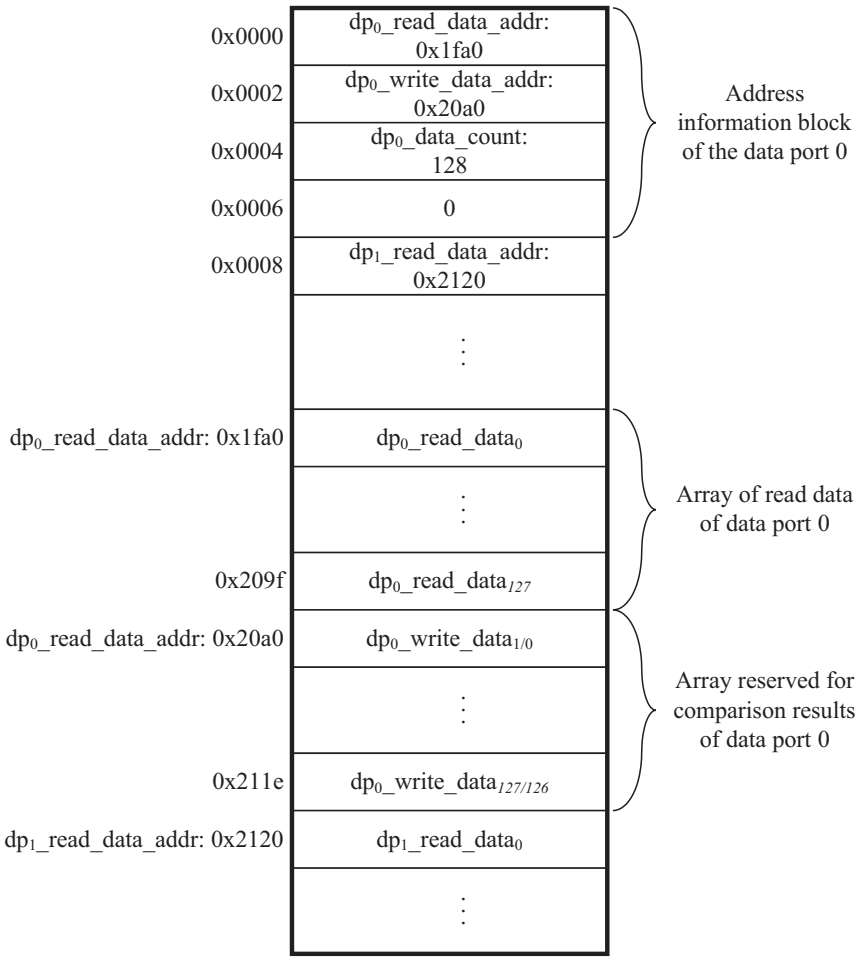


Figure 5.4: Data memory organization of NoTePAD

required information of the DP ϕ_0 (i.e., the ID is 0) is placed within the first three data words of the data memory. Furthermore, to ease the calculations, the address data blocks are placed at every forth data word. The resulting memory organization is exemplarily shown in Figure 5.4.

Another task of the *port controller* is to coordinate the operations of the shift registers and their interactions with the FIFOs. Basically, the registers have to be filled with the data from the output FIFO when the first data shall be applied or compared. Then, these registers have to be shifted for the next $n/k - 1$ test activities of the port. In case a DP is used for an input channel, the data from the value register has to be written to the input FIFO. Finally, when the next test operation is initiated, the shift registers have to be filled again with the data from the output FIFO. This procedure is repeated until all data words have been applied or compared. This is the case if the data word counter became zero.

Finally, the port controller implements a state machine that manages all the activities of the DP and delivers status information to the handshake ports. These activities are triggered by the operations that are defined by the associated HP. Obviously, these operations include the test operations for applying stimuli and for comparing responses of the DUT with expected data. Taking the direction of the port into account, the same opcode can be used for both operations. In addition to these primary operations, a few more are required to configure and control the port. As already pointed out, the address registers have to be initialized prior to the test execution. This requires three operations: set-read-data-address (`SRDA`), set-write-data-address (`SWRA`) and set-data-word-counter (`SDWC`). Furthermore, in order to prevent any data gap during the test, it is beneficial to fill the output FIFOs with data. Similarly, after the test execution, it has to be guaranteed that the data is flushed from the shift registers into the input FIFOs and from there into the memory. Thus, two more operations are required: fill-output (`FILL`) and flush-input (`FLSH`). Additionally, a DP has to be configured whether it is allowed to issue memory access requests. Otherwise, if a DP was not set up correctly, it may access invalid memory addresses. This mechanism also inhibits unused DPs to issue memory requests. Accordingly, a further operation, i.e., the toggle-memory-request (`TMRQ`) operation, is introduced which toggles a specific bit of the *port controller*. This bit indicates whether the port is allowed to issue memory access requests. Finally, an operation (`NOOP`) is required indicating that the port is in idle mode.

In the other direction, each DP has to deliver status information to its associated HP (via the *port switch*). As indicated previously, this especially includes completion information, since the duration of many operations is not fixed. Furthermore, the DP provides information indicating whether the port has data for executing the next data transfer. This information is evaluated by the HP prior to a data transfer.

In summary, the DPs provide the following interface signals to the HPs (via the *port switch*):

- The input `en` indicates whether the port is enabled.
- The input `opcode` determines the mentioned operations: `NOOP`, `SRDA`, `SWRA`, `SDWC`, `TMRQ`, `FILL`, `FLSH`, `TEST`.
- The input `dir` indicates the direction of a port (input, output).
- The output `opok` is the acknowledgment signal indicating the completion of the last operation assigned to the port.

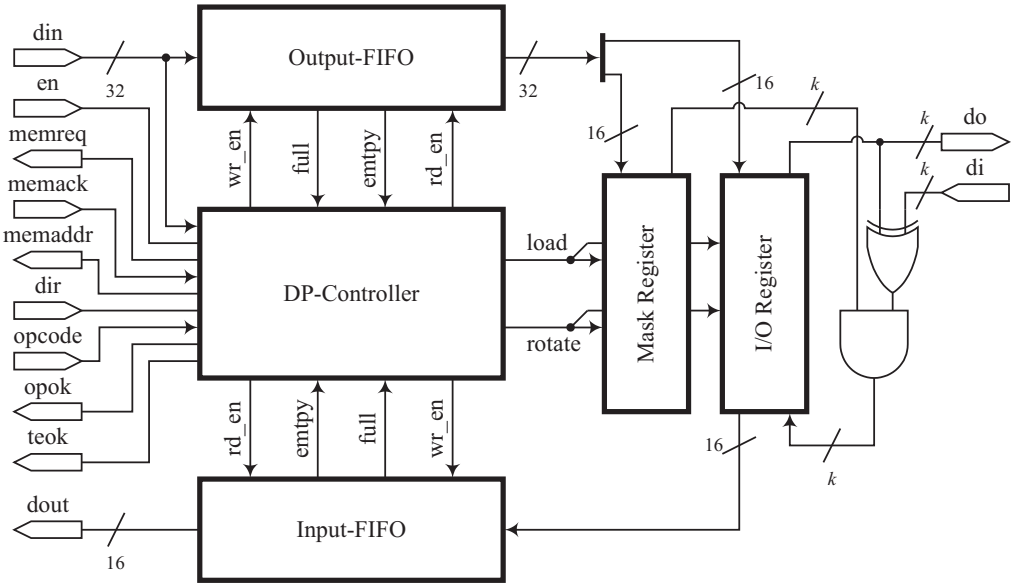


Figure 5.5: Architecture of a data port of NoTePAD

➢ The output `teok` indicates whether the port is able to execute the next test operation.

The final architecture of a DP resulting from these considerations is shown in Figure 5.5. For the implementation, the width of the shift registers was selected to comprise $n = 16$ bits. Consequently, the input FIFO has also a word width of 16 bits. The data words stored within the output FIFO are 32 bits wide, due to the necessity to store the data for the value and mask registers. The number k of inputs and outputs of the DPs is kept configurable in the HDL description of the component. This allows quick adaption of the number of available data I/Os during design time of the test processor if an FPGA is used for the implementation. This number has to be traded against the hardware overhead. Furthermore, one has to take into account that increasing k may lead to squandered I/Os. As afore mentioned, this is the case if the bit width of a channel is not a multiple of k. This also implies that the data belonging to the useless pins is wasted as it cannot be omitted to retain the data block alignment. Thus, increasing the number k of DPs, while keeping the total number of I/Os constant, potentially results in a waste of data memory, but also in a significant reduction of the hardware requirements. On the other hand, decreasing k reduces the waste of memory, but increases the area requirements. An evaluation of suitable configurations is delivered later in Chapter 7. To come right to the point, it has turned out that two or four I/Os per DP are suitable.

5.2.3 Design of the Handshake Ports

The main purpose of the handshake ports is to provide the handshake interface, comprising the input `hi` and the output `ho`, and to coordinate the flow of the selected protocol. However, as opposed to the previous TP implementations [Zeidler 2011, Zeidler 2012a], the HPs autonomously generate the handshake event sequence and fully control their associated DPs. To this end, the HPs implement a state machine that sends instructions to the DPs and generates all the handshake signal transitions depending on the chosen protocol. Accordingly, each HP requires a configuration register storing the protocol information p of the asynchronous channel $C = (H, D, v_0, p)$, the initial values $\texttt{hi}_{init}, \texttt{ho}_{init}$ of the handshake signals defined by the function v_0 and the direction of the port, i.e., input or output.

In order to control the HPs, these are connected to the *sequencer*. As defined in the global architecture, the *sequencer* sends the same opcode to all HPs. Furthermore, it enables the HPs that shall perform the currently selected operation. Similar to the completion indication of the DPs, the HPs acknowledge the execution of an operation via a respective signal. Furthermore, to configure the type of the protocol, the HPs are equipped with an additional configuration data input. Finally, the resulting interface to the *sequencer* comprises the following signals:

- The input `hp_opcode` determines the operation to be performed by the HP.
- The input `hp_en` defines whether the HP is enabled to execute the operation defined by `hp_opcode`.
- The input `hp_config` is used to define the protocol information and the initial values of the handshake interface signals.
- The output `hp_opok` indicates whether the HP has completed the last operation.

Corresponding to the functionality of the HPs, they provide a couple of operations. These include an operation to configure the protocol, i.e., set-asynchronous-handshake-protocol (`SAHP`), and another one to initiate a handshake-based data transfer (`TEST`). In addition, since the DPs are controlled by the HPs, the operations of the DPs have to be made accessible via respective operations of the HPs. Therefore, the HPs provide the operations `INIT`, `TMRQ`, `FILL` and `FLSH`. All of these instructions, except `INIT`, directly forward the according instructions to the associated DPs. The `INIT` instruction encapsulates the three DP operations `SRDA`, `SWRA`, and `SDWC` to setup the address and data count registers. To ensure that all registers have been configured correctly, the HPs wait until all their associated DPs have completed one operation before applying the next opcode.

Additionally, this instruction initializes internal signals of the HP which are needed for the detection and generation of protocol events. As an example, consider a 2-phase protocol. To detect the transition of the input `hi` an internal signal is used that stores the previous value of `hi`. This internal signal has to be initialized to the initial value $\texttt{hi}_{init}$ prior to the start of the test.

To apply the opcode to the DPs and to observe their states, the HPs provide the following interface signals connected with the associated DPs via the *port switch*:

- The output `dps_opc` is the opcode sent to the associated DPs.
- The output `dps_dir` indicates whether the HP and its associate DPs implement an input or an output channel.
- The input status signal `dps_opok` indicates whether all associated DPs have completed the last operation.
- The input status signal `dps_teok` indicates whether all associated DPs can execute the next test operation.

The latter two signals are computed by the *port switch* (see Section 5.2.4). For the sake of simplicity, these signals are combined to `dps_stat`. Similarly, the signal `dps_dir` is treated as part of the opcode.

Finally, the core of the handshake ports is the state machine that coordinates the operations of the HP and its associated DPs. Its most important tasks are the control and the monitoring of the handshake signals according to the selected handshake protocol. For the design of this state machine, separate synchronous automatons were created from the STGs of the supported protocols. Obviously, two automatons had to be created for each protocol: one for sending and one for receiving data. As mentioned, the most important bundled-data protocols are supported. For outgoing 4-phase data transfers, the broad data validity scheme is applied which covers all other schemes. For incoming data transfers, at least the early data validity is assumed which also covers the broad data validity. Finally, to support late data validity for incoming data transfers, an additional automaton has to be created. Furthermore, the information about the scheme has to be added to the protocol. However, since this scheme is rarely used, it is currently not supported by NoTePAD.

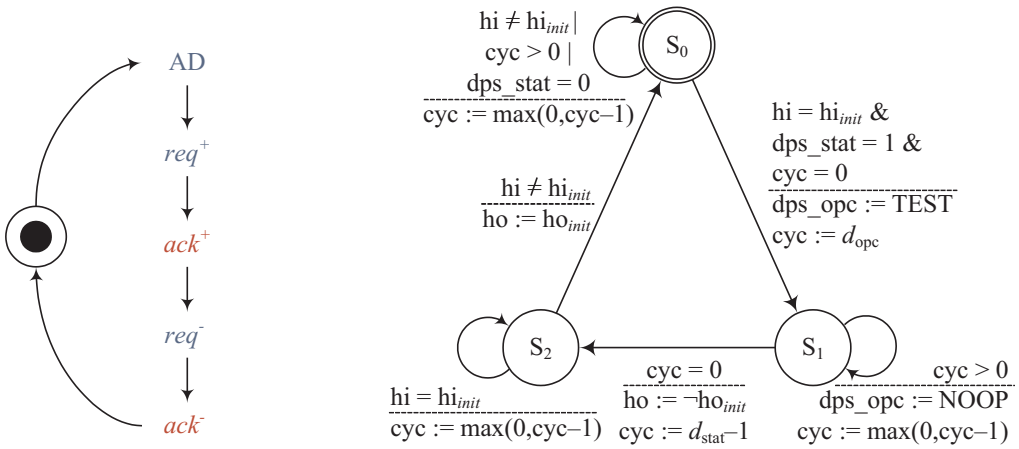
For the correct implementation of the FSMs, the cycle latency $d_{\texttt{opc}}$ required for the propagation of the opcode from the HP to the DPs and the latency $d_{\texttt{stat}}$ of the status signals from the DPs to the HPs have to be taken into account. These delays are, in turn, determined by the implementation of the *port switch*. Accordingly, one has to consider the following constraints:

➢ A handshake signal transition that shall occur after a DP operation has to be delayed by $d_{\texttt{opc}}$ cycles. To ensure that the transition really occurs after the DP operation, a further delay buffer can be added to the output `ho`.

➢ The evaluation of the status signal, indicating whether the next `TEST` operation can be issued, has to be delayed by at least $d_{\texttt{opc}} + d_{\texttt{stat}} + 1$ cycles after assigning the `TEST` operation to the DPs. The additional cycle is introduced by the response latency of the DPs. This ensures that the status signal has propagated to the HP.

Based on these considerations, the FSMs were created according to the respective STGs. The complete set of STGs of the protocols and the according FSMs can be found in Appendix A. However, the Figure 5.6 exemplarily illustrates the STG and the respective Mealy-FSM for sending data according to the 4-phase push protocol. In the STG, the blue transitions designate the transitions which are generated by the TP, whereas the red transitions are generated by the DUT.

For each state transition of the FSM, the significant input combinations and the related outputs are separated by a dashed line. Note that the inputs are basically boolean conditions which may be combined via logical AND (&) and OR (|) operators. Furthermore, one can see a variable `cyc` which denotes a cycle counter. This internal variable of the state machine is used to delay the state transition. In certain states, the HP evaluates the value of this counter and remains in these states as long as the counter is not zero. Thereby, the value assigned to the counter is determined by the constraints defined above. In the current implementation of the *port switch*, both $d_{\texttt{opc}}$ and $d_{\texttt{stat}}$ are one



(a) STG (b) FSM

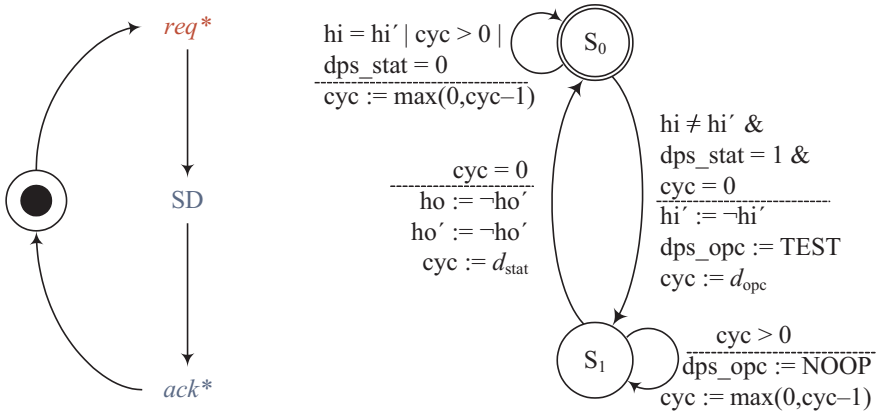
Figure 5.6: Automaton for the 4-phase push protocol from the sender point of view

cycle. Therefore, it is not necessary in some cases to delay the state transition. However, the FSMs show the values depending on the generic latency variables $d_{\texttt{opc}}$ and $d_{\texttt{stat}}$ for the sake of generality. Finally, a handshake is completed when the initial state is reached again. Thus, the completion of a handshake is reported to the *sequencer* when traversing a transition that leads to the initial state.

The STG shown in Figure 5.6 starts with the transition `AD`. This designates that valid data has to be applied by the TP. After that the TP can issue a request which is acknowledged by the DUT. Then, the request is reset by the TP. Finally, the DUT resets the acknowledgment as well.

In the respective FSM, the next valid data is applied via assigning the opcode `TEST` for the associated DPs. This is only allowed if all DPs are able to perform the next test operation (i.e., `dps_stat` = 1) and the input handshake signal `hi` is at its initial value. Then, the HP waits until the opcode reaches the DPs. Since all associated DPs were able to perform the test operation, it is not necessary in this state to wait for delivery of the status signal of the DPs. After $d_{\texttt{opc}}$ cycles the request is issued by setting the output handshake signal `ho` to the inverse of its initial value stored in $\texttt{ho}_{init}$. Finally, the HP waits for the acknowledgment of the DUT at the input signal `hi` before it resets the output `ho` to the initial value. However, before the next handshake can be performed, it has to be ensured that the status signal from the DPs has propagated to the HP. Thus, after the `TEST` operation has been executed by the DPs, at least $d_{\texttt{stat}}$ cycles have to go by. Therefore, it might be necessary to delay the execution of the next handshake.

Similarly, the Figure 5.7 shows the STG and the FSM for receiving data according to the 2-phase push protocol. In this case, the protocol events are represented by signal



(a) STG (b) FSM

Figure 5.7: Automaton for the 2-phase push protocol from the receiver point of view

transitions. Since there is no distinction between the rising and falling signal transitions, the transitions are labelled with asterisks. According to the protocol, the TP has to wait for the request of the DUT before it can strobe the data. In the STG, the strobe of data is indicated by the transition SD. After the receipt of the data, the TP sets the acknowledgment.

To realize the transition signaling in the synchronous FSM, two further internal variables hi' and ho' are introduced. These variables store the previous values of the handshake signals. During the initialization of the handshake ports, these signals are set to the initial values $\texttt{hi}_{init}$ and $\texttt{ho}_{init}$. If a transition is detected or generated, then the respective signal (hi' or ho') is inverted. Accordingly, the HP first waits for a transition of the input signal hi. This is the case if hi is unequal to hi'. If the transition is detected, the HP forces the DPs to strobe the data via assigning the opcode TEST. Again, the HP has to wait until the opcode has reached the DPs. Then, it sets the acknowledgment by generating a transition of the output signal ho. Finally, the HP has to wait for $d_{\texttt{stat}}$ cycles in order to fulfil the second constraint.

Based on the implementation of the FSMs, the number of cycles required for the execution of the handshakes can be analyzed. The FSMs are designed such that after assigning the TEST operation to the DP (which takes one cycle), the next handshake can be initiated at the earliest after $d_{\texttt{opc}} + d_{\texttt{stat}} + 1$ cycles. However, this is typically not the limiting factor. Instead, one has to consider the number of cycles consumed by the individual operations: As indicated, one cycle is required for setting the opcode for the DP. After that, the next handshake signal transition can be generated by the TP at the earliest after $d_{\texttt{opc}}$ cycles. The generation of the transition also consumes one cycle. Then, the transition on the handshake port input hi has to be detected, which in the best case requires two cycles. Thus, under the assumption that $d_{\texttt{opc}} = d_{\texttt{stat}} = 1$, five cycles are at least required for a 2-phase handshake. In case of 4-phase protocols, three further cycles are required: one for resetting the handshake signal and two further cycles for detecting the second transition of hi. Note that the order of these operations may vary in the different FSMs implementations, but not their presence.

5.2.4 Design of the Port Switch

The *port switch* is the central connection point between the handshake and the data ports. It stores the configuration that describes the association between these ports. This association comprises two functions A_{DP} and A_{HP}. $A_{\text{DP}} : \Omega \rightarrow \Phi \cup \{\emptyset\}$ maps the DPs $\Omega = \{\omega_0, \ldots, \omega_{l-1}\}$ to their associated handshake port. Since not all DPs might be used in a test program, A_{DP} might map some of the DP to $\emptyset$. In the opposite direction,

$A_{\mathrm{HP}} : \Phi \rightarrow \Omega^*$ (where Ω^* designates the power set of Ω) is the pseudo-inverse function to A_{DP}. This function maps the handshake ports $\Phi = \{\phi_0, \ldots, \phi_{m-1}\}$ to a set of their associated DPs. Again, not all HPs might be used in a configuration of a test.

To implement these mappings, the *port switch* is equipped with two register banks storing two different matrices M_{DP} and M_{HP} as shown in Equation 5.1.

$$M_{\mathrm{DP}} = \begin{pmatrix} s_0 \\ \vdots \\ s_{l-1} \end{pmatrix} \qquad M_{\mathrm{HP}} = \begin{pmatrix} a_{0,0} & \ldots & a_{0,l-1} \\ \vdots & \ddots & \vdots \\ a_{m-1,0} & \ldots & a_{m-1,l-1} \end{pmatrix} \tag{5.1}$$

To realize A_{DP}, the matrix M_{DP} stores the ID $s_i = \mathrm{id}(\phi_j) = j,\ i \in \{0, \ldots l-1\}$ of the associated HP $\phi_j = A_{\mathrm{DP}}(\omega_i)$ for each DP $\omega_i \in \Omega$. In addition to that, it is required to indicate whether a DP is mapped to an HP or to $\emptyset$. The latter case indicates that the certain DP is not used for the implementation of a channel. Therefore, one further register stores a bit mask $e = e_0 \ldots e_{l-1}$ such that $e_i = 1$ if $A_{\mathrm{DP}}(\omega_i) \neq \emptyset$. Otherwise, e_i is set to 0 which indicates that the port is not activated at all during the execution of a specific test. In order to implement A_{HP}, the matrix M_{HP} stores one bit vector $a_j = a_{j,0} \ldots a_{j,l-1}$ for each HP $\phi_j \in \Phi$. The bits of this vector designate which of the DPs are associated with the HP ϕ_j. Thus, if $a_{j,t} = 1$, then DP ω_t is associated with HP ϕ_j.

Depending on these configuration matrices, the *port switch* forwards the operation code from the handshake ports to the data ports and computes the status information that is delivered by the DPs to be sent to the HPs. To obtain the status information, the status signals `opok` and `teok` from the DPs are combined in the following way: Let $\mathtt{opok}_0, \ldots, \mathtt{opok}_{l-1}$ be the set of the signal `opok` of all DPs and $\mathtt{teok}_0, \ldots, \mathtt{teok}_{l-1}$ the set of the signal `teok`. Furthermore, let $\mathtt{dps\_opok}_j$ be the status signal for a HP ϕ_j indicating that all its associated DPs have completed their last operation. Similarly, $\mathtt{dps\_teok}_j$ designates the status signal for ϕ_j indicating that all its associated data ports are able to perform the next data transfer. Then, $\mathtt{dps\_opok}_j$ and $\mathtt{dps\_teok}_j$ can be derived from Equations 5.2 and 5.3.

$$\mathtt{dps\_opok}_j = (\mathtt{opok}_0 \vee \overline{a_{j,0}}) \wedge \ldots \wedge (\mathtt{opok}_{l-1} \vee \overline{a_{j,l-1}}) \tag{5.2}$$

$$\mathtt{dps\_teok}_j = (\mathtt{teok}_0 \vee \overline{a_{j,0}}) \wedge \ldots \wedge (\mathtt{teok}_{l-1} \vee \overline{a_{j,l-1}}) \tag{5.3}$$

Each of these resulting signals are fed to the respective handshake port. Accordingly, the *port switch* provides the following interface signals to the HPs:

- The inputs $\mathtt{opc\_to\_dp}_0, \ldots, \mathtt{opc\_to\_dp}_{m-1}$ determine the opcodes provided by the HPs to their associated DPs.

- The outputs $\texttt{dps\_opok}_0, \ldots, \texttt{dps\_opok}_{m-1}$ are the combinations of the status signals $\texttt{opok}_0, \ldots, \texttt{opok}_{k-1}$ of the DPs indicating that all associated DPs have completed their last operation.
- The outputs $\texttt{dps\_teok}_0, \ldots, \texttt{dps\_teok}_{m-1}$ are the combinations of the status signals $\texttt{teok}_0, \ldots, \texttt{teok}_{k-1}$ of the DPs indicating that all DPs can perform the next data transfer.

The interface to the DPs comprises the following signals:

- The output signals $\texttt{dp\_opc}_0, \ldots, \texttt{dp\_opc}_{l-1}$ are the opcodes for each DP delivered by the HPs.
- The input signals $\texttt{dp\_opok}_0, \ldots, \texttt{dp\_opok}_{l-1}$ and $\texttt{dp\_teok}_0, \ldots, \texttt{dp\_teok}_{l-1}$ are the status signals of the DPs.
- The output signals $\texttt{dp\_en}_0, \ldots, \texttt{dp\_en}_{l-1}$ are the enable signals $e_0 \ldots e_{l-1}$ for the DPs.

In order to set and reset the configuration, the switch provides two corresponding operations `PS_SET` and `PS_RST`, respectively. A further operation `PS_NOP` retains the values of the configuration matrices. To define the desired operation and the configuration of the matrices, the *port switch* has the following interface to the *sequencer*:

- The input `ps_opcode` defines the desired operation of the port.
- The input `hp_id` defines the ID of the handshake port if the `PS_SET` is executed.
- The input `dp_id` specifies the ID of the data port if the `PS_SET` is executed.

The latter two interface signals `hp_id` and `dp_id` are combined to `ps_config`. To define a port association, the opcode `PS_SET` and the IDs of the ports have to be provided. Then, the *port switch* sets the entries of the matrices as well as the enable signal e_i for the defined port ω_i.

The resulting architecture of the *port switch* is illustrated by the schematic shown in Figure 5.8. For the sake of simplicity, the signals $\texttt{dp\_opok}_i$ and $\texttt{dp\_teok}_i$ are combined to $\texttt{dp\_stat}_i$. As shown in the figure, the *port switch* comprises l multiplexers whose outputs depend on the register bank storing the matrix M_{DP}. Each of these multiplexers selects the opcode for one of the l DPs from the HPs. Furthermore, the block `DP Enable Register` stores the enable signals $e = e_0 \ldots e_{l-1}$ for each DP. For the opposite direction, the *port switch* comprises logic to generate the status signals for each of

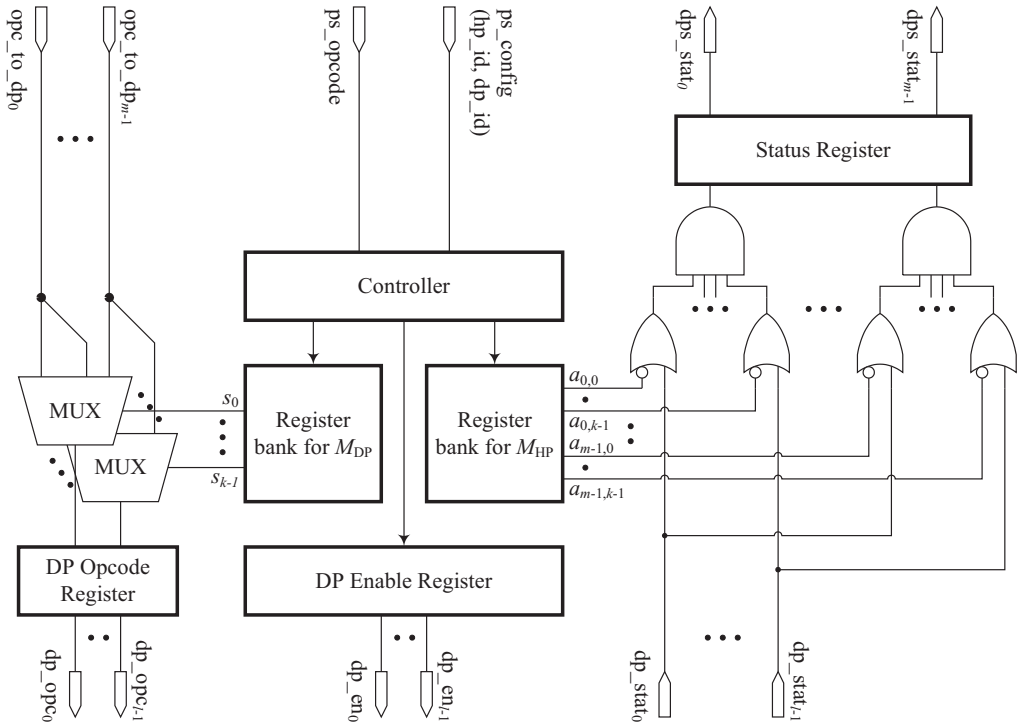


Figure 5.8: Port switch of NoTePAD

the HPs with regard to the matrix M_{HP}. Finally, the purpose of the `Controller` block is to coordinate the operations of the *port switch* and the control of the register banks.

To prevent timing problems all outputs of the *port switch* are registered. Obviously, this results in the afore mentioned latencies $d_{\mathtt{opc}} = d_{\mathtt{stat}} = 1$. However, in case of a large number of DPs, the computation of the status signals becomes a bottleneck due to the combinational path required for combining possibly several dozens of status signals from the DP. In this case, one can add a pipelined computation in order to achieve high clock frequencies. Of course, this increases $d_{\mathtt{stat}}$ which has to be considered in the handshake port implementation. A different approach could be that only subsets of the DP can be associated with the individual HP. This decreases the number of inputs of the combinational logic tree required for the status signal computation. Hence, the path delay is reduced. However, this also decreases the flexibility to combine an arbitrary subset of the data ports with one handshake port.

5.2.5 Architecture of the Memory Access Controller

The MAC connects all DPs with the data memory and coordinates the memory accesses. Therefore, it arbitrates between incoming read and write access requests from the DPs.

In addition to that, it also provides an interface for the *sequencer* by which data can be up- and downloaded before and after the test execution, respectively. Furthermore, this interface can be used by the *sequencer* to load and store data during test execution. In general, this allows the *sequencer* to implement a stack for executing complex programs including subroutine calls. However, typically the programs are almost completely sequential, except of special situations, e.g., a deadlock. The very few subroutine calls can also be implemented without a stack, since it is not intended that the depth of the call tree exceeds one subroutine call. Therefore, memory accesses during the test execution are very rare.

The major challenge of the implementation of the MAC is the arbitration of a high number of concurrently incoming requests. Apparently, in order to prevent deadlocks or throughput bottlenecks, all access requests from the DPs are considered to have the same priority. The only component that can issue a request with a higher priority is the *sequencer*. Even though memory accesses of this unit are very rare, a low memory response latency is required for this central control unit to ensure fast execution of the program.

To guarantee that all requests from the DPs are processed, the MAC implements a round-robin schedule mechanism. Therefore, the MAC scans its interfaces to the DPs for pending requests in a cyclic manner. The requests are processed one after the other and cycle by cycle. If a request of one DP is processed, then the next request of this DP is processed at earliest after all pending requests of other DPs. To ensure that a DP issues a new request only if no other request of this port is pending, the interfaces of the DPs and the MAC implement a synchronous handshake protocol. Thereby, the MAC has to distinguish between read and write request. Obviously, to complete a read request, a DP has to wait for the response of the memory. This is different for a write request, where the data to be written is transmitted along with the request itself. Thus, a write request is fully processed when it is accepted by the MAC. With respect to the data transmission, one further point to mention is the width of the data words written to and read from the memory, respectively. Due to the architectural concept of the DPs, the data written to the memory comprises $n = 16$ bits, whereas each data word read from the data memory is $2n = 32$ bits wide.

One essential challenge of the implementation of the MAC is the arbitration. In general, the arbitration between potentially hundreds of requests in one clock cycle would result in complex combinational circuitry and a significant reduction of the maximum clock frequency of the system. Therefore, the entire arbitration process is disassembled according to a divide-and-conquer approach. For this, the MAC is decomposed into a set of memory access arbiter (MAA) that are connected in a tree-like architecture as shown

in Figure 5.9. These arbiters are constructed in such a way that they are able to forward one of $r > 1$ incoming requests within one clock cycle. Thereby, the number r is kept generic which allows an adaption to the number of DPs during design phase of the processor. Obviously, increasing r reduces the arbiter stages and, therefore, the response latency for a read request as well. However, there is one limiting factor that strongly restricts r, i.e., the clock frequency. This is due to the complexity of the resulting combinational logic implementing the arbitration scheme. It exponentially increases with the increase of r and, therefore, limits the maximum operating frequency of the MAC. During the design phase, it has turned out that $r = 4$ request interfaces are suitable. This number results in an implementation of the MAC whose maximum operating frequency is well balanced with the other components of the processor.

Following the structure shown in Figure 5.9, the set of l DPs can be interconnected with the memory by using $\sum_{i=0}^{\widetilde{\log_r}(l)-1} r^i$ arbiters, where $\widetilde{\log_r}(x)$ delivers the smallest integer number y such that $y \geq \log_r(x)$. Apparently, for a well-balanced MAC l has to be a power of r. Thus, for the considered implementation of the arbiter with four incoming request interfaces $16, 64, 256, \ldots$ DPs are suitable. The resulting depth of the arbiter tree[1] is equal to $log_r(l)$. This number strictly affects the minimum latency in cycles required for accessing the memory from a DP. In the real implementation the latency is even larger, since additional cycles might be required for accessing the memory.

For the connection with either the DPs or the arbiters of a lower stage, each arbiter provides r input request interfaces. On the other side, each MAA provides one output request interface that is connected to the arbiter of a higher stage. These interfaces comprise the signals given in the following list. Thereby, the direction of the signals is given for the input and for the output interface separated by the slash.

- The input/output `req` indicates the presence of a memory request and its type.
- The input/output `data` is the data to be written to the memory.
- The input/output `addr` is the memory address of the data.
- The output/input `wrack` is the acknowledgment signal indicating the completion of a write access to the memory.

The only arbiter that differs from the others is the root arbiter connected to the memory. Besides the usual request interfaces for the MAA of the lower stage, it provides an additional interface for the *sequencer*. A memory access request at this interface is treated to have the highest priority. Therefore, these requests are processed immediately.

[1] The depth determines the number of arbiters on the path from the DPs to the memory.

Since data memory accesses from the *sequencer* are very rare, as will be shown later, this is a suitable solution. Furthermore, since the memory is expected to accept one read or write access per clock cycle, there is no acknowledgment signal at the output interface of this arbiter. This simplifies the design of the root arbiter.

In addition to the above mentioned interface signals, all MAAs, including the root arbiter, provide one read acknowledgment signal `rdack` for each input request port. However, instead of feeding these signals back to the lower arbiter stage (or to the DP) these signals are connected to some additional logic shown in the lower part of Figure 5.9. This circuitry generates the read access acknowledgments $\texttt{dp\_rd\_ack}_0, \ldots, \texttt{dp\_rd\_ack}_{l-1}$ for the DPs.

The purpose of this circuitry is to prevent that the read access acknowledgment needs to propagate the entire path back from the memory to the DP. Such a behavior would block all the arbiters on the path. Furthermore, the acknowledgment would require twice the number of cycles compared to the depth of the arbiter tree. Therefore, the read acknowledgment signals of all arbiters on the path from a DP to the memory are combined via AND-gates. Thereby, the registers store the combination of the read acknowledgments for each DP for the different $log_r(l)$ stages. Finally, the value of the read acknowledgment $\texttt{dp\_rd\_ack}_i(t)$ for the DP ω_i in cycle t is determined by the formula shown in Equation 5.4.

$$\texttt{dp\_rd\_ack}_i(t) = \bigwedge_{j=0}^{\log_r(l)-1} \texttt{rdack}_{j,i/r^j}(t-j-1) \tag{5.4}$$

The bus for the data read from the memory is directly connected to the respective ports of the DPs and the *sequencer*, respectively. For this reason, these signals are not shown in the figure. However, in practice this direct interconnection may impose considerable routing effort. In an FPGA implementation, each of the 32 data bus bits has to be fanned out in order to connect them to the inputs signals of potentially hundreds of DPs. In case of implementing the TP as an IC, a bus system can be used to decrease this routing effort.

5.2.6 Architecture of the Sequencer

The *sequencer* is the central control unit that coordinates the activities of the handshake ports and the *port switch*. It manages the flow of operation of the entire test system. One of its basic tasks is to detect deadlocks during the execution of asynchronous data transfers. The *sequencer* fetches the instructions from the program memory, decodes them, and controls and observes the HPs and the *port switch*. In addition to that, it also

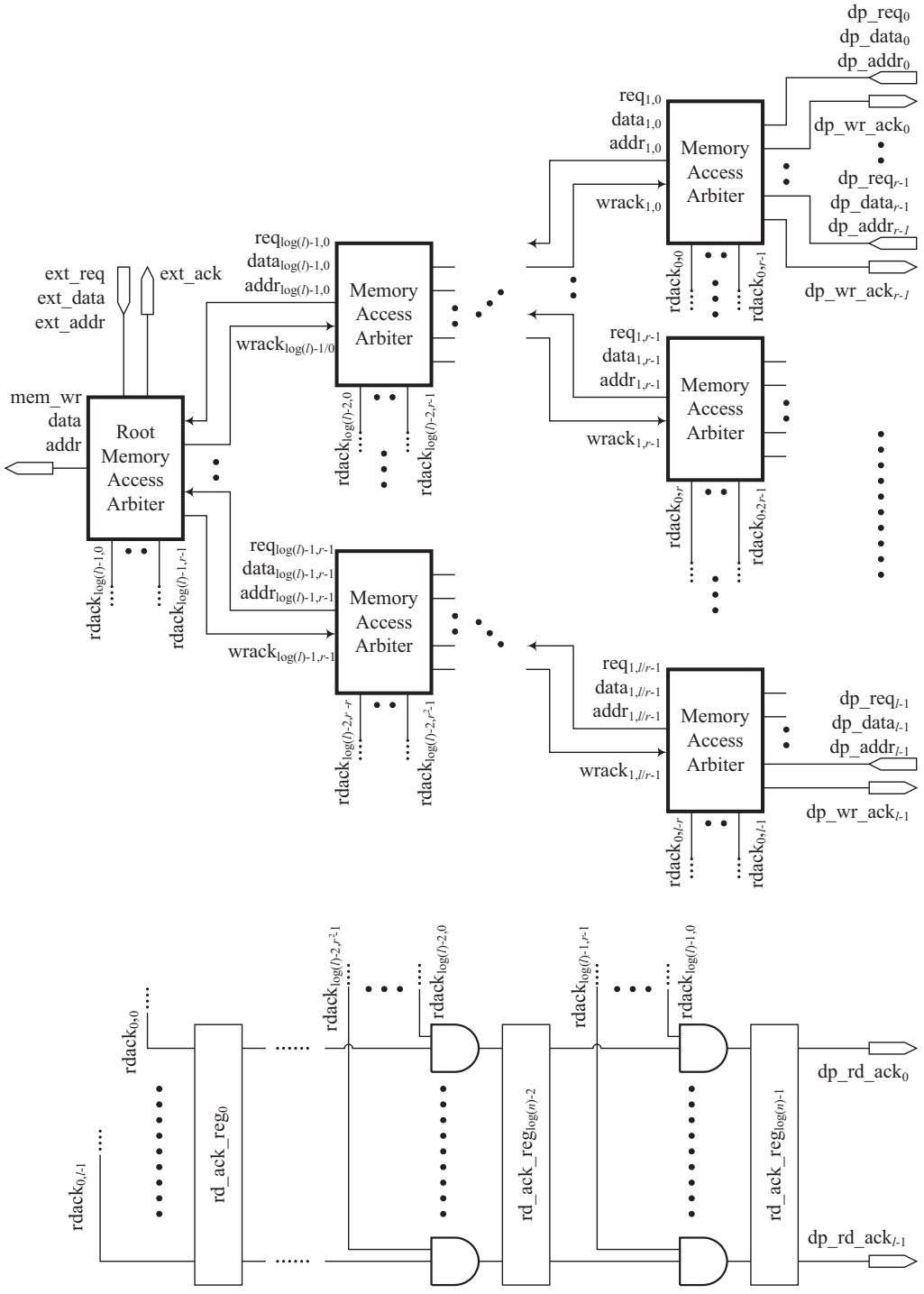


Figure 5.9: Memory access controller of NoTePAD

has access to the data memory via the afore mentioned interface of the MAC. Therefore, the *sequencer* is the central unit that provides access to both, the program and the data memory for the ETE via the external interface described in Section 4.2.2.2. Furthermore, the *sequencer* implements the interface to special control signals of the DUT, e.g., used to connect the reset signal.

Apparently, the *sequencer* is a specialized microprocessor. Its design is mainly driven by the required operations to control the components and to execute a program. The required functionality mainly includes instructions to embed the operations of the HPs and the *port switch*, as well as additional control flow instructions, and memory access and register manipulation instructions. The complete overview of these instructions is given later in the Section 5.3.

One essential point resulting from the instruction set is the width of the instructions. Obviously, the width of the instructions is determined by the number of instructions and respective data to pass to them. To anticipate, the processor supports 22 instructions. Thus, at least 5 bits are required to encode the operations. As shown later, the bit width of the data passed to the instructions is determined by the number of handshake ports and control signals. Finally, the instructions comprise 24 bits: 8 bits are used for the opcode and the remaining 16 for encoding the data belonging to these instructions. Of course, 8 bits for the opcode are much more than actually required, but this leaves room for further extensions.

In consideration of the complexity of the required operations, the *sequencer* has been decomposed into a pipeline of two stages in order to fit the clock frequency of the other components. The first one fetches the instruction from the memory. Therefore, it manages the program counter and the branch addresses delivered by the second stage. Furthermore, in order to program the processor prior to the test execution, this stage is connected to the interface of the ETE. Thus, when the program shall be downloaded, the processor is in an inactive state and directly forwards the data from the ETE interface to the memory.

The second stage decodes and executes the instructions. Thus, it is this block which controls and monitors the other components. The decode stage has a register file comprising eight 32-bit registers. Some of these registers are reserved for special purposes while others can be used for general purposes. Since many operations of the controlled components may require several cycles for completion, this stage needs the possibility to pause the execution of the program. Therefore, it provides a control signal to the fetch stage indicating whether to stall the pipeline. For example, in case of performing asynchronous data transfers, the *sequencer* stalls the pipeline, sets the operation code for the HPs, and selects the HP to be activated. Afterwards, the *sequencer* has to wait until all

activated HPs have performed the transfer and reported the completion at the respective status signal hp_opok.

However, stalling a pipeline may result in a deadlock of the entire program if at least one HP is not able to complete its operation. Therefore, two registers of the register file are used to implement a timeout counter scheme. One of these registers is used to define the upper bound of cycles to wait. The second register stores the current number of cycles exceeded since the start of the last operation. If the value of this register exceeds the value of the timeout register, then the execution of the current operation is interrupted. In order to enable adequate reactions to such unexpected behavior, the processor branches to a program address specified by a third register. However, information is required to identify the cause of the deadlock. Here, two things are of importance: the current program address and the information which of the ports did not complete the last operation. Thus, the value of the status signals hp_opok of the HPs have to be stored. Accordingly, two further registers are used to store the respective information. The first one simply stores the current program address and the second one stores a mask describing the values of the status signals. If a bit $b_i, i \in \{0, \ldots, m-1\}$ of this mask is logical-0, then the HP $\phi_i \in \Phi$ has not completed its last operation.

Based on these considerations, the register file comprises the following registers:

- CYCLCNT (r0) — is the accumulator for addition operations that is typically used for counting cycles and memory address calculations.
- TIMEOUT (r1) — timeout register storing the maximum number of cycles allowed for any kind of operations whose timing is not defined.
- JMPADDR (r2) — register storing the address to jump to in case of a timeout.
- RETADDR (r3) — register that stores the return address in case of any branch.
- MASKREG (r4) — register that receives the mask of status signals.
- TMPREG1 (r5) — first register for temporary data
- TMPREG2 (r6) — second register for temporary data
- TMPREG3/STACKPT (r7) — third register for temporary data that can be used by convention as stack pointer register.

The decode stage also implements the port comprising the control in- and outputs. This port simply comprises a set of inputs $\mathtt{ci}_0 \ldots \mathtt{ci}_{c-1}$ and a set of outputs $\mathtt{co}_0 \ldots \mathtt{co}_{c-1}$ that can be used in parallel. To prevent metastability, the inputs have to be migrated into

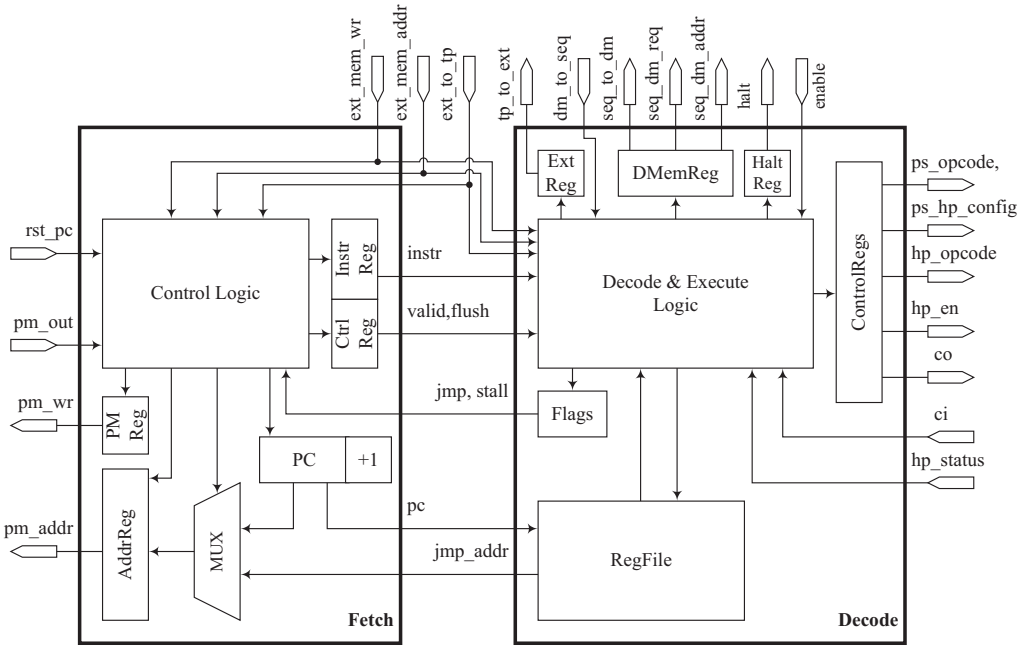


Figure 5.10: Sequencer component of NoTePAD

the clock domain of the processor, e.g., using synchronizers. Finally, the decode stage is also connected to the ETE interface allowing the patterns to be downloaded to the data memory.

In summary, the architecture of the *sequencer* is shown in Figure 5.10. The figure illustrates all the interface signals and the main components of the *sequencer*.

5.3 Instruction Set

The instruction set of NoTePAD covers operations to setup the processor, to execute the *test sequence* and to finally upload the test results. A majority of the required instructions is directly derived from the operations of the individual components. Thus, in order to control the *port switch*, the handshake and the data ports, the *sequencer* has to provide instructions encapsulating the operations of these components. Moreover, instructions are required that control the flow of operation during test execution. According to the demands of the *transfer protocol*, this includes instructions to wait for a specific time, e.g., to realize a time window between setting and releasing the reset signal of the DUT. Also, branch instructions and other control flow operations are required, e.g., for the synchronization with the ETE. Furthermore, instructions are needed that provide access to the registers of the *sequencer* and to the data memory.

According to their functionality, the instruction set of NoTePAD can be divided into several groups:

- control flow instructions
- port switch instructions
- handshake and data port instructions
- register manipulation instructions
- memory access instructions

The simplest instruction of the group of control flow instructions is the idle operation that can be used to wait for one test processor cycle before the next operation is executed. The next one is the wait instruction. This instruction stalls the pipeline, resets the cycle counter and afterwards increments this register each clock cycle until it reaches the value of the timeout register. Then, the stall signal is released and the processor continues the program execution.

Further required control flow instructions are branch operations. Typically, common microprocessors support various types of branch instructions, such as conditional and unconditional branches. However, since NoTePAD is intended to execute sequential programs with only a very few number of branches, it provides only one unconditional branch instruction. The branch destination address for this instruction has to be stored in the `JMPADDR` register. Furthermore, the instruction simultaneously stores the return address (i.e., the value of the program counter) in the `RETADDR` register and sets the program counter to the branch destination address. This is required to implement simple subroutines, e.g., a handler function for unexpected behavior of the DUT.

The last control flow instruction is required for the synchronization mechanism of the processor with the ETE. As defined in the concept (cf. Section 4.2.2.2), this operation is used to indicate that a specific stage of the program was reached. For example, the processor can indicate the completion of the *test sequence* and that the test result are ready for upload. Therefore, the processor provides the `halt` instruction. This instruction sets the ETE interface signal `halt` and stalls the pipeline. This state is kept until the `enable` control signal has been asserted by the ETE.

In sum, the control flow operations include the following instructions:

- `noop` — is the idle operation.
- `wait` — stalls the pipeline and waits for the number of cycles defined by the timeout register.

- `bral` — stores the return address in the `RETADDR` register and sets the program counter to the branch destination address specified by the `JMPADDR` register.

- `halt` — sets the `halt` signal of the processor and stalls the pipeline. The processor can be enabled to continue the operation by setting the control signal `enable` to logical-1.

The *port switch* instructions directly implement the two operations provided by this component. The first one is used to reset the entire port switch configuration and, therefore, deletes the port associations. The second instruction is used to define the association of one HP with a DP. For this, the IDs of the ports have to be passed to the instruction as immediate values[2]. Consequently, the port switch operations are the following:

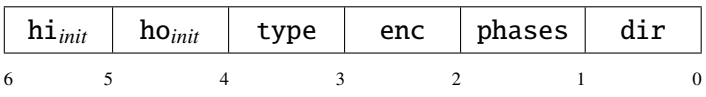
- `rsio` — resets the configuration of the *port switch*

- `conf hp=`*`imm6`*`, dp=`*`imm10`* — sets the configuration such that the DP defined by *imm10* is associated with the HP defined by *imm6*.

The largest set of instructions is dedicated to setup and control the data and handshake ports. Thus, these instructions directly implement the operations provided by the HPs. All of these instructions either provide a field specifying a mask or an ID in order to select the active ports of this instruction. Thereby, each bit h_i of the mask $h_{m-1} \ldots h_0$ represents the enable signal $\texttt{hp\_en}_i$ for one of the m handshake ports.

Furthermore, this group also includes two operations accessing the control in- and outputs. One of these instruction sets the control outputs and the other one is used to compare the input signals. Thereby, the comparison instruction is organized such that it stalls the pipeline of the processor until the input control signals are equal to an expected value passed to the instruction. In order to prevent deadlocks, this instruction also makes use of the timeout scheme. Both instructions provide a field for specifying a mask that determines which of the signals shall be set or compared, respectively. To summarize, the port instructions are the following:

- `sahp hp=`*`imm6`*`, protocol=`*`imm6`* — sets the handshake protocol configuration of the port with the specified ID. The protocol configuration is a six bit mask comprising the following bits:



[2]Immediate values represent numerical values in an instruction.

where

- $\texttt{hi}_{init}$ and $\texttt{ho}_{init}$ are the initial values of the handshake signals defined by the function v_0 of a channel,
- `type` defines the initiator type: push (`type` = 0) or pull (`type` = 1),
- `enc` indicates the encoding style: currently only single-rail (`enc` = 0) is supported,
- `dir` denotes the direction of the channel: input channel (`dir` = 0) or output channel (`dir` = 1)

➢ `init hpmask=`*imm16* — induces the handshake ports selected by the mask to consecutively activate the `SRDA`, `SWRA`, `SDWC` operations of their associated DPs. Thus, after executing this instruction all the DPs that are associated with the enabled handshake ports are fully configured to read and write data from/to the memory.

➢ `tmrq hpmask=`*imm16* — forces the selected HPs to allow or to disallow all their associated data ports to issue memory access request. This instruction has to be called after setting the *port switch* configuration and prior to the test execution in order to specify the active data ports.

➢ `fill hpmask=`*imm16* — induces the selected HPs to activate the `FILL` operation of the associated DPs. After the execution of this instruction the output buffers of all DPs associated with one of the selected HPs are filled with test patterns. This instruction can be used prior to a test to prevent bottlenecks during test execution due to temporarily unavailable data.

➢ `flsh hpmask=`*imm16* — induces the selected handshake ports to activate the `FLSH` operation of the associated DPs. This operation has to be executed after the *test sequence*. It guarantees that the input data (response/fault signatures) of all DPs associated with one of the selected HPs are written to the memory.

➢ `test hpmask=`*imm16* — forces the handshake ports selected by the mask (and their associated DPs) to perform one transfer. The instruction stalls the pipeline until either all selected HPs have completed the transfer or a timeout is detected. In the latter case, the processor branches to the address defined by the `JMPADDR` register.

➢ `ctri mask=`*imm8*`, value=`*imm8* — performs a bitwise comparison of the value with the control inputs specified by the mask. The instruction also stalls the

pipeline until either all selected control inputs are equal to the expected value or the timeout is detected.

➢ `ctro mask=`*`imm8`*`, value=`*`imm8`* — sets the selected output control signals to the specified value.

In order to manipulate the values of the registers, the processor provides three instructions: one instruction moves the value from one to another register, one instruction performs a simple addition and two further instructions set the value of a register. Thereby, one of the latter two instructions sets the lower 16-bits and the other one sets the upper bits of a register. These instructions comprise a field for specifying a 16-bit immediate value.

➢ `move` *`rd`*`,` *`rx`* — writes the value of register *`rx`* to register *`rd`*.

➢ `addu` *`imm16`* — adds the 16-bit immediate value *`imm16`* to the accumulator register.

➢ `setl/seth` *`rd`*`,` *`imm16`* — set the lower/upper 16-bit of register *`rd`* to the value of the 16-bit immediate value.

Finally, the last group of instructions realizes memory accesses functions. Three instructions implement data exchange operations between the registers and the memory. One aspect resulting from the architecture of NoTePAD is the width of the data of the memory interface. The data written to the memory has half the width of the data read from memory. Therefore, it requires two instructions for writing a 32-bit data word into the memory. One instruction stores the lower half and one the upper half of the bits of a register within the memory. In the other direction, only one instruction is required to read a 32-bit data word from the memory.

In addition to these basic memory access instructions, another instruction is used to transfer the content of the data memory to the ETE. In order to simplify the upload procedure, the instruction is designed such that an entire data array can be uploaded. Therefore, the start and the end addresses of the array have to be specified. Afterwards, the instruction can be called which reads the data words cycle by cycle from the memory and forwards them to the ETE interface. In order to compute the memory addresses, this instruction utilizes the adder required for the timeout scheme. This reduces the amount of arithmetical and multiplexing logic within the *sequencer*. As a consequence, prior to the execution of the upload instruction, the start and end address have to be stored in the cycle counter (`CYCCNT`) and the timeout (`TIMEOUT`) register, respectively.

Finally, the memory access instructions are the following:

- `slwd/shwd` *rx*, *ry* — stores the lower/upper 16-bits of the register *rx* at the memory address specified by register *ry*.
- `ldwd` *rd*, *rx* — reads a 32-bit data word from the memory address specified by register *rx* and writes the data into register *rd*.
- `dout` — loads the data at the address of the timeout counter register and forwards this data to the external interface. The timeout counter and the timeout boundary register have to be loaded with the start and the end address of the data array to be transferred. The instruction stalls the pipeline until the counter value exceeds the boundary register.

5.4 Tools Related to the Processor

Obviously, in order to create executable programs for the desired architecture, processor related tools, such as an assembler and a linker, are required. Therefore, an additional LISA model of the *sequencer* has been created besides the RTL-implementation of NoTePAD. This model includes all resources and instructions of the *sequencer* mentioned in the previous sections. Based on this LISA-model in combination with the tool suite of the SYNOPSYS PROCESSOR DESIGNER, an assembler and linker have been created that allow the generation of binary programs for the *sequencer*.

"Things should be made as simple
as possible, but not any simpler."

— Albert Einstein

Chapter 6

Test Program Generation

In this chapter, an implementation of the test program generation flow is discussed. An essential step of this flow is the generation of the *transfer protocol*. This chapter discusses how this protocol is generated from a standard logic simulation environment. The prerequisite for this is the utilization of a special package in the test bench modelling the environment of the design-under-test. A possible implementation of this package is presented. Finally, the *transfer protocol* has to be translated to a program for the test processor. Therefore, a mapping of the expressions of the *transfer protocol* to instructions of the proposed test processors is defined. This mapping is the base for the implementation of a compiler tool that generates assembly code for the NoTePAD solution.

6.1 The Channel Simulation Package

The heart of the test program generation approach is the *channel simulation package*. It implements the algorithm defined in Section 4.3.2 that serializes the transfers and generates the *transfer protocol* from a functional simulation of the DUT. Therefore, it provides an abstract data type that models generalized asynchronous channels. Moreover, the package includes the mentioned *transfer procedures* that implement the different handshake protocols for the interactions of a DUT with its environment. Therefore, these procedures take instances of the channel data type as input, perform the data transfer in the simulation and write the corresponding *transfer statement* into the protocol file. In order to realize this functionality in a reusable manner, the package was implemented as a VHDL package. To some extent, the implementation of the package is similar to a solution provided in [Sparsø 2001]. In their approach, two VHDL packages are described that provide an *abstract* and a *real asynchronous channel* implementation. These imple-

mentations differ in the way the handshaking is encoded. The *abstract channel model*, provided there, is based on abstract phases of the handshaking, e.g., idling, waiting for the request, waiting for the acknowledgment etc. The implementation of the *real channel package* relies on the actual implementation of the handshake signalling using explicit handshake signals. Although the solution presented in this chapter is similar to this *real channel package*, it differs in several aspects. Apart from the generation of the *transfer protocol*, the most important difference is that the *channel simulation package* implements various different handshake protocols rather than a particular one. Before the package is described in detail, the general approach of writing the test bench for the simulation of the DUT and the resulting issues for the *channel simulation package* shall be discussed.

6.1.1 Preconsiderations

As usual in test benches, the DUT has to be instantiated. Naturally, all resources for accessing the DUT have to be defined in the declaration section of the architecture of the test bench. This especially includes the signals and channels required to interface the DUT instance. As previously indicated, the same applies to the *transfer protocol* that shall be generated from the test bench. All resources used in the *test section* have to be defined in the *declaration section* of the *transfer protocol file*. This leads to the following issue. Before the definition of the resources can be written into the *transfer protocol*, the respective file has to be opened. Basically, opening and writing a file in the declaration part of a VHDL architecture is not a problem. However, the issue raises from the demand that certain internals of the package, such as the file object, shall be hidden from the user. This eases the entire flow due to the abstraction from internal activities of the package. For this reason, the required activities, such as opening the hidden protocol file, are encapsulated in procedures. However, procedures cannot be called in the declaration part of a VHDL architecture. The solution to this is the following: The declaration of the resources in the test bench is detached from writing this declaration into the *transfer protocol* file. Then, the writing of these declarations has to be done later in a process of the architecture body of the test bench. This also allows the declaration of channels that shall not be included into the *transfer protocol*. This is useful, e.g., if the test bench connects various designs that shall individually be tested. Then, the test bench can make use of instances of the channel data type, which will not appear in the *transfer protocol*.

Apart from writing the declaration, two more actions are required for the correct generation of the *transfer protocol*. As defined, a *transfer protocol* consists of a *decla-*

ration section and a *test section*. Thus, after declaring all resources, the *test section* has to be initiated before the transfers can be logged into the file. Finally, when all transfers have been executed, the protocol file has to be completed by ending the *test section* and closing the file. Although other solutions might be possible, all these activities should preferably be integrated into a separate process in the test bench. This process opens the protocol file, writes the resource declarations, initiates the *test sequence*, and closes the file as shown in Figure 6.1. Besides this control process, the test bench may comprise individual processes that access the different channels connecting the DUT. This is quite common for an asynchronous DUT. All these processes have to be synchronized with the control process at certain points. For example, as it is shown in Figure 6.1, the processes accessing the channels have to wait for the start of the *test sequence* before they are allowed to perform data transfers. These synchronization activities have to be taken over by the designer creating the test bench. However, the *channel simulation package* shall deploy mechanisms to ease the synchronization. Finally, if all these measures for declaring the resources and creating the *transfer protocol* have been heeded, the generation of the *test sequence* itself shall only be a matter of calling procedures to perform channel and signal data transfers.

6.1.2 Test Processor and Package Setup

Basically, the implementation of the simulation package shall be kept generic in order to abstract from the test processor implementation. However, some constructs are required that are directly related to the test processor and to the creation of the protocol. This especially concerns the various properties of the TP shown in Listing 6.1. Thus, the

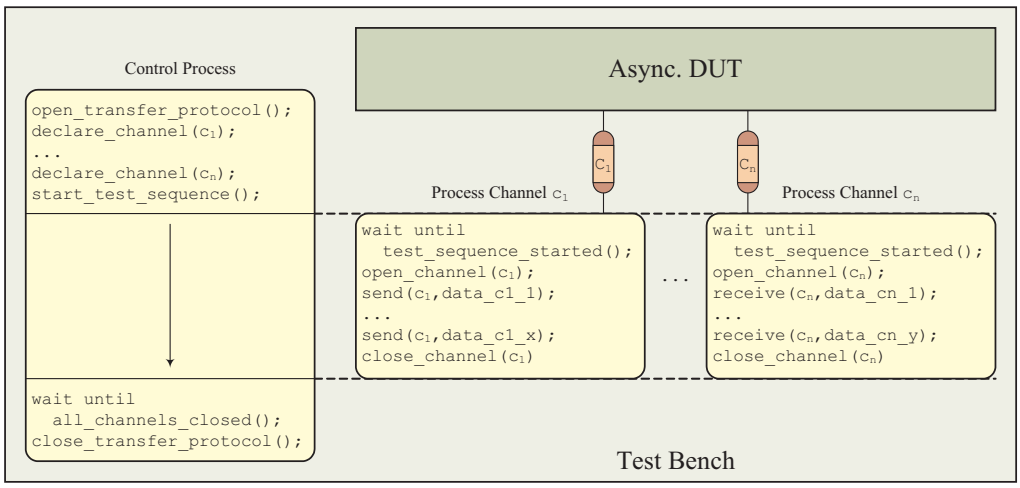


Figure 6.1: Processes of the test bench and their interaction with the DUT

Listing 6.1: Definition of test processor related constant

```
--! Period of time for one clockcycle of the processor.
constant TP_PERIOD    : time := 5 ns;
--! Maximum number of handshake ports of the processor.
constant MAX_CHANNELS : integer := 16;
--! Maximum number of io pins of the processor.
constant MAX_IO_PINS  : integer := 64;
--! Maximum number of either input or output signals.
constant MAX_SIGNALS  : integer := 8;
--! Maximum length of a name for a signal or channel.
constant MAX_NAME_LEN : integer := 20;
```

clock cycle period, the maximum number of handshake and data ports and the maximum number of control inputs and outputs have to be defined according to the architecture of the processor. These parameters have to be adapted to the concrete TP implementation used for the test. Besides these processor related parameters, a further one specifies the maximum length of the identifiers for channel and signal resources.

6.1.3 Procedures for Accessing the Transfer Protocol

As previously indicated, several things are required for the creation, the access, and the completion of the *transfer protocol*. The *transfer protocol* itself is a simple text file object in VHDL. The declaration of this file object is hidden within the package implementation. However, as it is illustrated in Figure 6.1, a procedure is required that opens a physical file for writing and associates it with the hidden file object. This is implemented by the `tp_open_transfer_protocol()` procedure. In addition to that, this procedure has to initiate the declaration section of the protocol. Thus, after a call to `tp_open_transfer_protocol()`, all the channel and signal resources used in the *test sequence* have to be declared. After that, the *transfer protocol* has to be switched to capture the *test sequence*. Therefore, the procedure `tp_start_test_sequence()` ends the declaration section and starts the *test section*. This procedure has to be called prior to any transfer on the declared resources. In order to inform the processes, which access the channels connect to the DUT, a synchronization function is required. This function, called `tp_test_sequence_started()`, has to indicate whether the *transfer protocol* was switched to the *test sequence*. Finally, when all transfers of the test bench have been processed, the protocol file needs to be closed. This is accomplished by the `tp_close_transfer_protocol()` procedure. The interfaces of all these required functions and procedures are shown in Listing 6.2.

Listing 6.2: Procedures and functions to access the *transfer protocol*

```
--! Opens and initializes the transfer protocol.
procedure tp_open_transfer_protocol( filename : string );
--! Starts the test sequence.
procedure tp_start_test_sequence;
--! Checks whether the test sequence has been started.
impure function tp_test_sequence_started return boolean;
--! Verifies whether the test sequence has been started.
procedure tp_assert_test_sequence;
--! Closes the transfer protocol.
procedure tp_close_transfer_protocol;
```

6.1.4 Model of the Handshake Protocol Type

As a prerequisite for modelling asynchronous channels, a model for the asynchronous protocol type is required. Therefore, several types are declared that represent the characteristics of an asynchronous channel. According to the abstract channel model, these characteristics include the initiator type (push/pull), the number of phases (2-phase/4-phase), and the data encoding (single-rail/dual-rail). Then, the protocol type itself is a combination of these characteristics implemented via a record type. As previously indicated, not all combinations of the properties and the corresponding protocols are supported by the test processor. However, this generic implementation allows future integration of additional protocols. Listing 6.3 illustrates the types for modelling handshake protocols.

Listing 6.3: Definition of the type for modelling the handshake protocol type

```
--! Type used to determine the initiator of a transfer
type INIT_TYPE is ( PUSH, PULL );
--! Type determining the number of phases (2-phase/4-phase)
type PHASE_TYPE is ( TWO, FOUR );
--! Type modelling the data encoding (single-rail/dual-rail)
type ENCODING_TYPE is ( SR, DR );
--! Type that models the protocol type, i.e., a combination
--! of the various handshake protocol properties.
type TP_PROTOCOL_TYPE is record
    --! Determines the initiator of the handshake.
    init   : INIT_TYPE;
    --! Determines the number of phases.
    phases : PHASE_TYPE;
    --! Defines the data encoding.
    enc    : ENCODING_TYPE;
end record;
```

6.1.5 Channel Resources

The model of the asynchronous channels in the simulation package is implemented as a record type. According to the definition of asynchronous channels, the type `TP_CHANNEL` comprises the handshake protocol information, the handshake signals and their initial values, and finally the data signals. Moreover, the direction of the channel from the perspective of the DUT is required for the mapping of the handshake signals to the in- and outputs of the handshake ports of the processor. In addition to this, the direction is, of course, also essential for the data transfers. It defines whether the channel is driven by the DUT or its environment.

Besides the characteristics inferred by the abstract channel model, a variety of other properties are required for the generation of the *transfer protocol* and the correct execution of the transfers. This includes a unique identification number, the name of the channel, the number of transfers, and the state of the channel. The name of the channel is used as the identifier of the channel in the *transfer protocol*. Typically, it is recommended to set the name to the same identifier as used in the test bench. The identification number is internally used by the *channel simulation package*, e.g., in the algorithm generating the *test sequence*. The user does not need to take care about this number. The same applies to the number of transfers. According to the *wait procedure* defined in Section 4.3.2.2, this member is internally required for determining whether a transfer has been performed on the channel. Finally, the state member is also used internally to indicate whether the channel has been correctly initialized and prepared for performing data transfers. Furthermore, this member indicates whether the channel will show up any further activity during test. Using this information, one can determine the point in time of the simulation when the protocol can be closed. Consequently, four states can be distinguished:

- The state `UNINITIALIZED` indicates that the channel has not been initialized.
- The state `INITIALIZED` indicates that the channel was correctly initialized but not opened for usage.
- The state `OPENED` indicates that the channel is ready for transferring data.
- The state `CLOSED` indicates that there will be no further activity on the channel.

The utilization of these states enables the verification of the usage of a channel and corresponding accesses. Therefore, all procedures that operate on channels check this member. In case of an invalid usage an error message is delivered.

Finally, the record type for the generalized asynchronous channels is shown in Listing 6.4. The listing also shows the array type TP_SET_OF_CHANNELS which is used to define sets of channels. This is, e.g., necessary for the *wait procedures* to specify sequence relations between transfers. Furthermore, the interface of the function tp_init_channel() is illustrated which eases the initialization of a channel.

Listing 6.4: Structure and initialization of channels

```
--! Type used to indicate a channel as an input or output.
type DIRECTION is ( INPUT, OUTPUT );
--! Type to indicate the state of a channel.
type CHANNEL_STATE is (UNINITIALIZED,INITIALIZED,OPENED,CLOSED);

--! This type describes an asynchronous channel.
type TP_CHANNEL is record
    --! ID of the channel. Used for internal identification.
    id         : integer;
    --! Name of the channel.
    name       : string(1 to MAX_NAME_LEN);
    --! Stores the number of transfers
    transferid : integer;
    --! Determines whether the channel is an input or output.
    dir        : DIRECTION;
    --! Determines the state of the channel.
    state      : CHANNEL_STATE;
    --! Stores handshake protocol information.
    hp         : TP_PROTOCOL_TYPE;
    --! The request signal of the channel.
    req        : std_logic;
    --! The acknowledgement signal of the channel.
    ack        : std_logic;
    --! The initial value of the request signal.
    req_init   : std_logic;
    --! The initial value of the acknowledgement signal.
    ack_init   : std_logic;
    --! The data bus of the channel.
    data       : std_logic_vector(MAX_IO_PINS-1 downto 0);
    --! Bit width of the data.
    size       : integer;
end record;
--! An array type for channels
type TP_SET_OF_CHANNELS is array (natural range <>) of TP_CHANNEL;
```

```
--! Initializes a channel with the specified parameters
function tp_init_channel(
    name        : in string;
    width       : in integer;
    dir         : in DIRECTION;
    init        : in INIT_TYPE;
    phases      : in PHASE_TYPE;
    enc         : in ENCODING_TYPE;
    req_init    : in std_logic := '0';
    ack_init    : in std_logic := '0' ) return TP_CHANNEL;
```

After declaring and initializing a channel in the declaration part of the test bench architecture, the channel can be registered in the *transfer protocol*. This is accomplished by the procedure `tp_register_resource()` which writes the channel properties into the *transfer protocol*. Obviously, the protocol file has to be opened using `tp_open_transfer_protocol()` before this procedure is called. From the functional point of view, the channel can be used to apply and receive data. However, according to the possible states the channel has to be opened to indicate that the channel is active. In addition to that, it might be necessary in some cases to set the data signals to predefined values before the first handshake is executed. Therefore, the procedure `tp_open_channel()`, which opens a channel, provides an optional parameter. This parameter designates the value assigned to the data signals of the channel without performing a handshake. After calling this procedure the channel is opened and allowed to perform transfers. Finally, as indicated in Figure 6.1, the channel has to be closed in order to indicate that the channel will not perform any further activity. Using this information, a process in the test bench can check whether channels handled in other concurrent processes are active. To ease this check the package provides two functions `tp_is_opened()` and `tp_is_closed()` whose interface is shown in Listing 6.5.

The most important functions are the *transfer procedures*. With respect to this, there are three possible activities. Obviously, two procedures have to be provided to write data to a channel (`tp_send()`), and to read data from a channel (`tp_recv()`). These procedures implement the different handshake protocols and write the *transfer statement* into the protocol file. However, these procedures can only be applied to channels having data signals. In order to support control channels without any data signal, a further procedure (`tp_sync()`) is required that only performs a handshake and adds a `SYNC`-statement to the protocol.

Furthermore, it might be necessary to wait for transfers on a set of channels before

Listing 6.5: Channel preparation procedures

```
--! Registers a channel and writes a declaration statement
--! into the transfer protocol.
procedure tp_register_resource(ch : in TP_CHANNEL);
--! Opens and initializes a channel according to the protocol.
procedure tp_open_channel(signal ch : inout TP_CHANNEL;
    data : in std_logic_vector := "U");
--! Closes a channel.
procedure tp_close_channel(signal ch : inout TP_CHANNEL);
--! Determines whether a channel was closed.
function tp_is_opened(signal ch : in TP_CHANNEL) return boolean;
--! Determines whether a channel was closed.
function tp_is_closed(signal ch : in TP_CHANNEL) return boolean;
```

continuing with the execution of a process. Therefore, the package provides two overloaded procedures[1] named `tp_wait()` that implement the *wait procedure* mentioned in Section 4.3.2.2. These procedures pause the execution of the calling process and wait until each specified channel has executed a handshake. To express this sequential behavior in the *transfer protocol*, the present concurrent transfer group is terminated. More details about that are given in Section 6.1.8. In summary, Listing 6.6 shows the prototypes of the procedure related to channel transfers.

Listing 6.6: Handshake procedures

```
--! Performs a handshake and writes data into the channel.
procedure tp_send(signal ch  : inout TP_CHANNEL;
    value : in std_logic_vector );
--! Performs a handshake and reads data from the channel.
procedure tp_recv(signal ch  : inout TP_CHANNEL;
    value : out std_logic_vector );
--! Performs only a handshake.
procedure tp_sync(signal ch  : inout TP_CHANNEL);
--! Waits for a transfer on the specified channel.
procedure tp_wait(signal ch  : in TP_CHANNEL);
--! Waits for transfers on all specified channels.
procedure tp_wait(signal chs : in TP_SET_OF_CHANNELS);
```

6.1.6 Signal Resources

A signal resource is quite similar to a channel. As shown in Listing 6.7, the corresponding data type comprises an identification number, a name, a direction, the data signals

[1] Overloaded procedures are procedures that have the same name, but a different parameter list.

Listing 6.7: Structure and initialization of signal resources

```
--! This type describes a signal or bus resource.
type TP_SIGNAL is record
    --! Id of the signal.
    id    : integer;
    --! Name of the signal.
    name  : string(1 to MAX_NAME_LEN);
    --! Direction of the signal.
    dir   : DIRECTION;
    --! Left bus range specifier.
    left  : integer;
    --! Right bus range specifier.
    right : integer;
    --! Data signals.
    data  : std_logic_vector(MAX_SIGNALS-1 downto 0);
    --! Bit width of the signal.
    size  : integer;
end record;

--! Defines a signal with the desired parameters
function tp_define_signal(
    name  : string;
    dir   : DIRECTION;
    left  : integer := 0;
    right : integer := 0 ) return TP_SIGNAL;

--! Writes a declaration statement into the transfer protocol.
procedure tp_register_resource( sig : in TP_SIGNAL );
```

and the according number of signals actually used. Furthermore, in order to cover entire data buses, the record type has two additional members `left` and `right`. These are used to describe the original bus range. For example, to declare a bus with the range `(3 to 8)`, `left` has to be set to 3 and `right` to 8. As well as for the channel record type, the simulation package provides a function `tp_init_signal()` for initializing a signal resource and a procedure `tp_register_resource()` to register a signal in the *transfer protocol*.

After the declaration, a signal resource can be used in the *test section*. Therefore, the package provides one procedure to assign a value to a signal resource. The other direction, thus, reading data from the signal resource is more ambiguous due to the timing relaxed nature of the desired test. Here, two different scenarios have to be distinguished. On the one hand, the signal may simply be read and compared in the moment of reaching that action in the *transfer protocol*. However, due to the lack of accurate timing the actual point in time of the comparison during test may differ from the one in the simula-

tion. Thus, such a simple comparison makes sense only in well-defined states, e.g., after the execution of a specific channel transfer.

On the other hand, it might be necessary to wait until the DUT delivers a specific value at a certain control output. In order to support both scenarios, the *channel simulation package* provides two functions, i.e., `tp_get_value()` and `tp_sync_value()` shown in Listing 6.8. The first one, `tp_get_value()`, directly reads the value of the signal during the simulation and writes a *signal comparison statement* into the *transfer protocol*. This means for the according test that the test processor has to directly check the value of the signal. If the received value does not match the expected one, then the test is treated as failed.

The functionality of the procedure `tp_sync_value()` is more sophisticated. It provides one optional parameter `expvalue`. If this parameter is defined, then the procedure waits until the value of the signal resource is equal to this parameter during simulation. Otherwise, the procedure simply reads the current value of the signal. In both cases, the procedure writes a *signal wait statement* into the *transfer protocol*. Thus, the test processor will wait until the value of the signal resource becomes either equal to the value of the signal computed by the simulation or the value defined by `expvalue`.

Listing 6.8: Accessing signal resources

```
--! Assigns a value to the specified signal.
procedure tp_assign(
    signal sig   : inout TP_SIGNAL;
           value : std_logic_vector);

--! Reads the value from the signal and writes an
--! according statement into the protocol.
function tp_get_value(signal sig : in TP_SIGNAL)
return std_logic_vector;

--! Reads a value from the signal possibly syncs this
--! value with the expected value.  In both cases a
--! wait statement is written to the protocol.
procedure tp_sync_value(signal sig : in TP_SIGNAL;
    actvalue : out std_logic_vector;
    expvalue : in  std_logic_vector := "X");
```

6.1.7 Miscellaneous Functions

Apart from the functions and procedures dedicated to processor resources, there are a couple of activities required to provide further control of the *transfer protocol* generation. One of these activities is the control of the timeout of transfer operations. Thus, one procedure is required that writes *timeout statements* and another one is required to write *timeout routine statements* into the *transfer protocol file*. Similarly, a method is required that waits for a certain time window and writes a corresponding *wait statement* into the protocol file. Finally, in some situations it might be required to explicitly terminate the present concurrent transfer group. In consideration of these activities, the *channel simulation package* provides the control functions shown in Listing 6.9.

Listing 6.9: Control functions

```
--! Terminates the present concurrent transfers.
procedure tp_sync;

--! Pauses the execution of the calling process for the
--! specified time and writes a wait statement into the file.
procedure tp_wait( waittime : time );

--! Writes a timeout statement into the transfer protocol file
--! that specifies the timeout for handshake and signal
--! comparison operations.
procedure tp_set_timeout( timeout : time );

--! Writes a timeout routine statement into the transfer
--! protocol file that specifies the name of the routine
--! that is called in case of a timeout.
procedure tp_set_timeout_routine( routine : string );
```

6.1.8 Implementation of the Sequence Generation Algorithm

As a first step of the implementation of the approach described in Section 4.3.2.2, one has to define how the *concurrent transfer groups* and the *test sequence* are represented by VHDL constructs and how these are stored in memory during simulation. Basically, the *concurrent transfer groups* are independent from each other and are sequentially created. Therefore, it is not required to simultaneously store the entire *test sequence* in the memory during the simulation. Instead, every group and even each transfer can directly be written to the protocol file. However, for this to be applicable, one has to add either the *concurrency* or the *sequence operator* between the separate statements. Of course, to

determine which of these operators should be inserted, one has to consider the transfers that are already in the presently considered transfer group designated by G_k in Listing 4.2. Thus, one has to keep G_k in the memory during the simulation. However, instead of buffering the entire information about the transfers, it is enough to store only the channels accessed by the transfers in G_k. At this point, the unique IDs of the channels come into play. Therefore, a structure is required that stores the IDs of the accessed channels. Apart from that, further information about the registered resources is required. This is necessary to prevent the *transfer procedures* to log transfers of unregistered channels. Therefore, the IDs of the registered channels are stored. To encapsulate all this information for the *test sequence* generation, the record type `TP_STATE` is introduced as shown in Listing 6.10. Prior to its definition, an array type is declared that is used to store the IDs of channel resources.

The most important part of Listing 6.10 is the record type `TP_STATE` itself. The type comprises a boolean value indicating whether the *transfer protocol* has been initialized, the number of used control in- and outputs, the number of data pins that are associated with registered channels, and two sets of channel IDs. Each of these sets is implemented using an instance of the defined array type and the number of IDs actually stored in the array. According to their names, the first set stores the IDs of the channels registered in the *transfer protocol*, whereas the second one stores the IDs of the channels that are accessed concurrently. Afterwards, a global instance `tp_proc` of `TP_STATE` is created that is used by the *transfer procedures*. The listing also shows the file object for the *transfer protocol*.

Listing 6.10: Internal data structures of the *channel simulation package*

```
--! Array type for storing the IDs of channels.
type TP_CHANNEL_IDS is array (integer range <>) of integer;
--! Type modelling the state of the test processor.
type TP_STATE is record
    --! Determines whether the TP is ready for the test sequence
    is_init         : boolean;
    --! Number of used input control signals
    num_in_signals  : integer;
    --! Number of used output control signals
    num_out_signals : integer;
    --! Number of used data pins
    num_data_pins   : integer;
    --! Number of registered channels
    num_registered_channels : integer;
    --! IDs of registered channels
```

```
    registered_channels : TP_CHANNEL_IDS(0 to MAX_CHANNELS-1);
    --! Number of concurrent accesses
    num_accessed_channels : integer;
    --! IDs of concurrently accessed channels
    accessed_channels : TP_CHANNEL_IDS(0 to MAX_CHANNELS-1);
end record;
--! Variable storing global status information.
shared variable tp_proc : TP_STATE;

--! The transfer protocol file object.
file tp_transferprotocol  : text;
```

Using these constructs, the procedure `log_transfer()` can be implemented that realizes the algorithm. Listing 6.11 shows this procedure. It takes as input the accessed channel and the textual description of the transfer, i.e., the corresponding *transfer statement*. This description has to be generated by the callers of this procedure.

Prior to the definition of the procedure a text line buffer is declared. The purpose of this buffer is the following. As mentioned, instead of buffering all the concurrent transfers, these are directly written to the protocol file. However, either the *concurrency operator* or the *sequence operator* has to be inserted between the last logged statement and the next one to be added. Therefore, each statement is first written into the buffer. When the next statement shall be logged, the according operator is added to the buffer which is afterwards written to the protocol file. After that, the buffer is set to the description of the current transfer.

Listing 6.11: Implementation of the concurrency check

```
--! Stores the next line to be written to the transfer protocol.
shared variable tp_buffer : line;

procedure log_transfer(ch : in TP_CHANNEL;
  transferdesc : inout line) is
begin
  -- Check whether the current channel is registered.
  -- If not, we don't need to log the transfer.
  if not contains(ch.id, tp_proc.registered_channels,
    tp_proc.num_registered_channels) then return;
  end if;

  -- Check the channel to be present in the concurrent group.
  if contains(ch.id, tp_proc.accessed_channels,
```

```
    tp_proc.num_accessed_channels)
  then
    -- If the channel already exists in the group add the
    -- sequence operator to the buffer, write the buffer
    -- to the file
    if ( tp_buffer /= null ) then
      write(tp_buffer, ";");
      writeline(tp_transferprotocol, tp_buffer);
    end if;
    -- ... and reset the concurrent group.
    tp_proc.num_accesses := 0;
  else
    -- ... otherwise add the concurrency operator and write
    -- the buffer to the file.
    if ( tp_buffer /= null ) then
      write(tp_buffer, ",");
      writeline(tp_transferprotocol, tp_buffer);
    end if;
  end if;

  -- Afterwards, write the description of the current
  -- transfer to the buffer.
  write(tp_buffer, transferdesc.all);

  -- Add the channel to the considered concurrent group.
  tp_proc.accessed_channels(
    tp_proc.num_accessed_channels ) := ch.id;
  tp_proc.num_accessed_channels :=
    tp_proc.num_accessed_channels + 1;
end;
```

The first step of the procedure `log_transfer()` checks whether the transfer has to be logged. This is the case if the ID of the accessed channel is in the set of registered channels of the global `tp_proc` object. Similarly, the second step checks whether the channel has already been accessed by any other transfer in the currently considered transfer group. If this is not the case, then the *concurrency operator* is added to the buffer. However, if the channel was already accessed, then the *sequence operator* is added to the buffer and the set storing the IDs of the accessed channels is cleared. In both cases the buffer, comprising the last statement, is written to the protocol file. Finally, the description of the current transfer is written to the buffer and the channel ID is added to the

set of concurrently accessed channels. Obviously, `log_transfer()` only adds the last statement to protocol file. The current transfer applied to the function is only stored in the buffer. For this reason, the buffer still contains the very last statement at the end of the simulation. Thus, the content of the buffer has to be written to the protocol file before closing it. This step is part of the procedure `tp_close_transfer_protocol()`.

Basically, the `log_transfer()` procedure has to be called by `tp_send()` and `tp_recv()` to log the transfer. However, these procedures are further decomposed. Two further procedures, i.e., `read_value()` and `write_value()`, are introduced that read and write the data value of a channel and generate the *transfer statement*. Finally, these procedures call `log_transfer()` as shown in Listing 6.12. Both procedures have an additional boolean parameter `sync_only`. This parameter determines whether to perform only a synchronization handshake. If this is the case, a `SYNC` *transfer statement* is

Listing 6.12: Reading and writing channel data

```
--! Writes the specified data to the data part of the channel.
procedure write_value(signal ch : inout TP_CHANNEL;
    data : in std_logic_vector; sync_only : boolean) is
    variable desc : line;
begin
    if sync_only then
        write(desc, "␣␣␣␣SYNC␣" & get_name(ch) );
    else
        write(desc, "␣␣␣␣" & get_name(ch) & "␣<=␣");
        write(desc, '"' & str(data) & '"');
        assign_data(ch, data);
    end if;
    log_transfer(ch, desc);
end procedure;

--! Reads the data value from a channel and assigns it
--! to the specified output variable.
procedure read_value(signal ch : inout TP_CHANNEL;
    data : out std_logic_vector; sync_only : boolean ) is
    variable desc : line;
begin
    if sync_only then
        write(desc, "␣␣␣␣SYNC␣" & get_name(ch) );
    else
        write(desc, "␣␣␣␣" & get_name(ch) & "␣=>␣");
        write('"' & str(ch.data(ch.size-1 downto 0)) & '"');
        assign_data(data, ch);
    end if;
    log_transfer(ch, desc);
end procedure;
```

written into the protocol file.

Finally, the *transfer procedures* implement the individual handshake protocols and call the procedures `read_value()`, `write_value()` to log the transfers. Listing 6.13 shows a simplified excerpt of the implementation of `tp_send()`. This procedure takes as input the channel object and the data value to be sent. In addition to that, the procedure also has the optional boolean parameter `sync_only`, which is forwarded to the `write_value()` procedure. By default `sync_only` is set to `false` such that a full data transfer is performed.

Listing 6.13: Implementation of the *transfer procedure* for sending data

```
procedure tp_send(
    signal ch : inout TP_CHANNEL;
        value : in std_logic_vector;
    sync_only : boolean := false ) is
begin

    -- Checks consistency
    assert tp_proc.is_init;
    assert ch.state = OPENED;
    assert ch.dir = INPUT;

    case ch.hp.init is
    when PUSH =>
        case ch.hp.phases is
        when TWO =>
            ...
        when FOUR =>
            case ch.hp.enc is
            when SR =>
                -- Wait until the acknowlegement is low
                if (ch.ack /= ch.ack_init) then
                    wait until ch.ack = ch.ack_init;
                end if;
                -- Assign the data to the channel
                write_value(ch, value, only_sync);
                -- Issue the request
                ch.req <= not ch.req_init after TP_PERIOD;
                -- Wait for completion of the transacation
                wait until ch.ack = not(ch.ack_init);
                -- Reset request to finish the handshake
                ch.req <= ch.req_init after TP_PERIOD;
```

```
            when DR =>
                ...
            end case;
        end case;
    when PULL =>
        ...
    end case;
    --! Update transfer ID
    ch.transferid <= ch.transferid + 1;
end procedure;
```

In the implementation of `tp_send()`, a variety of consistency checks are performed first. During the simulation, these checks test whether the *transfer protocol* and the specified channel have been opened properly. Afterwards, a set of `case`-statements selects the protocol according to the respective information stored in the channel object. Listing 6.13 exemplarily shows the implementation of the 4-phase single-rail push protocol. The first step of this protocol checks whether the next data transfer can be initiated. This is the case if the acknowledgment signal is in its inactive null-value, i.e., the value of the `ack_init` member of the channel. Afterwards, the data value is assigned to the data part of the channel via the call of `write_value()`. Consequently, the transfer is logged in the protocol file. After applying the data to the channel, the request is issued by setting the request signal of the channel to the inverse of its initial value `req_init`. To emulate the test processor environment to a certain extent, this assignment is delayed by the clock cycle period of the processor. Then, the procedure waits for the acknowledgment of the receiver before the request signal is reset. As a final step, the procedure increments the transfer counter of the channel in order to indicate that a transfer has been performed.

In the opposite direction, the *transfer procedure* `tp_recv()` reads data from a channel. Its parameters are almost identical to `tp_send()` except that the data value parameter is an output rather than an input. Listing 6.14 shows the fragment of the procedure implementing the 2-phase single-rail push protocol. In the first phase of the protocol the procedure waits for a transition on the request line. Afterwards, the data is read from the channel and the transfer statement is written to the protocol file via the call of `read_value()`. As a last step of the protocol, the receipt of the data is acknowledged by generating a transition on the acknowledgment line. Finally, the transfer counter is incremented.

Listing 6.14: Implementation of the *transfer procedure* for receiving data

```
procedure tp_recv(
    signal ch : inout TP_CHANNEL;
        value : out std_logic_vector;
    only_sync : boolean := false) is
begin

    -- Consistency checks
    assert tp_proc.is_init;
    assert ch.state = OPENED;
    assert ch.dir = OUTPUT;

    case ch.hp.init is
    when PUSH =>
        case ch.hp.phases is
        when TWO =>
            case ch.hp.enc is
            when SR =>
                -- Wait for the request
                wait until ch.req'event;
                -- Read the value from the channel
                read_value(ch, value, only_sync);
                -- Acknowledge the receipt
                ch.ack <= not(ch.ack) after TP_PERIOD;

            when others =>
                ...
            end case;

        when FOUR =>
            ...
        end case;
        ...
    end case;
    --! Update transfer ID
    ch.transferid <= ch.transferid + 1;
end procedure;
```

According to Section 4.3.2.2, a further requirement for the generation of the protocol is the *wait procedure*. For the sake of usability, there are two implementations of this procedure: one waits for a transfer on a single channel and the other one waits for transfers

on a set of channels. To detect a transfer, these functions make use of the `transferid` member of the channels. Corresponding to the conceptual algorithm described in Listing 4.3, the actual transfer IDs of the involved channels are stored first. Afterwards, the procedures wait until all IDs have been changed by the *transfers procedures* executed concurrently by other processes. Finally, when each specified channel has performed a transfer, the present *concurrent transfer group* is terminated via adding the *sequence operator*. Thus, any transfer performed after the call of `tp_wait()` is ensured to occur after the specified transfers in the *transfer protocol*, as well.

Listing 6.15: Implementation of the *wait procedures*

```
--! Waits for a transfer on a single channel
procedure tp_wait( signal ch : in TP_CHANNEL ) is
    variable current_transfer_id : integer;
begin
    tp_assert_test_sequence;
    current_transfer_id := ch.transferid;
    wait until ch.transferid /= current_transfer_id;
    tp_sync;
end procedure;

--! Type for storaging IDs of channels
type TRANSFER_IDS is array (natural range <>) of integer;

--! Helper function that checks whether the transfer IDs of
--! all specified channels change
function tp_all_channels_fired(
    signal chs : in TP_SET_OF_CHANNELS;
    ids : TRANSFER_IDS ) return boolean is
begin
    for i in chs'range loop
        if ids(i) = chs(i).transferid then
            return false;
        end if;
    end loop;
    return true;
end function;

--! Waits until each specified channel has performed at
--! least one handshake
procedure tp_wait( signal chs : in TP_SET_OF_CHANNELS ) is
    variable current_transfer_ids : TRANSFER_IDS(chs'range);
```

```
begin
    tp_assert_test_sequence;
    for i in chs'range loop
        current_transfer_ids(i) := chs(i).transferid;
    end loop;
    wait until tp_all_channels_fired(chs, current_transfer_ids);
    tp_sync;
end procedure;
```

6.2 Mapping of a Transfer Protocol to a Processor Program

With the help of the procedures introduced above, the *transfer protocol* can be generated from the functional simulation of the DUT. The next step in the flow is the generation of the test processor program from the created *transfer protocol*. This requires a mapping of the protocol statements to instructions of the test processor. Obviously, this step hardly depends on the test processor implementation. The following sections describe the entire procedure of creating a program for the provided NoTePAD architecture from the *transfer protocol*.

6.2.1 Preconsiderations Regarding the Program Generation

As outlined in the concept (see Section 4.3.2), the program is created in a platform dependent assembler format. Obviously, this direct mapping to assembler code has the disadvantage that the generated program cannot be reused for a different test processor architecture. An alternative approach is the generation of code in a high-level-language, such as C. However, high-level languages have several disadvantages. First, a C-compiler for the special processor would be required, whose creation is an exhaustive task. Furthermore, a library is needed that provides procedures to access the special functionalities of the processor. But the most important disadvantage is that high-level-languages demand a stack to realize context switches resulting from subroutine calls. However, in consideration of the desired programs, such context switches and, therefore, the stack as well, are not required due to the almost fully sequential nature of the generated programs. Therefore, the management of the stack leads to computation overhead which is not required. Moreover, the processor is basically not designed to execute programs derived from high-level languages. All these considerations lead to the decision to generate assembler code rather than platform independent high-level language code. This has the advantage that the generated code is kept compact with respect to the generated instruction sequences and resource management.

Basically, the entire program can be divided into different phases. Two of these phases are directly derived from the two sections of the *transfer protocol*: the *initialization phase* and the *test execution phase*. The *initialization phase* comprises all the required steps for setting up the processor. In the *test execution phase* the entire *test sequence* as described by the *transfer protocol* is carried out. Besides these two phases, two further steps are required that are not represented by a section in the protocol file, i.e., the finalization of the test and the upload of the test results. The *test finalization*

phase comprises additional steps to guarantee that all test results are available. After that the results have to be sent to the ETE in the *result upload phase*.

The *initialization phase* can be further divided into the *processor initialization* and the *test preparation phase*. In the *processor initialization phase* the resources of the processor are initialized, whereas the *test preparation phase* comprises activities to prepare the execution of an individual test, e.g., the reinitialization of the ports. In practice, it might be beneficial to separate the *processor initialization* from the other phases. For example, consider the case that a single device shall pass through a sequence of functional tests. Typically these tests utilize the same resources. In order to shorten the program and the test execution time, the initialization phase of the processor can be executed once before the entire sequence of tests. On the other hand, in case the processor resources have to be changed, the processor has to execute the initialization phase. In comparison to that, the *test preparation phase* has to be executed once for each test.

This separation of the entire program into different phases has some effects to the generation of the program. This especially concerns the creation of the program files. Thus, instead of creating one single file, the program is split into various files. One set of files includes the processor initialization and another set includes the test specific phases. This allows that the processor initialization can be executed separately from the other phases. Moreover, in consideration of generating assembler code, the programs are further divided for the following reason. Typically, assembler programs are separated into sections. The most important ones are the program section (typically called text section) and the data section. Obviously, the program section comprises the instructions to be executed by the processor, whereas the data section includes statically allocated data fields. During the link process, all possibly distributed sections of the same type are combined to a single one, which is finally mapped to memory resources.

To gain a clear separation between the sections, the generated programs are divided into various files. The program section, including all instructions, is written into one file or, in case of a separation of the *processor initialization phase*, into two files. In comparison to that, the data section is spread over several files. Each of these files includes the data for one data port. This is due to the agreement that the data for each data port is aligned in an array. Hence, separating this data into dedicated files allows that the data can directly written into the files. Otherwise, it would be required to collect all the data for each port during the translation process. According to this, the respective file for a data port, which is part of an output channel of the TP, includes the stimuli to be applied to the DUT. Compared to this, the file for a data port used as an input comprises the expected data and a memory array reserved for the responses/fault signatures. The size of the memory for the responses is determined by the product of the number of

transfers of the respective channel and the number of pins of the data port.

A further aspect that regards the DP data is their alignment in words which have to match the size of the words read from the data memory of the processor. For this reason, it might be necessary to buffer the data for each port until the size of the buffered data matches the size of a word read from the memory. For example, consider the case that each data port comprises 4 pins and the width of the data words read from the memory is 32 bits. For the proposed processor implementation, each data word, which is read from the memory and delivered to a data port, comprises 16 bits for the data value and 16 bits for the mask. Consequently, each of these words includes the required data for 4 data transfers. According to that, the data of 4 transfers have to be collected before the data word is written into the file. To further illustrate this, consider the sequence of channel transfers in Listing 6.16. Note that the data part of the channel comprises 15 bits. Assume that these transfers are the only ones performed on this channel. Furthermore, assume that the data ports associated with the channel are $\omega_3, \ldots, \omega_0$, where ω_3 realizes the most significant (leftmost) bits and ω_0 realizes the least significant (rightmost) bits.

Listing 6.16: Sequence of data transfers on a channel

```
1 DECLARE
      ch1 : OUT CHANNEL( PUSH, 4P, SR, '0', '0', 15 );
  END
  TEST
      ...
6     ch1 => "110 1000 0000 0001";
      ch1 => "010 1XX0 1110 1111";
      ch1 => "X01 0010 1000 0101";
      ...
  END
```

Corresponding to that, the data of the four data ports that are written to the data files is shown in Listing 6.17. Note that the data is combined in one file which is typically spread over four files. Apart from the test pattern data for the individual data ports, also the configuration of the address registers and the word counter has to be generated. According to the defined memory organization shown in Figure 5.4, this configuration has to be stored at the start of the data memory to ensure that the DPs can access the correct data. With respect to the test pattern data, the file includes the expected output responses and a memory reservation statement for each port. The latter one is used for storing the responses or fault signatures captured by the port. The statement itself comprises the start address label of the reserved data array, the size in bytes and the byte alignment.

Listing 6.17: Corresponding content of the files for the data ports

```
    .data
CONF_OF_DP0:
    .word (DP0_RDATA-DMEM_START)
    .word (DP0_WDATA-DMEM_START)
    .word ((DP0_RDATA_END-DP0_RDATA) >> 1)
    .word 0
CONF_OF_DP1:
    .word (DP1_RDATA-DMEM_START)
    .word (DP1_WDATA-DMEM_START)
    .word ((DP1_RDATA_END-DP1_RDATA) >> 1)
    .word 0
CONF_OF_DP2:
    .word (DP2_RDATA-DMEM_START)
    .word (DP2_WDATA-DMEM_START)
    .word ((DP2_RDATA_END-DP2_RDATA) >> 1)
    .word 0
CONF_OF_DP3:
    .word (DP3_RDATA-DMEM_START)
    .word (DP3_WDATA-DMEM_START)
    .word ((DP3_RDATA_END-DP3_RDATA) >> 1)
    .word 0

DP0_RDATA:
    .word 0x0fff05f1
DP0_RDATA_END:
    .bss DP0_WDATA, 4, 4

DP1_RDATA:
    .word 0x0fff08e0
DP1_RDATA_END:
    .bss DP0_WDATA, 4, 4

DP2_RDATA:
    .word 0x0f9f0288
DP2_RDATA_END:
    .bss DP0_WDATA, 4, 4

DP3_RDATA:
    .word 0x03770526
DP3_RDATA_END:
    .bss DP0_WDATA, 4, 4
```

For better illustration, consider the binary representation of the 32-bit data words:

	$m_{n/a}$	m_3	m_2	m_1	$v_{n/a}$	v_3	v_2	v_1
data of ω_3 :	0000	0011	0111	0111	0000	0101	0010	0110
data of ω_2 :	0000	1111	1001	1111	0000	0010	1000	1000
data of ω_1 :	0000	1111	1111	1111	0000	1000	1110	0000
data of ω_0 :	0000	1111	1111	1111	0000	0101	1111	0001

The left half of these 32-bit words comprises the bits for the mask, whereas the right half of the bits includes the data value. Thereby, the column labelled with m_1, m_2, m_3 denote the masks for the three data transfers and v_1, v_2, v_3 designate the value for the transfers. Since only three transfers are executed on this channel, the four leftmost bits of each of the half words ($m_{n/a}, v_{n/a}$) are not used and set to logical-0. Note that the undetermined bits (designated by *X*-values in Listing 6.16) cause the mask and the value bits at the corresponding positions to be set to logical-0, as well. Furthermore, it can be seen that the leftmost bits of the individual 4-bit blocks of the data port ω_3 are not used, since only 15-bits are required for the channel. Therefore, the mask and the value bits are also set to logical-0 at the respective bit positions.

6.2.2 Mapping to NoTePAD Instructions

Based on the above mentioned considerations, the mapping of the *transfer protocol* to the instructions of NoTePAD is discussed in this section. For this, the phases of the generated programs are described separately.

6.2.2.1 Processor Initialization

The first activity in the *processor initialization phase* is to reset the resource configuration. For the NoTePAD processor, this is achieved by executing the `rsio` instruction which resets the configuration of the *port switch* and deletes the associations of the data ports with the handshake ports. Afterwards, the new associations can be defined according to the declarations in the *transfer protocol*. Therefore, the channel and the signal declarations of a *transfer protocol* are processed one by one.

As defined in Section 4.3.2.3 a channel declaration has the following format:

<*name*> : <*dir*> CHANNEL(<*type*>, <*ph*>, <*enc*>, <*ri*>, <*ai*>, <*bits*>)

Three steps are required to setup a channel corresponding to the properties defined by its declaration in the protocol file. First, the channel resources, i.e., one handshake port and a set of data ports have to be allocated. This allocation has to take the maximum number of available ports and the number k of pins of each data port into account. Thereby, the number x of allocated DPs is determined by the smallest natural number such that $x \cdot k \geq b$, where b is the number of required bits for the channel (determined by <*bits*> in the above declaration). Afterwards, in the second step the allocated ports have to be associated with each other. This is accomplished by executing the `conf` instruction for each of the allocated data ports. Thus, let i be the ID of the next allocatable handshake port ϕ_i. Furthermore, let $\omega_0, \ldots, \omega_{y-1}$ be the set of already allocated data ports. Then, the

instruction `conf hp=`i`, dp=`j has to be generated for each $j \in \{y, \ldots, y+x-1\}$. Finally, in the last step the handshake protocol configuration needs to be defined for the allocated HP. Therefore, the configuration bit mask p is created as defined in Section 5.3 from the protocol parameters <*type*>, <*phases*>, <*enc*>, <*req_init*> and <*ack_init*>. This configuration mask is applied to the handshake port by executing the instruction `sahp hp=`i`, protocol=`p.

In comparison to channels, the initialization of signal resources requires only the allocation of control signals. Therefore, no explicit instruction has to be generated. Instead, the allocation only has to take care about the number of control pins. Thus, after the execution of the instructions for setting up the channels, the processor initialization is finished. As mentioned, these instructions have to be called only once, unless the required resources need to be changed. One important thing to note is that this resource allocation is also relevant for the interconnection of the processor with the DUT, since it determines the pin configuration.

An example of the generation of the *processor initialization phase* is given in the next section.

6.2.2.2 Test Preparation

As indicated before, the *test preparation phase* comprises test specific initializations which have to be executed once before each test. For the NoTePAD solution, this includes the initialization of the handshake and the data ports via the execution of the `init` instruction. Besides the initialization of the internal signals of the HPs, the instruction induces the selected handshake ports to consecutively apply the `SRDA`, `SWRA` and `SDWC` opcodes in order to initialize the registers of the associated data ports. To select the HPs, a mask has to be generated which has logical-1s at all positions that are equal to one of the IDs of the allocated HPs. As a result of the execution of the `init` instruction, all associated data ports request the required data from the memory as described in Section 5.2.2. Furthermore, an additional step is required to allow the used data ports to issue memory requests. Therefore, the `tmrq` instruction has to be called. This instruction receives the same mask as parameter as the `init` instruction.

Besides the initialization of the data port registers, it might be beneficial to fill the output FIFOs prior to the test run. This is useful if the DUT has a high data throughput. Thus, filling the FIFOs prior to the test execution might prevent bottlenecks due to empty output FIFOs. On the other hand, if the data rate of the DUT is low, then filling the FIFOs could increase the required test time as the data could be loaded while running the test. Therefore, this step is optional. In practice, one has to consider the throughput of the

DUT and the number of connected data ports in order to decide whether to execute the `fill` instruction.

Finally, the branch address register has to be set such that it points to a timeout routine. By default, the code implementing the *test finalization phase* is used as the timeout routine. To set the branch address register to the start address of this routine, a combination of the `setl` and `seth` instruction is used.

As an example, consider the fragment of a *transfer protocol* shown in Listing 6.18 and the corresponding assembly code shown in Listing 6.19. Note that the declaration of the `rst` signal does not result in any code. Furthermore, assume that each DP has four I/Os.

Listing 6.18: Declaration section of a *transfer protocol*

```
DECLARE
    rst  : IN  SIGNAL;
    cin  : IN  CHANNEL( PUSH, 4P, SR, '0', '0', 8 );
    cout : OUT CHANNEL( PUSH, 4P, SR, '0', '0', 8 );
END
```

Listing 6.19: Assembly code for the considered *transfer protocol* fragment

```
    .text
INIT_PROC:
    ; Reset port switch configuration
    rsio
    ; Associate the data ports with the handshake ports
    conf hp=0, dp=0
    conf hp=0, dp=1
    conf hp=1, dp=2
    conf hp=1, dp=3
    ; Configure handshake protocol
    sahp hp=0, protocol=0x5
    sahp hp=1, protocol=0x4
INIT_TEST:
    ; Initialize the ports
    init hpmask=0x0003
    ; Allow data memory requests
    tmrq hpmask=0x0003
    ; Load timeout routine
    seth r2, (END_OF_TEST >> 16)
    setl r2, (END_OF_TEST & 0xffff)
```

6.2.2.3 Test Execution

As defined by the format, the *transfer protocol* comprises various types of test activities including channel transfers, signal operations, and finally wait operations. To certain respects, the generation of the instructions implementing the *transfer statements* is trivial for the NoTePAD processor. In comparison to the RISC processor implementation provided in [Zeidler 2012a], each group of concurrent channel transfers is mapped to one single `test` instruction. Therefore, a bit mask representing the enable signals for the handshake ports has to be generated for each concurrent transfer group. Thus, the bit at position i of this mask is set to logical-1 if the handshake port ϕ_i is involved in the concurrent transfer group. Otherwise, the bit is set to logical-0. The resulting mask is applied as a parameter to the `test` instruction. To illustrate this, consider Listing 6.20 which shows an example of a *transfer protocol* that utilizes five channels: one synchronization channel `ctl`, one opcode channel `opc`, two operand channels `op1` and `op2`, and finally a result channel `res`. Assume that the handshake ports associated with each channel are allocated in the order of their declaration. Thus, the HP associated with the channel `ctl` is ϕ_0 and the one associated with the channel `res` is ϕ_4.

Listing 6.20: Example of a *transfer protocol*

```
DECLARE
    ctl : IN  CHANNEL( PUSH, 2P, SR, '0', '0', 0 );
    op1 : IN  CHANNEL( PUSH, 4P, SR, '0', '0', 4 );
    op2 : IN  CHANNEL( PUSH, 4P, SR, '0', '0', 4 );
    opc : IN  CHANNEL( PUSH, 4P, SR, '0', '0', 4 );
    res : OUT CHANNEL( PUSH, 4P, SR, '0', '0', 4 );
END
TEST
    SYNC ctl,
    opc <= "0000",
    op1 <= "0011",
    op2 <= "1011";
    SYNC ctl,
    opc <= "0100",
    op1 <= "0110",
    op2 <= "0011",
    res => "0011";
    res => "1001";
END
```

In the first concurrent transfer group only the first four channels are active. In the second group, all channels perform a handshake operation. Finally, in the last group only `res`, i.e., the fifth channel, is active. The generated code for this test sequence is given in Listing 6.21.

Listing 6.21: NoTePAD assembler code fragment for the test sequence of the considered *transfer protocol*

```
1       .text
EXEC_TEST:
        ...
        test hpmask=0x07
        test hpmask=0x17
6       test hpmask=0x10
```

The generation of instructions implementing *signal operations* is very similar. However, it is not possible to execute both types of signal operations simultaneously, since the input (comparison) and output (assignment) signal operations are realized by different instructions. In order to resolve conflicting situations, two means can be taken into account. On the one hand, one can define a default resolution approach that the assignment operations are executed before the comparison operation or vice versa. On the other hand, and this is probably the better option, one can resolve these conflicts during the generation of the protocol. Thus, instead of generating a group including both, signal assignments and signal comparisons, one should explicitly separate these activities in the test bench. This can be achieved by using either the `tp_sync()` or the `tp_wait()` procedures of the *channel simulation package*. The same applies to the combination of handshake and signal operations in one concurrent group. Since it is not possible in the current processor implementation to execute both activities at the same time, it is recommended to explicitly separate them into different concurrent groups. However, due to the lack of an exact timing in an asynchronous system, it is very unlikely that some handshake transfers have to take place exactly at the same time as a control signal operation. Therefore, this limitation is typically not critical. Apart from the mentioned restriction, several signal assignments or signal comparisons can be combined and executed simultaneously.

According to these considerations, the instructions realizing signal operations can be generated in a comparable manner as the one for handshake operations. Thus, for each concurrent group comprising signal operations, the indices of the control ports associated with the signals have to be determined. Using these indices a mask can be derived such that the bit e_i of the mask $e_0 \dots e_{c-1}$ (c is the number of supported control

pins) is set to logical-1 if i is one of the identified indices. Otherwise, the bit is set to logical-0. In the same way, the value of the signal is assigned to the bit vector $d_0 \ldots d_{c-1}$. Thus, let x be the value to be applied to the signal whose associated control signal is $\texttt{ci}_i$ or $\texttt{co}_i$, respectively. Then, d_i is set to x and e_i is set to logical-1. Depending on the signal operation to be performed, the resulting mask and value bit vectors are applied to either the comparison operation `ctri` or the assignment operation `ctro`. However, for a comparison operation an additional step is required for the translation. As defined, the test is treated as failed if the control input value does not match the expected value. In this case, the timeout routine shall be called whose start address is specified in the branch address register of the *sequencer*. To achieve this behavior, the value of the timeout register is simply set to 1 prior to the call of `ctri`. Thus, if the input signal does not match the expected value, then `ctri` immediately branches to the timeout routine. Finally, after the `ctri` instruction, the timeout register has to be restored to its previous value.

The last statements, which have to be translated, are the *wait statements*, *signal wait statements*, *timeout statements* and *timeout routine statements*. *Signal wait statements* are translated in a very similar way as *signal comparisons*. The only difference is that the timeout register is kept untouched. Consequently, the instruction waits for the signal to become equal to the specified value as long as the timeout is not exceeded.

Wait and *timeout statements* are translated in a different way. With recall to their definition in Section 4.3.2.3, these statements expect the specification of the time to wait. The translation of these statements is very similar, since they are mapped to the same mechanism implemented in NoTePAD.

Basically, both statements set the timeout register via calls of `setl` and `seth`. In case of the *timeout statement*, these are the only instructions to be generated. This is different for *wait statements*. For these statements, the additional `wait` instruction has to be called after setting the timeout register. Furthermore, the previous value of the timeout register has to be restored after the execution of the `wait` instruction. Basically, this can be achieved in two ways. One can use an additional register buffering the original value or one can set the timeout register again using `setl` and `seth`. Both possibilities result in the same overhead, since both require two additional instructions. In practice, for some cases, i.e., if either the lower or the upper 16 bits of the timeout register shall remain unchanged, then directly setting the timeout register can even be faster, since only one of the `setl` and `seth` instructions has to be called.

In order to determine the value of the timeout register according to the time t specified by the statements, the cycle time t_p of the processor is required. With this information, one can determine the corresponding number z of cycles to wait. Obviously,

the exact timing might not be met, since the precision is limited by the clock period of the processor. Accordingly, the wait time is approximated to the nearest multiple of the cycle period of the processor. Thus, the number z of cycles is determined as shown in Equation 6.1.

$$z = \begin{cases} z_t = {}^{t}/_{t_p},\ z_t \in \mathbb{N} & \text{when } t - z_t \cdot t_p < t - (z_t + 1) \cdot t_p \\ z_t = {}^{t}/_{t_p} + 1,\ z_t \in \mathbb{N} & \text{otherwise.} \end{cases} \tag{6.1}$$

However, there are some special cases for the correct translation of the *wait statements*. In order to match the required number of cycles to wait, the cycles needed for executing instructions have to be taken into account. This includes the register assignment and the execution of the `wait` instruction itself. Consequently, one has to subtract three cycles from z in order to gain the correct number to be written into the timeout register. However, if z is less or equal to three, then the described sequence of three instructions lead to an incorrect timing. To avoid this, the usual instruction sequence is replaced by $z \leq 3$ `noop` instructions. Obviously, in this case it is also not necessary to restore the timeout register.

If z is greater than three, then there are some special cases to be considered when restoring the timeout register. Naturally, the recovery of the original value is only necessary if the subsequent element in the *test sequence* does not manipulate the timeout register as well. On the other hand, assume that the next element in the sequence also implements a *wait statement*, where the number z of cycles to wait is $3 \leq z \leq 6$. In this case, the timeout recovery of the previous element can be redeemed with the number of cycles to wait in the next element.

Finally, *timeout routine statements* are translated again using a combination of `setl` and `seth`. In this case, the instructions are used to set the branch address register such that it points to the program address passed to the statement. The program address has to be specified via a label that defines the start of a routine of the generated code or a routine that has been externally defined. For the latter case to be applicable, the routine has to be written in assembler code and linked together with the other program files, when the program is translated to a binary.

An example of all these considerations is delivered in Listing 6.22. It shows a *transfer protocol* including *signal* and *channel assignments* as well as *wait* and *timeout statements*. For the translation of the protocol, it is assumed that the operating frequency of the processor is 200 MHz, thus, $t_p = 5$ ns.

The corresponding code for this protocol is illustrated in Listing 6.23. The first two instructions implement the *timeout routine statement*. These instructions set the value of the branch address register to the start address of the result upload routine. The next two

Listing 6.22: Sequence of *wait* and *timeout statements*

```
DECLARE
    rst : IN SIGNAL;
    opc : IN CHANNEL( PUSH, 4P, SR, '0', '0', 4 );
END
TEST
    SET TIMEOUT_ROUTINE END_OF_TEST;
    SET TIMEOUT 200 ns;
    opc <= "0000";
    WAIT FOR 10 ns;
    rst <= "1";
    WAIT FOR 24 ns;
    rst <= "0";
    WAIT FOR 15 ns;
    opc <= "1100";
END
```

instructions correspond to the first *timeout statement*. After that, a handshake on the opc channel is performed. Then, the two noop instructions implement the *wait statement* that cause the processor to wait for 10 ns. Then, the reset signal is set to logical-1. In the original *transfer protocol* this state is kept for 24 ns. But due to the clock period of 5 ns, the time is approximated to 25 ns. After this period, the reset signal is set to logical-0 again and the program execution is interrupted for 15 ns. However, before the handshake can be executed in the next step, the value of the timeout counter has to be restored which requires two cycles. These two cycles are subtracted from the required number of cycles to wait. According to the described procedure, this leads to one noop instruction and the two cycles for setting the timeout register. However, in the considered example an explicit window to wait is defined prior to the timeout definition. If this is not the case, the timeout recovery introduces an additional delay. In the current implementation this delay cannot be avoided. However, due to the asynchronous nature of the considered DUTs, such delays are typically negligible.

Listing 6.23: Assembler code for the sequence of *wait* and *timeout statements*

```
    .text
EXEC_TEST:
    ...
    ; Set timeout routine
    seth r2, (END_OF_TEST >> 16)
    setl r2, (END_OF_TEST & 0xffff)
    ; Set the timeout counter 200 ns
    seth r0, 0x0000
```

```
    setl r0, 0x0028
    ; Execute handshake
    test hpmask=0x0001
    ; Wait for 10 ns
    nop
    nop
    ; Set the reset signal
    sout mask=0x0001, value=0x0001
    ; Wait for 25 ns
    seth r0, 0x0000
    setl r0, 0x0002
    wait
    ; Set the reset signal
    sout mask=0x0001, value=0x0000
    ; Wait for 10 ns and set the timout counter to 100 ns
    nop
    seth r0, 0x0000
    setl r0, 0x0014
    ; Execute handshake
    test hpmask=0x0001
    ...
END_OF_TEST:
    ...
```

6.2.2.4 Test Finalization and Default Timeout Routine

After the test execution, some further steps are required to complete the test. This includes the flushing of the input FIFOs to ensure that all test results are written to the memory. After that, the DPs have to be disabled, such that they cannot issue further memory access requests. This is necessary to guarantee that the DPs do not issue invalid requests if the test program is reset. Furthermore, the test finalization is by default also used as timeout routine. Therefore, the mask register and the return address register are stored in the memory. This enables that the information about the cause of the timeout is accessible for the upload. Listing 6.24, shown in the next section, illustrates the required steps.

6.2.2.5 Result Upload

The last step of the translation process is the generation of the *result upload phase*. As mentioned previously, this phase is not directly part of the *transfer protocol*, but an

implicit step after the *test sequence*. In this phase, all results captured during the test have to be uploaded to the ETE via the dedicated interface in order to enable analysis of the test results.

As a first step in the phase, the ETE has to be synchronized with the TP. Thus, as mentioned in Section 4.2.2.2, the ETE has to be informed that the test is finished and that the TP is ready to upload the results. Therefore, the `halt` instruction is used. This instruction sets the `halt` output control signal of the TP to logical-1 and afterwards waits for the input control signal `enable` to be set to logical-1 by the ETE as well.

After synchronization, the upload of the test data can be started. This especially comprises the responses and fault signatures captured by the data ports. As defined previously, this data is aligned in arrays, i.e., one array for each DP. Thus, for each of the used DPs a sequence of instructions is required that transfers the array of data words to the ETE interface. The `dout` instruction of NoTePAD implements the desired functionality. As a prerequisite for this operation, the start address of the array has to be stored in the cycle counter and the end address in the timeout register. Again, the instructions `setl` and `seth` are combined to load these 32-bit addresses into the respective registers. These addresses are determined via labels in the assembler code. Finally, the program is stopped with an additional call of the `halt` instruction.

An example of the *result upload phase* for two data ports is shown in Listing 6.24.

Listing 6.24: Assembler code for uploading the results

```
    ; Forward declaration of labels related to the DPs
    .ref DP9_RDATA
    .ref DP9_RDATA_END
    .ref DP9_WDATA
    .ref DP9_WDATA_END

    .text
    ...
END_OF_TEST:
    ; Flush all data from the DPs to the memory
    flsh hpmask=0x0008
    ; Disable the data ports associated with the specified HPs
    tmrq hpmask=0x000f
    ; Load address for storage of timeout information
    setl r0, (MASK_AND_PC >> 16)
    seth r0, (MASK_AND_PC & 0xffff)
    ; Store the program address where a timeout has occurred
    slwd r3, r0
```

```
19      shwd r3, r0
        ; Increment address
        addu 2
        ; Store result mask
        slwd r4, r0
24      slwd r4, r0

    UPLOAD:
        ; Synchronize with ETE
        halt
29      ; Define start and end array addresses
        seth r0, (DP9_WDATA >> 16)
        setl r0, (DP9_WDATA & 0xffff)
        seth r1, (DP9_WDATA_END >> 16)
        setl r1, (DP9_WDATA_END & 0xffff)
34      ; Transfer the data to the ETE
        dout
        halt

        .data
39      ...
        ; Configuration for a data port
    CONF_OF_DP9:
        .word DP9_RDATA ; Address of the read-data array
        .word DP9_WDATA ; Address of the write-data array
44      .word (DP9_RDATA_END-DP9_RDATA >> 1) ; Number of words
        .word 0         ; Dummy to gain regular alignment

        ; Read-data array
    DP9_RDATA:
49      .word 0xffff6a81
        ...
    DP9_RDATA_END:
        ; Write-data array
    DP9_WDATA:
54      .bss DP9_WDATA, 256, 4
    DP9_WDATA_END:
    MASK_AND_PC:
        .bss MASK_AND_PC, 8, 4
```

6.2.3 Compiler for Generating Test Programs from Transfer Protocols

For the automatization of the program generation, a compiler tool, called `tpc`, has been developed. The tool implements the described mapping rules and generates assembler code for the NoTePAD architecture. It parses the entire protocol file and simultaneously builds up the statement tree which is traversed afterwards. This allows for the detection and cancellation of possible additional delays, e.g., introduced by *timeout statements*. The tool has various options to define architectural parameters and to control the generated output. The architectural parameters include the number of handshake and data ports, pins per data port etc. The output control options address, e.g., the separation of the *processor initialization phase* from the other phases, and the filling of the data port FIFOs. A more detailed description of the tool can be found in Appendix C.1.

"Forget mistakes. Forget failures.
Forget everything except what
you're going to do now and do it."

— William Durant

Chapter 7

Evaluation of the Concept

In this chapter, an evaluation of the entire concept is provided. Therefore, different aspects have to be considered. First of all, it has to be shown that the concept works in general. Therefore, the framework, including the proposed test processor and the respective tools, is applied to perform functional test of an asynchronous DUT. The selected design was implemented, simulated according to the pattern generation flow, and finally, connected with the TP to perform the tests. On the other hand, various parameters of the framework are evaluated in order to appraise its applicability in real test scenarios. The considered parameters include the speed of the execution of a program, the hardware requirements of the TP and other required resources, e.g., memory utilization of a program. Other classical properties, such as fault coverage, are not considered, since these properties are determined by the performed test in combination with a fault model, but not by the equipment executing the test.

7.1 Application of the Framework to an Asynchronous Device

As afore indicated, the first step of the evaluation is the application of the proposed framework to an asynchronous DUT. In this way, the general applicability of the approach to gain an elastic test of asynchronous handshake circuits is shown. Before showing the application of the framework, the DUT is introduced. After that, it is shown how the functional test programs were generated according to the proposed workflow.

For the evaluation, a NoTePAD implementation with 4 control inputs, 4 control outputs, 16 handshake and 16 data ports was used. Thereby, each of the data ports com-

prises 4 pins which results in 64 available data pins for realizing asynchronous channels. Furthermore, the TP was constrained to operate at a frequency of 200 MHz. This parameter was derived from the implementation of the NoTePAD architecture using a XILINX VIRTEX-5 VLX 155T FPGA. Finally, a demonstrator was built by combining the DUT and the test processor to one FPGA-design. This demonstrator was fully implemented, including synthesis, placement, and routing. Based on the resulting Verilog-netlist and the respective timing information of the demonstrator, various simulations were carried out that illustrate the execution of the test programs.

7.1.1 The Device-Under-Test

The design selected for the evaluation of the concept is an asynchronous QDI circuit implementing a 16-bit ALU. The ALU, whose structure is shown in Figure 7.1, implements logical operations such as bitwise-AND, -OR, -XOR, -INV, arithmetical operations like addition, subtraction and negation as well as a complex operation computing the greatest common divisor (GCD) of two unsigned 16-bit values. The circuit was implemented using dual-rail logic. According to this, every logical dual-rail gate was implemented using combinations of *C*-elements and OR-gates. Since real *C*-elements are not available in the FPGA, these were realized using the latch-based implementation shown in Figure 2.9d. Due to the completion detection of the dual-rail implementation which is sensitive to PVT variations, the circuit is expected to behave timing nondeterministic. Hence, it is an ideal DUT for showing the applicability of the provided TP solution.

The circuit has four interface channels: three input channels for two operands and the operation code as well as one output channel for the result. All channels operate using the 4-phase single-rail push protocol. As mentioned, the ALU internally works with dual-rail logic. However, the external interfaces use a 4-phase bundled-data protocol for the communication. Therefore, handshake protocol converters, as shown in Chapter B, are inserted at the boundary interfaces of the ALU. Furthermore, the DUT provides a reset signal `rst` for initialization purposes.

Basically, the ALU is composed of the mentioned protocol converters at the boundary interfaces, a processing unit (`alu`), a control unit (`alu_ctrl`) and a variety of multiplexers, demultiplexers and dual-rail registers arranged in a cyclic structure. The processing unit `alu` is the heart of the circuit. It calculates the results depending on its operands and control signals. For the coordination of the operation, the controller `alu_ctrl` delivers the control signals for the processing unit and for the other components. Therefore, it reads the chosen opcode and the value of the operands to set the values of the control signals accordingly.

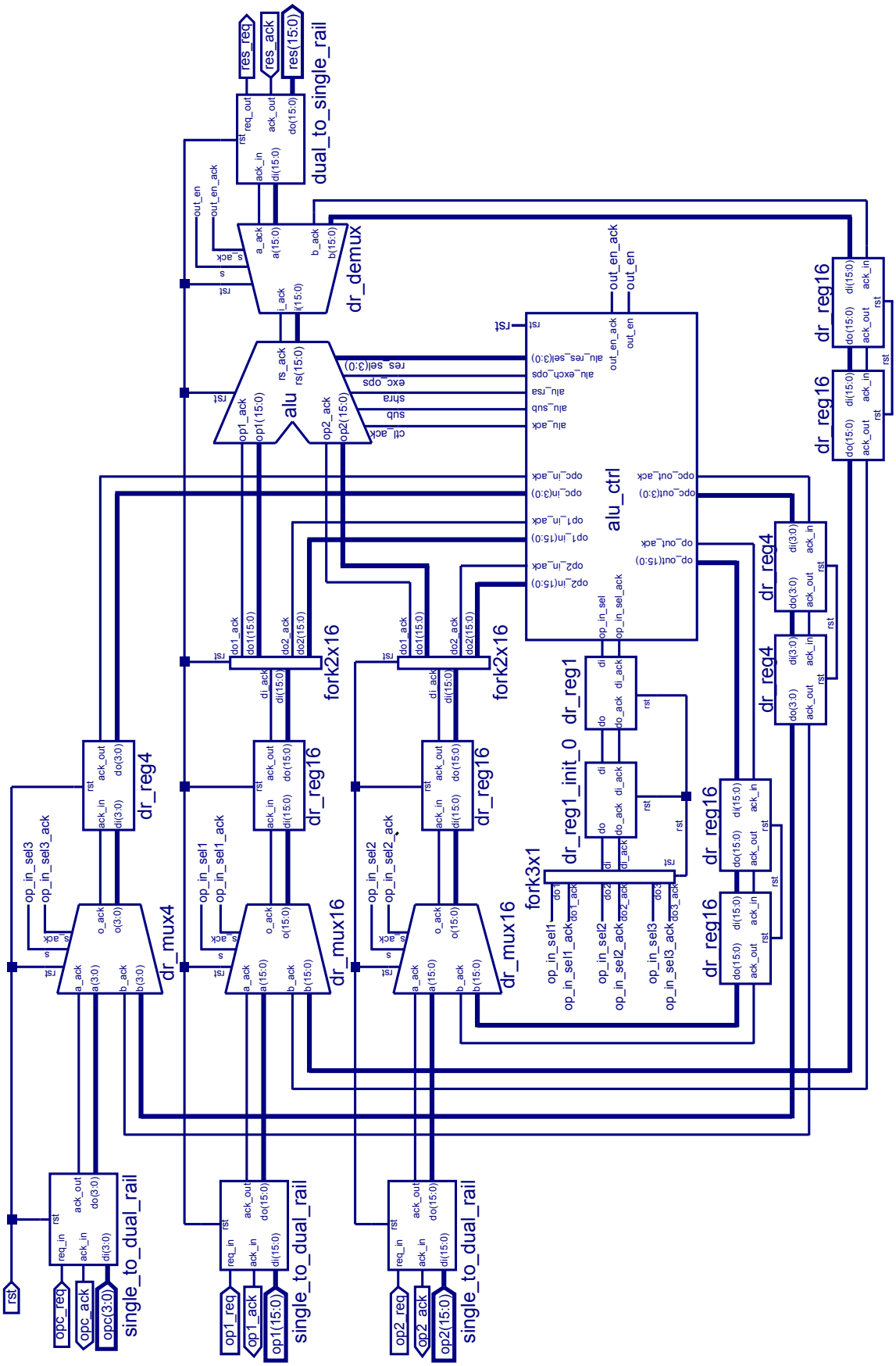


Figure 7.1: The design-under-test: an asynchronous 16-bit ALU

In case of logical or standard arithmetical operations, the ALU works as a pipeline. Thus, the opcode and the operands propagate from the primary inputs to the processing unit and to the controller. The controller then delivers the control signals to the processing unit which calculates the result. Finally, the result propagates to the output interface. For the computation of the GCD, the Euclidean Algorithm, as given in Listing 7.1, is used. As opposed to the standard implementation, e.g., shown in [Sparsø 2001], the structure is optimized such that the two comparisons ($a < b$ and $a \neq b$) of the operands a and b are performed by one comparator that is part of the controller. For the calculation of the difference between the operands, the processing unit is configured to calculate $r_t = \max(a, b) - \min(a, b)$. The minimum and maximum is determined via exchanging the operands depending on the comparison result delivered by the controller. Afterwards, the intermediate result r_t is fed to a demultiplexer that either forwards the result to one of the registers within the feedback loop or directly to the output. Thus, if a and b are equal, then r_t is fed to the output. Otherwise, r_t is written into the lower register of the feedback loop. In this case, the inner register within the feedback loop receive the other operand, i.e., $\min(a, b)$. Furthermore, the register in the middle of the loop receives the opcode to ensure that this data is available for the next iteration of the algorithm. This is repeated until both operands are equal.

Listing 7.1: Euclidean Algorithm for computing the GCD

```
function gcd(a, b : ℕ)
begin
  while a ≠ b do
    if a > b then
      a := a − b;
    else
      b := b − a;
    end if;
  done;
  return a;
end function;
```

7.1.2 Demonstrator

To illustrate the functionality of the test processor, a demonstrator was created by combining the considered asynchronous ALU with the NoTePAD processor implementation. The resulting design was integrated into a Xilinx Virtex-5 VLX 155T FPGA platform. The memories for the processor were implemented using integrated Block RAMs

(BRAMs) of the FPGA. For the program memory, a block of 1024×24-bit data words has been used. The data memory was realized using an 8192×16-bit BRAM.

All ports of the DUT were connected with the according ports of the processor to emulate a real test environment. For this interconnection, the pin information was derived from the resource allocation performed during the compilation of the *transfer protocol*. This information is described in a file (see Section D.5 in Appendix D) generated by the *transfer protocol compiler* `tpc`. Figure 7.2 shows the resulting block diagram with all interconnections.

7.1.3 Test Program Generation

Corresponding to the workflow, the considered DUT needs to be simulated in order to generate the test program. For this, either the RTL description or, if available, the netlist with corresponding timing information can be used. In this particular case, the netlist in combination with the respective timing information in Standard Delay Format (SDF) was used for the simulation. As a preliminary step of the simulation, the test bench has

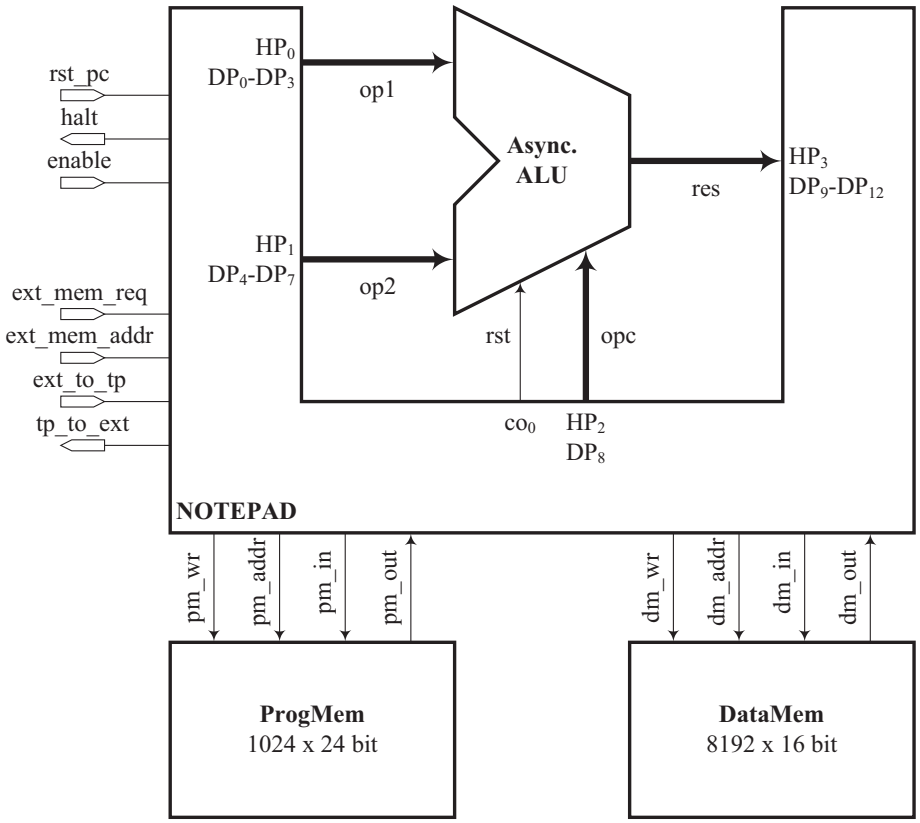


Figure 7.2: The demonstrator: NoTePAD connected with the asynchronous ALU

been adapted to use the *channel simulation package* proposed in Section 6.1.

Listing 7.2 shows the declarative part of the resulting test bench, whereas the entire test bench can be found in Appendix D.1. As the very first step in the test bench, the package is imported. Then, the listing continues as usual with the declaration of the test bench entity and the considered DUT. Afterwards, the channel and signal resources are declared that are used to apply and receive the patterns. In the considered scenario, a reset signal `rst` and the mentioned interface channels of the DUT are declared, i.e., `opc`, `op1`, `op2`, and `res`.

Listing 7.2: Declarative part of the test bench for the ALU

```
-- Include the test bench package
use WORK.tp_bench_pack.ALL;

entity tb_asynch_alu is
end tb_asynch_alu;

architecture tb_asynch_alu_arch of tb_asynch_alu is
    -- The asynchronous DUT
    component asynch_alu is
       port (
            rst     : in  std_logic;
            opc     : in  std_logic_vector (3 downto 0);
            opc_req : in  std_logic;
            opc_ack : out std_logic;
            op1     : in  std_logic_vector (15 downto 0);
            op1_req : in  std_logic;
            op1_ack : out std_logic;
            op2     : in  std_logic_vector (15 downto 0);
            op2_req : in  std_logic;
            op2_ack : out std_logic;
            res     : out std_logic_vector (15 downto 0);
            res_req : out std_logic;
            res_ack : in  std_logic );
    end component;

    -- Declaration of the reset control signal
    signal rst : TP_SIGNAL  := tp_init_signal("rst", INPUT);
    -- Declaration of the opcode, operand, and result channels
    signal opc : TP_CHANNEL :=
        tp_init_channel("opc",  4,  INPUT, PUSH, FOUR, SR);
    signal op1 : TP_CHANNEL :=
        tp_init_channel("op1", 16,  INPUT, PUSH, FOUR, SR);
    signal op2 : TP_CHANNEL :=
        tp_init_channel("op2", 16,  INPUT, PUSH, FOUR, SR);
    signal res : TP_CHANNEL :=
        tp_init_channel("res", 16, OUTPUT, PUSH, FOUR, SR);
```

Afterwards, the DUT is instantiated as shown in Listing 7.3. At this point, the signal and channel resources are connected with the ports of the DUT.

Listing 7.3: DUT instantiation

```
uut : asynch_alu
port map (
    rst     => rst.data(0),
    opc     => opc.data(3 downto 0),
    opc_req => opc.req,
    opc_ack => opc.ack,
    op1     => op1.data(15 downto 0),
    op1_req => op1.req,
    op1_ack => op1.ack,
    op2     => op2.data(15 downto 0),
    op2_req => op2.req,
    op2_ack => op2.ack,
    res     => res.data(15 downto 0),
    res_req => res.req,
    res_ack => res.ack );
```

According to the desired structure of the test bench described in Section 6.1.1, a separate control process `ctrl_proc` is introduced as shown in Listing 7.4. The first activity in this process is the creation of the protocol file. Afterwards, the signal and channel resources are registered. Then, after all resources were registered the *test sequence* needs to be initiated. To this end, the `tp_start_test_sequence()` procedure is called. Then, the DUT is initialized by generating a logical-1 pulse on the reset signal. Finally, the process waits until all channels are closed before the *transfer protocol* is completed via executing `tp_close_transfer_protocol()`.

Listing 7.4: Control process

```
ctrl_proc : process
begin
    -- Create the protocol file
    tp_open_transfer_protocol("async_alu");
    -- Register the signal and channel resources
    tp_register_resource(rst);
    tp_register_resource(op1);
    tp_register_resource(op2);
    tp_register_resource(opc);
    tp_register_resource(res);
    -- End declaration section and start the test sequence
```

```
        tp_start_test_sequence;
        -- Initialize the DUT via setting the reset signal
        tp_wait( 10 ns );
        tp_assign(rst, "1");
        tp_wait( 100 ns );
        tp_assign(rst, "0");
        tp_wait( 10 ns );
        -- Wait until all channels are closed
        wait until tp_is_closed(op1) and tp_is_closed(op2)
               and tp_is_closed(opc) and tp_is_closed(res);
        -- Close the tranfer protocol file
        tp_close_transfer_protocol;
        wait;
    end process;
```

The channels themselves are handled by individual processes as shown in Listing 7.5. Each of these processes starts with opening the respective channel using the procedure `tp_open_channel()`. Thereby, the processes that handle input channels call `tp_open_channel()` with the initial data value as parameter. For the output channel, this parameter is not required and, therefore, omitted. Afterwards, all these processes wait for the reset signal to become inactive in order to ensure that the circuit is initialized and ready to process data. Therefore, there is no need to first wait for the start of the *test sequence* via the call to `tp_test_sequence_started()`. Then, all input channels start with transferring data. In comparison to that, the process `proc_res` first waits until all input channels have performed at least one data transfer using the call to the *wait procedure* `tp_wait()`. This ensures that the first data transfer on the channel `res` occurs after the first data transfers on the input channels in the *transfer protocol*. Therefore, it is necessary to declare an array that includes the considered channels. After all input channels have performed a handshake, the process listens for activity on the output channel and performs the data transfer. In the given example, this is done seven times according to the number of performed operations[1]. When all data tokens have been sent and received, the individual channels are closed.

Eventually, the created test bench is simulated in order to generate the *transfer protocol*. Therefore, a standard logic simulator, such as Mentor Modelsim or Xilinx ISim can be used. The corresponding waveform of the simulation is illustrated in Figure 7.3. According to the behavior described in the test bench, the simulation starts with the pulse on the reset signal `rst`. Afterwards, data tokens are applied to the input channels of the

[1] Remark: Not all input tokens are shown in the example to save space.

Listing 7.5: Channel control processes

```
-- Process handling the opcode
proc_opcode : process
begin
    tp_open_channel(opc, "0000");
    wait until rst.data(0 downto 0) = "0";
    tp_send(opc, op_and);
    ...
    tp_close_channel(opc);
    wait;
end process;

-- Process handling the first operand
proc_op1 : process
begin
    tp_open_channel(op1, "0000000000000000");
    wait until rst.data(0 downto 0) = "0";
    tp_send(op1, "0000000000000001");
    ...
    tp_close_channel(op1);
    wait;
end process;

-- Process handling the second operand
proc_op2 : process
begin
    ...
end process;

-- Array of all input channels
input_channels <= opc & op1 & op2;
-- Process handling the result
proc_res : process
    variable result : std_logic_vector(15 downto 0);
begin
    tp_open_channel(res);
    tp_wait(input_channels);
    for i in 0 to 6 loop
        tp_recv(res, result);
    end loop;
    tp_close_channel(res);
    wait;
end process;
```

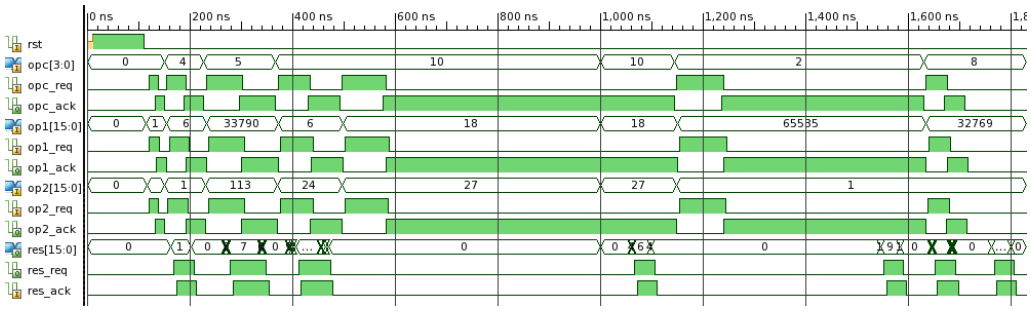


Figure 7.3: Simulation of the test bench with underlying netlist and timing of the ALU

DUT which afterwards delivers the responses of the chosen operations. In the considered example, the first three operations are a bitwise AND operation, an addition, and a subtraction. The fourth and fifth operations compute the GCD of 6 and 24 as well as of 18 and 27. These operations require considerable amount of time which is indicated by the long time window between the respective handshakes.

As a result of the simulation, the *transfer protocol* was created. The generated protocol is shown in Listing D.2 which can be found in Appendix D. In the next step, the protocol was translated using the *transfer protocol compiler*. Thereto, the defined parameters (frequency, number of ports etc.) of the TP were passed as options to the tool. The result of the compilation is provided in Listing D.3 also located in Appendix D. Afterwards, the assembler program was translated and linked using the dedicated processor tools (assembler and linker). The created binary file was, in turn, translated to memory map files which include the ASCII-descriptions of the memory content.

7.1.4 Test Results and Further Optimizations of the Generated Program

For the simulation of the demonstrator, the memory blocks were preinitialized with the data from the afore created memory map files. Afterwards, the demonstrator was synthesized, placed, and routed to gain the netlist and the respective timing information required for the simulation. As a final step, the SDF-file was edited such that the setup and hold timing checks of the first flip-flop of the synchronizers are disabled. This is necessary to emulate the metastability filtration capabilities of the synchronizers.

Figure 7.4 shows the resulting waveforms of the simulation. At first glance, these waveforms are very similar to the ones shown in Figure 7.3, which were gained from the initial simulation for creating the *transfer protocol*. However, on closer inspection one can notice little differences at the beginning of the *test sequence*. These differences are more emphasized in a zoomed view shown in Figure 7.5 and Figure 7.6. For example,

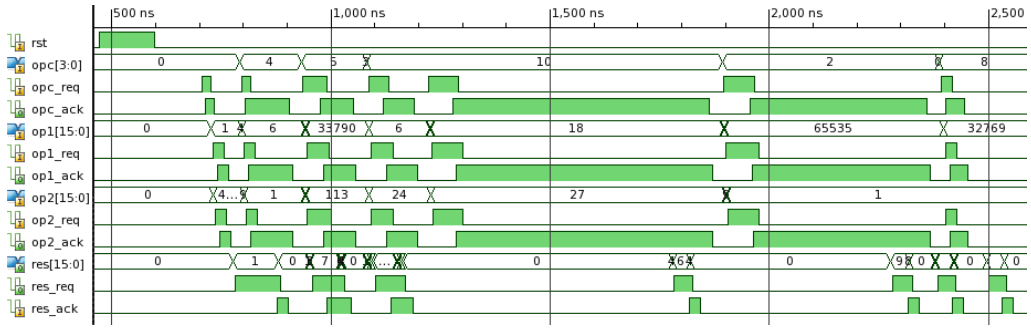


Figure 7.4: Simulation of the demonstrator executing the functional test program

the time window between the release of the reset signal and the first handshakes is much longer than the one in the initial simulation. Furthermore, the first handshakes on the input channels are sequentially initiated by the test processor. These effects are caused by the fact that the patterns were not completely loaded into the FIFO buffers of the data ports when the first `test` instruction is executed. Hence, the individual transfers are delayed until all associated data ports of a channel have received the patterns from the memory. In the considered example, the channel `opc` is the first that is ready to perform the transfers. This is because the channel `opc` is realized using one data port only which is filled with data first. Therefore, the transfer on this channel is executed first, followed by transfers on the channels `op1` and `op2`.

In principle, this effect is not critical for an asynchronous DUT, since its *event-driven* behavior compensates such irregularities. Nevertheless, this behavior can be prevented by filling the output FIFO buffers prior to the test execution. To this end, the *transfer protocol* is retranslated via invoking the *transfer protocol compiler* with the option to fill the data ports. This option inserts the `fill` instruction into *test preparation phase* of the test program. Afterwards, the new program was translated and simulated in the same way as before. Figure 7.7 shows the waveforms of the simulation. As can be seen, all handshakes on the input channels are initiated simultaneously as it is the case in the initial simulation shown in Figure 7.5.

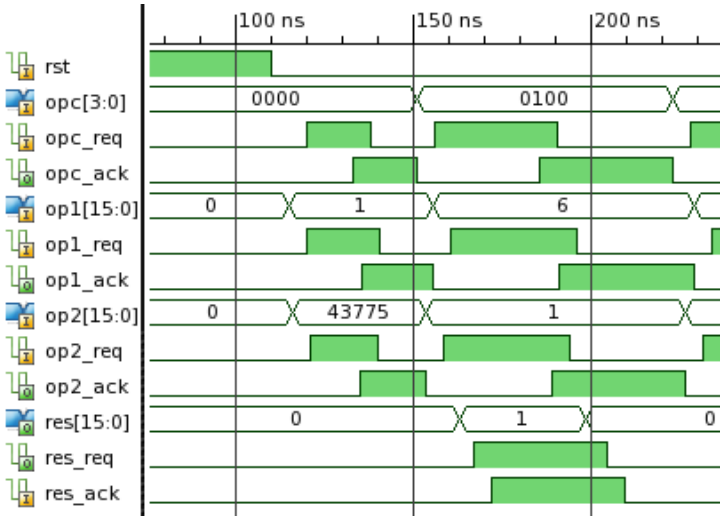


Figure 7.5: Start of the initial ALU simulation

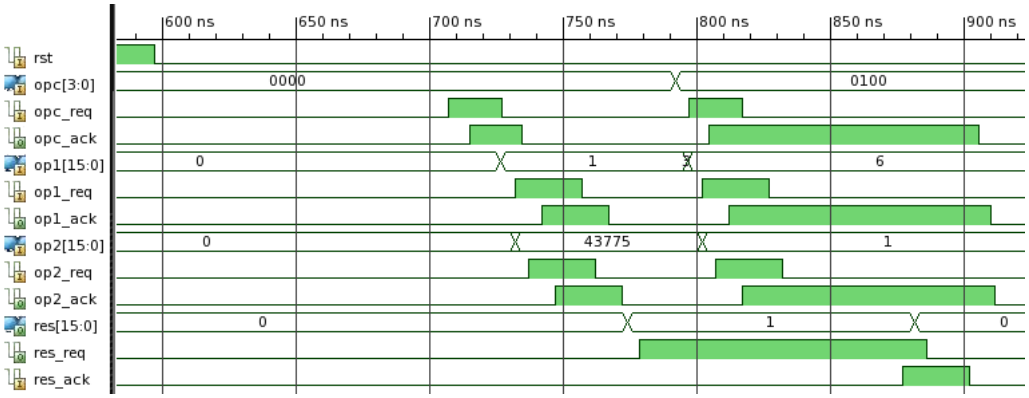


Figure 7.6: Start of the test sequence without preloading the data ports

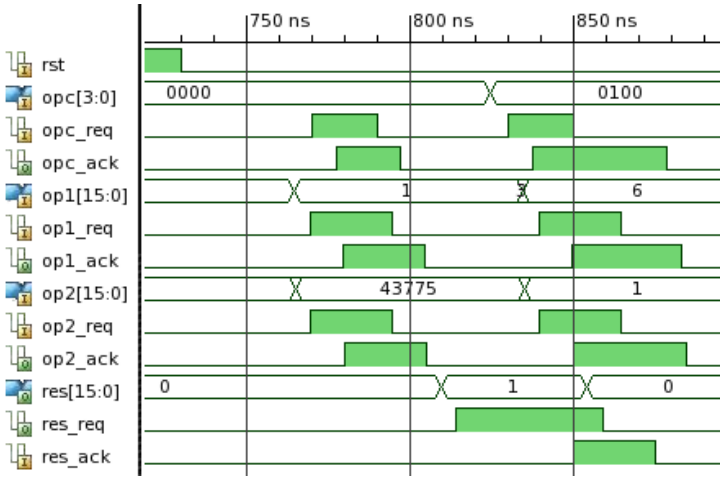


Figure 7.7: Simulation of the program with preloaded data ports

7.2 Evaluation of the Processor Implementation

Apart from its general applicability, the provided approach has to be evaluated in terms of its feasibility in real test scenarios. Therefore, the targeted processor architecture needs to be analyzed with respect to hardware requirements, operating frequency, and derived properties such as test execution time for given sets of transfers.

7.2.1 Hardware Requirements of the FPGA Implementation

At first, several implementations of the NoTePAD architecture were created and evaluated with respect to their hardware requirements. For these implementations, the number of handshake ports and the number of control inputs and outputs were fixed to 16, 8, and 8, respectively. As with the demonstrator, the XILINX VIRTEX-5 VLX 155T FPGA was used as target platform. This FPGA provides 97,280 registers (flip-flops/latches) and Lookup Tables (LUTs), 640 input/output buffers (IOBs), 424 18Kbit BRAMs. To gain maximum performance results, the clock frequency was set to 250 MHz for the synthesis. In order to obtain a well balanced MAC, the number of request interfaces of the MAAs was set to 2 for the implementations with 32 and 128 data ports. In all other cases, the MAAs were configured to have 4 request interfaces.

Table 7.1 shows the hardware requirements gained from the synthesis using XILINX ISE 13.3. The first column lists the total number of data pins that are available for the data part of the asynchronous channels. This number is determined by the number of data ports and the number of pins per data port which are given in parenthesis. The other columns show the FPGA primitive requirements of the individual NoTePAD implementations including the registers, LUTs, IOBs, and 18Kbit BRAMs. The table lists

Table 7.1: Results of the FPGA implementation of the NoTePAD architecture with varying number of data pins

Data Pins		max. Freq.	# Registers	# LUTs	# IOBs	# BRAMs
64	(16/4)	234.35 MHz	6790 (7%)	10462 (11%)	370 (54%)	32 (8%)
	(32/2)	200.00 MHz	13906 (14%)	19641 (20%)		64 (15%)
	(64/1)	202.38 MHz	24360 (25%)	33985 (35%)		128 (30%)
128	(32/4)	214.73 MHz	13874 (14%)	19783 (20%)	434 (64%)	64 (15%)
	(64/2)	201.93 MHz	24296 (25%)	33824 (35%)		128 (30%)
	(128/1)	142.32 MHz	53082 (54%)	67224 (69%)		256 (60%)
256	(64/4)	203.79 MHz	24232 (25%)	34720 (36%)	562 (83%)	128 (30%)
	(128/2)	160.74 MHz	52954 (54%)	67550 (69%)		256 (60%)

the absolut number of used components and the utilization related to the resources of the target FPGA.

As expected and confirmed by the illustrated results, the number of data ports strongly affects the hardware requirements of the NoTePAD implementation in terms of logic components (flip-flops, LUTs) and BRAMs. Furthermore, the increase of the number of pins per port affect almost only the total number of data pins. This is shown by the implementations having 64 data ports. Therefore, increasing the number of pins per port is a good measure to increase the total number of data pins without increasing the logic requirements. However, with respect to this parameter one has to trade between the total number of data pins, hardware requirements, flexibility, and performance. As already pointed out, increasing the number of pins per port potentially results in wasted pins and possibly memory space. Furthermore, the performance might be affected. For this, consider that the data port FIFOs have a fixed size. In the provided architecture each FIFO buffer can store the data for 64×16 transfers for one pin. Increasing the number of pins per data port obviously reduces the total number of transfers buffered by one FIFO. Now, consider a scenario where a large number of channels are constantly active. In this case, it might happen that one of the buffers runs empty, since the transfer data might not be exchanged fast enough between the active data ports and the data memory. This might cause bottlenecks due to data gaps in the execution of the program. As a countermeasure, the number of pins per port can be reduced while the number of data ports is increased. This leads to a processor design that is able to buffer data for more transfers. On the other hand, such an implementation has higher hardware requirements with respect to registers, LUTs, and BRAMs. This is shown by the TP implementations which have a constant number of data pins, but different numbers of data ports and pins per port.

One contradictory point with respect to the implementation results is related to the maximum clock frequency. In some cases, the frequency increases even though the implementation is more complex than others. As an example, consider the implementation pair of row 2 and 7 (32/2 and 64/4). In other cases, e.g., consider row 6 and 7 (128/1 and 128/2), the maximum clock frequency increases although the number of pins per data ports is increased. This is probably the result of the different placement and routing of the resulting circuits. However, a deep analysis of the cause of these effects is out of the scope of this work.

Nevertheless, there is one additional point with respect to the maximum frequency. It can be observed that the performance of the implementations with 128 data ports is significantly reduced in comparison to the other implementations. The cause of this performance loss is related to the implementation of the *port switch*. There, the status

information of 128 ports are combined to one status signal for each handshake port. The combinational path of the resulting logic limits the performance. As indicated before, one can implement a pipelined computation of the status signal to counteract this effect. However, this further increases the number of cycles for the signal propagation from the data ports to the handshake ports and, therefore, the number of cycles for the handshake execution as well.

Finally, to further estimate the value of the implementations, one has to consider the number of pins typically provided by commercial testers. Basically, this number is determined by the architecture of the test system and its purpose. Thus, the number of pins strongly varies between different test systems. However, 256 pins are a representative number in average. According to this, the version with 64 data ports and 4 pins per port is a suitable implementation, since it can operate at 200 MHz and provides a sufficient number of data pins.

7.2.2 Test Execution Properties

Besides the hardware requirements, some more parameters are relevant for the evaluation of the implementation. These parameters are related to the executed tests: the transfer rate per channel and the memory requirements. Therefore, two different types of experiments were performed for the analysis of the test execution. First, the different handshake protocol implementations were evaluated with respect to the number of cycles required for completing a single data transfer. To this end, a program including one channel for each supported handshake protocol has been created and simulated. As shown in the waveforms of Figure 7.8 the program sequentially executes four handshakes on each channel. The DUT was emulated by a test bench. Thereby, all signals driven by the test bench responded to the events of the TP with a delay of 1 ns. Since this delay is less than the clock period of the TP, the TP is the limiting factor during the handshake execution. Therefore, the minimum number of cycles, as well as the maximum transfer rates for the respective transactions, can be derived from this simulation. Table 7.2 shows the resulting numbers and the transfer rates at an operating frequency of 200 MHz. The results confirm the theoretical considerations made in Section 5.2.3. Thus, the minimum number of cycles for a transfer is mainly determined by the number of cycles required for the signal synchronization (via the synchronizers) and by the number of cycles for exchanging the information between the handshake and the data ports.

The next type of experiments considers the performance of the proposed test system in real scenarios for the asynchronous ALU. Therefore, several programs were created

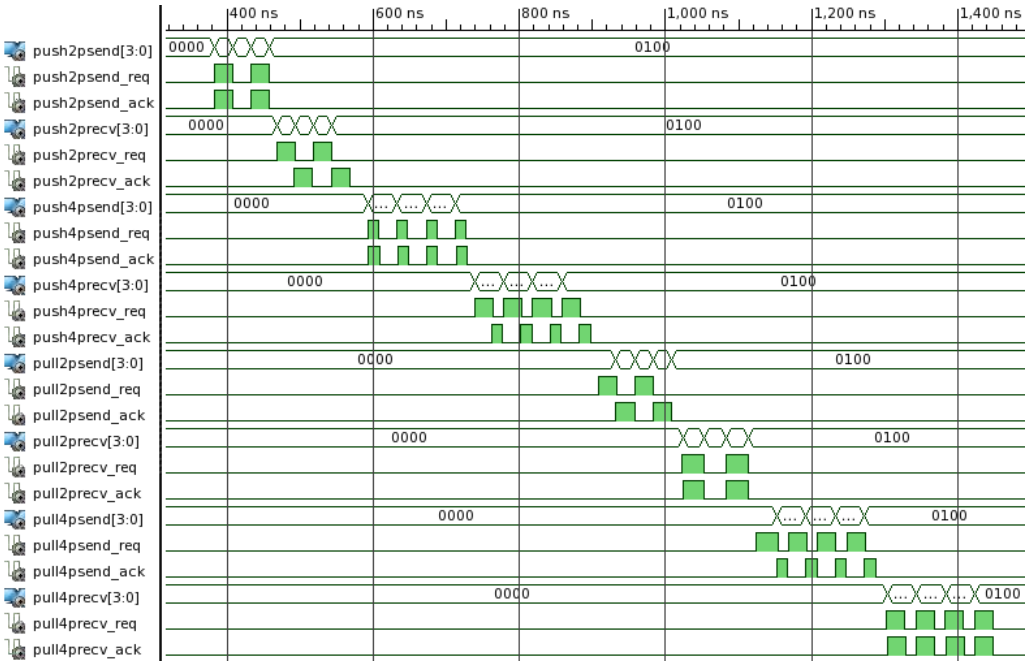


Figure 7.8: Simulation of all supported handshake protocols

Table 7.2: Minimum number of cycles for one handshake and respective transfer rates

	push		pull	
Direction	2-phase	4-phase	2-phase	4-phase
send	5 (40 MS/s)	8 (25 MS/s)	5 (40 MS/s)	8 (25 MS/s)
receive	5 (40 MS/s)	8 (25 MS/s)	6 (33 MS/s)	8 (25 MS/s)

that implement random tests for the considered DUT. These tests were executed using the same NoTePAD implementation as used in Section 7.1. Thus, the TP has 16 data ports each comprising 4 pins, and 16 handshake ports. In this configuration only 4 HPs were actually used. For this processor implementation, three sets of programs were created. These programs include all operations of the ALU except the GCD[2]. One set comprises 1k, the next 10k, and the last 100k ALU operations, respectively. Since each operation requires four transfers to complete, the individual programs include 4k, 40k and 400k data transfers. Each of these sets comprises one test program with and one program without the filling of the data ports with patterns prior to the test execution. In order to determine their runtime, the resulting programs were simulated using the RTL-model of the demonstrator. The results of all simulations are shown in Table 7.3 and

[2]The GCD operation may require considerable amount of time. Therefore, it is excluded from the tests to not distort the performance results

Table 7.3: Program runtime

# Operations	data port filling	# Cycles total	init.	test	upload	Exec. Time @ 200 MHz
1000	yes	9507	943	8034	530	47.54 µs
	no	8653	76	8047	530	43.27 µs
10000	yes	86007	943	80034	5030	430.04 µs
	no	85153	76	80047	5030	425.77 µs
100000	yes	851007	943	800034	50030	4255.04 µs
	no	850153	76	800047	50030	4250.77 µs

Table 7.4, respectively.

The former table shows the number of cycles consumed by the individual phases of the programs as well as the required time for executing the entire programs. Again, the operating frequency of the processor was set to 200 MHz. As expected, filling the data ports increases the required time for the test preparation phase, but reduces the time of the *test execution phase*. The obvious reason for this is that the patterns are already available when the test has started. However, for the considered DUT, the additional overhead does not justify the speedup of the test execution. It can be seen that the test execution is accelerated by only 13 cycles for all program sets. Against this benefit is the fact that filling the FIFOs constantly requires 867 cycles for all three program sets. This results in 943 cycles in total for the entire initialization phase compared to 76 cycles for the programs that renounce the filling of the buffers.

In consideration of the test execution, the table shows that approximately 8 cycles are required for executing one operation. This includes the application of the stimuli and the receipt of the response. As shown previously, the limiting factor is the handshake execution itself.

In order to illustrate the benefit of the test processor, consider the same test scenarios realized with a boundary scan. Therefore, assume that one scan chain connects all boundary registers of the considered DUT. Accordingly, the scan chain is composed of 52 scan cells. Thus, to perform one operation at least 37 cycles are required:

- $2 \times 16 + 4$ cycles are required to shift the patterns into the operand and the opcode registers, while concurrently scanning out the result register, and

- at least 1 cycle is required for executing the operation. Basically, this number might be even larger, since one has to assume the worst case timing of the DUT.

Consequently, the functional test using NoTePAD is 4.625 times faster than the boundary scan approach. However, this only applies if the scan test also runs at 200 MHz. If one assumes a more realistic scan clock frequency of approximately 50 MHz, then the test using NoTePAD is executed more than 18.5 times faster than the boundary scan approach.

Furthermore, to show the performance improvements in comparison to the solution provided in [Zeidler 2012a], comparable programs for this TP architecture were created. The programs include 4k and 40k data transfers, respectively, which were executed with a target frequency of the TP of 100 MHz. After simulation of these programs it turned out that the programs need 117,440 and 1,173,645 cycles to complete, respectively. Thus, each sequentially executed transfer requires approximately 30 cycles. This results in approximately 120 cycles for one operation of the ALU. In consideration of the operating frequency, the NoTePAD architecture is approximately 30 times faster than the solution provided in [Zeidler 2012a].

Besides the execution time, also the memory requirements are of importance, since these limit the amount of transfers within a single functional test. Furthermore, the size of the program also affects the test time, since the program including the patterns has to be loaded to the memory prior to the test. Accordingly, the program and the data memory requirements of the above mentioned programs were determined. The results are shown in Table 7.4. The program size is given in number of instructions where each instruction word comprises 24 bits. The programs with and without filling the data port FIFOs differ in only one instruction. Therefore, the table shows the results for the programs without filling the FIFOs only[3]. The data memory requirements are identical for each program pair. Finally, the time required for the program upload is determined by the number of cycles needed for storing the program and the data memory content. Thereby, each instruction is written at once in one cycle into the program memory, while two bytes of the data are written into the data memory in one cycle.

Similar to the test execution time, the memory requirements can be compared with the TP solution provided in [Zeidler 2012a]. Due to the software implementation of the handshake signalling, the programs are much larger compared to the programs of NoTePAD. This obviously also affects the time for uploading the programs. Thereby, each 32 bit instruction is written in one cycle into the program memory. The data is written in blocks of 4 bytes per cycle into the data memory. The program upload runs at 100 MHz. Finally, as indicated by the results shown in Table 7.5, the time for loading the entire program is approximately 3.5 times larger than for the NoTePAD solution.

[3] +1 indicates that the number of cycles for the program with filling the buffers has to be increased by one.

Table 7.4: Memory requirements of NoTePAD and respective time for the program upload

# Operations	# Instructions	Data Memory Requirements	Program Upload
1000	1064 (+1)	15,260 B	43.47 µs
10000	10064 (+1)	150,248 B	425.94 µs
100000	100064 (+1)	1,500,224 B	4250.88 µs

Table 7.5: Memory requirements of the TP provided in [Zeidler 2012a] and respective time for the program upload

# Operations	# Instructions	Data Memory Requirements	Program Upload
1000	11542	16,068 B	155.59 µs
10000	112238	160,068 B	1522.55 µs

7.3 A Test Scenario

Based on the evaluation of the test execution, a test scenario can be derived for the considered ALU. Suppose that 10k devices shall be tested by executing the program that performs 10k ALU operations. The upload of the program to the memories of NoTePAD requires 425.94 µs. This is negligible, since it is done only once before the test. Afterwards, the processor has to be configured, which is also only done once before the tests. Then, 4257.70 ms are required to test all devices, resulting in approximately 4.258 s for the entire test including the upload of the program. The test of all devices using the TP provided in [Zeidler 2012a] would require 117.37 s including the upload of the program. In comparison to that, consider the boundary scan approach again under the assumption that 37 cycles are needed for one operation. The scan clock is supposed to be 50 MHz. Then, approximately 74 s are required for executing the test for all devices. Note that the time for the change of the device, e.g., consumed by a wafer prober, is neglected.

However, one has to consider that typically various tests are executed. Then, the program has to be loaded to the test processor memory prior to each test. Suppose that 10 tests, each comprising 10k ALU operations, shall be executed for each device. In this scenario, the test of all devices using NoTePAD requires approximately 85 s, whereas the test using a boundary scan requires 740 s. Finally, the test using the solution in [Zeidler 2012a] would require 1326 s.

"Look at a day when you are supremely satisfied at the end. It's not a day when you lounge around doing nothing; it's when you've had everything to do, and you've done it."

— Margaret Thatcher

Chapter 8

Conclusions

This chapter summarizes the work and the major achievements. Furthermore, the limitations of the provided solutions are identified and discussed. Finally, an outlook about possible extensions and improvements is given that can be the subject of future activities.

8.1 Summary of the Work

This work is concerned with the realization of functional tests of asynchronous devices. The general issue in this context is related to the event-driven timing behavior of asynchronous handshake circuits that cannot be handled by common hardware testers. This applies to both, cycle-based and event-based testers currently available, since all these systems are basically not designed to react to signal events generated by the DUT. As a result of the lack of this property, an asynchronous handshake-based communication cannot be established between a DUT and a standard tester. This leads to problems with respect to functional tests.

This work tackles this issue and proposes a methodology based on a special test processor supporting asynchronous handshake communication. The key feature of the processor is a programmable port component that provides generic interfaces for the communication with asynchronous devices. This port component comprises two types of ports. The first type of ports are special handshake ports that implement various types of asynchronous handshake protocols. The second type of ports is used to realize the

data part of an asynchronous channel. These ports are designed to send, receive and compare data. In order to provide a generic solution, the ports can be combined in a configurable manner. This allows that arbitrary channels with different properties can be established between the DUT and the test processor.

Apart from the test processor itself, a workflow is proposed that describes how the test processor can be used in a standard test environment. One essential part of this workflow is the generation of programs for the provided test processor, which realize the desired functional tests of the DUT. The methodology for the program generation is based on the creation of a special test pattern file during logic simulation of the DUT. This so-called *transfer protocol* includes the description of the dataflow on the base of channel transfers rather than signal transitions. By this, the static timing of conventional test patterns is resolved which allows elastic tests. After its generation, the *transfer protocol* can be compiled to gain the desired processor program.

Based on the theoretical concept, a practical implementation is discussed. Therefore, a complete framework including a test processor implementation and respective tools for the generation of the test programs is presented. The provided processor is especially designed for the extension of commercial test equipment, but it can be used as a stand alone test system as well.

In order to show the feasibility of the approach, the entire framework was applied to an asynchronous DUT implementing a quasi-delay-insensitive ALU. It is shown how the individual steps of the framework are carried out to generate and execute the functional tests of the selected DUT. For the illustration of the functionality, a hardware demonstrator was created that embeds the DUT into the TP environment. This demonstrator was implemented using a XILINX VIRTEX-5 VLX 155T FPGA. After the implementation, the gained netlist was simulated with corresponding timing information. The results show that the concept works fine and that the tests exhibit the desired elastic timing behavior.

Besides the application of the framework, additional experiments have been carried out to evaluate further parameters of the infrastructure. This includes the required hardware resources of the processor as well as the runtime and memory utilization of different test programs. By this, the practical suitability of the approach is demonstrated. The experiments have shown that the proposed processor architecture can be implemented using existing FPGA hardware platforms and a maximum clock frequency of 200 MHz can be reached. Hence, the processor overcomes the performance limitations of alternative approaches and also fulfills demands, such as a flexibility, and controllability. This makes the proposed concept a suitable mechanism for performing functional tests of asynchronous circuits.

8.2 Summary of the Achievements

The theoretical part of the work includes the following achievements:

- ➢ a generic concept of a test processor for performing functional tests of asynchronous handshake interface devices,
- ➢ a workflow for generating elastic functional tests from a logic simulation using any standard simulation tool,
- ➢ a file format, called *transfer protocol*, describing elastic test patterns for handshake interface devices that prevents static cycle-based timing

Furthermore, the applicability of the approach has been shown by implementing the theoretical concept. In this context the achievements of the work comprise:

- ➢ an implementation of the generic test processor concept called NoTePAD (Novel Test Processor for Asynchronous Devices)
- ➢ a realization of the workflow by providing a framework including
  - a VHDL package that provides a generic model for asynchronous channels and respective functions to perform data transfers for the generation of *transfer protocols* during the functional simulation of the DUT,
  - a compiler tool translating *transfer protocols* to programs for the proposed NoTePAD architecture, and
  - a tool translating the binary programs for the processor to standard test patterns.

8.3 Impact of the Solution

The work introduced a powerful test technique for performing functional tests of asynchronous circuits. This concept builds the foundation to overcome one of the most limiting factors of the utilization of the asynchronous design style. With the help of the provided test processor, functional tests of asynchronous designs in the context of system debugging and production tests are now possible. Based on the provided concept, novel test systems can be realized that directly support asynchronous handshake protocols.

Furthermore, with the introduction and semi-automated generation of the *transfer protocol format*, a generic methodology for describing elastic test sequences has been proposed. Due to its abstract nature, the format can also be used by other test systems.

8.4 Limitations of the Approach

The approach aims at the test of timing nondeterministic circuits, i.e., circuits whose output sequences are not fixed with respect to their timing. However, the test of functionally nondeterministic circuits is a general problem at all. This has several reasons. The biggest issue to be solved in this context is the identification of the *right* response. For this, a magic predictor would be required. Furthermore, with respect to the generation of test patterns, there is the open question how a nondeterministic behavior can be simulated. Therefore, it is potentially required to extract all possible states and respective responses of the DUT reachable from a nondeterministic state transition.

Another limitation is related to the restriction to bundled-data protocols only. Although it has been illustrated (see Appendix B) how dual-rail circuits can be interfaced using protocol converters, it might be more efficient to directly support other data encoding styles. However, to ensure safe data transfers, the data ports may also be equipped with synchronization techniques in order to safely migrate the asynchronously arriving data signals into the clock domain of the test processor. Furthermore, the data ports have to be connected with each other in order to detect protocol events. Such a configurable interconnection of the data ports may impose considerable overhead.

A further restriction of the NoTePAD solution is the synchronous handshake implementation. To prevent metastability, synchronizers are added at the input handshake interfaces. Unfortunately, this considerably slows down the execution of transfers.

8.5 Outlook on Future Activities

With respect to the architecture of NoTePAD, the following enhancements can be taken into account:

- ➢ *Extensions to directly support further encoding styles.* Currently, the only way to support other encoding styles is to add protocol converters. Besides the hardware overhead, these converters add further latency. Thus, a direct support of the most important encoding styles, e.g., dual-rail and 1-of-*n* codes, would be beneficial.

- ➢ *Implementation of the handshake ports using asynchronous logic.* This prevents the slow synchronizer-based handshaking currently implemented and enables a fully asynchronous communication. As a result, the synchronizers connected to the input interface signal of the handshake ports are obsolete. This enhancement would considerably accelerate the asynchronous communication.

- ➢ *Support for source synchronous circuits.* Another class of circuits, which tend to be hard testable with standard hardware testers, are source synchronous circuits. These circuits align their outputs to a clock signal internally generated by the DUT. Similar to handshake circuits, it is necessary to react to these output clocks in order to capture the responses of the DUT. A possible solution is to further extend the handshake ports to support such output clocks of the DUT.

- ➢ *Integration of the functionality of control ports into data ports.* Currently, control signals that are not aligned to handshake ports are realized by special ports that have to be accessed separately. A more general approach could be the adaption of the data ports such that these could also be used to implement control inputs and outputs not associated with handshake ports. A possible solution could be the extension of the *port switch* such that the *sequencer* is able to send commands directly to subsets of data ports.

- ➢ *Generation of the port switch configuration.* The *port switch* is the mediator between the handshake and the data ports. It stores the association between these ports to form asynchronous channels. In the current implementation of NoTePAD, this configuration is programmable using respective instructions of the processor. Due to the required logic managing the port association, a delay of one cycle is required for the data exchange between the handshake and the data ports. For the execution of a handshake, this means that two cycles are squandered: One for sending the opcode from a handshake port to the data ports and one for transmitting the status signal from the data ports to the handshake port. This could be avoided if this configuration is directly implemented in hardware by hard-wired interconnections rather than programmable connections. This is possible, since the target platforms for the realization of NoTePAD are FPGAs. For this, the configuration of the *port switch* has to be generated from the pin configuration that defines the association between the handshake and the data ports. Then, for every different design to be tested, an adapted processor can be generated.

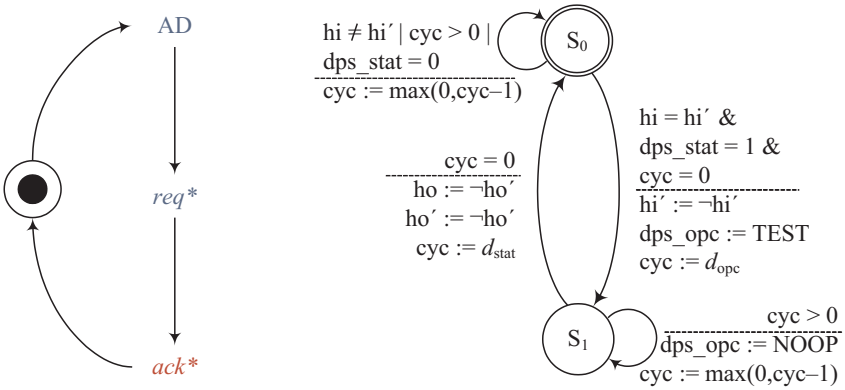
Besides further developments in the direction of the processor architecture, there are basically two further tools required which complete the entire tool suite for the framework:

- ➢ In consideration of the analysis of the test results, a tool is required that maps the responses of the processor to the respective channel transfers. This tool needs to consider the order of the uploaded port data and the bits per port. Using this information, it is trivial to map every bit of the uploaded data to a bit of a response of the DUT.

- Furthermore, for the integration of NoTePAD into an FPGA mounted onto a tester load board, it is required to map the FPGA interface pins to pins of the DUT. Therefore, a mapping of the FPGA pins to the pins of the socket and a further mapping from the socket to the DUT pins is required. Having this information, a constraint file (`.ucf`) comprising the pinning of the FPGA could be generated to route the ports of the processor to specific pins of the FPGA. The generation of the constraint file could easily be realized by a new software tool.

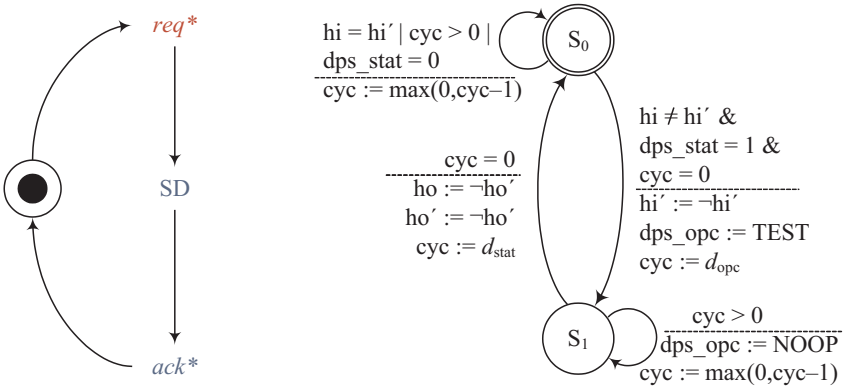
Appendix A

Handshake Protocol Implementations



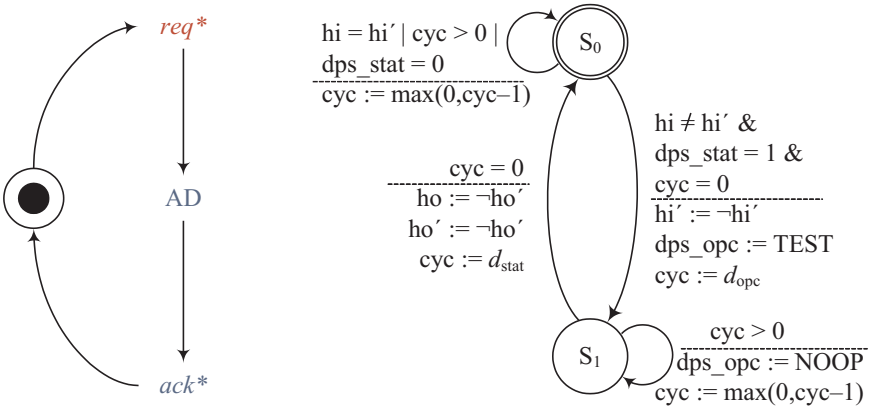
(a) STG (b) FSM

Figure A.1: Send 2-phase push protocol



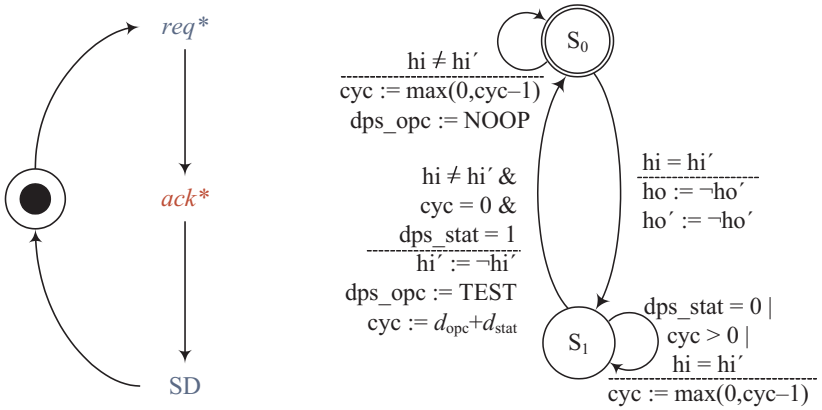
(a) STG (b) FSM

Figure A.2: Receive 2-phase push protocol



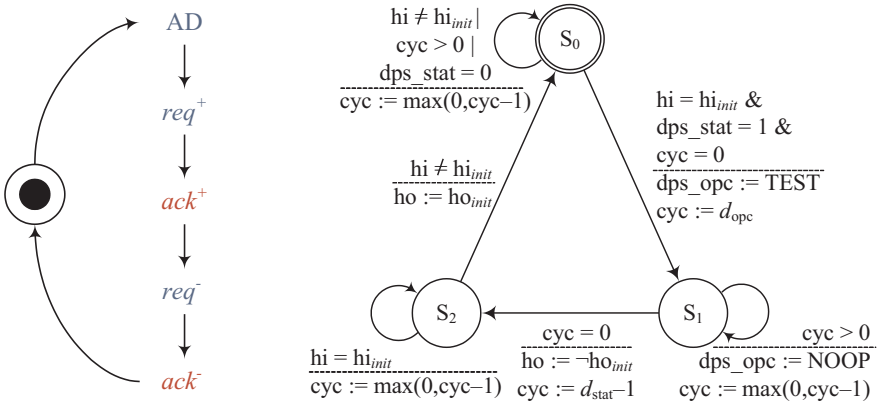
(a) STG (b) FSM

Figure A.3: Send 2-phase pull protocol



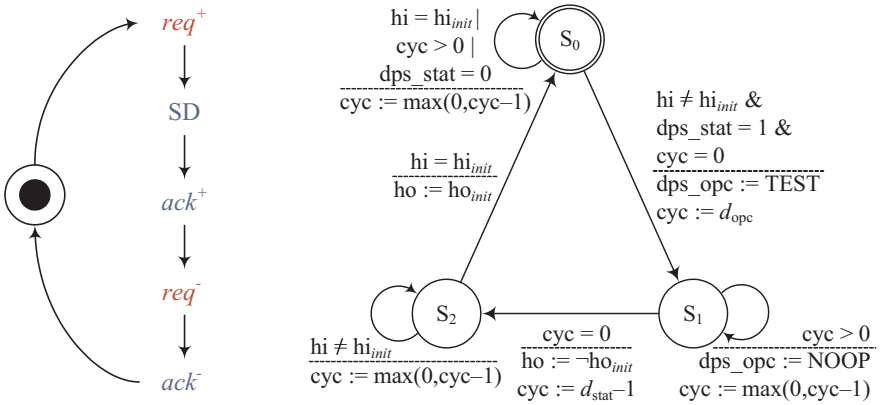
(a) STG (b) FSM

Figure A.4: Receive 2-phase pull protocol



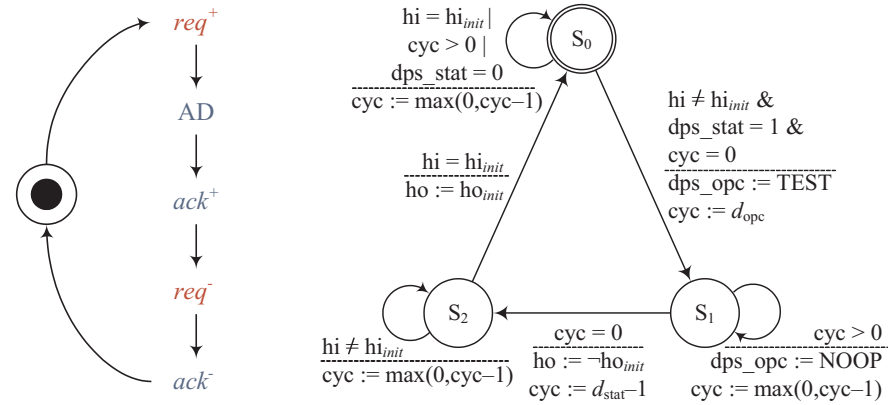
(a) STG (b) FSM

Figure A.5: Send 4-phase push protocol



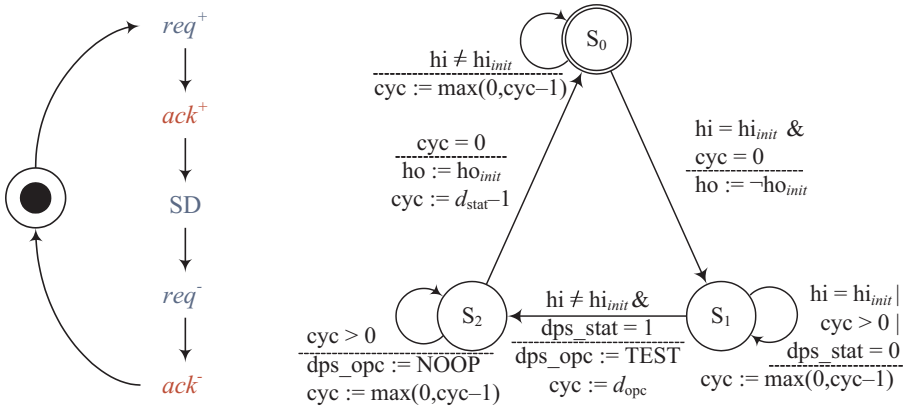
(a) STG (b) FSM

Figure A.6: Receive 4-phase push protocol



(a) STG (b) FSM

Figure A.7: Send 4-phase pull protocol

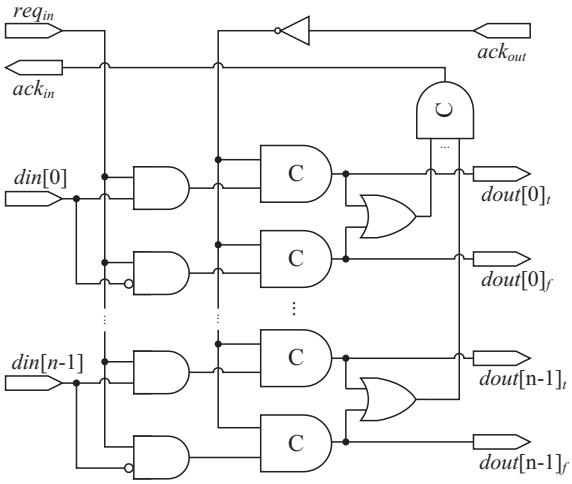


(a) STG (b) FSM

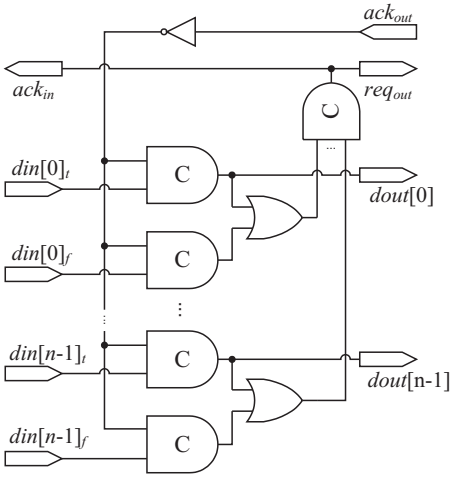
Figure A.8: Receive 4-phase pull protocol

Appendix B

Protocol Converters



(a) 4-phase single-rail to 4-phase dual-rail converter



(b) 4-phase dual-rail to 4-phase single-rail converter

Figure B.1: Protocol converters

Appendix C

Tools

C.1 Transfer Protocol Compiler

This tool compiles a *transfer protocol* to a NoTePAD assembler program. It is invoked via

```
tpc [options] filename
```

where `filename` is the name of the *transfer protocol* to be translated. The tool provides various options that can be divided into individual groups. The first group contains general options related to delivery of information about the translation process and the tool itself. The next group comprises options to define the architectural parameters of the target NoTePAD architecture. This includes the operating frequency, the number of handshake and data ports etc. Furthermore, there is a group of options that control the generated program output. One can define whether the initialization phase of the processor shall be separated from the *test sequence* and whether the expected signatures of data ports used as inputs shall be overwritten by gained fault signatures.

General option

`-h, --help` – Displays a help message.

`-v, --verbose` – Delivers more verbose output messages.

Options related to the test processor architecture

`-f, --frequency value` – Determines the operating frequency of the test processor. The value should be defined in MHz. If omitted the default of 200 MHz is assumed.

`-t, --timeout value` – Determines the default timeout assumed for parallel transfers. The value should be given in nanoseconds. Default is 1000 ns.

`-H, --hps number` – Determines the number of handshake ports.

`-i, --dps number` – Determines the number of data ports.

`-b, --bit-per-port number` – Determines the number of bits used per data port.

Output options

`-o, --output output-file` – Name of the output file

`-s, --split` – Separates the processor initialization from the test phase. If this option is set, `tpc` generates a separate program file including only the initialization phase of the processor.

`-O, --overwrite-data` – Determines whether to overwrite the expected responses with the actually received responses. This saves memory, but requires an additional download of the data between testing individual devices.

`-z, --zero-input` – Forces the expected data of input channels to be zero. This is useful when the results shall be received but not compared by the test processor.

`-p, --dp-preload` – Causes the data ports to be filled prior to the test execution. This may prevent bottlenecks due to data gaps during the test execution, but increases the test preparation time.

Output files

As a result of the compilation the tool creates several output files:

- program file(s) including the test processor program and the data block comprising the data port configurations. If the option `-s` is specified the tool separates the processor initialization from the *test sequence*. Furthermore, if the size of the generated program describing the *test sequence* exceeds 4 MB the entire program is distributed over multiple files each having a maximum size of 4 MB. This is due to the fact that the assembler has problems when translating files larger than the defined maximum size.

- data file(s) each comprising the data for one data port used.

- a pinning file including a description of the mapping of the channel resources used in the *transfer protocol* to actual ports of the test processor.

C.2 Memory Map Converter

A further tool that has been developed in the frame of this work is the memory map converter. This tool is used to translate a memory map file created from a binary program for the test processor into various output formats. This comprises memory initialization files (COE and MIF) for Xilinx memory blocks as well as EVCD pattern files. The tool is invoked via

```
cmm [options] filename
```

It also provides the above mentioned general options as well as options to define the output format.

General option

`-h, --help` – Displays a help message.

`-v, --verbose` – Delivers more verbose output messages.

Output options

`-o, --output output-file` – Name of the output file

`-f, --format {COE,MIF,EVCD}` – This option is used to define the output format when translating a memory map file. The format specification is a comma (',') separated list of the defined output formats. If omitted all output formats are created.

`-w, --bit-width number` – This option defines the width of the bit vectors written into a MIF file. This width has to match the word width of the memory block for which the output MIF file is created.

`-p, --is_program` – This option defines whether the input file describes the program or the data memory content. If the option is omitted, the input file is treated as data memory content. This affects the value of the write-enable signal for the two memory blocks.

Appendix D

Demonstrator

Listing D.1: Test bench for the asynchronous ALU

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.NUMERIC_STD.ALL;

library WORK;
use WORK.asynch_pack.ALL;
use WORK.asynch_alu_pack.ALL;
use WORK.tp_bench_pack.ALL;

entity tb_asynch_alu is
end tb_asynch_alu;

architecture tb_asynch_alu_arch of tb_asynch_alu is

    component asynch_alu is
       port (
            rst     : in  std_logic;
            opc     : in  std_logic_vector (3 downto 0);
            opc_req : in  std_logic;
            opc_ack : out std_logic;
            op1     : in  std_logic_vector (15 downto 0);
            op1_req : in  std_logic;
            op1_ack : out std_logic;
            op2     : in  std_logic_vector (15 downto 0);
            op2_req : in  std_logic;
            op2_ack : out std_logic;
            res     : out std_logic_vector (15 downto 0);
            res_req : out std_logic;
            res_ack : in  std_logic );
```

```
    end component;

    signal rst : TP_SIGNAL  :=
        tp_init_signal("rst", INPUT);
    signal opc : TP_CHANNEL :=
        tp_init_channel("opc",  4,  INPUT, PUSH, FOUR, SR);
    signal op1 : TP_CHANNEL :=
        tp_init_channel("op1", 16,  INPUT, PUSH, FOUR, SR);
    signal op2 : TP_CHANNEL :=
        tp_init_channel("op2", 16,  INPUT, PUSH, FOUR, SR);
    signal res : TP_CHANNEL :=
        tp_init_channel("res", 16, OUTPUT, PUSH, FOUR, SR);

    signal input_channels : TP_SET_OF_CHANNELS(2 downto 0);

begin

    uut : asynch_alu
    port map (
        rst     => rst.data(0),
        opc     => opc.data(3 downto 0),
        opc_req => opc.req,
        opc_ack => opc.ack,
        op1     => op1.data(15 downto 0),
        op1_req => op1.req,
        op1_ack => op1.ack,
        op2     => op2.data(15 downto 0),
        op2_req => op2.req,
        op2_ack => op2.ack,
        res     => res.data(15 downto 0),
        res_req => res.req,
        res_ack => res.ack);

    proc_ctrl : process
    begin
        tp_open_transfer_protocol("async_alu");
        tp_register_resource(rst);
        tp_register_resource(op1);
        tp_register_resource(op2);
        tp_register_resource(opc);
        tp_register_resource(res);
```

```
        tp_start_test_sequence;

        tp_wait( 10 ns );
        tp_assign(rst, "1");
        tp_wait( 100 ns );
        tp_assign(rst, "0");
        tp_wait( 10 ns );

        wait until tp_is_closed(op1) and tp_is_closed(op2)
               and tp_is_closed(opc) and tp_is_closed(res);
        tp_close_transfer_protocol;

        wait;
    end process;

    proc_opc : process
    begin
        tp_open_channel(opc, "0000");
        wait until rst.data(0 downto 0) = "0";
        tp_send(opc, op_and);
        tp_send(opc, op_add);
        tp_send(opc, op_sub);
        tp_send(opc, op_gcd);
        tp_send(opc, op_gcd);
        tp_send(opc, op_xor);
        tp_send(opc, op_sra);
        tp_close_channel(opc);
        wait;
    end process;

    proc_op1 : process
    begin
        tp_open_channel(op1, "0000000000000000");
        wait until rst.data(0 downto 0) = "0";
        tp_send(op1, "0000000000000001");
        tp_send(op1, "0000000000000110");
        tp_send(op1, "1000001111111110");
        tp_send(op1, "0000000000000110");
        tp_send(op1, "0000000000010010");
        tp_send(op1, "1111111111111111");
        tp_send(op1, "1000000000000001");
```

```
113         tp_close_channel(op1);
            wait;
        end process;

        proc_op2 : process
118     begin
            tp_open_channel(op2, "0000000000000000");
            wait until rst.data(0 downto 0) = "0";
            tp_send(op2, "1010101011111111");
            tp_send(op2, "0000000000000001");
123         tp_send(op2, "0000000001110001");
            tp_send(op2, "0000000000011000");
            tp_send(op2, "0000000000011011");
            tp_send(op2, "0000000000000001");
            tp_send(op2, "0000000000000001");
128         tp_close_channel(op2);
            wait;
        end process;

        input_channels <= opc & op1 & op2;
133
        proc_res : process
            variable result : std_logic_vector(15 downto 0);
        begin
            tp_open_channel(res);
138         tp_wait(input_channels);
            for i in 0 to 6 loop
                tp_recv(res, result);
            end loop;
            tp_close_channel(res);
143         wait;
        end process;

    end tb_asynch_alu_arch;
```

Listing D.2: Generated *transfer protocol* of the ALU

```
DECLARE

    rst : IN  SIGNAL;
    op1 : IN  CHANNEL( PUSH, 4P, SR, '0', '0', 16 );
    op2 : IN  CHANNEL( PUSH, 4P, SR, '0', '0', 16 );
    opc : IN  CHANNEL( PUSH, 4P, SR, '0', '0',  4 );
    res : OUT CHANNEL( PUSH, 4P, SR, '0', '0', 16 );

END

TEST

    WAIT FOR 10000 ps;
    rst <= "1";
    WAIT FOR 100000 ps;
    rst <= "0";
    WAIT FOR 10000 ps;
    opc <= "0000",
    op1 <= "0000000000000001",
    op2 <= "1010101011111111";
    opc <= "0100",
    op2 <= "0000000000000001",
    op1 <= "0000000000000110",
    res => "0000000000000001";
    opc <= "0101",
    op2 <= "0000000001110001",
    op1 <= "1000001111111110",
    res => "0000000000000111";
    opc <= "1010",
    op2 <= "0000000000011000",
    op1 <= "0000000000000110",
    res => "1000001110001101";
    opc <= "1010",
    op2 <= "0000000000011011",
    op1 <= "0000000000010010",
    res => "0000000000000110";
    opc <= "0010",
    op2 <= "0000000000000001",
    op1 <= "1111111111111111",
    res => "0000000000001001";
```

```
        opc <= "1000",
        op2 <= "0000000000000001",
43      op1 <= "1000000000000001",
        res => "1111111111111110";
        res => "1100000000000000";

    END
```

Listing D.3: Compiled assembler program

```
    .ref MASK_AND_PC
    .ref DP0_RDATA
    .ref DP0_RDATA_END
    .ref DP1_RDATA
    .ref DP1_RDATA_END
    .ref DP2_RDATA
    .ref DP2_RDATA_END
    .ref DP3_RDATA
    .ref DP3_RDATA_END
    .ref DP4_RDATA
    .ref DP4_RDATA_END
    .ref DP5_RDATA
    .ref DP5_RDATA_END
    .ref DP6_RDATA
    .ref DP6_RDATA_END
    .ref DP7_RDATA
    .ref DP7_RDATA_END
    .ref DP8_RDATA
    .ref DP8_RDATA_END
    .ref DP9_RDATA
    .ref DP9_WDATA
    .ref DP9_RDATA_END
    .ref DP10_RDATA
    .ref DP10_WDATA
    .ref DP10_RDATA_END
    .ref DP11_RDATA
    .ref DP11_WDATA
    .ref DP11_RDATA_END
    .ref DP12_RDATA
    .ref DP12_WDATA
    .ref DP12_RDATA_END
    .text
INIT_PROC:
    rsio
    conf hp=0, dp=0
    conf hp=0, dp=1
    conf hp=0, dp=2
    conf hp=0, dp=3
    conf hp=1, dp=4
    conf hp=1, dp=5
```

```
        conf hp=1, dp=6
        conf hp=1, dp=7
43      conf hp=2, dp=8
        conf hp=3, dp=9
        conf hp=3, dp=10
        conf hp=3, dp=11
        conf hp=3, dp=12
48      sahp hp=0, protocol=0x5
        sahp hp=1, protocol=0x5
        sahp hp=2, protocol=0x5
        sahp hp=3, protocol=0x4
    INIT_TEST:
53      init hpmask=0x000f
        tmrq hpmask=0x000f
        seth r2, (END_OF_TEST >> 16)
        setl r2, (END_OF_TEST & 0xffff)
    EXEC_TEST:
58      seth r1, 0x0003
        setl r1, 0x0d3e
        sout mask=0x0001, value=0x0001
        seth r1, 0x0000
        setl r1, 0x0009
63      wait
        seth r1, 0x0003
        setl r1, 0x0d3e
        sout mask=0x0001, value=0x0000
        noop
68      noop
        test hpmask=0x0007
        test hpmask=0x000f
        test hpmask=0x000f
        test hpmask=0x000f
73      test hpmask=0x000f
        test hpmask=0x000f
        test hpmask=0x000f
        test hpmask=0x0008
    END_OF_TEST:
78      flsh hpmask=0x0008
        tmrq hpmask=0x000f
        seth r0, (MASK_AND_PC >> 16)
        setl r0, (MASK_AND_PC & 0xffff)
```

```
        slhw r3, r0
83      slhw r3, r0
        addu 2
        slhw r4, r0
        slhw r4, r0
    UPLOAD:
88      halt
        seth r0, ((DP9_WDATA-DMEM_START) >> 16)
        setl r0, ((DP9_WDATA-DMEM_START) & 0xffff)
        seth r1, ((DP9_WDATA-DMEM_START+2) >> 16)
        setl r1, ((DP9_WDATA-DMEM_START+2) & 0xffff)
93      dout
        seth r0, ((DP10_WDATA-DMEM_START) >> 16)
        setl r0, ((DP10_WDATA-DMEM_START) & 0xffff)
        seth r1, ((DP10_WDATA-DMEM_START+2) >> 16)
        setl r1, ((DP10_WDATA-DMEM_START+2) & 0xffff)
98      dout
        seth r0, ((DP11_WDATA-DMEM_START) >> 16)
        setl r0, ((DP11_WDATA-DMEM_START) & 0xffff)
        seth r1, ((DP11_WDATA-DMEM_START+2) >> 16)
        setl r1, ((DP11_WDATA-DMEM_START+2) & 0xffff)
103     dout
        seth r0, ((DP12_WDATA-DMEM_START) >> 16)
        setl r0, ((DP12_WDATA-DMEM_START) & 0xffff)
        seth r1, ((DP12_WDATA-DMEM_START+2) >> 16)
        setl r1, ((DP12_WDATA-DMEM_START+2) & 0xffff)
108     dout
        halt
        halt
        .end
```

Listing D.4: Data section of the program

```
        .data
    DMEM_START:
    CONF_OF_DP0:
4       .word (DP0_RDATA-DMEM_START)
        .word 0
        .word ((DP0_RDATA_END-DP0_RDATA) >> 1)
        .word 0
    CONF_OF_DP1:
9       .word (DP1_RDATA-DMEM_START)
```

```
    .word 0
    .word ((DP1_RDATA_END-DP1_RDATA) >> 1)
    .word 0
CONF_OF_DP2:
    .word (DP2_RDATA-DMEM_START)
    .word 0
    .word ((DP2_RDATA_END-DP2_RDATA) >> 1)
    .word 0
CONF_OF_DP3:
    .word (DP3_RDATA-DMEM_START)
    .word 0
    .word ((DP3_RDATA_END-DP3_RDATA) >> 1)
    .word 0
CONF_OF_DP4:
    .word (DP4_RDATA-DMEM_START)
    .word 0
    .word ((DP4_RDATA_END-DP4_RDATA) >> 1)
    .word 0
CONF_OF_DP5:
    .word (DP5_RDATA-DMEM_START)
    .word 0
    .word ((DP5_RDATA_END-DP5_RDATA) >> 1)
    .word 0
CONF_OF_DP6:
    .word (DP6_RDATA-DMEM_START)
    .word 0
    .word ((DP6_RDATA_END-DP6_RDATA) >> 1)
    .word 0
CONF_OF_DP7:
    .word (DP7_RDATA-DMEM_START)
    .word 0
    .word ((DP7_RDATA_END-DP7_RDATA) >> 1)
    .word 0
CONF_OF_DP8:
    .word (DP8_RDATA-DMEM_START)
    .word 0
    .word ((DP8_RDATA_END-DP8_RDATA) >> 1)
    .word 0
CONF_OF_DP9:
    .word (DP9_RDATA-DMEM_START)
    .word (DP9_WDATA-DMEM_START)
```

```
    .word ((DP9_RDATA_END-DP9_RDATA) >> 1)
    .word 0
CONF_OF_DP10:
    .word (DP10_RDATA-DMEM_START)
    .word (DP10_WDATA-DMEM_START)
    .word ((DP10_RDATA_END-DP10_RDATA) >> 1)
    .word 0
CONF_OF_DP11:
    .word (DP11_RDATA-DMEM_START)
    .word (DP11_WDATA-DMEM_START)
    .word ((DP11_RDATA_END-DP11_RDATA) >> 1)
    .word 0
CONF_OF_DP12:
    .word (DP12_RDATA-DMEM_START)
    .word (DP12_WDATA-DMEM_START)
    .word ((DP12_RDATA_END-DP12_RDATA) >> 1)
    .word 0
    .bss MASK_AND_PC, 8, 4

DP0_RDATA:
    .word 0xffff6e61, 0x0fff01f2
DP0_RDATA_END:
DP1_RDATA:
    .word 0xffff0f00, 0x0fff00f1
DP1_RDATA_END:
DP2_RDATA:
    .word 0xffff0300, 0x0fff00f0
DP2_RDATA_END:
DP3_RDATA:
    .word 0xffff0800, 0x0fff08f0
DP3_RDATA_END:
DP4_RDATA:
    .word 0xffff811f, 0x0fff011b
DP4_RDATA_END:
DP5_RDATA:
    .word 0xffff170f, 0x0fff0001
DP5_RDATA_END:
DP6_RDATA:
    .word 0xffff000a, 0x0fff0000
DP6_RDATA_END:
DP7_RDATA:
```

```
    .word 0xffff000a, 0x0fff0000
DP7_RDATA_END:
DP8_RDATA:
    .word 0xffffa540, 0x0fff082a
DP8_RDATA_END:
DP9_RDATA:
    .word 0xffff6d71, 0x0fff00e9
DP9_RDATA_END:
    .bss DP9_WDATA, 4, 4
DP10_RDATA:
    .word 0xffff0800, 0x0fff00f0
DP10_RDATA_END:
    .bss DP10_WDATA, 4, 4
DP11_RDATA:
    .word 0xffff0300, 0x0fff00f0
DP11_RDATA_END:
    .bss DP11_WDATA, 4, 4
DP12_RDATA:
    .word 0xffff0800, 0x0fff0cf0
DP12_RDATA_END:
    .bss DP12_WDATA, 4, 4
```

Listing D.5: Pinning information

```
SIGNAL rst  : co(0);

CHANNEL op1 : req => ho(0),
              ack => hi(0),
              data(15 downto 0) => do(15 downto 0);
CHANNEL op2 : req => ho(1),
              ack => hi(1),
              data(15 downto 0) => do(31 downto 16);
CHANNEL opc : req => ho(2),
              ack => hi(2),
              data(3 downto 0) => do(35 downto 32);
CHANNEL res : req => hi(3),
              ack => ho(3),
              data(15 downto 0) => di(51 downto 36);
```

List of Figures

List of Tables

Listings

Bibliography

[Abramovici 1990] Miron Abramovici, Melvin A. Breuer and Arthur Friedman. Digital Systems Testing and Testable Design. IEEE Press, 1990. [cited at p. 35, 36, 37, 40]

[Advantest 2013] Advantest. *V93000 SOC*, 2013. http://www1.verigy.com/ate/products/V93000/index.htm. [cited at p. 3, 44]

[Ali 1996] Md. Liakot Ali, Zahari Mohamed Darus, Mohd Alauddin Mohd Ali and Iftekhar Ahmed. *Test processor ASIC design*. In Proceedings of the IEEE International Conference on Semiconductor Electronics (ICSE '96), pages 261–265, November 1996. [cited at p. 53]

[Ali 2002] Mohd Alauddin Mohd Ali, Syed Zahidul Islam and Md. Liakot Ali. *Test processor chip design with complete simulation result including reseeding technique*. In Proceedings of the IEEE International Conference on Semiconductor Electronics (ICSE '02), pages 218–221, December 2002. [cited at p. 53]

[Altaf-Ul-Amin 1999] Md. Altaf-Ul-Amin and Zahari Mohamed Darus. *VHDL design of a test processor based on mixed-mode test generation*. In Proceedings of the Ninth Great Lakes Symposium on VLSI, pages 244–245, March 1999. [cited at p. 53]

[Alves 1998] Vladimir C. Alves, Felipe M. G. Franca and Edson P. Granja. *A BIST Scheme for Asynchronous Logic*. In Proceedings of the 7th Asian Test Symposium (ATS'98), pages 27–32, Washington, DC, USA, 1998. Universidade Federal do Rio de Janeiro, Brazil, IEEE Computer Society. [cited at p. 41]

[Anderson 1991] James H. Anderson and Mohamed G. Gouda. *A New Explanation of the Glitch Phenomenon*. Acta Informatica, vol. 28, no. 4, pages 297–309, April 1991. [cited at p. 19]

[Beigne 2006] Edith Beigne and Pascal Vivet. *Design of On-chip and Off-chip Interfaces for a GALS NoC Architecture*. In Proceedings of the IEEE International Symposium on Asynchronous Circuits and Systems (ASYNCH '06), vol-

ume 0, pages 172–183, Los Alamitos, CA, USA, 2006. IEEE Computer Society. [cited at p. 33]

[Bhutada 2007] Rani Bhutada and Yiannos Manoli. *Complex Clock Gating with Integrated Clock Gating Logic Cell*. In Proceedings of International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS'07), pages 164–169, 2–5 September 2007. [cited at p. 9]

[Branover 2004] Alex Branover, Rakefet Kol and Ran Ginosar. *Asynchronous Design By Conversion: Converting Synchronous Circuits into Asynchronous Ones*. In Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '04), volume 2, pages 870–875, Washington, DC, USA, February 2004. VLSI Systems Research Center, Technion – Israel Institute of Technology, Haifa, Israel, IEEE Computer Society. [cited at p. 26]

[Chaney 1973] Thomas J. Chaney and Charles E. Molnar. *Anomalous Behavior of Synchronizer and Arbiter Circuits*. IEEE Transactions on Computers, vol. C-22, no. 4, pages 421–422, 1973 1973. [cited at p. 20]

[Chapiro 1984] Daniel M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems*. PhD thesis, Stanford University, October 1984. [cited at p. 8, 31]

[Chelcea 2000] Tiberiu Chelcea and Steven M. Nowick. *Low-latency asynchronous FIFO's using token rings*. In Proceedings of the 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC '00), pages 210–220, 2000. [cited at p. 33]

[Chu 1987] Tam-Anh Chu. *Synthesis of self-timed VLSI Circuits from Graph-theoretic Specifications*. PhD thesis, Massachusetts Institute of Technology, 1987. [cited at p. 26]

[Cortadella 1996] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno and Alex Yakovlev. *Petrify: A Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers*. In Proceedings of the 11th Conference on Design of Integrated Circuits and Systems, Barcelona, November 1996. [cited at p. 26]

[Cortadella 2004] Jordi Cortadella, Alex Kondratyev, Luciano Lavagno, Kelvin Lwin and Christos P. Sotiriou. *From Synchronous to Asynchronous: An Automatic Approach*. In Proceedings of the Conference on Design, page 21368, Washington, DC, USA, 2004. Univ. Politècnica de Catalunya, Barcelona, Spain; Cadence

Berkeley Labs Berkley and San Jose, CA, USA; Politecnico di Torino, Torino, Italy; Cadence Berkeley Labs; ICS-FORTH Crete, Greece, IEEE Computer Society. [cited at p. 26]

[Cortadella 2010] Jordi Cortadella, Luciano Lavagno, Djavad Amiri, Jonas Casanova, Carlos Macian, Ferran Martorell, Juan A. Moya Vicen, Luca Necchi, Danil Sokolov and Emre Tuncer. *Narrowing the margins with elastic clocks*. In Proceedings of the IEEE International Conference on IC Design and Technology (ICICDT), pages 146–150, 2010. [cited at p. 10]

[Couranz 1975] George R. Couranz and Donald F. Wann. *Theoretical and Experimental Behavior of Synchronizers Operating in the Metastable Region*. IEEE Transactions on Computers, vol. C-24, no. 6, pages 604–616, June 1975. [cited at p. 20]

[Darus 1997] Zahari Mohamed Darus, Iftekhar Ahmed and Md. Liakot Ali. *A test processor chip implementing multiple seed, multiple polynomial linear feedback shift register*. In Proceedings of the Sixth Asian Test Symposium (ATS '97), pages 155–160, November 1997. [cited at p. 53]

[Dike 1999] Charles Dike and Edward Burton. *Miller and noise effects in a synchronizing flip-flop*. IEEE Journal of Solid-State Circuits, vol. 34, no. 6, pages 849–855, jun 1999. [cited at p. 21]

[Dinh Duc 2002] Anh Vu Dinh Duc, Jean-Baptiste Rigaud, Amine Rezzag, Antoine Sirianni, Joao Fragoso, Laurent Fesquet and Marc Renaudin. *TAST CAD Tools*. In Proceedings of the 2nd Asynchronous Circuit Design Workshop (ACiD'02), 2002. [cited at p. 25]

[Edwards 2000] Douglas A. Edwards and Andrew Bardsley. *Synthesizing an Asynchronous DMA Controller with Balsa*. Journal of Systems Architecture, vol. 46, no. 14, pages 1309–1319, December 2000. [cited at p. 25]

[Edwards 2002] Douglas A. Edwards and Andrew Bardsley. *Balsa: An Asynchronous Hardware Synthesis Language*. The Computer Journal, vol. 45, no. 1, pages 12–18, January 2002. [cited at p. 25]

[Fan 2010] Xin Fan, Miloš Krstić, Christoph Wolf and Eckhard Grass. *A GALS FFT processor with clock modulation for low-EMI applications*. In Proceedings 21st IEEE Intl. Conf. Application-specific Systems, Architectures and Processors (ASAP), 2010. [cited at p. 8]

[Fang 2005] David Fang, John Teifel and Rajit Manohar. *A High-Performance Asynchronous FPGA: Test Results*. In Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '05), pages 271–272, Washington, DC, USA, April 2005. Computer Systems Laboratory, Cornell University, USA, IEEE Computer Society. [cited at p. 10]

[Frost 2007] Raik Frost, David Rudolph, Christan Galke, Rene Kothe and Heinrich T. Vierhaus. *A Configurable Modular Test Processor and Scan Controller Architecture*. In Proceedings of the 13th IEEE International On-Line Testing Symposium (IOLTS '07), pages 277–284, July 2007. [cited at p. 53, 54, 75]

[Fuhrer 1999] Robert M. Fuhrer, Steven M. Nowick, Michael Theobald, Niraj K. Jhay, Bill Linz and Luis Plana. *MINIMALIST: An Environment for the Synthesis, Verification and Testability of Burst-Mode Asynchronous Machines*. Technical report, Columbia University, Computer Science Deptartment, 1999. [cited at p. 26, 31]

[Furber 1994a] Steve B. Furber, Paul Day, Jim D. Garside, Nigel C. Paver, Steve Temple and John V. Woods. *The design and evaluation of an asynchronous microprocessor*. In Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD '94), pages 217–220. Department of Computer Science, University of Manchester, UK, 10–12 October 1994. [cited at p. 10]

[Furber 1994b] Steve B. Furber, Paul Day, Jim D. Garside, Nigel C. Paver and John V. Woods. *AMULET1: A Micropipelined ARM*. In Digest of Papers. Compcon Spring '94, pages 476–485. Department of Computer Science, University of Manchester, UK, 28 February–4 March 1994. [cited at p. 8]

[Furber 1997] Steve B. Furber, Jim D. Garside, Steve Temple, Jianwei Liu, Paul Day and Nigel C. Paver. *AMULET2e: An Asynchronous Embedded Controller*. In Proceedings of the 3rd International Symposium on Advanced Research in Asynchronous Circuits and Systems, pages 290–299. Department of Computer Science, University of Manchester, UK, IEEE Computer Society, 1997. [cited at p. 8, 10]

[Furber 1998] Steve B. Furber, Jim D. Garside and D.A. Gilbert. *AMULET3: A High-Performance Self-Timed ARM Microprocessor*. In Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors

(ICCD '98), pages 247–252. Department of Computer Science, University of Manchester, UK, 5–7 October 1998. [cited at p. 8]

[Furber 1999] Steve B. Furber, Jim D. Garside, P. Riocreux, Steve Temple, Paul Day, Jianwei Liu and Nigel C. Paver. *AMULET2e: An Asynchronous Embedded Controller*. Proceedings of the IEEE, vol. 87, no. 2, pages 243–256, 1999. [cited at p. 8]

[Furber 2000] Steve B. Furber, Douglas A. Edwards and Jim D. Garside. *AMULET3: A 100 MIPS Asynchronous Embedded Processor*. In Proceedings of the International Conference on Computer Design, pages 329–334. Department of Computer Science, University of Manchester, UK, 17–20 September 2000. [cited at p. 8]

[Galke 2002] Christian Galke, Matthias Pflanz and Heinrich T. Vierhaus. *A Test Processor Concept for Systems-on-a-Chip*. In Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors, pages 210–212, 2002. [cited at p. 53, 54, 75, 82, 83]

[Garside 2000] Jim D. Garside, John Bainbridge, Andrew Bardsley, Douglas A. Edwards, Steve B. Furber, Jianwei Liu, David W. Lloyd, S. Mohammadi, Jeffrey S. Pepper, Oleg A. Petlin, Steve Temple and John V. Woods. *AMULET3i - An Asynchronous System-on-Chip*. In Proceedings of the 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems, pages 162–175, Washington, DC, USA, April 2000. Department of Computer Science, University of Manchester, UK, IEEE Computer Society Press. [cited at p. 8]

[Gill 2005] Gennette Gill and Montek Singh. *Synthesizing Asynchronous Burst-Mode Machines without the Fundamental-Mode Timing Assumption*. In ACM/IEEE International Workshop on Timing Issues, 2005. [cited at p. 30]

[Ginosar 2003] Ran Ginosar. *Fourteen Ways to Fool Your Synchronizer*. In Proceedings of the 9th International Symposium on Asynchronous Circuits and Systems, pages 89–, Washington, DC, USA, 2003. IEEE Computer Society. [cited at p. 21]

[Gürkaynak 2002] Frank K. Gürkaynak, Thomas Villiger, Stephan Oetiker, Norbert Felber, Hubert Kaeslin and Woflgang Fichtner. *A Functional Test Methodology for Globally-Asynchronous Locally-Synchronous Systems*. In Proceedings of the 8th International Symposium on Asynchronous Circuits and Systems, pages 181–189. Integrated Systems Laboratory, ETH Zürich, Switzerland, IEEE Computer Society, 8–11 April 2002. [cited at p. 3, 46]

[Hawkins 1994] C.F. Hawkins, J.M. Soden, A.W. Righter and F.J. Ferguson. *Defect classes-an overdue paradigm for CMOS IC testing*. In Test Conference, 1994. Proceedings., International, pages 413–425, 1994. [cited at p. 36]

[Heath 2003] Matthew W. Heath and Ian G. Harris. *A Deterministic Globally Asynchronous Locally Synchronous Microprocessor Architecture*. In Proceedings of the 4th International Workshop on Microprocessor Test and Verification: Common Challenges and Solutions, pages 119–124. University of Massachusetts Amherst, USA, May 2003. [cited at p. 49]

[Heath 2004] Matthew W. Heath, Wayne P. Burleson and Ian G. Harris. *Eliminating Nondeterminism to Enable Chip-Level Test of Globally-Asynchronous Locally-Synchronous SoC's*. IEEE Computer Society, 2004. citeseer.ist.psu.edu/heath04eliminating.html. [cited at p. 50]

[Heath 2005] Matthew W. Heath, Ian G. Harris and Wayne P. Burleson. *Synchro-Tokens: A Deterministic GALS Methodology for Chip-Level Debug and Test*. IEEE Transaction on Computers, vol. 54, no. 12, pages 1532–1546, December 2005. [cited at p. 49]

[Hoare 1978] Charles Antony Richard Hoare. *Communicating Sequential Processes*. Communications of the ACM, vol. 21, no. 8, pages 666—677, August 1978. [cited at p. 25]

[Huffman 1955] David A. Huffman. *A study of the memory requirements of sequential switching circuits*. Technical report 293, Massachusetts Institute of Technology, Research Laboratory of Electronics, March 1955. [cited at p. 30]

[Hulgaard 1994] Henrik Hulgaard, Steven M. Burns and Gaetano Borriello. *Testing Asynchronous Circuits: A Survey*. Integration, the VLSI Journal, vol. 19, no. 3, pages 111–131, March 1994. [cited at p. 37, 38]

[IEEE 1364-1995] IEEE Std 1364-1995. *IEEE Standard Hardware Description Language Based on the Verilog(R) Hardware Description Language*. 1995. [cited at p. 44]

[IEEE 1364-2001] IEEE Std 1364-2001. *Standard Verilog Hardware Description Language*. 2001. [cited at p. 44]

[IntellaSys 2008] IntellaSys. *SEAforth®40C18*, 2008. http://www.intellasys.net/index.php?option=com_content&task=view&id=60&Itemid=75. [cited at p. 8]

[ITRS 2012] ITRS. *International Technology Roadmap for Semiconductors 2012 – Design*. Technical report, Semiconductor Industries Association, 2012. [cited at p. 2]

[Jährig 2004] Ralf Jährig and Walter Anheier. *Automatisierte Erzeugung von asynchronen Schaltungen*. In AUSTROCHIP 2004, Villach, Austria, pages 13–20. Institut für Theoretische Elektrotechnik und Mikroelektronik ITEM, August 2004. [cited at p. 26]

[Jin 2009] Gang Jin, Lei Wang and Zhiying Wang. *A New Description Language for Data-Driven Asynchronous Circuits and its Design Flow*. In Proceedings of the Pacific-Asia Conference on Circuits, Communications and Systems (PACCS'09), pages 322–325, 2009. [cited at p. 26]

[Kabir 2009] M.A. Kabir and Liakot Ali. *Design of GLFSR based test processor chip*. In Proceedings of the IEEE Student Conference on Research and Development (SCOReD), pages 234–237, November 2009. [cited at p. 53]

[Keezer 2005] David C. Keezer, Carl Gray, A.M. Majid and N. Taher. *Low-cost multi-gigahertz test systems using CMOS FPGAs and PECL*. In Design, Automation and Test in Europe, 2005. Proceedings, pages 152–157 Vol. 1, 2005. [cited at p. 55]

[Kermani 2001] Bahram Ghaffarzadeh Kermani, William James Smurthwaite and James Frank Vomero. *Testing Asynchronous Circuits*, 2001. [cited at p. 43, 45]

[Khoche 1994] Ajay Khoche and Erik Brunvand. *Testing micropipelines*. In E. Brunvand, editor, Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems, pages 239–246, Washington, DC, USA, 1994. Department of Computer Science, University of Utah, Salt Lake City, IEEE Computer Society. [cited at p. 37]

[Khoche 1995] Ajay Khoche and Erik Brunvand. *A partial scan methodology for testing self-timed circuits*. In E. Brunvand, editor, Proceedings of the 13th IEEE VLSI Test Symposium (VTS'95), pages 283–289, Washington, DC, USA, 1995. Department of Computer Science, University of Utah, Salt Lake City, IEEE Computer Society. [cited at p. 40]

[King 2004] Matthew L. King and Kewal K. Saluja. *Testing Micropipelined Asynchronous Circuits*. In Proceedings of the 2004 International Test Conference (ITC'04), pages 329–338, Washington, DC, USA, October 2004. Department of Electrical and Computer Engineering, University of Wisconsin, Madison, WI, USA, IEEE Computer Society. [cited at p. 37]

[Kinniment 2002] David J. Kinniment, Alex Bystrov and Alex Yakovlev. *Synchronization circuit performance*. IEEE Journal of Solid-State Circuits, vol. 37, no. 2, pages 202 –209, feb 2002. [cited at p. 21]

[Kishinevsky 1997] Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, Alexander Saldanha and Alexander Taubin. *Partial Scan Delay Fault Testing of Asynchronous Circuits*. In Proceedings of the 1997 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'97), pages 728–735, Washington, DC, USA, 1997. The University of Aizu, Aizu-Wakamatsu, Japan; Politecnico di Torino, Torino, Italy;Cadence Berkeley Laboratories, Berkeley, USA, IEEE Computer Society. [cited at p. 37]

[Krstić 2005a] Miloš Krstić and Eckhard Grass. *BIST technique for GALS systems*. In Eckhard Grass, editor, Proceedings of the 8th Euromicro Conference on Digital System Design, pages 10–16. IHP microelectronics Frankfurt (Oder), Germany, IEEE Computer Society, 2005. [cited at p. 3, 47]

[Krstić 2005b] Miloš Krstić, Eckhard Grass and Christian Stahl. *Request-driven GALS technique for wireless communication system*. In Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC '05), pages 76–85, 2005. [cited at p. 33]

[Krstić 2007] Miloš Krstić, Eckhard Grass, Frank Gürkaynak and Pascal Vivet. *Globally Asynchronous, Locally Synchronous Circuits: Overview and Outlook*. IEEE Design Test of Computers, vol. 24, no. 5, pages 430–441, 2007. [cited at p. 8, 31, 32]

[Krstić 2011] Miloš Krstić, Xin Fan, Eckhard Grass, Luca Benini, M. R. Kakoee, Christoph Heer, B. Sanders, A. Strano and Davide Bertozzi. *Moonrake Chip - GALS Demonstrator in 40 nm CMOS*. In Proceedings of the International Symposium on System-on-Chip (SoC), 2011. [cited at p. 8]

[Lines 2004] Andrew Lines. *Asynchronous interconnect for synchronous SoC design*. vol. 24, no. 1, pages 32–41, 2004. [cited at p. 8]

[Majid 2005] A.M. Majid and David C. Keezer. *An improved low-cost 6.4 Gbps wafer-level tester*. In Proceedings of the 7th Electronic Packaging Technology Conference (EPTC'05), volume 2, pages 6 pp.–, 2005. [cited at p. 55]

[Majid 2010] A.M. Majid and David C. Keezer. *Stretching the limits of FPGA SerDes for enhanced ATE performance*. In Proceedings of the IEEE Design, Automa-

tion Test in Europe Conference Exhibition (DATE'10), pages 202–207, 2010. [cited at p. 55]

[Martin 1989] Alian J. Martin. Formal Development of Programs and Proofs, chapter Formal Program Transformations for VLSI Circuit Synthesis, pages 59–80. Addision-Wesley, 1989. [cited at p. 17]

[Martin 1991] Alian J. Martin and Pieter J. Hazewindus. *Testing Delay-Insensitive Circuits*. In Proceedings of the University of California/Santa Cruz conference on Advanced research in VLSI, pages 118–132. Computer Science Department, California Institute of Technology, Pasadena, USA, MIT Press, 1991. [cited at p. 36]

[McCluskey 1963] Edward J. McCluskey. *Fundamental mode and pulse mode sequential circuits*. In Proceedings of the IFIP Congr. Inform. Processing, page 725, 1963. [cited at p. 30]

[McCluskey 1986] Edward J. McCluskey. Logic design principles - with emphasis on testable semicustom circuits. Prentice Hall series in computer engineering. Prentice Hall, 1986. [cited at p. 21]

[Menon 1978] Premachandran R. Menon and Stephen G. Chappell. *Deductive Fault Simulation with Functional Blocks*. IEEE Transaction on Computers, vol. C-27, no. 8, pages 689–695, August 1978. [cited at p. 35]

[Miller 1965] Raymond Edward Miller. Switching Theory Volume 2: Sequential Circuits and Machines. Wiley J., New York, 1965. [cited at p. 29]

[Muttersbach 2000] Jens Muttersbach, Thomas Villiger and Wolfgang Fichtner. *Practical Design of Globally-Asynchronous Locally-Synchronous Systems*. In Proceedings of the 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems, ASYNC '00, pages 52–, Washington, DC, USA, 2000. IEEE Computer Society. [cited at p. 33]

[Nielson 1997] Lars S. Nielson. *Low Power Asynchronous VLSI Design*. PhD thesis, Department of Information Technology, Technical University of Denmark, 1997. [cited at p. 10]

[Nowick 1995] Steven M. Nowick. *Automatic Synthesis of Burst-Mode Asynchronous Controllers*. Technical report, Department of Electrical Engineering and Computer Science, Stanford University, December 1995. [cited at p. 31]

[Petlin 1995a] Oleg A. Petlin and Steve B. Furber. *Scan Testing of Asynchronous Sequential Circuits*. In Proceedings of the 5th Great Lakes Symposium on VLSI (GLSVLSI'95), pages 224–229, Washington, DC, USA, 1995. Department of Computer Science, University of Manchester, UK, IEEE Computer Society. [cited at p. 41]

[Petlin 1995b] Oleg A. Petlin and Steve B. Furber. *Scan Testing of Micropipelines*. In Steve B. Furber, editor, Proceedings of the 13th IEEE VLSI Test Symposium, pages 296–301, Washington, DC, USA, May 1995. Department of Computer Science, University of Manchester, UK, IEEE Computer Society. [cited at p. 37, 41]

[Petlin 1997] Oleg A. Petlin and Steve B. Furber. *Built-In Self-Testing of Micropipelines*. In Proceedings of the 3rd International Symposium on Advanced Research in Asynchronous Circuits and Systems, pages 22–29. Department of Computer Science, University of Manchester, UK, IEEE Computer Society, 7–10 April 1997. [cited at p. 47]

[Plana 2011] Luis A. Plana, David Clark, Simon Davidson, Steve Furber, Jim Garside, Eustace Painkras, Jeffrey Pepper, Steve Temple and John Bainbridge. *SpiNNaker: Design and Implementation of a GALS Multicore System-on-Chip*. J. Emerg. Technol. Comput. Syst., vol. 7, no. 4, pages 17:1–17:18, December 2011. [cited at p. 8]

[Roncken 1996] Marly Roncken and Eric Bruls. *Test Quality of Asynchronous Circuits: A Defect-oriented Evaluation*. In Proceedings of the International Test Conference (ITC'96), pages 205–214. Philips Research Laboratories, Eindhoven, Netherlands, IEEE Computer Society, October 1996. [cited at p. 36]

[Roncken 2000] Marly Roncken, Ken S. Stevens, Rajesh Pendurkar, Shai Rotem and Parimal Pal Chaudhuri. *CA-BIST for asynchronous circuits: a case study on the RAPPID asynchronous instruction length decoder*. In Proceedings of the 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC'00), pages 62–72, 2000. [cited at p. 3, 47]

[Rutten 1997] J.W.J.M. Rutten and Michel R.C.M. Berkelaar. *Improved state assignment for burst mode finite state machines*. In Proceedings of the 3rd International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNCH '97), pages 228 –239, apr 1997. [cited at p. 31]

[Schöber 2001] Volker Schöber. *Der testfreundliche Entwurf asynchroner Schaltungen*. PhD thesis, Fachbereich Elektortechnik und Informationstechnik, Universität Hannover, Germany, June 2001. [cited at p. 41]

[Seitz 1980] Charles L. Seitz. Introduction to VLSI Systems, chapter System Timing, pages 218–262. Addision-Wesley Publishing Company, 1980. [cited at p. 7, 18, 66]

[Semiat 2003] Yaron Semiat and Ran. Ginosar. *Timing measurements of synchronization circuits*. In Proceedings of the Ninth International Symposium on Asynchronous Circuits and Systems (ASYNCH '03), pages 68 – 77, may 2003. [cited at p. 21]

[Shams 1996] Maitham Shams, Jo C. Ebergen and Mohamed I. Elmasry. *A comparison of CMOS implementations of an asynchronous circuits primitive: the C-element*. In Low Power Electronics and Design, 1996., International Symposium on, pages 93–96, 1996. [cited at p. 18]

[Shang 2006] Delong Shang, Alex Yakovlev, Frank Burns, Fei Xia and Alex Bystrov. *Low-cost Online Testing of Asynchronous Handshakes*. In Proceedings of the 12th IEEE European Test Symposium (ETS'06), pages 225–232, Washington, DC, USA, May 2006. School of EECE, University of Newcastle upon Tyne, UK, IEEE Computer Society. [cited at p. 39]

[Shi 2006] Feng Shi and Yiorgos Makris. *A Transistor-Level Test Strategy for C^2MOS MOUSETRAP Asynchronous Pipelines*. In Proceedings of the 12th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNCH'06), page 57, Washington, DC, USA, March 2006. Electrical Engineering Department, Yale University, USA, IEEE Computer Society. [cited at p. 36]

[Singh 2007] Montek Singh and Steven M. Nowick. *MOUSETRAP: High-Speed Transition-Signaling Asynchronous Pipelines*. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, vol. 15, no. 6, pages 684–698, June 2007. [cited at p. 13]

[Solutions 2004] Handshake Solutions. *HT-80C51 microcontroller*. ARM – Device Database, 2004. http://www.keil.com/dd/chip/3931.htm. [cited at p. 8]

[Sparsø 2001] Jens Sparsø, Steve Furber, René van Leuken, Reinder Nouta and Alexander de Graaf. Principles of Asynchronous Circuit Design: A Systems Perspective. Kluwer Academic Publishers, Boston, 2001. [cited at p. 9, 12, 16, 17, 29, 30, 43, 48, 63, 109, 150]

[Sutherland 1989] Ivan E. Sutherland. *Micropipelines*. Communications of the ACM, vol. 32, no. 6, pages 720–738, June 1989. [cited at p. 8, 13]

[Tanenbaum 2007] Andrew S. Tanenbaum. Modern Operating Systems. Prentice Hall, 3rd édition, 2007. [cited at p. 22]

[Taylor 2007] Samuel M. Taylor. *Data-Driven Handshake Circuit Synthesis*. PhD thesis, University of Manchester, 2007. [cited at p. 26]

[te Beest 2002] Frank te Beest, Ad Peeters, Marc Verra, Kees van Berkel and Hans Kerkhoff. *Automatic Scan Insertion and Test Generation for Asynchronous Circuits*. In Proceedings of the International Test Conference (ITC '02), pages 804–813, Washington, DC, USA, 2002. University of Twente, MESA+ Research Institute, Testable Design and Testing Group, Enschede, Netherlands; Philips Research Laboratories, Eindhoven, Netherlands; Eindhoven University of Technology, Eindhoven, Netherlands, IEEE Computer Society. [cited at p. 40, 41]

[Tiempo 2008] Tiempo. *Tiempo Asynchronous TAM16 Core IP*, 2008. http://www.tiempo-ic.com/uploads/Docs/TAM16_Datasheet.pdf?page=uploads/Docs/Tiempo%20TAM16%20IP%20Data%20Sheet%201.2.pdf. [cited at p. 8]

[Tiempo 2012] Tiempo. *Tiempo Asynchronous Circuits System Verilog Modeling Language*. In Proceedings of the IEEE 18th International Symposium on Asynchronous Circuits and Systems (ASYNC'12), volume 0, pages 105–112, Los Alamitos, CA, USA, 2012. IEEE Computer Society. [cited at p. 25]

[Tiempo 2013] Tiempo. *Cryptoprocessors cores*, 2013. http://www.tiempo-ic.com/products/ip-cores/cryptoprocessors.html. [cited at p. 8]

[Unger 1983] Stephen H. Unger. Asynchronous Sequential Switching Circuit. Krieger Publishing Co., Inc., Melbourne, FL, USA, 1983. [cited at p. 30]

[van Berkel 1991] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs and Frits D. Schalij. *The VLSI-programming language Tangram and its translation into handshake circuits*. In Proceedings European Conference on Design Automation (EDAC), pages 384–389, Washington, DC, USA, February 1991. Philips Research Laboratories, Eindhoven, Netherlands, IEEE Computer Society. [cited at p. 25]

[van Berkel 1993] Kees van Berkel. Handshake circuits: an asynchronous architecture for VLSI programming. Cambridge University Press, New York, NY, USA, 1993. [cited at p. 25]

[Wang 2006] Alice Wang, Benton H. Calhoun and Anantha P. Chandrakasan. Subthreshold Design for Ultra Low-Power Systems (Series on Integrated Circuits and Systems). Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. [cited at p. 10]

[Wolf 2011] Christoph Wolf, Steffen Zeidler, Miloš Krstić and Rolf Kraemer. *Overview on ATE Test and Debugging Methods for Asynchronous Circuits*. In 12th International Workshop on Microprocessor Test and Verification (MTV'00), pages 16–21, December 2011. [cited at p. 4]

[Woods 1997] John V. Woods, Steve B. Furber, Jim D. Garside, Steve Temple, Paul Day and Nigel C. Paver. *AMULET1: An Asynchronous ARM Microprocessor*. IEEE Transaction on Computers, vol. 46, no. 4, pages 385–398, April 1997. [cited at p. 8]

[Yun 1992a] Kenneth Y. Yun and David L. Dill. *Automatic synthesis of 3D asynchronous state machines*. In Digest of Technical Papers. IEEE/ACM International Conference on Computer-Aided Design (ICCAD'92), pages 576–580, 1992. [cited at p. 26]

[Yun 1992b] Kenneth Y. Yun, David L. Dill and Steven M. Nowick. *Synthesis of 3D Asynchronous State Machines*. In Proceedings of the International Conference of Computer Design (ICCD'92), pages 346–350. IEEE Computer Society Press, 1992. [cited at p. 31]

[Yun 1994] Kenneth Y. Yun. *Synthesis Of Asynchronous Controllers For Heterogeneous Systems*. PhD thesis, Stanford University, 1994. [cited at p. 26, 31]

[Yun 1996] Kenneth Y. Yun and Ryan P. Donohue. *Pausible clocking: a first step toward heterogeneous systems*. In Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'96), pages 118–123. Department of Electrical & Computer Engineering, California University, San Diego, USA, 1996. [cited at p. 8, 33]

[Yun 1999] Kenneth Y. Yun and A.E. Dooply. *Pausible clocking-based heterogeneous systems*. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 7, no. 4, pages 482–488, 1999. [cited at p. 33]

[Zeidler 2010] Steffen Zeidler, Alexandre Bystrov, Miloš Krstić and Rolf Kraemer. *Online testing of bundled-data asynchronous handshake protocols*. In Proceedings of the 16th IEEE International On-Line Test Symposium (IOLTS'10), pages 261–267, 2010. [cited at p. 39]

[Zeidler 2011] Steffen Zeidler, Christoph Wolf, Miloš Krstić, Frank Vater and Rolf Kraemer. *Design of a Test Processor for Asynchronous Chip Test*. In Proceedings of the IEEE Asian Test Symposium (ATS '11), pages 244–250, November 2011. [cited at p. 5, 75, 83, 89]

[Zeidler 2012a] Steffen Zeidler, Christoph Wolf, Miloš Krstić and Rolf Kraemer. *Eng: Design of a Novel Test Processor for Functional Tests of Asynchronous Circuits; Deu: Entwurf einer neuen Testprozessorlösung für den Funktionaltest asynchroner Schaltungen*. In Proceedings of the Test und Zuverlässigkeit Workshop (TuZ'12), February 2012. [cited at p. 5, 75, 83, 89, 137, 164, 165, 198]

[Zeidler 2012b] Steffen Zeidler, Christoph Wolf, Miloš Krstić and Rolf Kraemer. *Functional Pattern Generation for Asynchronous Designs in a Test Processor Environment*. In Proceedings of the 21st IEEE Asian Test Symposium (ATS'12), pages 296–301, November 2012. [cited at p. 5]

[Zhang 2010] Zhen Zhang. Performance Analysis of Synchronization Circuits. Master's thesis, School of Computer Science, University of Manchester, 2010. [cited at p. 21]

[Zhou 2006] Jun Zhou, David J. Kinniment, Gordon Russell and Alex Yakovlev. *A Robust Synchronizer*. In Proceedings of the IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'06), pages 442–443, 2006. [cited at p. 21]

[Zhou 2011] Rong Zhou, Kwen-Siong Chong, Bah-Hwee Gwee and J.S. Chang. *Quasi-delay-insensitive compiler: Automatic synthesis of asynchronous circuits from verilog specifications*. In Proceedings of the IEEE 54th International Midwest Symposium on Circuits and Systems (MWSCAS'11), pages 1–4, 2011. [cited at p. 25]

[Zivojnovic 1996] Vojin Zivojnovic, Stefan Pees and Heinrich Meyr. *LISA - Machine Description Language and Generic Machine Model for HW/SW Co-Design*. In Proceedings of the IEEE Workshop on VLSI Signal Processing, pages 127–136, 1996. [cited at p. 78]