TECHNISCHE UNIVERSITÄT
ILMENAU

Towards an Embedded Board-Level Tester
Study of a Configurable Test Processor

Dissertation

Zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing)

vorgelegt in der Fakultät für Informatik und Automatisierung der Technischen Universität Ilmenau

von Herrn Ing. Jorge Hernán Meza Escobar,
geboren am 18.01.1984 in Cali, Kolumbien

Datum der Einrichtung: 15.06.2016 (vorliegende Revision vom 29.11.2016)

Datum der Verteidigung: 21.11.2016

Gutachter:
1. Prof. Dr.-Ing. habil. Andreas Mitschele-Thiel, Technische Universität Ilmenau
2. Prof. Dr.-Ing. Sebastian Michael Sattler, Friedrich-Alexander Universität Erlangen-Nürnberg
3. Prof. Dr. Raimund-Johannes Ubar, Tallinn University of Technology

urn:nbn:de:gbv:ilm1-2016000605

Acknowledgments

First, I would like to express my deepest and sincere gratitude to my advisors Dr.-Ing. Heinz-Dietrich Wuttke and Prof. Dr.-Ing. habil. Andreas Mitschele-Thiel. Thank you for letting me be part of the ICS group, for the excellent scientific guidance of my thesis, and for your constructive reviews, suggestions, and discussions, which significantly improved this work.

I would also like to express my gratitude and appreciation to my whole family, especially to my parents Jorge and Leonor, my brother Kike, and my sister Leonor. Thank you for your support and for encouraging me to pursue my dreams. Special thanks go to Pamela for all her love, support, and encouragement. I would also like to thank my cousin Grace Lewis for her grammar and text corrections.

I would like to thank the HW/SW systems group, especially Steffen Ostendorff and Jörg Sachße. The embedded board-level tester and the test processor would surely not be able to work without your contributions! My special thanks go to Dr.-Ing. Karsten Henke and all ICS group members. Thanks for having an open ear for discussing various aspects of my theoretical and practical work.

I would like to thank the organizational and technical staff, Nicole Sauer, Nadine Wolf, and specially Jürgen Schmidt. I would also like to thank the reviewers for investing their time in reading and evaluating this work.

Finally, I would like to express my gratitude and recognize that this research would not have been possible without the financial assistance of the Thüringer Aufbaubank (TAB), the Zentrales Innovationsprogram Mittelstand (ZIM), and the TU Ilmenau. I am also very thankful to Göpel Electronic and Thomas Wenzel for the opportunity to develop this research with the collaboration of an industry partner.

Abstract

The demand for electronic systems with more features, higher performance, and less power consumption increases continuously. This is a real challenge for design and test engineers because they have to deal with electronic systems with ever-increasing complexity maintaining production and test costs low and meeting critical time to market deadlines.

For a test engineer working at the board-level, this means that manufacturing defects must be detected as soon as possible and at a low cost. However, the use of classical test techniques for testing modern printed circuit boards is not sufficient, and in the worst case these techniques cannot be used at all. This is mainly due to modern packaging technologies, a high device density, and high operation frequencies of modern printed circuit boards. This leads to very long test times, low fault coverage, and high test costs.

This dissertation addresses these issues and proposes an FPGA-based test approach for printed circuit boards. The concept is based on a configurable test processor that is temporarily implemented in the on-board FPGA and provides the corresponding mechanisms to communicate to external test equipment and co-processors implemented in the FPGA. This embedded test approach provides the flexibility to implement test functions either in the external test equipment or in the FPGA. In this manner, tests are executed at-speed increasing the fault coverage, test times are reduced, and the test system can be adapted automatically to the properties of the FPGA and devices located on the board.

An essential part of the FPGA-based test approach deals with the development of a test processor. In this dissertation the required properties of the processor are discussed, and it is shown that the adaptation to the specific test scenario plays a very important role for the optimization. For this purpose, the test processor is equipped with configuration parameters at the instruction set architecture and microarchitecture level. Additionally, an automatic generation process for the test system and for the computation of some of the processor's configuration parameters is proposed. The automatic generation process uses as input a model known as the device under test model (DUT-M).

In order to evaluate the entire FPGA-based test approach and the viability of a processor for testing printed circuit boards, the developed test system is used to test interconnections to two different devices: a static random memory (SRAM) and a liquid crystal display (LCD). Experiments were conducted in order to determine the resource utilization of the processor and FPGA-based test system and to measure test time when different test functions are implemented in the external test equipment or the FPGA. It has been shown

that the introduced approach is suitable to test printed circuit boards and that the test processor represents a realistic alternative for testing at board-level.

Zusammenfassung

Der Bedarf an elektronischen Systemen mit zusätzlichen Merkmalen, höherer Leistung und geringerem Energieverbrauch nimmt ständig zu. Dies stellt eine erhebliche Herausforderung für Entwicklungs- und Testingenieure dar, weil sie sich mit elektronischen Systemen mit einer steigenden Komplexität zu befassen haben. Außerdem müssen die Herstellungs- und Testkosten gering bleiben und die Produkteinführungsfristen so kurz wie möglich gehalten werden.

Daraus folgt, dass ein Testingenieur, der auf Leiterplatten-Ebene arbeitet, die Herstellungsfehler so früh wie möglich entdecken und dabei möglichst niedrige Kosten verursachen soll. Allerdings sind die klassischen Testmethoden nicht in der Lage, die Anforderungen von modernen Leiterplatten zu erfüllen und im schlimmsten Fall können diese Testmethoden überhaupt nicht verwendet werden. Dies liegt vor allem an modernen Gehäuse-Technologien, der hohen Bauteildichte und den hohen Arbeitsfrequenzen von modernen Leiterplatten. Das führt zu sehr langen Testzeiten, geringer Testabdeckung und hohen Testkosten.

Die Dissertation greift diese Problematik auf und liefert einen FPGA-basierten Testansatz für Leiterplatten. Das Konzept beruht auf einem konfigurierbaren Testprozessor, welcher im On-Board-FPGA temporär implementiert wird und die entsprechenden Mechanismen für die Kommunikation mit der externen Testeinrichtung und Co-Prozessoren im FPGA bereitstellt. Dadurch ist es möglich Testfunktionen flexibel entweder auf der externen Testeinrichtung oder auf dem FPGA zu implementieren. Auf diese Weise werden Tests at-speed ausgeführt, um die Testabdeckung zu erhöhen. Außerdem wird die Testzeit verkürzt und das Testsystem automatisch an die Eigenschaften des FPGAs und anderer Bauteile auf der Leiterplatte angepasst.

Ein wesentlicher Teil des FPGA-basierten Testansatzes umfasst die Entwicklung eines Testprozessors. In dieser Dissertation wird über die benötigten Eigenschaften des Prozessors diskutiert und es wird gezeigt, dass die Anpassung des Prozessors an den spezifischen Testfall von großer Bedeutung für die Optimierung ist. Zu diesem Zweck wird der Prozessor mit Konfigurationsparametern auf der Befehlssatzarchitektur-Ebene und Mikroarchitektur-Ebene ausgerüstet. Außerdem wird ein automatischer Generierungsprozess für die Realisierung des Testsystems und für die Berechnung einer Untergruppe von Konfigurationsparametern des Prozessors vorgestellt. Der automatische Generierungsprozess benutzt als Eingangsinformation ein Modell des Prüflings (device under test model, DUT-M).

Das entwickelte Testsystem wurde zum Testen von Leiterplatten für Verbindungen zwischen dem FPGA und zwei Bauteilen verwendet, um den FPGA-basierten Testansatz und die Durchführbarkeit des Testprozessors für das Testen auf Leiterplatte-Ebene zu evaluieren. Die zwei Bauteile sind ein Speicher mit direktem Zugriff (static random-access memory, SRAM) und eine Flüssigkristallanzeige (liquid crystal display, LCD). Die Experimente wurden durchgeführt, um den Ressourcenverbrauch des Prozessors und Testsystems festzustellen und um die Testzeit zu messen. Dies geschah durch die Implementierung von unterschiedlichen Testfunktionen auf der externen Testeinrichtung und dem FPGA. Dadurch konnte gezeigt werden, dass der FPGA-basierte Ansatz für das Testen von Leiterplatten geeignet ist und dass der Testprozessor eine realistische Alternative für das Testen auf Leiterplatten-Ebene ist.

Table of Contents

List of Tables

List of Figures

Acronyms

ACK	Acknowledge
ALU	Arithmetic logic unit
AOI	Automated optical inspection
ATE	External automatic test equipment
AXI	Automated X-ray inspection
ASIC	Application-specific IC
BGA	Ball grid array
BIDIR	Bidirectional
BILBO	Built-in logic block observer
BIST	Built-in self-test
BScan	Boundary scan
CISC	Complex instruction set computer
CPI	Clock cycles per instruction
CPU	Central processing unit
CS	Chip select
DI	Device interface
DDR	Double data rate
DR	Data register
DRAM	Dynamic RAM
DUT	Device under test
DUT-M	DUT-model
DT	Data transfer
EDA	Electronic design automation
ESW	Embedded software
FBT	FPGA-based testing
FBTS	FPGA-based test system
FPGA	Field programmable gate array
FPT	Flying probe test
FSM	Finite state machine
GPR	General purpose register
HDL	Hardware Description Language
HW	Hardware
I	Index
I1 - I3	Interface 1 – Interface 3
I/O	Input/output
IC	Integrated circuit

ICT	In-circuit test
ILP	Instruction level parallelism
IP	Intellectual property
IPC	Instructions per clock cycle
IR	Instruction register
ISA	Instruction set architecture
JTAG	Joint test action group
L1 - L5	Layer 1 - Layer 5
LCD	Liquid crystal display
LE	Logic element
LFSR	Linear feedback shift register
LSB	Low significant bit
LUT	Look-up table
MISR	Multiple input signature register
MSB	Most significant bit
PBT	Processor based testing
PC	Program counter
PCB	Printed circuit board
RAM	Random access memory
RISC	Reduced instruction set computer
ROBSY	Reconfigurable on-board test system
RTL	Register-transfer level
SDR	Single data rate
SFR	Special function register
SMT	Surface mount technology
SoC	System on chip
SP	Stack pointer
SRAM	Static RAM
SW	Software
TAP	Test access port
TCK	Test clock
TDI	Test data in
TDO	Test data out
THT	Through-hole technology
TMS	Test mode select
TRST	Test reset
VLIW	Very large instruction word
VLSR	Variable length shift register
VHDL	VHSIC HDL
VHSIC	Very-high-speed integrated circuit

1 Introduction

1.1 Motivation

The world is more and more dependent on electronic systems. Today, they are a fundamental tool in medicine, healthcare, communication, automotive, navigation, space exploration, agriculture, etc. In all these fields, the demand for electronic systems with more features, higher performance, and less power consumption increases continuously. This is a real challenge for design and test engineers because they have to deal with electronic systems with ever-increasing complexity maintaining production and test costs low and meeting critical time to market deadlines.

For a test engineer, this means that manufacturing defects must be detected as soon as possible. Therefore, the manufacturing stage of every electronic system is escorted by several test phases hierarchically organized for early defect detection and diagnose. To fulfill this purpose, several test techniques have been developed and successfully used. But these techniques are not able to cope with test requirements of modern electronic systems, and in the worst case, they cannot be used at all.

At the board-level, this is mainly due to modern packaging technologies, a high device density, and the high operation frequencies of modern printed circuit boards. One example of this phenomenon is boundary scan (BScan) [1], which is a popular test technique developed thirty years ago and since then successfully used. Unfortunately, it is reaching its limits due to the properties of modern printed circuit boards. Basically, it is inadequate to handle defects with dynamic behavior because it cannot execute tests at the normal operation frequencies (at-speed test) of devices located at the board. Additionally, the high number of devices and pins per device results in very long test times and high test costs. Therefore, there is a growing interest in the research community and industry to investigate and develop new board-level test techniques that improve test quality metrics and reduce costs.

New developed test techniques and test standards tend to embed more of the test functions in the electronic system, whether the electronic system represents a printed circuit board (PCB) or a system entirely implemented in silicon such as a system on chip (SoC). In this way, access to internal structures and functions of the electronic system is enhanced, improving the observability, controllability, and test quality in a cost effective way.

Two emerging embedded test techniques at the board-level have gained considerable importance. The first one corresponds to processor-based testing (PBT), where processors or microcontrollers located on the board are used to execute part of the test functions. The second technique is known as FPGA-based testing (FBT). Instead of processors or microcontrollers, this technique uses field programmable gate arrays (FPGAs) to implement part of the test functions. Both techniques are very attractive because they do not require the modification or addition of new components to the board, and allow the detection of defects with a dynamic behavior.

In the case of PBT, the difficulty lies in the development of mechanisms for the automatic generation of descriptions necessary to access different processors or microcontrollers from the external automatic test equipment (ATE). The access should be guaranteed independently if the processor or microcontroller is implemented as a stand-alone integrated circuit or as part of a SoC. In the case of FBT, the complexity lies in the development of the test system implemented in the FPGA, which is known as the FBT system (FBTS). For the development of such a system, it is necessary to provide an answer to the following questions:

- What is the structure of the FBTS?
- Which components are used for its implementation?
- How should the FBTS communicate with the ATE?
- Which mechanisms can be used to facilitate the work of test engineers?

In this dissertation the research is linked to the development and implementation of a configurable test processor as part of an FBT approach.

1.2 Problem statement and research questions

This dissertation targets the study and development of a configurable test processor as part of a new FBT approach. The FBT approach should be able to execute tests at-speed and in short time, adapt to the PCB and test requirements, and hide implementation details from test engineers in order to facilitate its utilization. The goal is to provide a better understanding of whether the use of a test processor as part of an FPGA-based testing approach is a good alternative to tackle test challenges imposed by modern PCBs, so that classical test techniques such as boundary scan can be complemented.

Although there are some FBT approaches found in the literature (Section 3.2), none of them makes use of a test processor or is able to provide all the advantages that the FBT

approach proposed in this dissertation provides. Based on these considerations, the central research questions are defined as follows:

- Is it possible to design a resource and computational efficient FBT approach and obtain the advantages already mentioned based on a test processor?
- What are the architecture characteristics of an FBTS that use a test processor as central component?
- Which adaptation and abstraction mechanisms should be included in order to customize the FBTS to the test scenario and hide implementation details?
- Which instruction set architecture and microarchitecture features should be included in the test processor?
- What are the test functions that the test processor can efficiently execute in terms of resource utilization and execution time?
- Is it possible to use this approach in real manufacturing scenarios?
- Are the results concerning test quality metrics better in comparison to other FBT approaches?
- What are the limitations?

Literature research shows that these questions have not been answered in the related work so far, which ultimately motivates the development of this dissertation.

1.3 Publications related to this work

In the context of this work several papers have been published on national and international conferences. In [2, 3] the general concept of the FBT approach and targeted test processor is presented. The provided solution describes the layer concept and a first proposal for the FBTS architecture and automatic generation process. In [4] the concept is further developed and emphasis is given to the automatic generation of the complete test system. In [5] the main subject is the test processor and the use of configuration options at the instruction set architecture and microarchitecture level as a mechanism for the processor adaptation to the PCB and test requirements. In [6] the processor is used as part of a novel verification approach of register transfer level (RTL) designs. Finally, [7] presents results of the approach with focus on the test processor.

The publications cover basic aspects of the FBT approach proposed as part of this dissertation. This dissertation provides an extension of the work and a more detailed analysis of the FBTS, test processor, and automatic generation process. These aspects are extended with further improvements of the concept.

1.4 Structure of the dissertation and contributions

The structure of the dissertation is divided into eight chapters, as indicated in Figure 1.1.

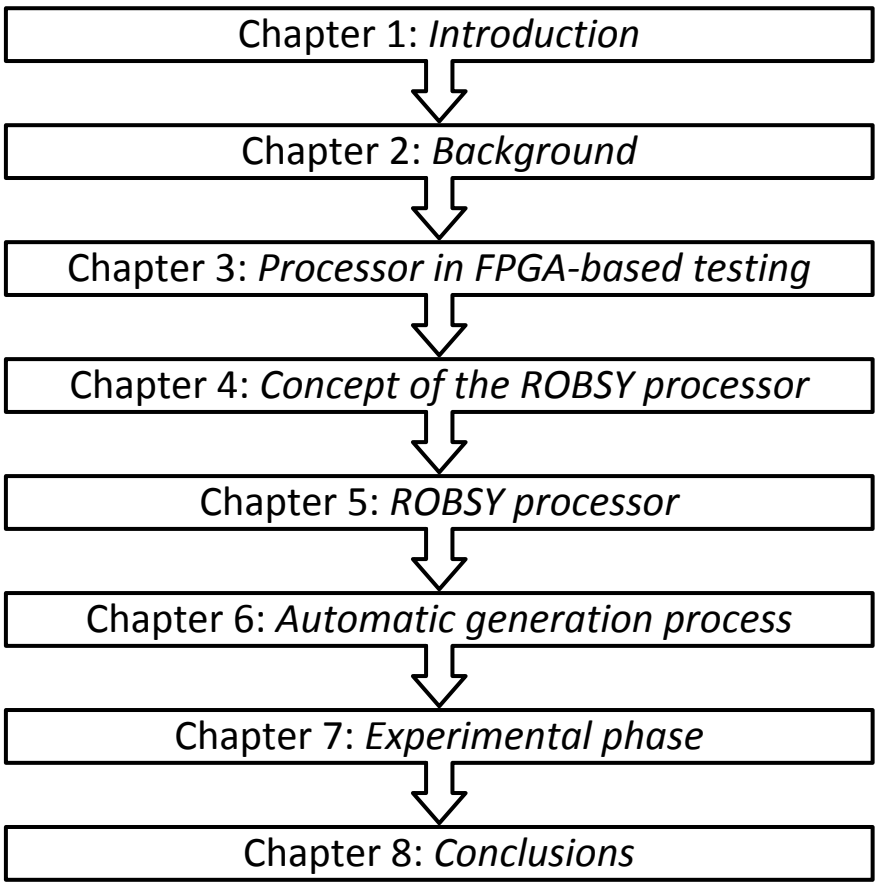


Figure 1.1: Structure of the dissertation

Chapter 2 provides the theoretical basis of the dissertation. It presents an overview on testing, testing of printed circuit boards, and test techniques. Additionally, it provides information about the state of the art of soft-core processors and test processors.

Chapter 3 begins with a discussion about FPGA-based testing and the state of the art of FPGA-based testing approaches. The approaches are reviewed and classified under the aspects of external/embedded test instrumentation, application speed, test time, and design methodology. Based on the literature review, the FPGA-based testing approach proposed in this dissertation is presented. At the end of Chapter 3, an analysis of the use of a processor in FPGA-based testing is performed.

Chapter 4 presents the concept of the ROBSY (Reconfigurable On-Board test System) processor based on the analysis performed in Chapter 3. This chapter discusses the processor general design aspects, specialization for testing, and adaptation mechanisms to the PCB and test requirements.

Chapter 5 presents the implementation details of the ROBSY processor. It shows its instruction set architecture, microarchitecture, and debug-interface. Additionally, Chapter 5 presents the configuration parameters supported by the processor.

Chapter 6 provides an overview of the automatic generation process for an FBTS composed of a processor/co-processor pair. The automatic generation process includes the generation of software for the ATE, embedded software for the ROBSY processor, and hardware descriptions that represent the co-processor. Furthermore, the automatic generation process is in charge of adapting the processor by defining the value of the configuration parameters known as dependent configuration parameters.

Chapter 7 presents the experimental results obtained with the proposed FBT approach. The chapter analyzes the implementation of test functions on the ATE and FPGA, as well as the effect of the processor configuration parameters known as the dependent and independent configuration parameters. It analyzes the resource utilization of the FPGA, the time that the FBTS is active during the whole test execution, and the total test time. At the end of the chapter the results obtained with the ROBSY test system are compared against the results obtained with a generic test instrument and with FBTS variants implemented based on the Nios II cores. The results show the advantages of the ROBSY approach and the ROBSY processor.

Finally, Chapter 8 summarizes the dissertation and provides an outlook on further research opportunities.

2 Background

2.1 Testing, what is it?

2.1.1 Introduction

In order to provide a proper definition for the word testing, it is necessary to take a look at the general development flow of an electronic system presented in Figure 2.1. In the figure, the specification of the electronic system represents the starting point of the development flow, in which the requirements of the electronic system, as seen from its environment, are defined. This comprises the definition of the product's overall functionality, interfaces to the environment, performance requirements, etc. [8, 9].

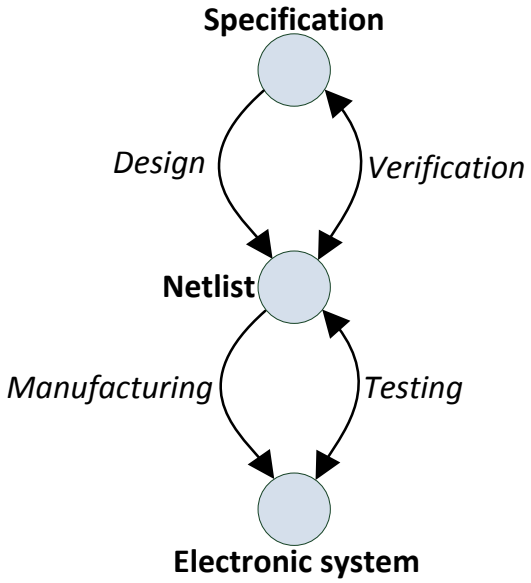


Figure 2.1: Development flow of an electronic system

Based on the specification, the internal structure of the system is defined during the design stage. In this stage several solutions can be explored, based on different technologies, architectures, hardware/software partitions, and so on [10]. The outcome of the design stage is documented in a structural description or netlist of the physical objects and their interconnections. Depending on the level of abstraction, these objects are transistors, gates, integrated circuits (ICs), or even multiple printed circuit boards (PCBs).

The second stage is the fabrication of the electronic system, which is known as the manufacturing stage. Depending on the level of abstraction, the manufacturing stage

might deal with the fabrication of ICs based on silicon wafers or the fabrication of PCBs, which involves the assembly of ICs in an insulated base with firmly attached conductors.

It is possible that the resulting electronic system behaves in an undesired way due to errors introduced during its development, or errors appearing during electronic system operation. In the first case, these errors were introduced during the elaboration of the specification, design or manufacturing stage. In the second case, these errors appear in the field.

In order to avoid or minimize the probability of incorrect behavior and detect possible causes as soon as possible, additional stages are included in the development flow, which are known as validation, verification, and testing [9].

2.1.2 Errors and defects

An error is essentially the observed effect of an incorrect behavior, which may be caused by diverse factors such as specification-errors, design-errors, fabrication-errors, fabrication-defects, or physical-failures [9, 11].

- Specification-errors occur due to an ambiguous or incomplete specification. In many cases, they are caused by an operation scenario not properly described in the specification, or even worse, an operation scenario not described at all [9].
- Design-errors are produced during the design of the electronic system. They are originated by an incorrect interpretation of the specification, violations of design rules, usage of defective design tools, etc. [9].
- Fabrication-errors and fabrication-defects appear during the manufacturing process. The former are directly attributable to a human act, such as the use of wrong devices, incorrect wiring, improper soldering, etc., while the latter are not directly attributable to a human act because they result from an imperfect manufacturing process [11].
- Physical-failures occur during the operation of the electronic system, and are caused due to wear out, aging, and environmental factors affecting the electronic system [11].

Fabrication-errors, fabrication-defects and physical-failures are collectively referred to in the literature as physical faults [11], manufacturing faults [9], or defects [12, 13]. They can be found anywhere: in a die, in one or multiple layers, in packages, in PCBs, etc. Additionally, they can be presented in arbitrary areas and can have different electrical properties, which means that they manifest themselves in different ways such as changing

a logical value on a node, increasing the steady state supply current, changing timing properties, or causing discrepancy in other parameters.

In this dissertation, the term *defect* is adopted to refer to fabrication-errors, fabrication-defects, and physical-failures, and the term *error* is the observed incorrect behavior produced by a defect.

2.1.3 Validation, verification, and testing

There are different ways to look for errors and defects, depending on the electronic system development stage.

Validation is carried out to detect and minimize specification-errors. During the validation process, the specification is analyzed to determine if it describes the desired behavior and whether it is complete, unambiguous, and consistent [9].

As seen in Figure 2.1, verification is performed during the design stage. The goal is to continually prove if the structural model resulting from the design stage fulfills all the functional and nonfunctional requirements provided in the specification [9, 14]. There are different ways to perform verification, either using formal verification methods (model and equivalence checking) or functional verification.

In contrast, testing is the process responsible for detecting all defects (fabrication-errors, fabrication-defects, and physical-failures) introduced during the manufacturing process or operation of an electronic system [11, 12]. In this context, testing is an experiment in which the system is exercised and the resulting response is analyzed to ascertain whether it behaved correctly. If incorrect behavior is detected, a second goal of testing may be to diagnose the defect causing the misbehavior. Testing requires the generation, application, acquisition, and analysis of test patterns, which are defined depending on required test quality metrics and allowed costs (Section 2.1.6).

2.1.4 Fault and fault models

Ideally, a test should detect all defects produced in the manufacturing stage and let only functionally good devices pass. Unfortunately, the number of potential defects that appear during the manufacturing stage can be very large, and their effect on properties and behavior of the electronic system can be very complex and difficult to understand.

Faults and fault models are used to address this problem.

A fault is defined as a representation of the defect at an abstract function level [13], which means that it is a representation of the effect of a defect on the operation of the system. Consequently, a fault model is a mathematical or formal description of a fault. Fault models bridge the gap between the physical reality and mathematical abstracts, and they reduce test complexity, given that [11]:

- Many defects can be represented by the same fault model.
- Some fault models can be used for different technologies because they are technology independent.
- Tests derived from fault models may be used for defects whose effect in the circuit behavior is not completely understood or is too complex to be analyzed.

Unsurprisingly, there is a trade-off when working with fault models. The more simple and abstract the fault model is, the more difficult it is to associate the fault with a real defect, which makes defect location and diagnosis more difficult. Therefore, one alternative is to work at different abstraction levels, starting with the development of tests for fault models at high abstraction levels, and then develop tests targeted at low-level faults that were not covered by the high-level fault models [15]. Which fault model is best and which type of testing is necessary depends on technology and defect manifestation.

In the next sections, fault models relevant for this thesis are presented: stuck-at, bridging, and dynamic fault models.

2.1.4.1 Stuck-at faults

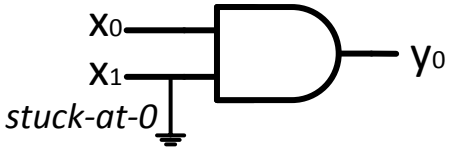
The stuck-at fault is considered a static fault model because the defects that it represents are manifested at all frequencies. It is the logic-level fault model most commonly used in research as well as industry, being an industrial standard since 1959. Its death has been predicted but several reasons and properties have made the stuck-at fault model to continue to be used for testing [13]:

- Simplicity: It is easy to apply to a device under test (DUT).
- Logical behavior: Fault behavior is determined logically, so simulation is straightforward and deterministic.
- Measurability: The defect represented by a stuck-at fault is easy to detect.
- Adaptability: It can be used on transistors, gates, registers, systems, etc.

Stuck-at faults are mapped to interconnections between devices. Under faulty conditions, the affected line is assumed to be stuck-at a logic level (0 or 1) and the value cannot be altered by input stimuli. According to the value of the affected line, the fault is called

either stuck-at-0 or stuck-at-1. Usually, open lines or lines shorted to ground or power behave like stuck-at faults [16].

Figure 2.2 shows an example of a stuck-at-0 fault located at the input of an AND gate.



x_1	x_0	y_0	y'_0
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	0

Figure 2.2: Stuck-at-0 AND gate with truth table [17]

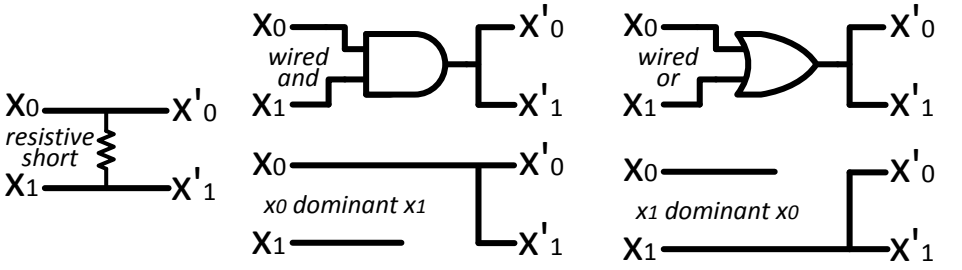
In general, several stuck-at faults can be simultaneously present. A DUT with n lines can have $3^n - 1$ possible stuck-at line combinations. This is because each line can be in one of the three states: stuck-at-1, stuck-at-0, or fault-free. Clearly, even a moderate value of n will generate an enormously large number of multiple stuck-at faults. However, it is a common practice to work with single stuck-at faults [11, 12, 17, 18]. In this case, an n-line DUT can have at most $2n$ single stuck-at faults.

2.1.4.2 Bridging faults

Same as the stuck-at fault model, a bridging fault is considered a static fault model because the defects that it represents are manifested at all frequencies. It is also a very common fault model and it is still one of the fault models most widely accepted in industry [16]. A bridging fault considers that two or more lines are unintentionally connected.

There are two bridging fault models that are frequently used in practice: the wired-AND/wired-OR bridging fault model and the dominant bridging fault model [17]. The wired-AND bridging fault is also known as 0-dominant bridging fault because a 0 on one of the faulty lines determines the logic value on both lines in the same way as a logic 0 at any input of an AND gate. Similarly, the wired-OR bridging fault is also known as 1-dominant bridging fault.

The dominant bridging fault model was developed to more accurately reflect the behavior of some shorts, in which the logic value at the destination end of the shorted lines is determined by the line with the strongest drive capability. As a consequence, the driver for one line “dominates” the driver for the other line. Figure 2.3 shows the four bridging fault models.



fault free		wired AND/OR				dominant			
		wired AND		wired OR		x0 dominant x1		x1 dominant x0	
x1	x0	x'1	x'0	x'1	x'0	x'1	x'0	x'1	x'0
0	0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	1	1	0	0
1	0	0	0	1	1	0	0	1	1
1	1	1	1	1	1	1	1	1	1

Figure 2.3: Wired AND/OR and dominant bridging fault models with truth table [17]

In general, several bridging faults can be simultaneously present in the DUT. In this case, a DUT with n lines can have $5^n - 1$ possible bridging fault combinations. However, in the same way as stuck-at faults, it is a common practice to work with single bridging faults [11, 12, 17, 18]. An n-line DUT can have at most $4n$ single bridging faults.

2.1.4.3 Dynamic faults

In contrast to stuck-at and bridging faults, dynamic faults represent defects that show up only at high frequencies. In order to detect this kind of fault, it is necessary to execute test at-speed, which means applying patterns at the operation frequency of the DUT [12]. Additionally, while stuck-at and bridging faults require the application of a single pattern for their detection, dynamic faults typically require the application of two patterns: the first one for value initialization and the second one for the activation of the fault. They can be separated into two main classes, known as delay faults and signal integrity faults.

Delay faults

Delay fault models are associated to interconnection or component defects that cause the non-fulfillment of timing specifications. This means, defects that cause that a DUT whose operation is logically correct, to not perform at the required working frequency [13, 17]. In this case, an expected signal value is delayed and the actual measured output signal is therefore not correct within a pre-specified timing constraint. An example of this behavior is illustrated in Figure 2.4 using a NOR gate with a delay fault, whose output value changes from 0 to 1 after a larger time interval in comparison to the time interval defined in the timing specification.

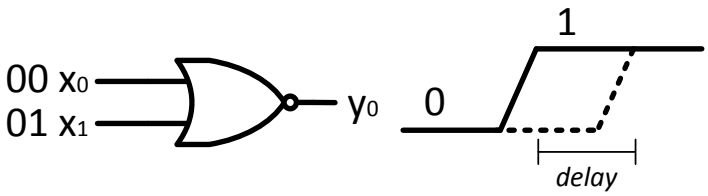


Figure 2.4: Example of a delay fault [13]

At the transistor level, defects modeled as delay faults can result from under- or over-etching during the fabrication process, which produces transistors with channel widths that are much narrower or channel lengths that are much longer than intended, such that, some paths through the circuit may not meet performance specifications. Other causes are a drop in the power supply, shifts in the transistor threshold voltage, increment of parasitic capacitance, high resistance shorts, etc. [13].

At the gate level, delay faults are further categorized into specific delay fault models, such as gate-delay faults, transition-delay faults, path-delay faults, line-delay faults, and segment-delay faults [12, 13, 19, 20].

The gate-delay fault model is a quantitative model in which delays are represented by time intervals. Each gate has a pre-specified nominal delay, but a faulty gate is characterized by a considerably larger delay. Therefore, the gate delay fault is an added delay of certain magnitude in the propagation of a rising or falling transition from the gate inputs to the gate output. The number of gate-delay faults is twice the number of gates.

The transition-delay fault model is one of the basic delay fault models. Faults according to this model make slow signal changes on a line. There are two possible fault types, slow-to-rise and slow-to-fall. In this case, it is assumed that a defect on a line is large enough to affect any path that includes it. The total number of faults is twice the number of lines.

The path-delay fault model considers the cumulative delay of paths from primary inputs to primary outputs, and therefore it is a more realistic fault model in comparison to the gate-delay and transition-delay fault models. Same as for transition-delay faults, two faults (rising and falling transitions) have to be considered for each path. Because many paths exist at the gate level, the use of this model is limited to a selected subset of paths specified as the critical paths.

Segment-delay faults consider slow-to-rise and slow-to-fall defects on segments, whose length L represents a chain of combinational gates. If L is equal to 1 then the segment-delay fault is identical to the transition-delay fault, and if L is the maximum logic depth, it is identical to the path-delay fault. The goal of the segment-delay fault is to reduce the

number of faults that have to be considered in comparison to the path-delay fault. On the other hand, the line-delay fault model considers rising or falling delays on a given line. In contrast to the transition-delay fault model, where the defect affects any path that includes the line, the line-delay fault is propagated only to the longest path of the circuit.

A detailed discussion about delay faults is presented in [19] and recommended for further reading. The discussion includes delay fault properties, advantages and disadvantages, and a classification.

Signal integrity faults

In comparison to delay faults, signal integrity faults are related to signal integrity issues of the DUT interconnections [21–24]. These issues are due to coupling effects between interconnections, ground bounce, power supply noise, electromagnetic interference, reflections, electro migration problems, and even high resistance in the interconnections produced by cracks or narrow width of lines caused by fabrication errors. Their effect on the DUT is the appearance of signal delays and signal distortions (noise) in the form of glitches, overshoots, and undershoots.

A glitch is produced when a signal that should maintain a constant value changes its value to the opposite value temporarily. Signal overshoots and undershoots elevate or drop the signal value respectively, stressing the DUT and increasing the probability of early failures. Figure 2.5 illustrates glitch, overshoot, and undershoot distortions.

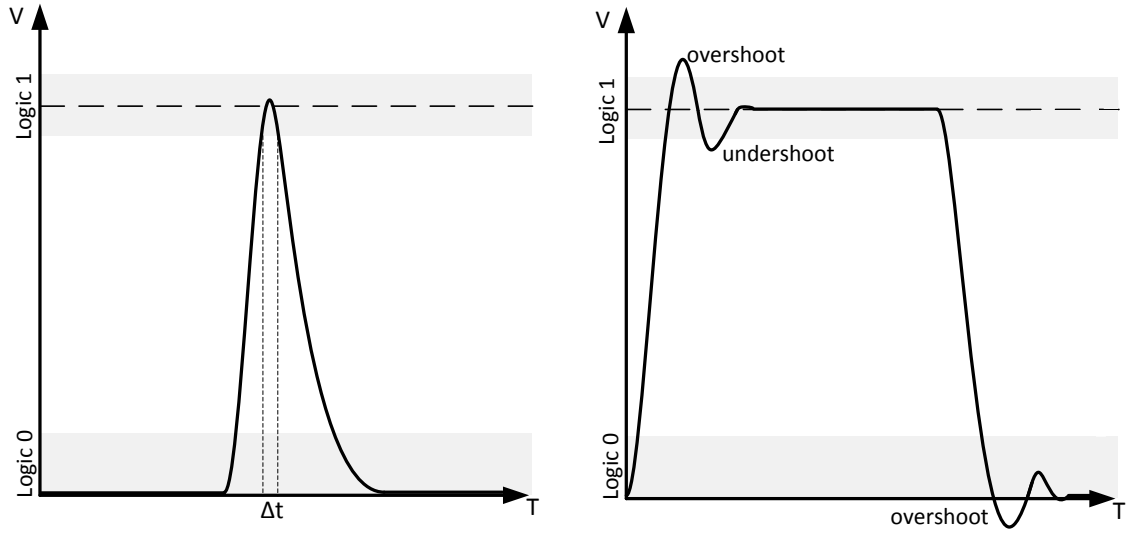


Figure 2.5: Glitch, overshoot, and undershoot distortions [22]

Testing of signal delays and crosstalk effects produced by coupling requires the use of signal integrity fault models. The most frequently used fault model is the maximal aggressor fault model [25]. Figure 2.6 shows one victim and two aggressor lines, where

the coupling between aggressor and victim lines is represented by a generic component Z. Here, the worst case results in multiple aggressor lines having the same transition at the same time. In this case, the voltage level on all aggressor lines is assumed to be the same, which results in no influence between the aggressor lines, whereas all aggressors couple energy to the victim, as shown by the arrows in Figure 2.6.

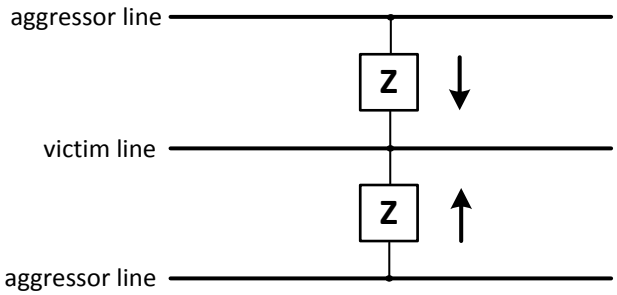


Figure 2.6: Maximum aggressor fault model

The impact of coupling effects depends on the properties of the transition on the aggressor line as well as the properties of the victim line. While a signal change on the aggressor line originates the fault effect, the signal on the victim line can be stable as well as changing in the same or opposite direction of the aggressor lines. Because of coupling, different effects can occur, ranging from a speedup or slow-down of the affected signal to hazards on stable signals. Table 2.1 shows the possible combinations.

Victim signal	Aggressor signal(s): rising	Aggressor signal(s): falling
Stable 0	Positive glitch	Negative overshoot (negative glitch)
Rising	Speed-up	Slow-down
Stable 1	Overshoot (positive glitch)	Negative glitch
Falling	Slow-down	Speed-up

Table 2.1: Effects of aggressor signals [22]

The speed-up phenomenon will normally not cause problems for proper functioning of the DUT, while the effect of a slow-down of the victim line signal causes at least a delay. The negative (positive) glitch on the stable 0 (1) victim signal will not alter the DUT logical behavior but may stress it, resulting in a shorter life time. And the positive (negative) glitches on stable 0 (1) victim signal may cause an incorrect logical behavior of the DUT.

2.1.5 Diagnosis

Diagnosis consists of locating the faults in a structural model of the DUT. In other words, diagnosis maps the observed misbehavior of the DUT into faults affecting its devices or interconnections [11].

When faults are detected, diagnostic procedures are used for two main purposes, the identification and replacement of faulty devices, or the improvement of the quality of the manufacturing process [17]. In the first case, a diagnostic procedure is carried out in order to replace the faulty component. This can take place during the manufacturing process or after the DUT fails in the field. In the second case, diagnosis provides information about faults that are likely to occur during the manufacturing process. This information is used to tune the manufacturing process in order to improve its quality.

There are two main methods used for diagnosis, sequential and combinational fault diagnosis [13]. Sequential fault diagnosis, also known as adaptive testing, is a process, in which the fault location is carried out step by step. In this case, the next step depends on the result of the previous step, a behavior that is represented in the form of a diagnostic tree. An example is shown in Figure 2.7, where the diagnostic tree is composed of a set of potential faults {F1, F2, F3, F4, F5, F6, F7} and a set of test patterns {T1, T2, T3, T4}. Every test pattern is a node in the diagnostic tree that defines the next step to carry out (next pattern to apply) depending on the test result for the actual pattern (pass P or fail F). In this case, the set of faults that remains after each step represents faults that are equivalent (undistinguishable) under the currently applied test set.

On the other hand, combinational fault diagnosis is based on explicit fault dictionaries that associate each fault to a set of test patterns [13]. To locate a fault, it is necessary to match the test results with one of the precomputed expected results stored in the fault dictionary. This method is called combinational fault diagnosis, because a look-up process in the fault dictionary is carried out to identify the corresponding fault.

There are two main types of fault dictionaries and two methods used to store them [26]. In a “full response dictionary” the DUT response is stored for every fault and every test pattern. On the other hand, in a “pass/fail dictionary” the pass/fail result is stored for every fault and every test pattern. The two storage methods are the “list storage method” and the “two-dimensional storage method”. The former is a list that stores for every test pattern the number of faults detected and additional information depending on the fault dictionary type. The latter is a two dimensional matrix, in which each row corresponds to a fault and each column to one of the applied patterns. In this case, the information stored in each matrix element depends on the type of fault dictionary.

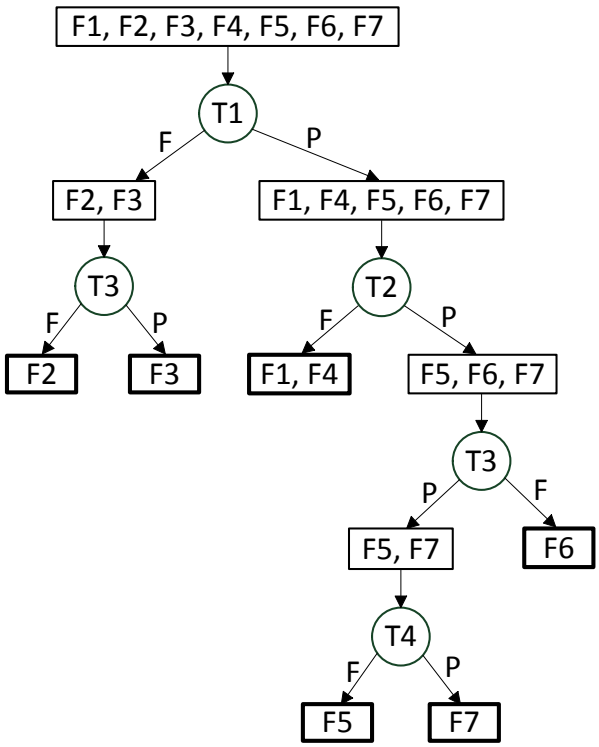


Figure 2.7: Diagnostic tree [13]

Table 2.2 presents the test set (four test patterns) for a DUT with 4 inputs and two outputs with the corresponding fault free response.

Test pattern Index	Test pattern	Fault-free response
0	10010	00
1	01111	00
2	11010	11
3	10101	11

Table 2.2: Test set for DUT

In Table 2.3, the "full response fault dictionary" for ten single stuck-at faults in the DUT is presented. For every test pattern, the list of faults detected by each test pattern together with indices of the faulty DUT outputs is stored.

Test pattern index	Number of faults	Faults			
0	4	1(0,1)	3(0)	4(0)	7(1)
1	2	0(0)	2(0,1)		
2	3	3(0)	5(0,1)	9(0,1)	
3	4	2(1)	6(0)	8(1)	9(0)

Table 2.3: Full response list fault dictionary

In Table 2.4, the "pass/fail two dimensional fault dictionary" is presented. In this case, for every fault and test pattern a single bit is stored. The bit '1' value indicates that the fault is detected by the given pattern.

Fault	Test pattern index			
	0	1	2	3
0	0	1	0	0
1	1	0	0	0
2	0	1	0	1
3	1	0	1	0
4	1	0	0	0
5	0	0	1	0
6	0	0	0	1
7	1	0	0	0
8	0	0	0	1
9	0	0	1	1

Table 2.4: Pass/fail two dimensional fault dictionary

2.1.6 Test quality metrics and test costs

Test quality metrics and test costs are used for two main purposes: as a mean to specify test requirements and as a way to determine the advantages and disadvantages of different test techniques.

Test quality metrics are employed to quantify the quality of a given test or test technique. In this case, the main test quality metrics used are fault coverage, vertical testability and diagnostic resolution.

Fault coverage provides information about the effectiveness of a given test. In its simplest form, this metric is defined as the ratio of the number of detected faults to the total number of faults considered. Here, it is important to know the targeted fault types (e.g. dynamic or static faults) and the existence of additional factors that influence the fault coverage, such as faults that are undetectable or potentially detectable [17].

Vertical testability is related to the application range of a specific test technique. It defines the application level in which the given test technique can be used. For example, built-in self-test (BIST) is a test technique with high vertical testability because it can be used at the wafer, IC, printed circuit board, and even at the system level [17].

Diagnostic resolution indicates the level of fault location achieved by a specific test or test technique [17]. For example, a printed circuit board test that is able to map a fault to an interconnection of an IC has a higher diagnostic resolution than a printed circuit board test that is just able to identify the IC with the faulty interconnection.

Cost is a measurable quantity that plays a very important role in testing. In [27], the authors consider that an ideal test solution should not add any costs to an electronic

system during its design or manufacturing. However, they claim that this is not a real situation because test costs are a significant part of the overall development of modern electronic systems due to their complexity, high-speed, and the lack of test access. Test costs are mainly a function of the test time and resources spent on test development and execution.

Higher test quality translates to higher test costs. Therefore, it is necessary to achieve a balance between the required test quality and costs [12, 15].

2.2 Printed circuit board testing

The main focus of this dissertation is manufacturing test of printed circuit boards (PCBs) in the digital domain. Therefore, this section presents details about the PCB manufacturing process and aspects related to PCB testing.

2.2.1 Manufacturing process

The manufacturing process of PCBs is examined from the point of view of the PCB assembly process, which consists of populating bare boards with the corresponding devices. A bare board is composed of layers of conductors separated by insulating layers that together provide the electrical connections and physical structure for mounting and holding devices. The conductors are typically made of copper, and the insulators are usually some form of fiberglass composite with epoxy resin [28, 29].

Devices assembled on the bare board are classified as passive or active devices [29]. The former are resistors, capacitors, and inductors, while the latter are devices capable of controlling voltages and generating signals with switching behavior (e.g. diodes, transistors, and integrated circuits (ICs)). Active and passive devices use two different technologies for their attachment to the bare board: through-hole technology (THT) and surface mount technology (SMT), respectively. THT attaches devices by inserting their leads to the board through mounting holes. In SMT, the devices do not have leads. They are designed in such a way that they can be directly attached to the surface of the board.

Figure 2.8 shows an example of a board assembly line for mounting THT and SMT devices. Here, some of the stages are dedicated to the assembly of the bare board, while other stages are used for testing purposes. The inclusion of test stages at different points of the assembly line is realized in order to identify different types of faults as soon as possible. This is based on the “rule of ten”, which states that finding a fault at any production stage costs 10 times more than finding the same fault at a previous stage. This

rule can be applied to the manufacturing of a complete electronic system, as well as to the PCB manufacturing process [30].

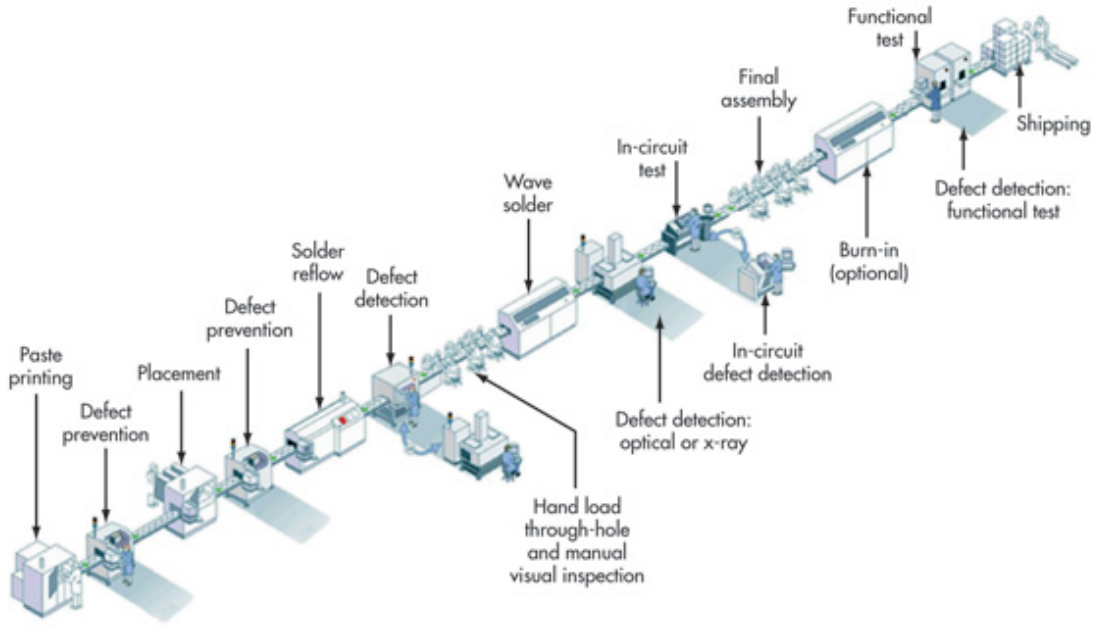


Figure 2.8: Example of an assembly line [31]

In comparison to THT devices, SMT devices are mounted to the board at earlier stages of the assembly process [29]. In Figure 2.8, the first six stages from left to right are used to mount SMT devices, and the two following stages are used to mount THT devices. These stages are presented in Sections 2.2.1.1 to 2.2.1.3 in more detail.

2.2.1.1 Application of the solder paste

In this stage, the solder paste is applied to the metal contacts of the bare board in which SMT devices will be mounted. This can be done by pneumatic dispensing using special syringes, or by paste printing methods based on special printers and a stencil to define the locations of the solder paste. The application of the solder paste can be skipped if the wave soldering method is used to solder SMT devices to the bare board.

This stage is normally followed by a test stage (first defect prevention stage in Figure 2.8), which checks the amount and location of the solder paste.

2.2.1.2 Placement

In this stage, the SMT devices are mounted on the bare board by matching their metal terminations with the board regions covered with the solder paste. In case that the wave soldering method is used to solder the SMT devices, the attachment is done using an adhesive to hold the devices in the correct position.

On the other hand, the placement of THT devices does not involve the application of a solder paste to the board. In this case, leads are inserted through the board holes, and then they are bent, formed, and cut in order to provide the required attachment. For THT ICs, the bending, forming, and cutting actions are not necessary.

In the same way as for solder paste application, a test stage follows the placement. This stage checks that devices are not missing and that they are properly oriented and aligned.

2.2.1.3 Soldering

After device placement, the soldering stage is carried out. During this stage, the board and metallic terminations of the devices are joined together in an intermolecular bond to the board interconnections in order to guarantee the electrical continuity of the joint. Two of the automatic methods used for this purpose are reflow and wave soldering. Both methods are classified as mass soldering methods because they are able to make several solder joints simultaneously.

- Wave soldering is the standard method used for leaded THT devices. It is done by passing the populated board held in a horizontal position over the crest of an artificially created wave of molten solder. Once the board is exposed to the molten solder, the joints are created.
- Reflow soldering creates a joint by re-melting the previously applied solder paste. This is carried out by the application of heat, which is produced through convection of a hot gas, infrared radiation, or heat panels.

After soldering and final assembly process, several test stages are carried out to test the PCB. Their purpose is to avoid shipping faulty PCBs and increase the quality of the manufacturing process [32]. In this thesis, all the test actions carried out during the assembly process are referred to as board-level testing.

2.2.2 Board-level testing

Over the last decade, board-level testing has changed its focus from finding component failures towards finding defects produced during the PCB manufacturing process [29, 33]. This has led to the development of test strategies that assume that devices populating the PCB work properly, and therefore the dominant manufacturing defects are located at the soldering joints [16, 32].

2.2.2.1 Defects at the board level

The occurrence of defects is caused by process variations during the PCB assembly. They are caused by inaccurate placement of devices, solder melting problems, low temperatures, excessive exposure to heat, oxidation, organic residues, and the usage of excessive flux [29].

Examples of defects appearing in the PCB are shorts between joints and cracks (Figure 2.9), lifted leads or leads with insufficient solder (Figure 2.10), and misaligned or tombstone devices (Figure 2.11). Other types of defects are missing or misplaced devices and damage of the bare board during the assembly process.

These defects manifest themselves in dissimilar ways, causing the PCB to operate outside its specification. At the logic level, their effect on the PCB operation can be modeled by stuck-at, bridging, or dynamic faults.

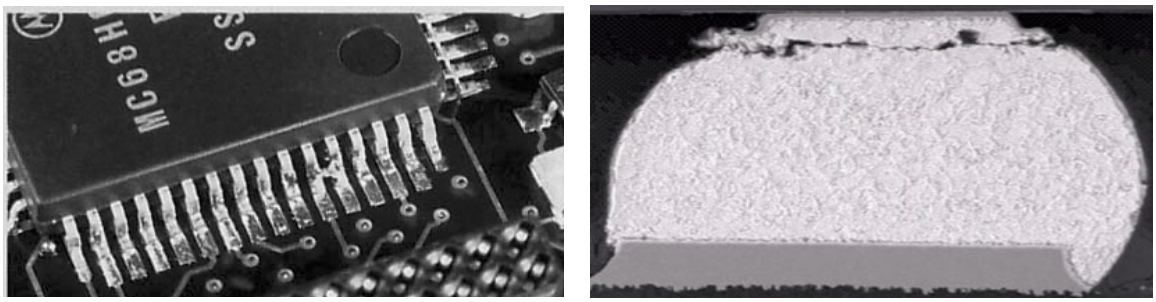


Figure 2.9: Short defect [29] and crack in ball grid array solder joint [34]

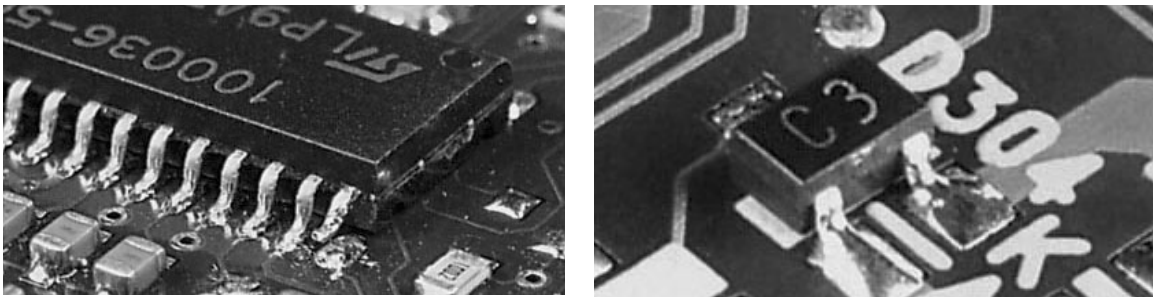


Figure 2.10: Lifted lead and insufficient solder defects [29]

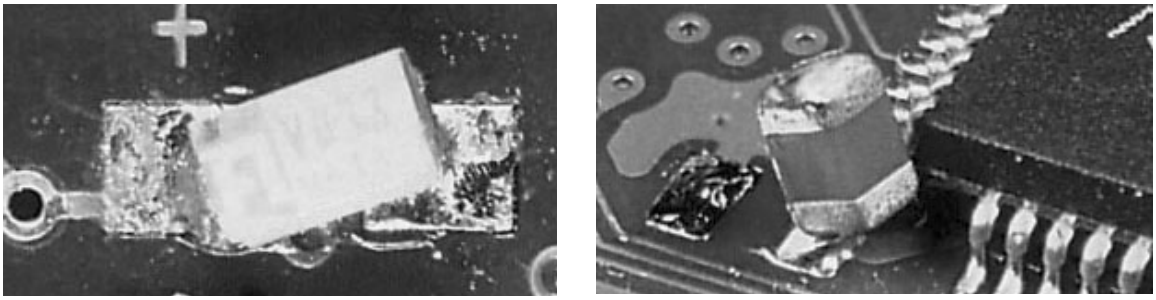


Figure 2.11: Misaligned and tombstone component defects [29]

2.2.2.2 Test strategies

Modern PCBs are considered to have high design and manufacturing complexity. They are characterized by several wiring layers, small dimensions, and a dense population of devices with different packaging technologies. In order to obtain competitive test quality metrics, it is necessary to employ test strategies that rely on multiple and complementary test techniques [35, 36].

A general test strategy is presented in Figure 2.12. It comprises three main steps, which are known as inspection, structural test, and functional test. In Figure 2.8, manual inspection, X-ray inspection and optical inspection are part of the inspection step, whereas in-circuit test (ICT) forms part of the structural test step. After these two steps are carried out, functional tests are typically applied to the PCB. Section 2.2.3 presents board-level test techniques in more detail.

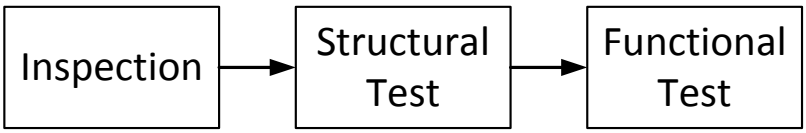


Figure 2.12: General test strategy during PCB assembly

2.2.3 Board-level test techniques

In this section an overview of widely used test techniques is provided. For this purpose, a description of the main properties of these techniques is presented, with the aim of showing their purpose, advantages and disadvantages.

2.2.3.1 Inspection

Inspection relies on visual or image information about the PCB that is used to evaluate if it is faulty or not. Therefore, it is not necessary to power up the PCB or to physically access it with some test instrumentation. However, inspection by itself cannot determine if a PCB works, because the PCB is not stimulated with any input signals and no output signals are measured. It can only determine that a PCB looks correct [33].

Visual inspection is usually used after application of the solder paste, placement, and soldering stages. In the first case, it inspects the quality of the solder paste, while in the second case it looks for missing devices, alignment, and orientation problems. Here, the repair costs are low because the soldering stage has not been carried out. Visual inspection is done manually by visual inspectors that make use of inspection equipment such as magnifiers or microscopes.

The inspection actions carried out after the soldering stage target the solder joints between the board and the devices. For this purpose, automatic inspection techniques are utilized because they provide better reliability in comparison to a manual process. One of the automatic inspection techniques is known as automated optical inspection (AOI). This technique takes an image of an entire or a part of the PCB, and matches it against a proven correct variant. The limitation of AOI is that it is not effective in examining hidden solder joints such as solder joints of ICs with ball grid array (BGA) or land grid array (LGA) packages. To overcome this problem, automated X-ray inspection (AXI) is used. AXI is able to look through the ICs and the PCB, allowing the inspection of hidden solder joints, double-sided boards and even inner interconnection layers [29, 33].

2.2.3.2 Structural test

Smaller devices and denser populated boards complicate and increase the costs of the inspection process due to the need for a high image resolution, which increases the camera positioning and image-processing time. This slows down the inspection process and increases the likelihood of falsely flagging defect-free PCBs as defective [33]. Therefore, structural test techniques are incorporated into the test strategy in order to electrically stimulate the PCBs and find defects not detected with inspection techniques.

Structural test techniques check the structure of the PCB instead of the function that the PCB should perform. The idea is to apply suitable test patterns targeted at sensitizing specific faults in a way that a faulty circuit will produce an erroneous response. Their advantage over functional tests is that these techniques can be used to obtain high fault coverage based on a small set of patterns, reducing test time dramatically [12].

Structural tests make use of classical test techniques, which are generally divided into two main classes, invasive and noninvasive.

Invasive test techniques

Invasive test techniques are characterized by external test instrumentation that makes physical contact with different parts of the PCB (test points). Two popular invasive test techniques are known as in-circuit testing (ICT) and flying probe testing (FPT).

ICT was one of the first structural test techniques on the market [32]. Figure 2.13 shows the typical architecture of an in-circuit tester. The tester consists of a test fixture with a bed of nails, a vacuum port used to firmly attach the PCB to the test equipment, and a receiver connected to an external computer in control of the testing process. The bed of nails comprises multiple fixed test probes that are used to access the PCB test points.

ICT can provide very high fault coverage in cases where good external test access is provided. Practically all structural faults, such as opens, shorts, wrong or defective devices, can be covered. Additionally, ICT is easily automated and provides good diagnostic resolution. Unfortunately, the cost of ICT is very high due to the expensive equipment and the PCB-specific test fixture required for every new PCB. Additionally, the increasing density and continuous miniaturization of PCBs, as well as the use of SMT ICs with BGA or LGA packages significantly limit test access [32].

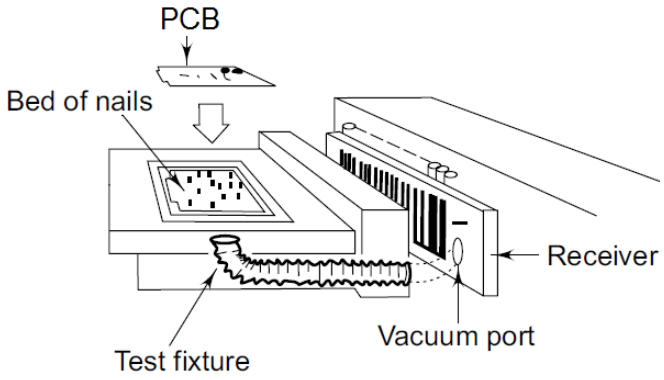


Figure 2.13: ICT equipment [29]

The effort and costs related to the adaptation of the test fixture to the PCB led to the development of the FPT technique [32]. In this case, instead of having a fixed test fixture, high-speed test probes that move across the PCB are used. The probes can reach test points and devices located across the board, and therefore they can perform all kind of electrical tests. In this case, it is necessary to program the movement of the probes.

The FPT equipment is more flexible and independent of the target PCB, reducing test development costs. Nevertheless, its main advantage is also its main disadvantage, because the movement of the probes clearly slows down the test execution in comparison to ICT [32].

The main disadvantage of both test techniques is the physical contact required to access the PCB, making their use very problematic in modern PCBs. This led to the development of noninvasive test techniques.

Noninvasive test techniques

Noninvasive test techniques do not use external test instrumentation to contact test points on the PCB in order to have access to it. They use embedded test resources located in the ICs, which are accessed through external automatic test equipment (ATE).

The most popular noninvasive test technique is known as Boundary Scan (BScan). In 1985, the Joint Test Access Group (JTAG), made up of companies primarily from Europe

and North America, was set up with the goal of developing a new technology that could provide test access to the PCB without the utilization of physical test probes. The technology received positive feedback from industry, and therefore it was adopted by the Institute of Electrical and Electronics Engineers (IEEE) and standardized as "IEEE 1149.1 Test Access Port and Boundary Scan Architecture" in 1990. Several revisions have been made to the standard, with IEEE 1149.1-2013 being the latest one of them [1]. The standard is often referred to as JTAG, due to its history of development [37]. In comparison to ICT and FPT, BScan allows easier and faster test development, making it possible to obtain better test quality metrics and reduce test costs [38, 39].

Figure 2.14 shows the BScan architecture included in BScan compliant ICs. The access to the IC is done by means of a serial test bus, which provides access to the IC and input/output (I/O) pins.

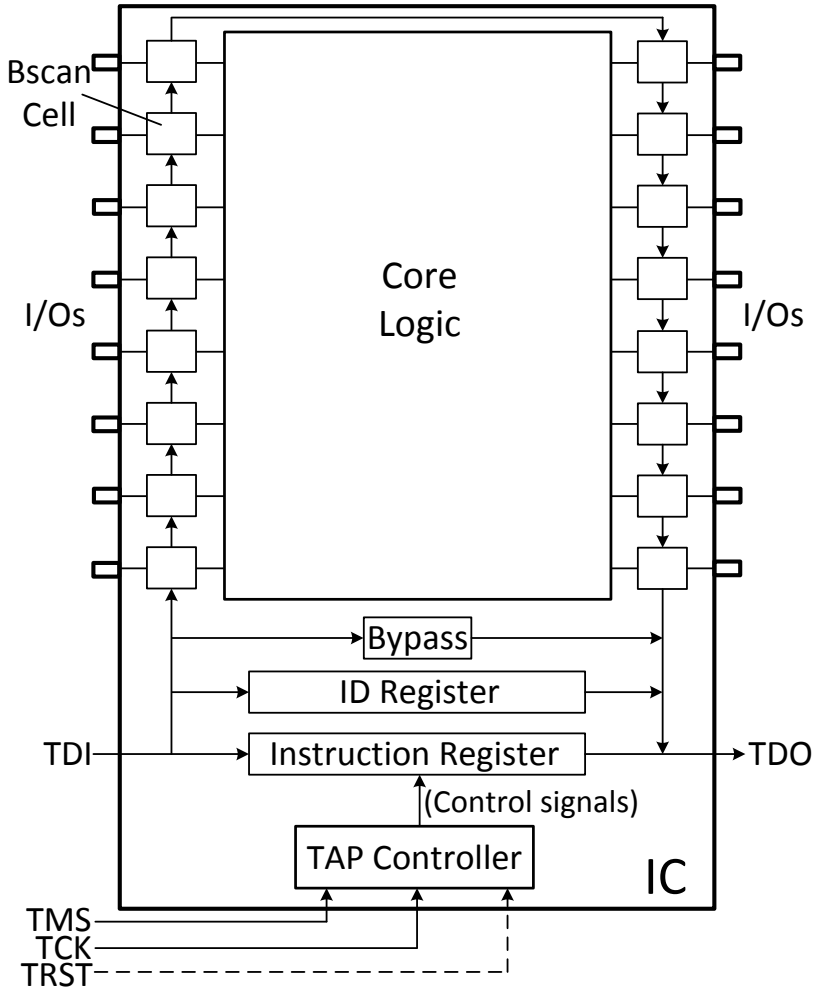


Figure 2.14: Simplified architecture of an 11.49.1 compliant IC [37]

The Test Access Port (TAP) comprises five pins, which are used to transfer information to the embedded test resources. Test Clock (TCK), Test Mode Select (TMS), and the optional pin Test Reset (TRST) are input pins used to synchronize and control the transfer

of information. Test Data In (TDI) and Test Data Out (TDO) are used to shift data in and out of the IC serially.

In the IC die, there is a finite state machine known as the TAP Controller. It responds to the TCK, TMS, and TRST values, generating control signals used to control the instruction register (IR) and data registers (DR) forming part of the BScan architecture.

The instruction Register (IR) is accessed through the TDI and TDO pins, and it is used to load instructions that set the mode of operation of one or more Data Registers (DRs). Figure 2.14 shows three DRs, which are known as the BScan, Bypass, and ID registers. The BScan DR is composed of serially connected BScan cells located between the IC pins and core logic. Its main purpose is to separate the pins from the core logic in order to control and observe the pins' activity during the test execution. The Bypass DR consists of a single cell used to reduce the shifting path (TDI to TDO) of an IC. On the other hand, the Device Identification (ID) DR is optional, and it is used to identify the IC.

The BScan compliant ICs are connected together forming a serial scan chain or scan path, which is used to shift in test patterns and shift out test responses. The main limitations of this technique are the serial structure of the chain and low TCK frequencies. They cause low coverage values for dynamic faults and prohibitively long test times, affecting the test quality and cost [40, 41].

The time it takes to shift in and out a single test vector is known as the test access time, which is calculated as presented in [30]. For this purpose, Equation (2.1) describes the total length of the BScan chain C_{Total} based on n BScan compliant devices, each of them with a BScan register of length l_i.

$$C_{Total} = \sum_{i=1}^{n} l_i \tag{2.1}$$

If F_{TCK} is the clock frequency of TCK, then the test access time T_{ACC} is equal to:

$$T_{ACC} = \frac{C_{Total}}{F_{TCK}} + \delta_s + \delta_h + 6 \cdot \frac{1}{F_{TCK}} \tag{2.2}$$

In this case, δ_s represents the software delay and δ_h the hardware delay of the ATE. The additional 6 TCK cycles include the transition of the TAP state machine from the Shift-DR TAP state to the Update-DR state and then back to the Shift-DR state. The application speed achieved with BScan F_{APP} is the opposite to T_{ACC} and is calculated as follows:

$$F_{APP} = \frac{1}{T_{ACC}} \tag{2.3}$$

Equations (2.2) and (2.3) are an important way to calculate the test time and application speed achieved with BScan for a given PCB. For this purpose, δ_s and δ_h are typically approximated to $7 \cdot \frac{1}{F_{TCK}}$.

Nowadays, the standard is not only used for testing purposes but also as a multipurpose access port for communication, debugging, or even programming of ICs [42, 43]. A detailed description of the BScan operation and architecture is presented in [1, 37] and recommended for further reading.

2.2.3.3 Functional test

After all devices and connections of the PCB are structurally tested, it is necessary to check the functional behavior of the board. This is performed by functional tests at the end of the manufacturing process. A functional tester exercises the PCB through its edge or test connector [33]. The tester applies a signal pattern that resembles the normal operation of the board, and then examines output pins to ensure a valid response. In some cases, in order to complement edge and test connectors, a modified bed of nails is used for observation purposes only.

In contrast to structural tests, a functional test is performed in a normal operation mode, providing the capability to check the board in close to real life situations. This means that it can be used to verify the board performance, mimicking its behavior in the end product. A functional test is carried out at full speed, thereby uncovering racing and other dynamic problems that escape static or low-speed tests.

This technique is very expensive because it cannot be standardized. Typically, there are no fully automatic generation tools, making it necessary to develop all tests manually for each type of PCB, which is a very complex and time consuming task. Additionally, a functional test only can determine whether the board functionality corresponds to its specification or not. This does not include any diagnostic information, and therefore it is not possible to locate the defects that produce the incorrect behavior [33, 44].

2.3 Embedded board-level test techniques

Test strategies for modern PCBs cannot rely on classical structural and functional test techniques due to the high test costs and the impossibility to guarantee good test quality metrics in a modern PCB manufacturing process. Columns two to four of Table 2.5

summarize main properties and limitations of these techniques. The information was obtained from [30, 35, 45, 46].

It can be observed that ICT and FPT have high static fault coverage and test automation values, but they have access and cost limitations. BScan provides better test access to the PCB given that it is a noninvasive test technique. But it is not able to detect dynamic faults and requires long test times. Functional test provides high functional fault coverage values, making it possible to detect defects that appear at the DUT operation speed. However, it is difficult to automate, provides low diagnosis resolution, and has high costs.

	FPT	ICT	BScan	Functional	Embedded
DUT access mechanism	Flying probes	Fixed nails	Scan cells	Edge connector	Embedded
Test access	Low	Low	High	Low	High
Static fault coverage	High	High	High	Uncountable	High
Dynamic fault coverage	No	High	No	Uncountable	High
Functional fault coverage	Uncountable	Uncountable	Uncountable	High	High
Diagnosis	High	High	High	Low	High
Test time	Very high	Low	High	High	Low
Test automation	High	High	High	Low	High
Test cost	High	Very high	Low	High	Low

Table 2.5: Comparison of different test techniques

Based on the comparison of test techniques, there is no ideal technique that meets all test requirements. Therefore, in order to overcome the limitations of classical test techniques, modern test strategies complement them by means of embedded test techniques. Embedded test techniques have a high level of access to the PCB and ICs, and they are able to execute tests at-speed, in short time, with high diagnosis resolution, and at low cost. The last column of Table 2.5 represents the ideal properties of an embedded test.

During the last years a large amount of research has been done in this area, whether altering the BScan technique [40, 47], using embedded custom test circuits [41, 48] or smart devices already available in the PCB. The use of smart devices already available on the PCB has gained considerable importance in the last decade because it does not require the modification or addition of new devices to the PCB. In this case, these devices correspond to processors or field programmable gate arrays (FPGAs) that are also used during normal operation of the PCB.

2.3.1 Processor-based testing

Processor emulation was a popular approach in the 80's and early 90's [45, 49, 50]. It relied on the insertion of processor specific pods into the processor socket that allowed

replacing the processor with an external processor emulator. In this way it was possible to take control over the PCB bus for testing and debugging purposes. However, the decreasing physical access and increasing clock rates of modern PCBs made emulators difficult or impossible to design.

It was not until the advent of on-chip debug interfaces that the idea of processor emulation was reconsidered, giving birth to Processor-based testing (PBT). The debug interface, embedded in the microprocessor die, provides the required control and observation features necessary to carry out traditional emulation actions. For example, it can be used to stop the processor, read/write memory and I/Os, set breakpoints, execute single steps, and trace code. This technique is commonly implemented using the JTAG interface as the main communication link between the debug interface and the ATE, and it is available in commercial tools such as VarioTap [45] and Processor-controlled Test [49, 50].

In PBT, the processor on the PCB becomes an embedded tester. It is in charge of accessing the ICs and executing test functions. During the test execution, the ATE takes control over the processor via its debug interface and performs tests based on two possible operation modes, which are known as online and offline modes [46, 51–53].

In offline mode, the complete test program is translated into a set of microinstructions that is loaded into the processor memory. The translation and loading process is carried out by the ATE, which is also in charge of starting the program execution. During the test execution, the processor works independently and does not interact continuously with the ATE. The test program stores pass or fail results in one or more general purpose registers accessible through the debug interface. These results are read by the ATE for further evaluation and diagnosis.

Due to the autonomous operation of the processor, the offline operation mode requires a large memory space to store the complete test algorithms, test patterns, and analysis functions. However, in some cases, it is possible to make use of special algorithms as part of the test program (e.g. walking one, counting sequence, pseudo random pattern sequence, etc.) that generate driving and expected values on the fly, reducing the memory requirements.

In online mode, test steps are executed separately under strict control of the ATE. For this purpose, the test program is split into a sequence of steps and a special interpreter is loaded into the processor internal memory. Test steps are basically a sequence of microinstructions that describe the actions necessary to apply a test pattern to a given DUT. They are transferred to the processor and received by the interpreter, which is in

charge of executing them. After each step, results are stored in the processor registers and retrieved by the ATE.

The main drawback of online mode in comparison to offline mode is the speed at which test patterns are applied. The online mode is considerably slower due to the continuous communication overhead between the ATE and the processor, and the application of a sequence of test patterns at-speed is not guaranteed. The latter can be an issue for the detection of dynamic faults [46].

PBT is a very interesting test technique due to low test costs and the potential to execute tests at-speed. In order to reach the ideal properties of an embedded test (Table 2.5), it is necessary to automate the development flow and provide support for a large spectrum of debug interfaces and processors [46, 51–53]. The fulfillment of these tasks has been proven to be very difficult, and therefore additional research is still needed [46].

2.3.2 FPGA-based testing

Instead of using processors, FPGA-based testing (FBT) employs field programmable gate arrays (FPGAs) located on the PCB for testing purposes. On this subject, different approaches have been investigated [30, 54–58], and commercial tools such as ChipVorx [59] and FPGA-controlled test [60] are already on the market.

In the same way as PBT, FBT has the potential to execute tests at-speed, and therefore it can be used for detection of static and dynamic faults. It is a low cost approach because it does not require the addition of test points, special devices, or expensive ATE. In this case, the main challenge is the development of a test system embedded in the FPGA that is able to fulfill the properties of an embedded test technique as presented in Table 2.5.

Given that FBT is the main research topic of this dissertation, Chapter 3 presents a detailed analysis of FBT features, available approaches, and the approach followed in this work.

2.4 Soft-core and test processors

This section presents soft-core processors and test processors found in the literature. Soft-core processors are configurable cores that are synthesized and instantiated on the FPGA just like any other FPGA-based design. Compared to hard-core processors, soft-core are more flexible because the hardware description of the soft-core processor can be changed. However, they have lower performance because they are implemented using the

configurable blocks and routing resources of the FPGA. Test processors are application-specific processors developed for testing purposes. In comparison to general purpose processors, they are tailored and specialized for the execution of test functions in order to obtain higher performance values and reduce test cost.

2.4.1 Soft-core processors

Soft-core processors are classified into two basic categories:

- Proprietary soft-core processors
- Open source soft-core processors

Proprietary soft-core processors are developed by intellectual property (IP) design companies or FPGA vendors. They are fully verified and have good software development tool support, libraries, and documentation. Their main disadvantages are the cost of license fees, restricted access to the source code, limited support for vendor-specific or family-specific FPGAs, and that developers are forced to use vendor-specific development tools for object code generation, debugging, verification, and design instantiation purposes.

On the other hand, open source soft-core processors are mainly developed for educational and research purposes. They follow the GNU philosophy, making the source code available and editable without any license fees. Their implementation is not restricted to a specific FPGA family or vendor, and development tools are in most cases open source. Depending on processor popularity and acceptance, they might not be fully operational or fully verified.

Open source soft-core processors are further divided into processors with an original instruction set architecture (ISA) or with a cloned ISA. The former have their own ISA, while the latter are developed based on popular ISAs or a subset of them. Cloned soft-core processors are compatible with libraries and development tools of the original processor. On the other hand, the development of all libraries and software tools is necessary for processors with their own ISA.

Table 2.6 shows the most relevant proprietary and open-source soft-core processors found in the literature. The selection was made based on their popularity in the research community, development stage (stable or beta release), and existing support. The columns *Type* and *Clone* map the processor in one of the categories. The *ISA* column provides information about the ISA, and the *Release stage* column shows the status of the current release. The number of bits necessary for coding the content of a register or an instruction

is shown in the *Data width* and *Instruction width* columns, respectively. Multiple data width values indicate either the existence of various processor versions (Ensilica 1600 and 32X0) or a configurable data width (Proteus). The value 8x in the instruction width of the T48 μcontroller indicates instructions of variable length (byte multiples), which are typical in CISC processors.

The *Pipeline stages* column presents the number of pipeline stages supported by the processor. A hyphen indicates a multi-cycle implementation (no pipeline). Multiple values indicate various processor versions (Nios II, Amber) or a configurable pipeline (Microblaze, Xtensa LX6, Plasma, and Proteus). The *Configurable/extensible* column indicates support for configuration parameters and extension of the instruction set. Configuration parameters are used to activate/deactivate complete processor modules and define the value of properties such as number of registers, memory sizes, number of pipeline stages, etc. Extension of the instruction set indicates support for implementation of new instructions.

Finally, the *FPGAs* column indicates the supported FPGAs. As shown in the table, the Nios II, Microblaze, and Picoblaze processors are restricted to a specific vendor, while the Ensilica and Xtensa LX6 can be implemented in FPGAs supported by the vendor. Open-source processors do not have any FPGA restrictions, but they may require editing the source code in order to use vendor-specific components such as multipliers, RAM blocks, JTAG interfaces, etc.

2.4.2 Test processors

Application-specific processors developed exclusively for testing purposes are of great interest to the research community. They are equipped with specialized test functions in order to improve test execution time and adjust their properties to the test scenario.

They are classified into two basic categories:

- Hardwired test processors.
- Programmable test processors.

2.4.2.1 Hardwired test processors

The first category comprises processors that are not instruction programmable. They are equipped with hardwired linear feedback shift registers (LFSRs) for generation of patterns, signature registers for the analysis of test responses, and special interfaces.

Soft-core processor	Type	Clone	ISA	Release stage	Data width	Instruction width	Pipeline stages	Configurable/ extensible	FPGAs	Begin-end development
Nios II [61]	proprietary	no	Nios II	stable	32	32	-, 5, 6	yes/yes	Altera FPGAs only	2004-2015
Microblaze [62]	proprietary	no	Microblaze	stable	32	32	3, 5	yes/yes	Xilinx FPGAs only	2002-2015
Picoblaze [63]	proprietary	no	Picoblaze	stable	8	18	-	no/no	Xilinx FPGAs only	2003-2014
Ensilica cores [64]	proprietary	no	eSI_RISC	stable	16, 32	16, 32	5	yes/yes	vendor[1]	2009-2011
Xtensa LX6 [65]	proprietary	no	Xtensa	stable	32	16, 24	5, 7	yes/yes	vendor[1]	1999-2015
T48 µcontroller [66]	open source	yes	MCS-48 (CISC)	stable	8	8x	-	no/no	all	2004-2009
Plasma [67]	open source	yes	MIPS I (TM)	stable	32	32	2, 3	yes/no	all	2001-2013
Copyblaze [68]	open source	yes	Picoblaze	stable	8	18	-	no/no	all	2011-2013
Leon 3 [69]	open source[2]	yes	SPARC-v8	stable	32	32	7	yes/no	all	2003-2015
aeMB [70]	open source	yes	Microblaze	beta	32	32	3	yes/no	all	2004-2009
Amber [71]	open source	yes	ARM-7	stable	32	32	3, 5	no/no	all	2010-2013
OpenRisc 1200 [72]	open source	no	OpenRisc 1000	stable	32	32	5	yes/yes	all	2001-2015
Mico8 [73]	open source	no	Lattice Mico8	stable	8	18	-	yes/no	all	2005-2013
Mico32 [74]	open source	no	Lattice Mico32	stable	32	32	6	yes/no	all	2006-2015
Proteus [75]	open source	no	Proteus	beta	8, 16	16	3, 4[3]	yes/yes	all	2009-2009

Table 2.6: General purpose soft-core processors

[1] Supported FPGAs are defined by the soft-core processor vendor.
[2] Open source license if the processor is used for education or research purposes.
[3] The soft-core processor uses two clock cycles per pipeline stage.

The first member is the Testchip [76], which was developed for research and educational purposes to test VLSI circuits in 1990. The goal was to reduce test costs by replacing a high-cost ATE with a low-cost ATE and a Testchip. The Testchip is included in the PCB as an additional IC to carry out test functions typically performed by the high-cost ATE. The Testchip is used for pattern generation, pattern application, test result acquisition, and signature analysis. The adaptation to a DUT is carried out based on configuration registers and random access memory (RAM) accessed through a bus interface in order to configure the test lengths and weights of the pattern generators. The pattern generators produce weighted random patterns, which are applied in parallel to the primary inputs of the DUT and serially (via scan-path) to the internal registers of the DUT. A signature analyzer is able to compact the acquired parallel and serial test responses into a signature, which is sent to the ATE for further analysis.

Figure 2.15 shows the architecture of the Testchip. The control unit is in charge of the coordination of the test execution and communication with the ATE. The pattern generators are implemented as LFSRs, and they use RAM to store the configuration information of the pattern sequences. The shift register is in charge of serial-to-parallel and parallel-to-serial conversions for the execution of scan-path tests. The signature register generates the signatures based on the test responses.

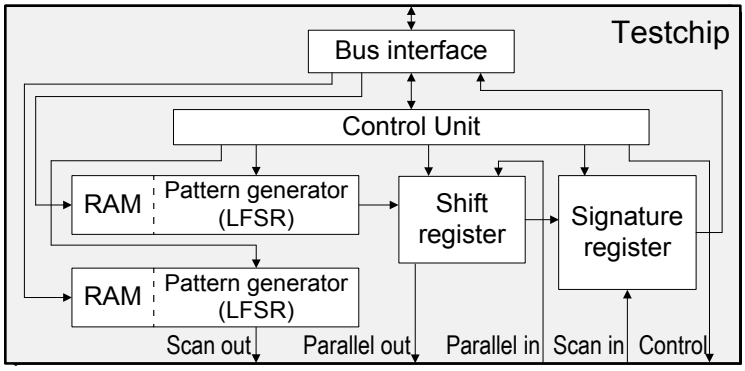


Figure 2.15: Testchip architecture [76]

A similar test processor, called the Test processor chip, is presented in [77–80]. The concept, field of application, and adaptation mechanism share the same concepts of the Testchip. In this chip, the pattern generation is based on multiple polynomial LFSR reseeding [77–79] and generalized LFSRs [80].

2.4.2.2 Programmable test processors

The second group of test processors is instruction programmable, which makes it possible to implement test algorithms in a more flexible way. They are used to test ICs, and they -or at least some of their modules- are embedded in the DUT. Additionally, they have

special circuits for the generation and application of patterns, configuration options for adaptation to the test scenario, and serial interfaces for the execution of scan-path tests.

The test processor presented in [19, 81–86] is a standard 16-bit reduced instruction set computer (RISC) used for testing the components and interconnections of a system on chip (SoC). It is embedded in the SoC in order to improve the test observability and controllability, and reduce test costs. It is equipped with special test functions, and different versions of the processor are available for adaptation purposes.

In [81] the test processor is introduced to the scientific community for the first time. It is presented as a minimum-size processor designed using the hardware description language VHDL. This version supports 32 basic instructions and two special test instructions for pattern generation and compaction. The special instructions use two registers of the register file to implement an LFSR or a multiple input signature register (MISR) based on the built-in logic block observer (BILBO) scheme. The test processor supports deterministic instruction execution given that it is not equipped with pipeline stages or cache memories. A specialized I/O controller applies patterns and acquires test responses. The I/O controller has four parallel ports, one serial scan-path port, and dedicated circuitry to perform a fast comparison of test responses. For this purpose, special structures known as bus reflectors [82] are placed at the other end of the interconnections.

Figure 2.16 shows the architecture of the test processor, which comprises four basic modules: memory, control-path, data-path, and I/O controller. Registers g and h of the register file are used to implement a LFSR or MISR. The I/O controller has five ports to access the DUT and ATE, a configuration register to configure the ports, and test, rollback, and switch (TRS) controllers that perform the fast comparison of patterns.

In [82–84] emphasis is given to a hierarchical test scheme. The processor is equipped with self-testing and on-line test features that improve its reliability. For this purpose, two different processor versions are presented. The first version has specialized structures for the processor self-test. The second version includes online-testing capabilities for the control- and data-path. In [19, 85, 86] more importance is given to the adaptability to the test and area requirements of the SoC. For this purpose, four main processor versions are presented. The basic version is equipped with eight registers and no interrupts. The second version is equipped with 16 registers, interrupts, and self-test structures. The third version supports online test features, and the fourth version supports special hardware for multiplication and division. A fifth processor version with pipelining is also mentioned, but there is no further information about it.

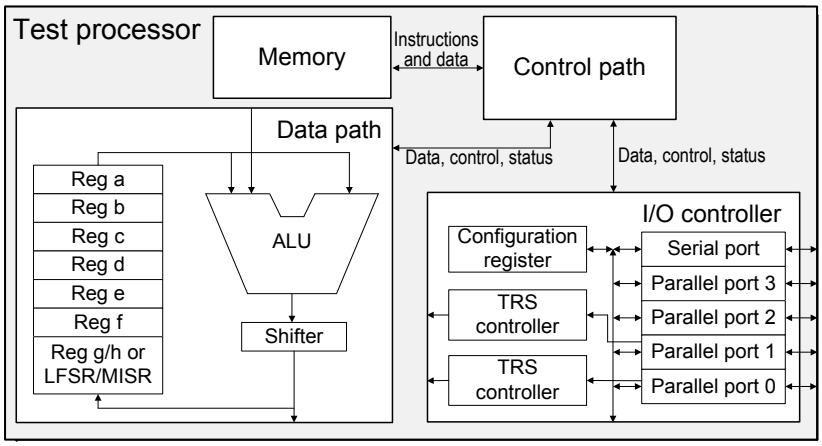


Figure 2.16: Simplified architecture of the test processor presented in [81]

Another instruction programmable processor was developed for the functional test of asynchronous circuits [87–89]. It supports the timing and arbitration indeterminism of asynchronous circuits. The first versions [87, 88] are based on the test processor introduced in [81], while the version presented in [89] follows a different approach.

The test processor in [87] is a 16-bit RISC processor designed with the processor description language known as language for instruction set architecture (LISA). It can be implemented in the SoC, ATE, or as a combination of both options. It supports 53 instructions with two operands per instruction. Some of these instructions are used for the implementation of asynchronous handshake protocols, generation of patterns, and compaction of test responses. The processor has separate data and program memories, four pipeline stages with result forwarding, and sixteen registers, in which four of the registers are used to implement two LFSRs or MISRs. It is equipped with an I/O interface with eight I/O ports and four asynchronous handshake ports that are used to generate the asynchronous signaling. The asynchronous handshake ports can be configured, allowing adjustable and programmable communication.

The work presented in [88] tries to solve the problems of the previous version, namely the pattern storage capacity and pattern application speed. In order to improve the storage capacity, the ISA of the processor is extended to 32 bits. Additionally, it uses three operands per instruction, and supports a variety of logic, arithmetic, and control flow operations. It uses any of the registers of the register file as a LFSR/MISR, and is equipped with 16 I/O ports and asynchronous handshake ports. In order to improve the pattern application speed, the processor is equipped with special instructions that perform complex operation sequences. For pattern application purposes, a single instruction can read a pattern from memory, or generate the pattern using an LFSR, and subsequently transfers the pattern to one of the processor I/O ports. For result acquisition purposes, a

single instruction acquires a test response from one of the I/O ports, and subsequently it can compare the test response to a value stored in memory, or compact it using a MISR.

In [89] the concept presented in [87, 88] is reorganized to focus exclusively on the application and acquisition of patterns, and the processor is exclusively implemented in the ATE. The idea is to avoid the limitations of the previous test processor versions, in which the data transfers between DUT and test processor are still the bottleneck of the approach. In order to avoid these limitations, the test processor proposed in [89] relies on three independent units: a port switch, a memory access controller, and a sequencer.

The port switch couples asynchronous handshake ports with I/O ports, allowing a quick configuration of the ports and the possibility to carry out parallel data transfers between the DUT and test processor. The memory access controller coordinates and performs independent data transfers between the data memory and I/O ports. The sequencer is a small instruction programmable unit, which is used to coordinate the flow of data and configure the port switch and memory access controller. It is described in VHDL, has a 32-bit data bus and a 24-bit instruction bus, two pipeline stages, and eight registers. It supports a limited instruction set with 22 instructions, most of them implementing complex sequences necessary to configure and control the two other units.

Figure 2.17 shows a simplified illustration of the test processor architecture. The data and program memories are independent from each other, and there are communication links between the sequencer, handshake (HS) ports, memory access controller, and port switch. Although not shown in the figure, access to the test processor is carried out by means of an external interface included in the sequencer.

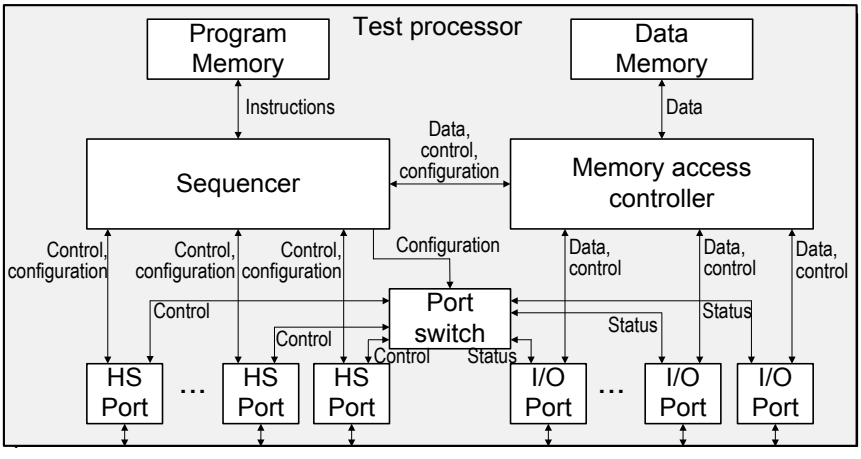


Figure 2.17 Simplified architecture of the test processor presented in [89]

Table 2.7 shows a summary of the test processors. The first column of the table is the processor name, the second column maps the processor to one of the categories, the third column is the application field, and the last column presents some of their main features.

Test processor	Category	Target	Features
Testchip [76]	Hardwired	On board device test	– Pattern generation and analysis based on weighted LFSRs – Communication with ATE via bus interface – Access to DUT's primary I/Os and registers (via scan-path) – Adaptation based on register and RAM
Test processor chip [77–80]	Hardwired	On board device test	– Same concept of Testchip – Different LFSR implementations o LFSR reseeding and generalized LFSRs
Test processor for SoC test [81]	Program-mable	SoC test	– 16-bit RISC processor described in VHDL o Deterministic execution (no pipeline and no cache) o 32 basic instructions + 2 special test instructions – 8 general purpose registers o 2 registers for LFSR/MISR using BILBO scheme – Configurable I/O controller for ATE/DUT communication
Test processor for SoC test [82–84]	Program-mable	SoC test	– 2 processor versions with fixed ISA based on [81] o 1st version with self-test features o 2nd version with on-line test features – 2 pairs of registers for LFSR/MISR
Test processor for SoC test [19, 85, 86]	Program-mable	SoC test	– 4 processor versions with fixed ISA o Basic [81], self-test [82–84], on-line test [82–84] o 4th version with multiplication and division – No serial port for scan-path test (separate scan-path module)
Test processor for asynchronous chip test [87]	Program-mable	Asynchro-nous chip test	– 16-bit RISC processor based on [81] and described in LISA o 4 pipeline stages with result forwarding (no cache) o 53 instructions o 8 registers (2 pairs of registers for LFSR/MISR) – 12 I/O ports – Configurable asynchronous handshake ports
Test processor for asynchronous chip test [88]	Program-mable	Asynchro-nous chip test	– 32-bit RISC processor based on [87] and described in LISA o 4 pipeline stages with result forwarding (no cache) o Instructions based on complex sequences o LFSR/MISR operations with all registers o Load/store for transfers between memory and I/Os – 32 I/O ports – Asynchronous handshake ports (not configurable)
Test processor for asynchronous chip test [89]	Program-mable	Asynchro-nous chip test	– Three main modules designed in VHDL o Port switch couples asynchronous and synchronous ports o Memory access controller coordinates memory I/O transfers o Sequencer manages data flow during test – Sequencer architecture o 32-bit data and 24-bit instruction coding o 2 pipeline stages (no cache) o 22 complex instructions to configure/couple ports o 8 registers reserved for special purposes o No LFSR/MISR support

Table 2.7: Test processors

3 Processor in FPGA-based testing

3.1 FPGA-based testing

In this section an introduction to FPGA-based testing is provided. For this purpose, building blocks, test phases, and approaches found in the literature are presented.

3.1.1 Building blocks

A test system comprises all hardware and software components used during the test process. In the case of FBT, the test system is composed of two main components: The external automatic test equipment (ATE) and the system embedded in the FPGA, which is known in this dissertation as FPGA-based test system (FBTS). Figure 3.1 shows a top view of the PCB and test system. In this case, each IC populating the PCB and directly connected to the FPGA is known as a device under test (DUT).

The ATE operates as the interface between the test engineer and the PCB. It is the platform used to develop the FBTS, configure the FPGA, visualize test results, and monitor, control and execute test functions. It is equipped with a test controller for the configuration of the FPGA and communication via the available test interface, and appropriate software tools for visualization, test development, processing, and communication purposes. These tools are commonly integrated in a framework that facilitates the design and development.

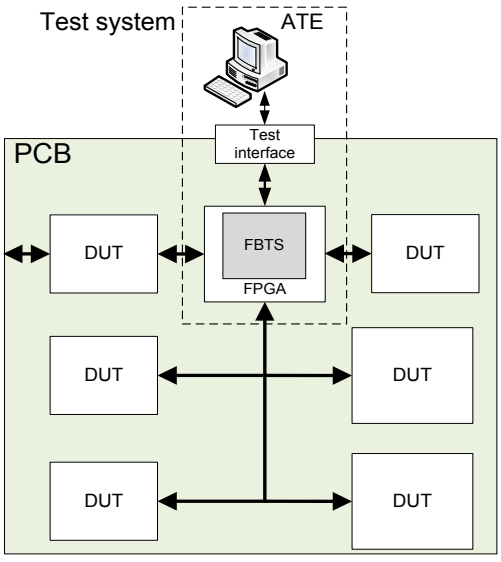


Figure 3.1: FBT building blocks

The FBTS corresponds to the part of the test system that is embedded in the PCB. It is responsible for accessing the ICs connected to the FPGA and executing test functions. After the test process is completed, the FPGA is reconfigured for its original operation or for the execution of functional tests.

3.1.2 Test scenario, application field, DUT, and faults

An important term used in this dissertation is test scenario. It represents the *PCB properties* and *set of test requirements* that are defined for a specific case. In FBT, the *PCB properties* correspond to the FPGA, ATE-FPGA communication infrastructure, access mechanisms to DUTs, working frequencies, timing and electrical characteristics of pins, number of interconnections, etc. The *set of test requirements* is defined by the test engineer and represents the selection of interconnections under test as well as the definition of test quality metrics such as fault coverage, test time, and diagnosis resolution.

In order to define the application field of FBT, it is important to remember that test strategies used at the board-level assume that devices populating the PCB work properly. The predominant manufacturing defects are caused by missing or misplaced devices, and problems at the soldering joints (Section 2.2.2). Defects caused by missing, misplaced or incorrectly assembled devices are targeted by inspection and classical structural test techniques. Therefore, the application field of FBT is the detection and diagnosis of defects located at the PCB interconnections and soldering joints.

In order to perform tests, FBT requires accessing and using the functionality provided by devices connected at the other side of the interconnections given that test resources are only located at the FPGA side. Although these devices are labeled as DUTs, it is important to clarify that they provide the required accessibility and functionality to perform the interconnection test, but tests do not target defects in their internal structure.

As already mentioned, FBT is used to improve the quality of a given test strategy, in cases where other test techniques are not able to fulfill the test requirements. For this purpose, FBT takes advantage of the direct access to DUTs and interconnections, the potential to perform tests at the FPGA clock frequency, and the low costs associated with FPGA reuse. As a consequence, FBT offers a great potential for the reduction of test time and the execution of tests at-speed.

3.1.3 Test phases

Figure 3.2 shows the three test phases required for FBT, which are known as pre-test, test, and post-test phases. The pre-test phase is carried out on the ATE before test execution and consists of two essential steps. The first step is the analysis of the PCB properties and test requirements and the design and evaluation of potential solutions. The second step is compilation of the software and hardware descriptions and FPGA configuration.

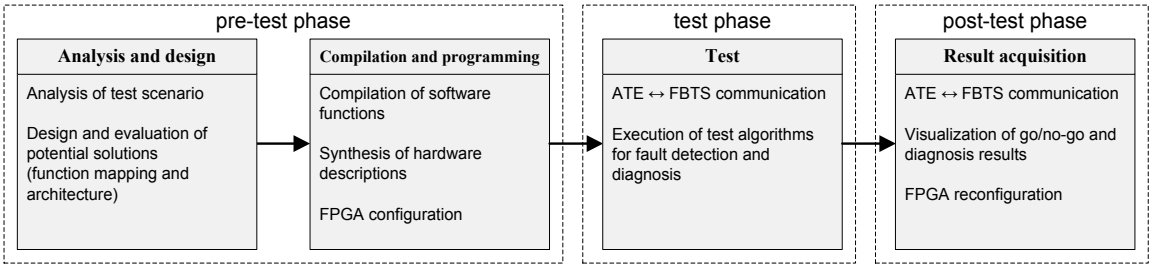


Figure 3.2: FBT test phases

The test phase comprises the execution of the test and communication between ATE and FBTS. The post-test phase comprises the acquisition and presentation of final test results. At the end of this phase the FPGA can be reconfigured for normal operation or for the execution of functional tests. In the following subsections, each of the test phases is discussed in greater detail.

3.1.3.1 Pre-test phase

The analysis of the test scenario is the first step in the development of the FBTS. An example of two test scenarios for a PCB equipped with an FPGA and memory devices is as follows.

- *Test scenario 1*: Detection of static faults at the interconnections between the FPGA and memories. Diagnosis resolution at device-level is sufficient, and test execution time does not represent a critical issue.
- *Test scenario 2*: Detection of static and dynamic faults at the interconnections between the FPGA and memories. It is necessary to diagnose faults at the interconnection-level and minimize the test execution time.

After the analysis of the test scenario, potential solutions are designed and evaluated. A potential solution for the first test scenario is the implementation of a short scan chain in the FPGA, leaving all test and DUT-related access functions on the ATE. On the other hand, the implementation of test and access functions in the FBTS is more appropriate for the second test scenario because it has more demanding test requirements.

The compilation and programming step is in charge of synthesizing the FBTS, compiling test functions for the ATE, and configuring the FPGA. The compilation and synthesis is done by means of electronic design automation (EDA) tools and it is carried out once for a specific test scenario. On the other hand, the FPGA configuration has to be carried out for each PCB under test.

3.1.3.2 Test phase

During the test phase two main operations are carried out: ATE/FBTS communication and test execution.

ATE/FBTS communication refers to the exchange of control commands, status information, test patterns, and test results through the test interface. Depending on the functions implemented in the FBTS, a different type and amount of data is transmitted.

During the test execution the PCB is exercised with test patterns and the acquired test responses are analyzed for the detection and localization of faults. For this purpose, pattern generation, pattern application, and pattern analysis tasks are carried out.

Pattern generation produces pattern sequences that target a specific fault coverage (number of faults and fault types) and diagnosis resolution. These patterns are generated before the test execution and stored in memory, or during test execution.

Pattern application applies test patterns to the DUT and acquires the corresponding test responses. In comparison to structural test techniques such as BScan, FBT requires the use of the DUT access functions for the application of test patterns and acquisition of test responses. This has large repercussion on the test process because it makes it necessary to consider the DUT functional and timing properties. If the access functions are not implemented properly, this can lead to low fault coverage or even the incapability to perform any test.

Pattern analysis evaluates obtained test responses. This evaluation is performed during the execution of the test algorithm for fault detection and diagnosis. It can be performed only if the DUT is able to deliver test responses.

Section 3.4.3 provides more information about the type of test patterns and test responses that are used with FPGA-based testing approaches.

3.1.3.3 Post-test phase

The last phase is the post-test phase. During this phase the ATE acquires the final results from the FBTS and processes this information in order to present final test results to the

test engineer. In this way, the test engineer obtains information about detected faults and their location.

After the processing and presentation of test results, a final (optional) step is the reconfiguration of the FPGA for normal operation or execution of functional tests.

3.1.3.4 Comparison to boundary scan and processor-based testing

The test phases of FBT and other test techniques such as BScan and PBT have essential differences. Most of them are found in the pre-test phase given that BScan and PBT do not require the design of any hardware.

Figure 3.3 shows the typical test phases in BScan. In this case, there is no embedded test system to design because the TAP controller and corresponding instruction and data registers already are part of the BScan-compliant devices. The pre-test phase is just used to define the BSDL models and describe the BScan shifting operations. On the other hand, the test and post-test phases are very similar to the FBT phases, with the difference that it is not necessary to configure an FPGA or program an embedded tester.

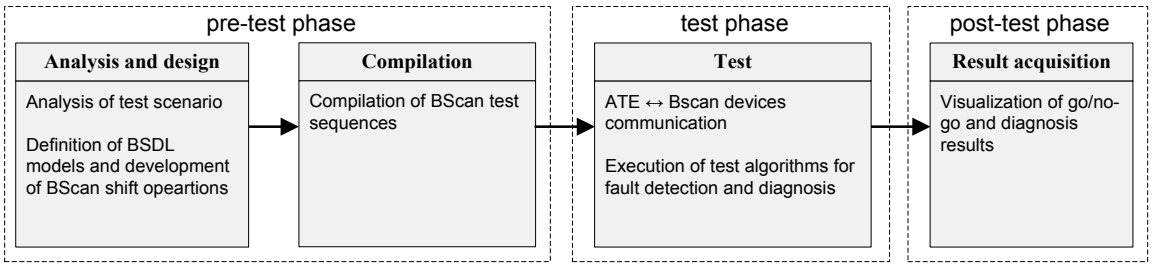


Figure 3.3: BScan test phases

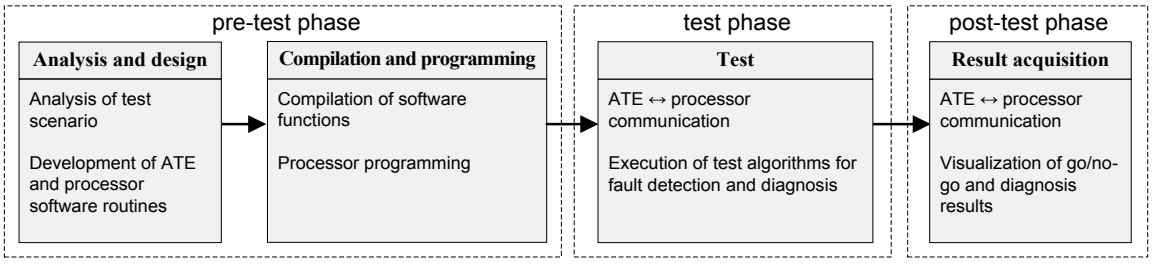


Figure 3.4: PBT test phases

Figure 3.4 shows the test phases of PBT. As can be seen in the figure, PBT has more aspects in common with FBT. The design step includes the development of software routines that correspond to the test or interpreter functions used in offline and online operation modes. The software routines are subsequently compiled and the corresponding machine code downloaded in the processor memory. The main difference with FBT is the design of hardware modules. This is not necessary in PBT because the central processing unit (CPU), debug-interface, and DUT controllers are already available in the die.

3.1.4 Design automation in FPGA-based testing

One main feature (besides good test quality metrics) that makes a test technique successful is the support of an automatic development and test execution process. This is very important for a test engineer because it reduces costs, accelerates the test process, and minimizes errors caused by human intervention. Additionally, it makes it unnecessary to have test engineers with high expertise and knowledge of the implementation details of the test technique.

The BScan pre-test phase is considered to have low complexity, and therefore it is relatively easy to automate. For example, software test tools such as CASCON from Göpel Electronic offer the option to automatically generate and execute BScan infrastructure and interconnection tests [90]. On the contrary, the diversity of processor architectures and debug interfaces makes the automation of PBT a very difficult task [46].

The automation of the FBT pre-test phase is even more challenging in comparison to the other two test techniques. The reason for this is that it is necessary to design and implement the complete FBTS infrastructure, which requires the definition of hardware modules as well as the access to the FPGA pins and test interface. In Section 3.2 the FBT approaches found in the literature are analyzed. They differentiate themselves in the architecture properties of the FBTS and in the way the pre-test phase automation challenges are solved.

3.2 FPGA-based testing in the literature

The term *FPGA test instrument* is commonly found in the literature for FPGA-based systems that perform test and measurement (T&M) tasks. In [30] different ways to classify FPGA test instruments are proposed: traditional/virtual/synthetic, post-manufacturing/in-field, for test/measurement/debug/configuration, and with a focus on application speed/session time.

In this dissertation, the field of FPGA test instrumentation is restricted to instruments used during the PCB manufacturing and with focus in the application speed and session time (test time). Figure 3.5 shows the classification proposed for this restricted case.

The classification groups FPGA test instruments into two classes: external FPGA test instruments and embedded FPGA test instruments. External FPGA test instruments are located on the ATE, and they are used for the generation of signal stimuli and acquisition of test responses. They are further divided into single purpose and synthetic instruments.

Single purpose instruments are low cost solutions in which the FPGA is configured once for the implementation of a specific function, such as a pattern generator, logic analyzer, bus monitor, or as a device tester [91–93]. On the other hand, synthetic instruments represent instruments that are more flexible. In this case, the FPGA test instrument has a standardized interface to the PCB, making it possible to reconfigure the FPGA with different T&M functions defined by the test engineer [94, 95].

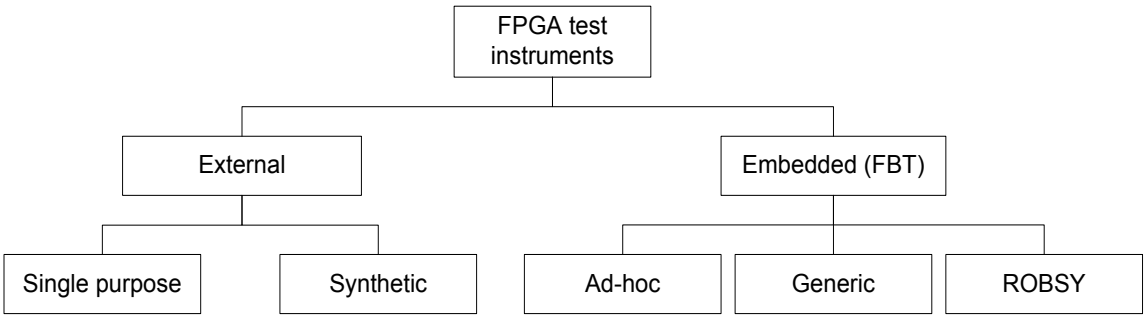


Figure 3.5: FPGA test instruments classification

On the other hand, embedded FPGA test instruments are located on the PCB. They are divided into three subcategories based on the design methodology. Ad-hoc and generic FPGA test instruments (Sections 3.2.1 and 3.2.2) group test instruments found in the literature, while ROBSY represents the FBT approach proposed in this dissertation (Section 3.3).

3.2.1 Ad-hoc FPGA test instruments

Ad-hoc FPGA test instruments are specialized FBTS manually developed for a specific test scenario. In industrial applications, they are frequently used where classical test techniques are not able to fulfill test requirements. They target the test of interconnections and main functions of peripheral devices such as memories and communication interfaces. They are developed based on a typical FPGA design process, in which the test engineer is in charge of the manual design of the instrument. This means that the synthesis step is mandatory and that the complete pre-test phase is responsibility of the test engineer. The test engineer should have deep knowledge of FPGA design and board-level testing.

In [96] two different ad-hoc test instruments are presented. The first one is known as *in-system programmable built-in assisted test* (ISP BIAT), which includes basic functions in the FPGA to alleviate the access to regions of the PCB. The FPGA is used as a signal pass-through device, making it possible to access PCB interconnections that otherwise could not be accessed, or, the FPGA is used to implement simple access functions to write to or read from memory devices in order to test their interconnections.

The second test instrument is known as ISP built-in self-test (ISP BIST). In this case, more complex functionality is embedded in the FPGA in order to perform test execution and diagnostic functions required during the test and post-test phases. An example of this approach is the development of a complete embedded memory tester that can apply patterns and acquire and evaluate test responses. As a consequence, the communication overhead between ATE and FPGA is very low.

The most important feature of ad-hoc FPGA test instruments is the flexibility for defining the FBTS architecture and for mapping test functions either on the ATE or FPGA. As a consequence, they provide the option to develop and customize a high performance FBTS that is able to fulfill strict test requirements.

However, the use of ad-hoc FPGA test instruments has several drawbacks:

- Manual design or adaptation for each new test scenario is required.
- EDA tools and qualified FPGA designers are a must.
- Debugging actions for detecting and correcting design errors are a must.
- Long design time and high test costs have to be considered.

3.2.2 Generic FPGA test instruments

The concept of generic FPGA test instruments was conceived in order to address the high design time and costs required for ad-hoc instrumentation. The idea is to use pre-developed instruments that support configuration options for their adaptation to the specific test scenario. The configuration options are set either before synthesis or after the instrument is configured in the FPGA.

In [30, 54, 55, 57, 58] this approach is known as FPGA embedded virtual instrumentation. It uses pre-developed instruments with a fixed mapping of test functions in the FBTS and ATE. The FPGA is used as an access mechanism to the DUT, while the ATE performs all the test and diagnostic operations required during test and post-test phases. This means that DUT and test algorithm related operations are not implemented in the FPGA, but as software routines in the ATE.

FPGA embedded virtual instruments are adapted to the properties of the DUTs and PCB by setting configuration options after the FPGA is configured. This means that design and synthesis steps are not required for every new test scenario, minimizing design time, design errors, and costs. The FPGA embedded virtual instruments found in the literature are employed for clock frequency measurement, memory interconnection test, and in-

system programming. They are implemented based on variable length shift registers (VLSR), accumulating buffers, counters, and primitive pattern comparison functions.

The implementation of low-level access functions (DUT native protocols) and test algorithms in the FPGA is necessary in order to fulfill strict test and timing requirements. The implementation of these functions as part of the FPGA embedded virtual instruments increases their complexity because it is necessary to support mechanisms that adapt the instrument functional and timing behavior on the fly (after the instrument is configured in the FPGA). This aspect represents a huge challenge for FPGA embedded virtual test instrumentation and is not discussed in any of the examples found in the literature.

Another type of generic test instrument is the FPGA-based universal embedded digital test instrument presented in [97]. In this case, the FPGA is used not only as an access mechanism, but also as a platform for the implementation of low-level DUT native protocols, pattern generation, and result analysis functions. The DUT access, pattern application and low-level analysis functions are implemented in a generic way independently of the test scenario. However, blocks of code representing the DUT native protocols and pattern generators have to be manually rewritten, and the parameters that define number of pins and size of internal memories have to be properly set depending on the DUT. This means that the adaptation to a specific test scenario is carried out during the pre-test phase and before the synthesis process takes place.

The main advantage of generic FPGA test instruments is the simplification of the tasks carried out during the pre-test phase. The design and compilation steps are completely avoided for FPGA embedded virtual instruments and the design effort for FPGA-based universal embedded digital test instruments is reduced. In this way, the design time and costs are significantly reduced in comparison to ad-hoc test instruments.

However, the advantages obtained with generic FPGA test instruments come at a price:

- They are not a solution optimized for a particular test scenario, which makes it impossible to guarantee the execution of tests at-speed.
- The execution of test and diagnosis operations on the ATE makes it necessary to exchange a considerable amount of data through the typically low-speed test interface, which leads to long test times.

3.2.3 Summary of embedded FPGA test instruments

Table 3.1 summarizes the principal properties of the two embedded FPGA test instruments found in the literature. For comparison purposes, the BScan test technique is included in the table.

	Classical Boundary Scan	Ad-hoc FPGA test instruments	Generic FPGA test instruments
ATE/FPGA function mapping	Fixed	Flexible	Fixed
ATE/FPGA data exchange	Very high	Low	High
Function reuse	Yes	No	Yes
Synthesis step	No	Yes	Yes/No
FPGA designers	No	Yes	Yes/No
Configuration step	No	Yes	Yes
Optimized for test scenario	No	Yes	No
Fault coverage	Static	Dynamic	Dynamic in some cases
Design time	Short	Long	Short
Test time	Very long	Short	Long
Test costs	Low	High	Intermediate/low

Table 3.1: Properties of boundary scan, ad-hoc and generic FPGA test instruments

ATE/FPGA function mapping gives an idea of the flexibility to implement test functions on the ATE or FPGA. ATE/FPGA data exchange refers to the amount of data that is transmitted between the ATE and FPGA. Function reuse shows the possibility to use the same embedded test functions for different test scenarios.

The rows synthesis step and FPGA designers indicate if it is necessary to synthesize the test instruments and employ FPGA designers during the pre-test phase. The configuration step indicates if it is necessary to configure the FPGA. The remaining properties indicate if the test instrument is optimized for the specific test scenario, and they give an idea of the test quality metrics achievable and relative test costs.

3.3 ROBSY approach

The main idea of the Reconfigurable On-Board test SYstem (ROBSY) is to group the main advantages of ad-hoc and generic FPGA test instruments into a single solution. For this purpose, it is necessary to develop a tailored and specialized FBTS keeping the design effort as low as possible. In order to implement such a test system, the ROBSY approach focuses on three main aspects:

- Flexible mapping of test functionality either at the FPGA or ATE side.
- Customization of FBTS.

- Automatic design, evaluation, and implementation mechanism.

The flexibility improves the adaptability of the test system to different test scenarios and allows the execution of tests at-speed and reduction of the amount of data exchanged between the ATE and FBTS. The latter is very important for the reduction of the test time because test interfaces such as JTAG are typically the bottleneck of the test process [56].

The customization provides the option to tailor the FBTS for the specific test scenario, fine-tuning the resource usage and execution speed. In this way, it is possible to fulfill strict test requirements, and improve the test coverage in comparison to generic FPGA test instruments and BScan.

The support for automation hides the design and implementation details from the test engineer, reducing the design complexity, design time, design errors, and test costs. This makes it unnecessary to have test engineers with FPGA design skills during the FBTS development process, but EDA tools and the corresponding software licenses are still required. This means that the development costs remain higher compared to pre-developed generic FPGA test instruments.

Table 3.2 shows a comparison of the ROBSY approach to ad-hoc and generic FPGA test instruments.

	ROBSY	Ad-hoc FPGA test instruments	Generic FPGA test instruments
ATE/FPGA function mapping	Flexible	Flexible	Fixed
ATE/FPGA data exchange	Low	Low	High
Function reuse	Yes	No	Yes
Synthesis step	Yes	Yes	Yes/No
FPGA designers	No	Yes	Yes/No
Configuration step	Yes	Yes	Yes
Optimized for test scenario	Yes	Yes	No
Fault coverage	Dynamic	Dynamic	Dynamic in some cases
Design time	Short	Long	Short
Test time	Short	Short	Long
Test costs	Intermediate	High	Intermediate/Low

Table 3.2: Properties of the ROBSY approach

3.3.1 Modeling the test functionality

In order to automate the design step, enable mapping of test functions in a flexible way, and support design reuse, it is necessary to rely on a high-level model that abstracts away low-level details of the test system, enabling test engineers without FPGA expertise to

take advantage of this approach. This model-based design mechanism is the main reason to classify ROBSY as a separate class of embedded FPGA test instruments.

3.3.1.1 Layer concept

The core of the model is a layer concept that splits up the complexity of test algorithms and DUT native bus protocols in small and independent functions and interfaces [2–4, 56]. Figure 3.6 illustrates the organization of the layers and interfaces.

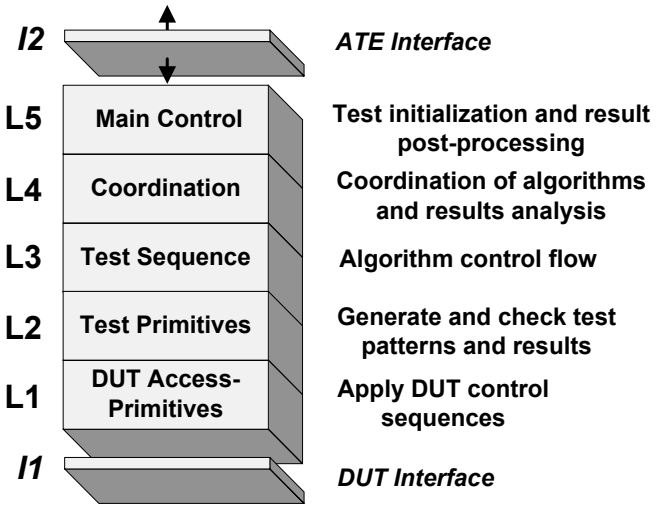


Figure 3.6: Organization of the layers and interfaces

Interface 2 (I2) represents the interface between the PCB and the ATE, and interface 1 (I1) represents the interface between the test system and DUT. I1 is a fixed interface, while I2 can be placed between any two layers. For example, the interface configuration presented in Figure 3.6 corresponds to all layers implemented in the FPGA, with the ATE merely used for the configuration of the FPGA. On the other hand, I2 right above I1 represents the case in which all layers are implemented in the ATE and the FBTS is only used as an access mechanism. This configuration can also represent BScan and some configurations of BIAT and embedded FPGA virtual instrumentation.

The test algorithms and access functions for a specific DUT are described within the five layers. Each layer uses the functions supplied by the lower layers and provides its functionality to the upper layers. The complexity of the layers ranges from low-level functions for accessing the DUT (DUT native bus protocols) to high-level functions for control and coordination of test algorithms.

The *DUT access-primitives layer* (L1) describes low-level functions responsible for access to the DUT within specification. For this purpose, it is necessary to consider the DUT native protocols, electrical properties of the DUT I/Os, and the timing properties of

the DUT buses. This layer describes the control sequences required to write data to, or read data from, a DUT.

Layers L2 and L3 describe functions related to the test algorithms. The *test primitives layer* (L2) is in charge of the pattern generation and the low-level analysis of test results. The latter comprises the comparison of test patterns with test responses and the generation of signatures. The exact operations depend exclusively on the test algorithms, and they are represented by primitives such as shift, count, rotate, and compare.

The *test sequence layer* (L3) is in charge of the test algorithm control flow. It invokes L2 primitives, and it makes decisions on the next step to execute depending on the results delivered by layer L2. Additionally, it provides information to the upper layers about the test status and results of the test algorithms.

Layers L4 and L5 are used for high-level tasks such as managing test algorithms and visualizing test results. The *coordination layer* (L4) is in charge of coordinating the execution of all test algorithms used for a given DUT and analyzing results delivered by L3. For example, this layer allows the analysis of diagnosis information for the location of the faulty interconnection or group of interconnections. Finally, the *main control layer* (L5) is in charge of the initialization of the test process and post-processing of results. The latter is required for the proper visualization of test results.

3.3.1.2 DUT-model

The test engineer describes the test algorithms and DUT properties in a high-level model, which encapsulates the test functions based on the layer concept. The model is known as DUT-model (DUT-M). It is used to hide implementation details of the FPGA-based design, enable reuse of test functions for different test scenarios, and serve as input to the automatic generation process of the test system.

The DUT-M facilitates the use of the ROBSY approach by test engineers with no deep knowledge of FPGA design, therefore reducing design time and costs in comparison to ad-hoc test instruments. It is specified using the ROBSY test description language (RTDL), which is a programming language developed as part of the ROBSY approach and with syntax and semantics very similar to ADA. It is tailored for the ROBSY approach, making it possible to generate hardware and software descriptions from the DUT-M. Additionally, it can be used as an executable model for verification.

The DUT-M is divided into three main sections:

- DUT interface.

- Pins properties.
- Layer procedures.

The DUT interface describes main properties of the DUT data, address, and control buses (I1 of the layer concept). It provides information about the name and direction (in, out, bidirectional) of pins belonging to each bus, as well as the default and active values. The pins properties describe the electrical properties of the pins (pullup, pulldown, etc.) and their location in the netlist of the PCB.

The layer procedures describe the functionality of layers L1 to L5. L1 describes the timing relations of the DUT data, address, and control buses that are required to implement the DUT access functions. For this purpose, Difference-Bound-Matrixes and action sequences are used to specify the timing behavior of the DUT. More information about this subject is found in [98, 99].

The L2-L5 procedures are described by means of standard high-level constructs. The DUT-M supports local and global variables, whose width is defined explicitly during their declaration. Additionally, the declaration of vectors (two dimensional arrays) is supported by global variables.

3.3.2 FBTS architecture

The FBTS architecture is the central part of the ROBSY approach. It allows the implementation of tests functions in the FPGA and the execution of tests at-speed. It is based on a processor co-processor structure and is equipped with interfaces to the ATE and DUTs [3, 4]. In the following subsections, the structure of the FBTS and the relationship between the layer concept and the FBTS architecture is presented.

3.3.2.1 FBTS domains

In order to define a proper structure for the FBTS, it is necessary to answer the following question:

- What can be tested at the same time using the ROBSY approach?

With the goal of providing an answer to this question, Figure 3.7 shows a PCB with an FPGA as the central computing element. This PCB configuration corresponds to an ideal test scenario for FBT because the FPGA has access to all devices on the PCB based on four separate buses.

Given that PCB buses are completely independent from each other, it is possible to test the interconnections of each bus at the same time. However, the interconnections of DUTs belonging to the same bus cannot be tested at the same time. The reason for this is the existence of shared interconnections that only allow accessing a single DUT at a time. Therefore, the test of separate PCB buses can be carried out in parallel, but the test of the interconnections to DUTs belonging to the same bus has to be carried out sequentially.

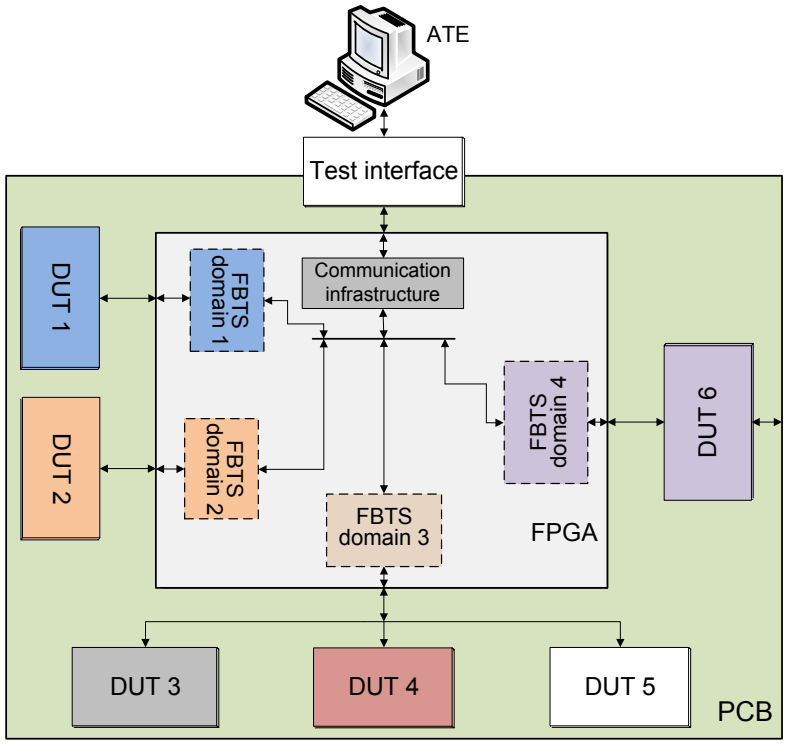


Figure 3.7: FBTS domains

In order to support this parallelism, the FBTS is divided into independent sub-systems, each of them responsible for a single PCB bus. These sub-systems are known as FBTS domains, and they work independent from each other. In Figure 3.7, four FBTS domains are shown.

In addition to the FBTS domains, it is necessary to include a communication infrastructure for exchanging information between the ATE and FBTS. The information exchange is carried out for each domain and through the physical test interface available on the PCB. This does not affect the independent operation of the FBTS domains, but requires a communication infrastructure for handling the coordination of all domains.

3.3.2.2 Main components and processor co-processor interface

In the ROBSY approach, the two main components of each FBTS domain are the processor and co-processor. The processor is a pre-defined programmable component

with an instruction set that can execute embedded software routines. The co-processor is a hardwired component without an instruction set, which is used to accelerate test execution and guarantee the execution of tests at-speed. In this way, each layer implemented in the FBTS can be mapped either to the processor or to the co-processor.

Taking these two components into consideration, a third interface is included in the layer concept. This interface represents the hardware/software partition of layer functions implemented in the FBTS. In Figure 3.8 this interface is labeled as the processor/co-processor interface (I3), and in the same way as I2, it can be placed between any two layers. The only restriction necessary to maintain the logical organization of the layer concept is that I3 must be always located above I1 and below I2. For example, the configuration presented in Figure 3.8 shows that layers L1 and L2 are mapped to the co-processor and implemented in hardware (HW), layers L3 to L5 mapped to the processor and implemented in embedded software (ESW). In this configuration, there are no layers mapped to the ATE and implemented in software (SW). The ATE is used to configure the FPGA only.

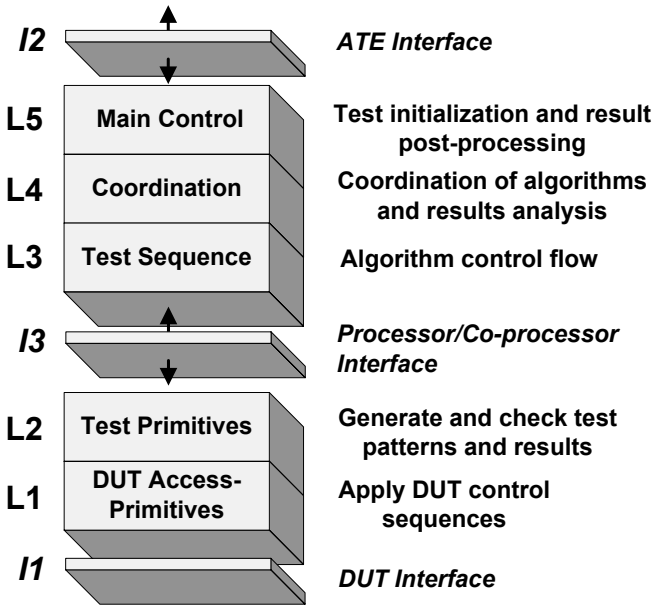


Figure 3.8: Layer concept with processor co-processor interface

There is no need for a processor in the FBTS if interfaces I2 and I3 are located between the two same layers. In the same way, there is no need for a co-processor in the FBTS if I3 is located in the same position as I1. Section 3.4 presents a discussion of the reasons why interfaces I2 and I3 or interfaces I1 and I3 should not be located in the same place.

3.3.2.3 Architecture example

The structure in Figure 3.9 is proposed in order to support independent FBTS domains and a processor/co-processor pair for implementing layers belonging to a DUT.

This structure consists of a single processor per domain and a co-processor per DUT. The use of a single processor per FBTS domain is sufficient for performing tests of all interconnections because only one processor/co-processor pair will be active at the same time (Section 3.3.2.1). The example in Figure 3.9 corresponds to domain 3 in Figure 3.7. In this case, a single processor and three co-processors are used for testing the interconnections to DUT 3, DUT 4, and DUT 5 sequentially.

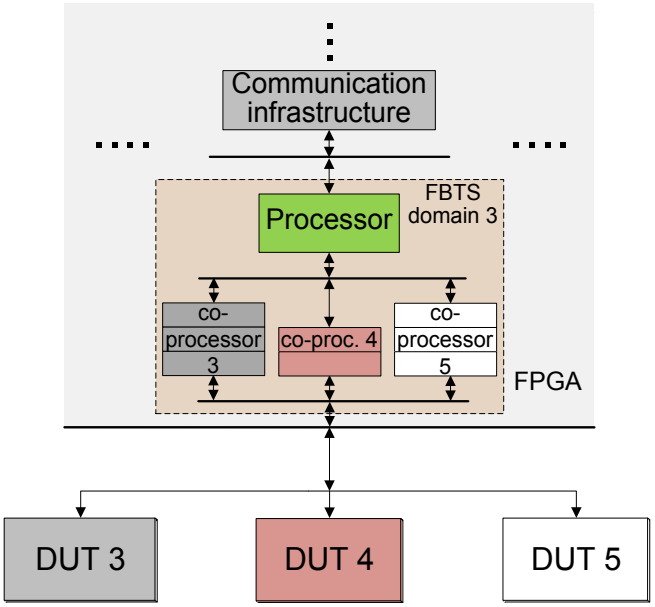


Figure 3.9: Architecture of an FBTS domain

Figure 3.10 shows the complete PCB and test system. In this case, the FBTS comprises three FBTS domains with three processors, each of them connected to one or more co-processors. Embedded software routines (for the processor) and hardware descriptions (for co-processors) are generated based on the corresponding DUT-M. Here, the size of the co-processor indicates a different location of I3. DUT 1 and DUT 2 are part of a single FBTS domain, even though they are connected to two separate PCB buses. If resources are limited, it is possible to combine more than one domain to minimize the number of processors that are part of the FBTS. The trade-off is the increase in the test time because two separate domains would be tested sequentially.

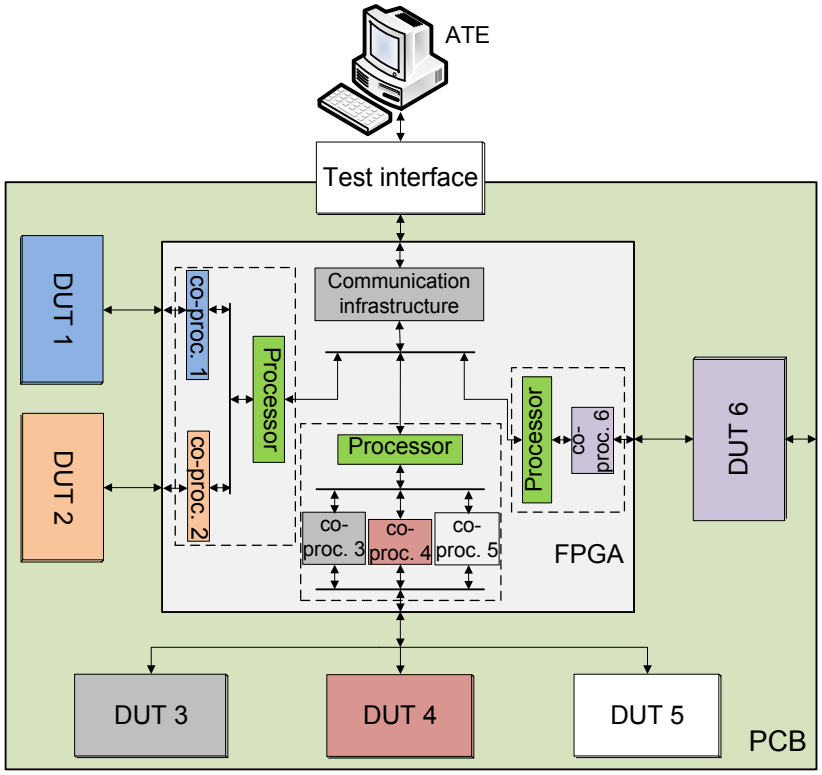


Figure 3.10: Complete FBTS architecture

3.4 Processor in the FPGA-based test system

Relying on a processor and ESW to carry out part of the test functions has a direct impact on the automation process, test quality, resource utilization, and costs. Whether this impact is beneficial for the ROBSY approach or not, and how beneficial it can be, are the main research questions that this dissertation answers. In order to start providing an answer to these questions, it is necessary to analyze the effects of including a processor as part of the FBTS.

3.4.1 Processor impact

The use of a processor in any digital system is justified by the flexibility that it provides for the implementation or update of functions without making any changes to the hardware of the system. This improves the level of reusability of the system for different applications and reduces the design and implementation costs [100].

In FPGA-based design, including a processor offers the possibility to change system functionality without the need to change the hardware descriptions of the system and reconfiguring the FPGA. However, these arguments are not strong enough to justify the inclusion of a processor in the ROBSY FBTS given that the flexibility provided by

FPGAs and the automatic generation capabilities required in ROBSY can substitute the flexibility provided by a processor.

There are five additional arguments that speak in favor of having a processor as part of the ROBSY FBTS:

- A processor represents an efficient mechanism to implement test functions that do not need to be executed at high speeds.
- A processor reduces the amount of data exchanged between ATE and FPGA.
- A processor provides a standard communication mechanism for controllability and observability of test execution.
- A processor is a pre-verified component that does not need any further verification after the FBTS is automatically generated.
- Debugging of ESW requires less effort than debugging of HW.

The processor represents an efficient mechanism to implement test functions because it uses the same resources to perform arithmetic, logic, and control flow operations. In this case, the same arithmetic logic unit (ALU) and registers are used to carry out the operations that describe different functions and variables described in the DUT-M. This contrasts with the number of resources required to implement the same operations in HW, which depends on the binding and allocation strategies used by the hardware compiler. As a consequence, the processor guarantees that the test functions implemented in ESW use a relatively constant and predictable number of resources independently of the amount and complexity of the test functions.

By having the possibility to implement test functions in the FBTS efficiently, it is possible to move more test functions from the ATE side to the FPGA side. This reduces the amount of data that has to be transferred through the test interface because it is not necessary anymore to transfer complete patterns and test responses, but just short commands to control the test execution and receive information about the test status. As a consequence, the test time is reduced given that the test interface is typically the bottleneck of embedded test approaches.

Independent of the test functionality implemented in the FBTS, the processor provides the same standard communication mechanism from the point of view of the ATE. This mechanism is also independent of the FPGA and relies on the processor debug-interface. The debug-interface offers all the advantages of PBT to the ROBSY approach, making it possible to stop the processor, write to or read from memory and I/Os, set breakpoints, execute single steps, and trace code.

The processor is a component developed before the automatic generation process of the FBTS takes place. This means that the verification of its functionality is carried out by the processor designers during the development of the ROBSY approach and not in the field by test engineers.

Finally, debugging of ESW requires less effort than debugging of HW. The execution of ESW can be performed in a stepwise manner, which allows the observation of the processor status after the execution of each instruction. This is performed based on the processor debug-interface, which is used to control the execution flow. In the case of hardware debugging, multiple modules running in parallel have to be observed and control at the same time. This requires complex debugging mechanisms such as logic analyzers and the definition of hardware events. In addition, it requires a deep knowledge of the structure of the FBTS, which goes against one of the goals of ROBSY: hiding implementation details from the test engineer.

But there are arguments that also speak against the use of a processor:

- Successful built-in self-test (BIST) [17] approaches do not use processors for the implementation of embedded test functions.
- The execution of test functions by a processor takes a longer time than the execution of test functions by a hardwired unit.
- A processor does not guarantee the execution of tests at-speed.
- Processor instructions that are not used during test execution represent a non-optimal use of resources.

The arguments against the inclusion of a processor make it questionable if a processor is really necessary for the execution of embedded test functions in the ROBSY approach. They leave as the only alternative the implementation of the embedded test functionality by means of special hardwired units as is the case of BIST. However, the implementation of embedded test functions by means of special hardwired units is not necessary in all cases, and there are mechanisms that can be used to reduce the negative effect that the processor could have on test time and non-optimal use of resources.

The implementation of the higher layers does not have a significant influence on the fulfillment of critical test requirements such as dynamic fault coverage and test time. Therefore, the implementation of these layers in HW does not provide any advantage in terms of test execution time, resource utilization, and test coverage.

An alternative would be to implement the layers that have a minor impact in the fulfillment of test requirements on the ATE side. However, this would cause an increase

in the communication overhead between the FBTS and the ATE, which translates to longer test times. Therefore, the processor presents a very good alternative to implement non critical layers in the FBTS and reduce the communication overhead between the FBTS and ATE.

Additionally, BIST techniques are used for testing the ICs in which the BIST circuitry is implemented. This means that there is no guarantee that the BIST circuitry is free of faults, and therefore it is necessary to use basic and small circuitry that can be self-tested. In the case of ROBSY, the FPGA is considered to be already tested and fault free, which makes it possible to include complex components as processors that do not have to be self-tested.

There is no doubt that the execution of test functions in ESW takes longer than the execution of test functions in specialized hardware units such as co-processors. In the processor, test functions are translated to instructions executed in sequential order, in which each instruction requires multiple clock cycles for fetching, decoding, and executing. However, the specialization of the processor for testing purposes is a mechanism that can be used to reduce the time spent in the execution of ESW.

Concerning the execution of tests at-speed, it is necessary to include co-processors for this purpose because it is the only possible way to guarantee the fulfillment of strict timing requirements of the DUTs. The non-optimal use of resources due to processor functions not used during test execution can be avoided by implementing mechanisms in the processor that allow its adaptation to the specific test scenario.

Certainly, it is necessary to execute experiments in order to evaluate the arguments presented in this section. For this purpose, Chapter 7 presents different experiments carried out with the ROBSY test system. Sections 3.4.2 to 3.4.6 present an analysis of the layers that are more suitable for the FBTS and the implementation in ESW, as well as the possible location of the interfaces I1, I2, and I3.

3.4.2 Analysis of L1

In the ROBSY approach test resources are located at one side of the interconnections (FPGA side), which means, that it is necessary to make use of the functionality and access mechanisms provided by the DUT to carry out the test. This is the same case of cluster testing in BScan [16], in which non-BScan compliant DUTs are addressed during the test execution at one side of the interconnections.

In the ROBSY approach, it is necessary to include a DUT controller as part of the FBTS, which might be implemented either as ESW running on the processor or as a hardwired unit that is part of the co-processor. In order to analyze both implementation options, this section is divided into two parts. The first part presents DUTs typically connected to an FPGA, and the second part analyzes the implementation of L1 in ESW or HW.

3.4.2.1 Devices under test

Table 3.3 presents DUTs typically connected to an FPGA. The DUTs presented in the table range in complexity from basic devices such as push buttons (Button PTS645) and single digit displays (HDSM-281x) to complex devices such as Ethernet transceivers (PHY 88E1111) or video decoders (ADV7180). They are classified into three main classes based on the theoretical capability of BScan to test their interconnections:

- At-speed (BScan tests at the maximum DUT operation frequency).
- Yes (BScan tests below the maximum DUT operation frequency).
- No (BScan tests outside the DUT specification).

The DUT classification is based on a TCK frequency of 20 MHz and a BScan chain with 394 cells. For these values, the test application speed of BScan using Equations (2.2) and (2.3) is 49 kHz[1] and the maximum clock frequency achieved with BScan, which is the half of the test application speed, is 24.5 kHz. Both values are used in order to map DUTs into one of the groups. For synchronous DUTs, maximum clock frequency achieved with BScan is used.

The first column in Table 3.3 is the name of the DUT. The second column shows the number and type of interfaces of each DUT. A device interface represents a group of pins used to exchange data with the FPGA. DI1 and DI2 indicate the presence of different interfaces (different groups of pins) in the same DUT, while the postfix is used to differentiate between working modes or protocols supported by the same interface. For instance, the audio codec WM8731 has two interfaces, DI1 and DI2. DI1 is used for initialization and configuration purposes and it supports either the SPI or I2C protocol. On the other hand, DI2 is used to exchange audio information between the FPGA and DUT, and it supports I2S as the single communication protocol.

The third column represents the device class, which groups the DUT in one of the classes. Simple DUTs working at low frequencies such as pushbuttons and LEDs can be tested with BScan at their maximum operation frequency, while devices with higher operation frequencies cannot be tested at their maximum operation frequency or even within their

[1] $T_{acc} = \frac{394+7+6}{20 \cdot 10^6 Hz} = 20.35\mu s \rightarrow F_{app} = \frac{1}{T_{acc}} \approx 49kHz \rightarrow F_{app\_clk} = \frac{F_{app}}{2} \approx 24.5kHz$

specification. The fourth column shows the direction of the data flow from the point of view of the FPGA. The keywords *In* and *Out* represent unidirectional ports and the *I/O* keyword represents unidirectional ports with data flowing in both directions. Bidirectional ports are represented by the keyword *Bidir*.

The interface type and protocol column shows if the corresponding interface is a parallel or serial interface and the protocol supported. The common bus column shows if there are common interconnections for transmission of control, address, and data values. In this case, a typical common bus is a serial interface.

The clock source column shows if the corresponding DUT works in a synchronous or asynchronous way. A hyphen in this column represents an asynchronous DUT without an explicit clock source (e.g. SRAM IS61WV102416) or that the synchronization information is implicit in the data stream (e.g. Image sensor Viimagic 9221 in asynchronous working mode). For synchronous DUTs, the *In* and *Out* keywords provide information about the clock direction from the point of view of the FPGA. *In* indicates that the clock source comes from the DUT, while *Out* indicates that it comes from FPGA.

3.4.2.2 Analysis

L1 is a critical layer for the execution of tests at-speed because it defines the access rate to the DUTs, and consequently the coverage in terms of dynamic faults. The analysis of Table 3.3 leads to the following conclusions:

- The implementation of the DUT controller in ESW might be sufficient to perform tests at-speed for DUTs that are part of the at-speed BScan test class. In this case, the access operations are very simple, maximum operation frequencies are located in the kHz range, and there is no need for special hardware to meet critical timing requirements. A soft-core processor working in the MHz range (typical for FPGAs) has enough processing power to emulate the DUT bus protocol, react to incoming signals (e.g. PS/2 interface clock), and execute functions of the upper layers. However, these DUTs does not represent the main application field of FBT, given that they can be fully tested using BScan.
- The implementation of the DUT controller in ESW does not guarantee the execution of tests at-speed for DUTs that are part of the other two classes. It might not be even possible to implement access functions in ESW for some of these DUTs because their operation frequencies are located in the MHz range (image sensor and DRAMs) or are restricted to specific values (video decoder and Ethernet PHY).

Device under test	Device interface	BScan test	Operation frequency(MHz)	Ports direction	Interface type	Interface protocol	Common bus	Clock source
Button PTS645 [101]	DI1	At-speed	-	In	Parallel	-	No	-
1-digit display HDSM-281x [102]	DI1	At-speed	-	Out	Parallel	-	No	-
Mouse/keyboard interface [103]	DI1	At-speed	0.010-0.016	In	Serial	PS/2	Yes	In
I/O expander TCA9535 [104]	DI1	Yes	0-0.4	Bidir	Serial	I2C/SMB +Int.	Yes	Out
Temperature sensor Max6682 [105]	DI1	Yes	0-5	In	Serial	SPI	No	Out
Gyroscope ADIS16266 [106]	DI1	Yes	0.01-2.50	I/O	Serial	SPI +Int.	Yes	Out
LCD module CFAH1602B [107]	DI1	Yes	0-2	Bidir	Parallel	Register based	Yes	-
SDcard interface v2 [108]	DI1.sd	Yes	0-50	Bidir	Serial/Parallel	SD mode	Yes	Out
	DI1.spi	Yes	0-50	I/O	Serial	SPI	Yes	Out
SRAM IS61WV102416 [109]	DI1	Yes	0-100	Bidir	Parallel	SRAM	No	-
SSRAM IS61NVP102418 [110]	DI1	Yes	0-250	Bidir	Parallel	SSRAM	No	Out
EEPROM 24LC32 [111]	DI1	Yes	0-0.4	Bidir	Serial	I2C	Yes	Out
FLASH S29GL032N [112]	DI1	Yes	0-11	Bidir	Parallel	NOR FLASH	Yes	-
FLASH NAND01G [113]	DI1	Yes	0-66	Bidir	Parallel	NAND FLASH	Yes	-
Audio codec WM8731 [114]	DI1.spi	Yes	0-12.5	Out	Serial	SPI	Yes	Out
	DI1.i2c	Yes	0-0.52	Bidir	Serial	I2C	Yes	Out
	DI2.audio	No	12-20	I/O	Serial	Audio (I2S)	No	In/Out
Video decoder ADV7180 [115]	DI1.i2c	Yes	0-0.4	Bidir	Serial	I2C+Int.	Yes	Out
	DI2.pixel	No	27	In	Parallel	Pixel stream	No	In
Image sensor Viimagic 9221 [116]	DI1.spi	Yes	0-25	I/O	Serial	SPI	Yes	Out
	DI2.pixel	No	450	I/O	Serial	LVDS	No	In/-
Ethernet PHY 88E1111 [117]	DI1.mdio	Yes	0-8.3	Bidir	Serial	MDIO +Int.	Yes	Out
	DI2.GMII	No	2.5,25,125	I/O	Parallel	GMII	No	In/Out
	DI2.SGMII	No	625	I/O	Serial	SGMII(LVDS)	Optional	In/Out
Triple video DAC ADV7123 [118]	DI1	No	0.5-240	Out	Parallel	Data stream	No	Out
SDR-SDRAM 42S16320B [119]	DI1	No	5-166	Bidir	Parallel	SDR-SDRAM	Yes	Out
DDR-SDRAM P3R1GE3JGF [120]	DI1	No	125-400	Bidir	Parallel	DDR-SDRAM	Yes	Out

Table 3.3: DUTs directly connected to FPGA

Abbreviations: DI: *Device interface.* PS/2: *Personal system/2*. I2C: *Inter-integrated circuit*. SMB: *System management bus*. Int.: *Interrupt.* SPI: *Serial peripheral interface*. Bidir: *bidirectional.* SD: *Secure digital.* I/O: *in/out.* I2S: *Integrated interchip sound*. LVDS: *Low-voltage differential signaling*. MDIO: *Management data I/O*. GMII: *Gigabit media-independent interface*. SGMII: *Serial GMII*. SDR: *single data rate.* DDR: *double data rate.*

- It is necessary to include special hardwired blocks in order to support high-speed data streams (triple video DAC, SD card, image sensor) and proper hardwired interfaces for LVDS channels and DDR data transfers. These features cannot be supported using SW or ESW routines.

As can be seen, the implementation of L1 functions in ESW does not guarantee the execution of tests at-speed, and in some cases, it does not guarantee the execution of any test at all. The only proper way to ensure the execution of tests at-speed is by implementing the L1 functions in HW based on the co-processors of the FBTS.

3.4.3 Analysis of L2

The implementation of L2 in the FBTS corresponds to the inclusion of pattern generation and analysis functions in the FPGA. Patterns are generated by performing a set of arithmetic and logic operations during the test execution or are defined before the test is executed. Based on the classification proposed in [17], patterns used in the ROBSY approach are defined as follows:

- *Algorithmic test patterns*: They are developed to detect specific fault models and are generated using a sequence of arithmetic and logic operations. They have a short length that grows at a linear or algorithmic rate depending on the number of interconnections. Examples of algorithmic test sequences are walking one and zero, modified counting, true/complement, interleaved true/complement counting, and maximum aggressor fault model sequences [16, 22, 41].
- *Pseudo-random test patterns*: They are characterized by having properties similar to those of random patterns but are composed of repeatable sequences. They are generated using shifting and XOR operations and implemented using linear feedback shift registers (LFSRs).
- *Exhaustive/pseudo-exhaustive test patterns*: They produce every possible combination of values (exhaustive) or a sub-set of combinations (pseudo-exhaustive). These patterns can be generated by counters.
- *Deterministic test patterns*: They are defined before the test execution given that they cannot be easily obtained in an algorithmic way. They are developed to target a specific set of faults or considering the operating properties of the DUT. They have to be loaded into the FPGA and stored in some kind of memory.

Algorithmic test patterns are strongly used for interconnection testing, with walking one/zero and true/complement counting sequences being the most popular ones due to the fault coverage values that can be obtained with a small set of patterns [9, 25]. For the detection and diagnosis of dynamic faults, walking sequences, interleaved

true/complement sequences, and maximum aggressor fault model sequences are usually used [22, 23, 41, 121]. On the other hand, pseudo-random, exhaustive, and pseudo-exhaustive patterns are a good alternative when algorithmic test patterns are not sufficient to fulfill the test requirements. If a large amount of patterns is required to exercise the DUT, then the use of pseudo-random and pseudo-exhaustive patterns is a good alternative.

In the ROBSY approach, the use of these patterns depends on the properties of the specific DUT and the required fault coverage and diagnosis resolution. In the first case, they can be used only if the correct operation of the DUT or PCB is not compromised. A good example is the utilization of patterns that unintentionally misconfigure the DUT pins or change the behavior of the DUT. As seen in Table 3.3, DUTs that can be tested based on this type of patterns are mainly parallel memories (SRAM, SSRAM, SDRAM, EEPROM, FLASH), communication devices (data interface of PHY device), and basic devices (I/O expander, SDcard interface, and digital to analog converters). If the fault coverage and diagnosis resolution obtained is still not good enough, then deterministic patterns can be added to the pattern set.

On the other hand, if the DUT is very sensitive to pattern values, then deterministic patterns are the only alternative. This is necessary, for example, for testing interconnections of DUT interfaces used for configuration purposes, such as the interface DI1 of the gyroscope, audio codec, video decoder, image sensor, and PHY devices. In the same way, deterministic patterns are required for in-system programming of FLASH memories, and for testing the DI2 interconnections of the audio codec and image sensor.

In cases that the acquisition of test responses from the DUT is possible, pattern analysis functions are required. In the ROBSY approach, these functions are known as:

- *Equality comparison*: Test responses are compared to the expected values.
- *Signature analysis*: Test responses are compacted in a final signature, which is compared to the expected signature. This solution is commonly implemented using LFSRs and minimizes storage and computation requirements.

Equality comparison is the pattern analysis method most commonly used during the test execution. In this case, a test response is compared with an expected value, which is either generated based on pattern generation methods or stored in memory. If the obtained value is not the expected one, an error flag is set in order to stop the test execution or for further diagnostics purposes. On the other hand, the signature analysis is used to compact test responses into a single signature, which is compared with the expected one at the end of the test execution.

Mapping L2 and upper layers in the processor or co-processor of the FBTS does not affect the test coverage for dynamic faults if L1 is able to apply test patterns and acquire test responses at-speed. For this purpose, it is necessary to generate the test patterns and analyze test responses at the rate needed, or it is necessary to include mechanisms for the accumulation of test patterns and test responses between L1 and L2. It is very difficult to guarantee a specific rate for pattern generation and analysis if L2 functions are implemented in ESW. On the other hand, accumulation mechanisms between L1 and L2 represent a good alternative for the execution of tests at-speed regardless of the implementation mechanism of layers L2 to L5.

To summarize, implementing L2 functions in the FBTS involves typical pattern generation sequences, the storage of deterministic patterns, and comparison and compaction operations for the analysis of the test responses. If proper measures are taken, mapping these functions in the processor or co-processor does not affect fault coverage metrics. However, it will affect test time and resource utilization.

3.4.4 Analysis of L3

The implementation of L3 in the FBTS requires control flow operations such as branches and loops, which are used to describe the sequence of control steps presented in each test algorithm. Control flow operations can be implemented in HW by means of finite state machines and extra logic used to trigger state transitions. Alternatively, these operations can be realized in ESW, using control flow instructions such as unconditional and conditional branches.

If proper mechanisms between L1 and L2 are used to guarantee the execution of tests at speed, mapping L3 functions in ESW does not affect fault coverage metrics. In the same way as L2, it will affect test time and resource utilization.

3.4.5 Analysis of L4 and L5

Layers L4 and L5 are used for high level tasks related to a DUT. The functions involved are mainly the coordination of multiple test algorithms, diagnostics operations, test initialization, and post-processing of test results for visualization purposes.

L4 can be implemented in the FBTS using the same mechanism as L3 (finite state machines and control flow instructions). However, it is necessary to make use of arithmetic and logic operations to perform the look-up process in fault dictionaries, if the diagnosis involves the utilization of fault dictionaries.

The processor data memory can be used to store fault dictionaries, load and store operations to access the signatures in the fault dictionaries, and comparison instructions to analyze the signatures. In the case of co-processors, it is necessary to include memory blocks as part of the co-processor in order to support the utilization of fault dictionaries.

On the other hand, L5 has to be implemented on the ATE side. This is the only way to provide test engineers with basic control over the test execution and an appropriate visualization of test results (e.g. location of faults in PCB layout).

Layers L4 and L5 do not require any special test-related operations for their implementation. Control flow structures, load and store operations, and conventional arithmetic and logic operations are sufficient for their implementation. They have influence on test time and resource utilization, but they do not have any influence on the achievable test coverage.

3.4.6 Analysis of interfaces I1, I2 and I3

I1 represents the physical interface of the FPGA to the DUT. This interface is strongly coupled with L1, and cannot change its position. It comprises the pins and resources required to link the FBTS to the PCB interconnections, and it has to be implemented using FPGA I/O blocks.

Interfaces I2 and I3 can change their position. This has a significant impact on the test system because it defines how test functions are mapped in the FBTS and ATE. As a consequence, the position might affect the type and amount of information exchanged between the processor and co-processor and FBTS and ATE. I2 includes the ATE interface controller, the test interface of the PCB (e.g. JTAG), the communication infrastructure of the FBTS, and the debug-interface. This interface has to provide an efficient mechanism for the controllability and observability of the test execution. I3 represents the communication link between processor and co-processor, which can be implemented as a loosely- or a tightly-coupled system. In a loosely-coupled system the co-processor is completely independent of the processor, and in a tightly-coupled system, the co-processor is encapsulated as part of the processor.

The implementation of I3 as a loosely-coupled system is based on a system bus, in which every co-processor is mapped in the processor I/O address space. On the other hand, the implementation of I3 as a tightly-coupled system is based on specialized instructions to access the co-processor, which means that the co-processor is part of the processor data-path. Both approaches have already been investigated [122], and it was shown that loosely-coupled systems are more suitable for FPGAs. Therefore, the implementation of I3 as a loosely-coupled system is the preferred option for the ROBSY approach.

There are 21 possible layer partitions depending on the location of interfaces I2 and I3. They are obtained by placing I2 in the same location or above I3:

- 6 partitions when I3 is located below L1.
- 5 partitions when I3 is located between L1 and L2.
- 4 partitions when I3 is located between L2 and L3.
- 3 partitions when I3 is located between L3 and L4.
- 2 partitions when I3 is located between L4 and L5.
- 1 partition when I3 is located above L5.

However, the partitions are reduced to 10 if the constraints presented in sections 3.4.2 to 3.4.6 are considered (L5 implemented on the ATE and L1 in the co-processor). In this case, the 10 partitions are:

- 4 partitions when I3 is located between L1 and L2.
- 3 partitions when I3 is located between L2 and L3.
- 2 partitions when I3 is located between L3 and L4.
- 1 partition when I3 is located between L4 and L5.

Moreover, if the processor should at least carry out the functions of one of the layers, the number of partitions is reduced to 6 because I3 cannot be placed in the same location of I2. Figure 3.11 shows the 6 possible partitions.

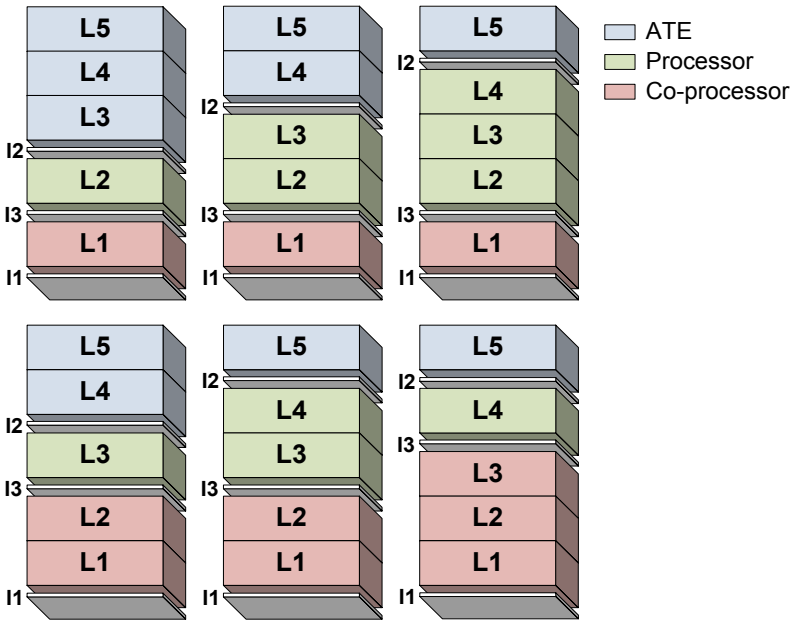


Figure 3.11 ROBSY layer partitions

In this dissertation, the processor is always part of the FBTS and L5 and L1 are always implemented on the ATE and co-processor, respectively.

3.5 Summary

In this section a new FPGA test instrumentation class known as ROBSY was presented. It differs from ad-hoc and generic FPGA test instrument classes in three main features:

- Automatic generation of the FBTS based on DUT-Ms.
- Implementation of test functions in FBTS or ATE based on a layer concept.
- FBTS architecture composed of loosely-coupled processor co-processor pairs organized in domains.

Co-processors are dedicated computation engines tailored to accelerate parts of the test execution. They are hardwired modules composed of data and control-paths whose hardware description is automatically generated during the development of the FBTS. They should be used at least for the implement of L1 in order to guarantee the execution of tests at-speed.

It was demonstrated that the inclusion of a processor as part of the FBTS is an alternative worth considering. The processor is a pre-designed component that is part of the FBTS that can be used for the execution of layers L2-L4 functions and the communication to the ATE. The implementation of layers in ESW does not affect the execution of tests at-speed, and therefore it does not compromise the coverage of dynamic faults. Implementing these layers in ESW influences the resource utilization of the FPGA and the test execution time.

From the design and automatic generation point of view, it is a ready-to-use component that uses a relatively constant and predictable amount of resources and does not require verification of its correct operation after the FBTS is automatically generated. However, executing test functions in ESW requires reserving FPGA resources for the processor and can degrade the test time due to the overhead of instruction cycles, making the fulfillment of strict requirements related to the utilization of resources and test execution time a concern. Therefore, it is necessary to:

- Add adaptation mechanisms to the processor that are transparent to the test engineer.
- Include proper support for the communication to the ATE and co-processor.
- Evaluate the implementation of the layers in ESW for different test scenarios.

4 Concept of the ROBSY processor

4.1 Introduction

Chapter 3 presented a discussion about the impact of including a processor as part of the FBTS. Part of this discussion was centered on the layer concept, defining the layers that can be implemented in ESW, and the support that a processor offers to interfaces I2 and I3. It was shown that the processor represents a promising alternative for the implementation of the following functions:

- I2 and I3 interfaces.
    - Information exchange with co-processors.
    - Information exchange with the ATE.
- L2-L4 layers.
    - Pattern generation and analysis.
    - Control flow of test algorithms and diagnosis.
    - Coordination of test algorithms.

Furthermore, it was shown that adaptation plays an important role in the ROBSY approach. Finely tuning the processor to the specific test scenario leads to an efficient usage of FPGA resources and improves test time. This chapter presents the concept of the ROBSY processor, which is based on three main pillars:

- Analysis of general design aspects.
- Processor specialization for testing.
- Processor adaptation mechanism.

Section 4.2 presents general design aspects of the processor, including the definition of processor requirements, analysis of processors available in the literature, and main design decisions. Section 4.3 presents the specialization options considering the test application field, test operations, and interfaces I2 and I3. Finally, Section 4.4 presents a discussion of the adaptation mechanism supported by the ROBSY processor.

4.2 General design aspects

The definition of the ROBSY processor requirements is presented at the beginning of this section. Based on these requirements, processors found in the literature (Section 2.4) are

analyzed as potential candidates for the FBTS. At the end of this section, a discussion about fundamental design decisions is presented.

4.2.1 ROBSY processor requirements

Before starting with the design of the ROBSY processor, it is necessary to define its main functional and non-functional requirements. The former describe the functionality that the processor should provide to the ROBSY test system, while the latter describe attributes of the processor more related to performance and implementation costs.

The main functional requirements are:

i. The processor should have an instruction set that allows the execution of test functions related to layers L2, L3, and L4.
ii. The processor should be equipped with an interface to the ATE. This interface should provide the option to control and observe program execution and exchange data.
iii. The processor should be equipped with an interface to the co-processors. This interface should scale well and allow the attachment and management of co-processors in a simple way.

Requirement (i) defines the processor as an instruction programmable machine capable of fetching, decoding, and executing instructions. Requirements (ii) and (iii) are necessary for communication to the ATE and co-processors. In the first case, a standard test interface that provides the option to connect multiple processors should be considered. In the second case, an on-chip bus used as communication channel to exchange information between processor and co-processors should be considered.

The main non-functional requirements of the processor are:

i. The processor should be able to adapt to the specific test scenario. This means, adapting the functionality, resources, and performance depending on the FPGA, DUTs, and test quality metrics.
ii. It should be designed for the implementation in FPGAs.
iii. The hardware description should be portable to FPGAs from different families and vendors in order to support a wider range of test scenarios.
iv. The processor should be designed considering test time as the main test quality metric and with the aim of achieving efficient resource utilization.
v. There are no requirements concerning energy consumption or security issues.

Non-functional requirement (i) basically highlights the importance of adapting the processor to the specific test scenario. This includes adaptation mechanisms that allow tuning the processor to the layers implemented in ESW, DUT properties, available FPGA, and test requirements. The adaptation also includes the processor specialization for testing. This means equipping the processor with special test instructions and avoiding the implementation of functions that are not required during test execution.

Requirement (ii) implies that design decisions have to consider the FPGA as the target platform. This means that the processor is implemented by means of look-up tables, flip flops, memory blocks, multipliers, etc. Furthermore, the processor has to be completely encapsulated in the FPGA. This means that it is not possible to use external components (e.g. memories) placed on the PCB because there is no guarantee that the interconnections to external components are defect free.

Requirement (iii) guarantees the use of the ROBSY approach in a wide range of PCBs. FPGAs from different families and vendors are equipped with diverse hardware blocks that do not necessarily share the same properties. Additionally, synthesis is carried out based on different synthesis tools.

Requirement (iv) shows that test time and resource utilization are the main metrics that can be influenced with the processor. The program execution time of a processor is known as CPU time, and is approximated to [123]:

$$CPU\_time = Instruction\ count \cdot CPI \cdot Clock\ cycle\ time \tag{4.1}$$

Equation (4.1) shows that the number of instructions (instruction count), clock period (clock cycle time), and clock cycles per instruction (CPI) have a direct influence on CPU time. The three values are not independent, and in most cases improving one of them leads to the deterioration of the other two [123]. Therefore, it is always necessary to find a balance between them.

On the other hand, the resource utilization defines the number of processors and FBTS domains that can be implemented in a given FPGA. An efficient resource utilization of the processor provides the option to implement more functions in the FPGA.

Finally, there are no requirements (v) concerning energy consumption and security. It is assumed that these requirements are not critical during the PCB manufacturing test.

4.2.2 Analysis of pre-designed processors

One of the first decisions to make when including a processor as part of any digital system is whether to use a pre-designed processor or develop a new one. This issue is also considered in the ROBSY approach, and requires the analysis of processors available in the literature based on the requirements presented in section 4.2.1.

Section 2.4 introduced popular soft-core processors and specialized test processors that are found in the literature. Sections 4.2.2.1 and 4.2.2.2 present an analysis of the use of these processors as part of the ROBSY approach.

4.2.2.1 Analysis of soft-core processors

The utilization of proprietary soft core processors in the ROBSY approach was discarded from the very beginning. These processors would limit the scope of ROBSY to FPGAs of a specific family or vendor and would increase nonrecurring costs due to license fees.

Open-source soft-core processors require a more detailed analysis. They do not increase nonrecurring costs due to license fees and provide the option to edit the source code either for portability to different FPGAs or for customization of the processor. For the analysis, they are divided into two groups of processors:

- High-end processors.
- Low-end processors.

The Plasma, Leon 3, aeMB, Amber, OpenRisc 1200, and Mico32 processors are part of the high-end class. They are used in a wide range of applications characterized by high performance and functional requirements. These processors are equipped with 32-bit data and instruction buses, a large amount of registers, configurability based on the activation or deactivation of modules, and features for supporting operating systems. However, high-end processors do not represent a good alternative for the ROBSY approach. They are over-featured for testing purposes and their ISA does not provide the required adaptability.

Floating point units, memory management and protection units, and different execution modes are not necessary for testing, and therefore have a negative impact on the resource utilization of the FBTS. If a test scenario requires the implementation of FBTS with several domains, the instantiation of multiple high-end processors would be only possible in high-capacity FPGAs. Therefore, the field of applicability of the ROBSY approach would decrease.

In the same way, the adaptability provided by these processors is limited. Adaptation mechanisms with a high impact on the resource utilization are not supported. These high-impact mechanisms allow the configuration of the number of bits used for the data bus, instructions, memories, and I/O addressing. As a consequence, it would be not possible to make an efficient use of resources in test scenarios of low complexity.

The T48, Copyblaze, Mico8, and Proteus processors are part of the low-end class. They have a simpler ISA with instructions coded using a low number of bits (8x, 16, 18), short data bus widths (8, 16), low number of registers, and no operating system support. They are exclusively conceived to execute simple tasks that do not require much memory and data processing capabilities.

The low flexibility of the T48, Copylaze, and Mico8 ISAs does not provide the adaptability required for the ROBSY approach. The ISA of the Proteus provides greater flexibility, making it possible to work with two data width configurations (8 or 16-bit). However, the limited number of bits available to code an instruction restricts the number of available registers, the range of immediate values, and the depth of the program memory. This limits the memory capacity of the processor to store code and data, and therefore would limit the implementation of layer functions in ESW.

To conclude, high-end and low-end general soft-core processors are characterized by standard ISAs equipped with features that are not necessary for ROBSY or that do not provide enough adaptability. Additionally, the specialization for testing purposes is not provided in any of the two cases. This would impact test time and resource utilization, making their use in the ROBSY approach unsuitable.

4.2.2.2 Analysis of test processors

From the test processors presented in Section 2.4, the programmable test processors are the only ones relevant for this dissertation. Hardwired test processors are more related to the co-processors and are not further analyzed.

The relevant features for the ROBSY approach of the processor concept used for SoC test [19, 81–86] are:

- 16-bit RISC with fixed ISA.
- Four processor versions for adaptation purposes.
- Scan-path test and fast pattern comparison.
- I/O ports used as communication mechanism to ATE.
- Self-test and on-line test support.
- Pattern generation and analysis based on LFSRs and MISRs.

The first five features are not of any interest to the ROBSY approach. A processor with a fixed ISA and an adaptation mechanism that relies on four processor versions is not appropriate for ROBSY because the obtained flexibility is very limited. A special I/O scan-path controller with fast pattern comparison does not have any use in ROBSY. Scan-path tests are not carried out at the board level and the utilization of a fast pattern comparison approach requires special test components (reflectors) at the other end of the PCB interconnections. Including special components in the PCB for testing purposes is opposite to the ROBSY approach.

The use of I/O ports for communication to the ATE does not fulfill the requirements of the ROBSY processor. The observability and controllability of the processor from the ATE side is critical in the ROBSY approach, and it cannot be obtained based on I/O ports. Self-test capabilities are not a requirement in ROBSY because the FPGA is assumed to be defect free. In the same way, on-line test capabilities are not necessary. It is assumed that the tests are performed in a controlled environment and that they take a very short time. Therefore, the probability of transient or intermittent defects is very low.

On the contrary, the use of pattern generation and analysis mechanisms is an attractive feature for ROBSY. This feature can make the implementation of L2 functions in the processor more attractive because it reduces complexity and the time required for the processor to generate pseudo-random patterns. However, the generation of other type of patterns, such as algorithmic and deterministic sequences, should be also considered.

The test processor for asynchronous circuit testing [87–89] requires an independent analysis of the two proposed concepts. The first concept is based on a RISC processor, while the second concept is based on a minimalistic processing unit, known as the sequencer. The RISC processor has the following features:

- 16/32-bit RISC processor with 4 pipeline stages.
- Configurable asynchronous handshake protocols.
- I/O ports used as communication mechanism to ATE.
- Standard instruction set for arithmetic, logic, and control operations.
- Instructions based on complex sequences to generate and apply test patters, and to compact and acquire test responses.

The 32-bit processor version provides more capacity for demanding test scenarios in comparison to the processor for SoC test, but this still might lead to non-optima resource utilization for test scenarios that do not require large data widths. Besides, the processors do not support other adaptation mechanisms. The use of configurable asynchronous handshake protocols is not necessary because L1 functions should be implemented by the

co-processors. As already mentioned, the use of I/O ports for the communication mechanism to the ATE is not appropriate for the ROBSY approach.

On the other hand, pipelining and a large instruction set for arithmetic, logic, and control operations are attractive features for the ROBSY processor. The former improves the CPI, while the latter facilitates the implementation of layers L2 to L4 functions in ESW. In the same way, specialized instructions to generate and analyze patterns are an attractive feature. On the other hand, the use of instructions composed of complex sequences to apply and acquire patterns should be further analyzed (Section 4.2.3.2).

The second test processor developed for asynchronous chip tests is optimized for the application of patterns and acquisition of test responses. In this case, the unit with a closer relation to the ROBSY processor is the sequencer, which has the following features:

- Minimalistic architecture with 2 pipeline stages.
- Limited instruction set with most of the instructions specialized for the configuration and control of other units.
- No pattern generation or analysis instructions.
- Non-standard communication interface to the ATE.

The sequencer is not in charge of generating patterns or processing data obtained during test execution. Its tasks are mainly the configuration of other units and coordination of test execution. The ATE is responsible for off-line pattern generation and the analysis of test responses. As a consequence, the functionality provided by the sequencer is very limited, and does not fit with the functional requirements of the ROBSY processor. The use of a non-standard communication interface to the ATE goes against the ROBSY processor requirements. Such an interface cannot be implemented in every PCB because it would require the modification of the PCB just for testing purposes.

To conclude, test-processors found in the literature do not represent good alternatives for the ROBSY approach. They are used for testing at the device-level, and therefore are not optimized for board-level testing. As a consequence, the design of a test processor that fulfills the requirements presented in section 4.2.1 is inevitable. Nevertheless, there are some features included in programmable test processors that are considered. These features are the use of configuration mechanisms for adaptation purposes, a standard instruction set for control flow, arithmetic, and logic operations, specialized instructions for pattern generation and analysis, and pipelining.

4.2.3 Fundamental design choices

For the design of the ROBSY processor, it is necessary to discuss fundamental design choices. They comprise the design abstraction level, the design philosophy of the processor architecture, and technology constraints related to the use of FPGAs as implementation platforms.

4.2.3.1 RTL as design abstraction level

Today, register transfer level (RTL) is the abstraction level commonly used for the design of FPGA-based systems. However, there is another processor design methodology that allows working at a higher abstraction level. Its goal is to reduce the design time and complexity in comparison to RTL, providing the option to explore different design alternatives faster [124, 125]. It uses processor description languages also known as architecture description languages (ADLs), which allows the specification of the processor behavior and its structure. In this case, depending on the ADL language, it is possible to automatically generate the RTL description, as well as the required development tools (compiler, assembler, instruction set simulator, and debugger).

However, working at a higher level of abstraction reduces the control over design parameters such as timing, resource utilization, and power consumption. Therefore, a decrease in the quality of these values is expected. This is shown in [126, 127], where the authors claimed that the RTL descriptions automatically generated from an ADL description are often of poor quality. In [128], the authors compare an ADL implementation to a handwritten RTL implementation of the same processor, and they show that a 20%-30% overhead in area, clock frequency and power consumption is obtained with the ADL implementation. Another example corresponds to the test processor used for asynchronous chip test (Section 2.4.2). The first two test processor versions [87, 88] were developed using the LISA ADL, while the third version [89] was developed at the RTL level using the Very High Speed Integrated Circuit Hardware Description Language (VHDL). The reason given in [89] for going back to the design at the RTL level was the optimization of processor performance.

The design of an efficient processor in terms of resource utilization and operation speed is crucial for the ROBSY processor. Additionally, the automatic generation of a compiler based on ADL is not suitable for the ROBSY approach because available ADL tools cannot generate a compiler for the RTDL language. For these reasons, the ROBSY processor is designed at the RTL level using VHDL.

At this point it is important to mention that the support for different FPGA families and vendors is achieved by using generic synthesizable VHDL constructs that are understood

by different synthesis tools. These constructs allow the synthesis tools to infer the proper device-specific hardware blocks such as memory blocks [129]. However, there are hardware blocks that cannot be inferred, such as JTAG interfaces. In this case, it is necessary to include the proper VHDL description depending on the FPGA.

4.2.3.2 RISC design philosophy

The reduced instruction set computer (RISC) design philosophy [130, 131] is selected for the design of the ROBSY processor. This is the design philosophy followed by most processor designers, and it has been adopted even by Intel for the implementation of modern x86 processors [132].

The RISC design philosophy achieves good performance metrics by means of simple instructions executed in few clock cycles at high clock frequencies. This approach is very efficient because compilers tend to use simple instructions to synthesize high-level constructs. Taking this into consideration, the RISC design principles used for the design of the ROBSY processor are:

- *Simple instructions:* Simple instructions require few clock cycles for their execution. It simplifies the processor design and achieves high clock frequencies because the use of microcode or complex instruction decoders is not necessary.
- *Load-store architecture:* Dedicated load and store instructions with simple addressing modes enable arithmetic and logic instructions to be independent of memory or I/O accesses. This simplifies the instruction set and the design of the processor control unit, facilitating a fast computation of operand addresses and the achievement of high clock frequencies.
- *Large register set:* A large register set is necessary to support register-register operations. This minimizes the overhead associated with procedure calls and provides plenty opportunities for the compiler to optimize the register usage.
- *Fixed length instruction format:* Fixed length instructions allow an efficient fetching and decoding of instructions, and minimizes the design complexity of the processor.

The main limitation of the RISC design philosophy is the need for more instructions to execute the same operations in comparison to processors based on the complex instruction set computer (CISC) philosophy. This is translated into the use of more memory for the storage of instructions. Fortunately, algorithms used for interconnection test are relatively short and do not require large amounts of instruction memory. Additionally, this effect is reduced by the use of an efficient compiler.

4.2.3.3 FPGA technology constraints

The design of the ROBSY processor based on the RISC philosophy offers the option to exploit design techniques such as pipelining and instruction level parallelism (ILP). Pipelining takes advantage of the parallelism that exists among the actions needed to execute an instruction, overlapping the execution of instructions. On the other hand, ILP represents a group of techniques that exploits the parallelism among instructions. Widespread ILP techniques are superscalar and very large instruction word (VLIW) techniques [123, 132].

The improvement level achieved by using pipelining and ILP techniques is measured in terms of the processor CPI or the inverse of CPI, which is a metric known as instructions per cycle (IPC) [123]. Pipelining approximates IPC of a scalar processor to one, while ILP increases this value above one.

However, the use of an FPGA as the technology basis for the implementation of the FBTS imposes some limitations on the design techniques that can be used efficiently in terms of clock frequency and resource utilization. In [131] the authors claimed that superscalar or VLIW techniques are not practical for FPGAs. This is due to the FPGA limitations for the implementation of multi-port register files, and the low performance achieved when the control logic required for out-of-order execution is implemented in an FPGA. In more recent investigations [133–135] this radical belief has been changing because of the higher capacities of modern FPGAs and the development of new methods for an efficient implementation of multiport register files. However, resource utilization is still very high in comparison to scalar processors.

The high resource utilization necessary for ILP techniques goes against the ROBSY approach. It limits the amount of processors or FBTS domains that can be implemented in an FPGA, and restricts the use of the ROBSY approach to high-capacity FPGAs. Therefore, the ROBSY processor is a scalar RISC processor with a single-issue in-order microarchitecture and pipelining support. As shown in Section 2.4, this is the methodology followed by most soft-core and test processors found in the literature.

4.3 Processor specialization for testing

Processor specialization for testing is essential for the ROBSY approach. The goal is to improve test time and make an optimal usage of FPGA resources. The former reduces the time the PCB spends on the tester, while the latter facilitates fitting the processor in FPGAs of low capacity and implementing a multi-domain FBTS.

Processor specialization includes three main aspects: tailoring the processor functionality for testing, defining standard and special test instructions for the implementation of layers L2 to L4, and defining the support for interfaces I2 and I3.

4.3.1 Tailoring the processor for testing

There are features of standard processors that are not strictly necessary for the ROBSY processor. These features consume resources and increase the complexity of the design. Therefore, it is necessary to identify them to avoid their inclusion in the processor.

4.3.1.1 Machine data types and widths

Machine data types correspond to the native data types supported by a processor. In the case of ROBSY, the support for fixed-point arithmetic is sufficient. Fixed-point values and corresponding arithmetic and logic operations are ideal for describing test patterns, test responses, signatures, and test operations. Layers L2 to L4 typically work with unsigned values. However, signed addition and subtraction operations do not require additional modules in the processor. This is realized by means of two's complement values and sign and overflow conditions.

The support for floating point data types is not necessary for the ROBSY processor. Layers L2 to L4 do not require floating point arithmetic or logic operations because they cannot be used to represent patterns, signatures, or interconnections in an efficient way. In the same way, the support for character or Boolean data types is not necessary. Character-based or string-based operations are not essential for layers L2 to L4, and Boolean values can be easily represented using fixed-point values.

The only composite data type required for the ROBSY approach corresponds to vectors of fixed-point values that are used to describe and store deterministic patterns and fault dictionaries efficiently. Deterministic patterns and fault dictionary signatures are generated during the pre-test phase, and therefore do not suffer any transformation during test execution. As a consequence, equipping the processor with arithmetic and logic operations for vector processing is not necessary. Standard load and store instructions are sufficient to handle vectors.

The data width defines the number of bits that constitute a fixed-point value in the processor. As will be presented in Section 4.4.2, the ROBSY processor should support different data widths for adaptation purposes. Therefore, a possible alternative is to have a processor with support for standard widths (8-bit bytes, 16-bit words, 32-bit double words, etc.). However, this requires a large instruction set because single instructions for

the same arithmetic, logic, load, and store operations have to be defined for each data width. This increases the instruction coding requirements and the complexity of the processor's decoding logic. For this reason, the ROBSY processor is limited to a single data width value, avoiding the need for multiple instructions for the same operations.

To summarize, the ROBSY processor is designed to execute unsigned fixed-point arithmetic and logic operations. Vector processing is not required, but standard load and store operations are necessary to access a set of deterministic patterns or the content of a fault dictionary. In terms of data width, the ROBSY processor is limited to a single configurable data width value.

4.3.1.2 Operating system support

An operating system represents a group of programs that forms an abstract interface between the hardware resources of the processor and the application program. Its main job is to provide for an orderly and controlled allocation of hardware resources among the application programs that are competing for them [136]. Therefore, it deals with dynamic scheduling of processes, abstraction of memory and I/O, hardware protection, etc. In order to support an operating system, a processor should be equipped with memory management and protection units, different operation modes, privilege instructions, and fast context switching features.

However, scheduling and allocation of resources based on an operating system is not necessary for the ROBSY approach. The ROBSY processor runs a single application program defined by the DUT-M that can be properly scheduled during the pre-test phase in a static way. In addition, other aspects of the ROBSY approach that go against the use of an operating system are:

- The coordination of test algorithms is completely defined in the DUT-M.
- The test execution of DUTs belonging to the same domain is performed one DUT at a time in order to guarantee the fulfillment of timing requirements of the DUT.
- Each domain of the FBTS works independently, which means that there is no need for inter-processor synchronization or communication.
- Abstraction mechanisms to access the DUTs do not have to be considered. L1 provides the mechanism to define the timing and functional behavior of the access functions. This makes it possible to provide reliable statements about the test coverage of dynamic faults.

To conclude, there is no need to use an operating system as part of the ROBSY approach.

4.3.1.3 Memory hierarchy

Modern processors are equipped with data and program memories organized in a hierarchical way to minimize the latency of fetching instructions and data. These levels are built based on processor registers, internal cache memories, and slower external memories such as SRAM, DRAM, and disk storage devices.

In the ROBSY approach, external memories are not used because the interconnections to these devices might contain defects. Therefore, the ROBSY processor makes use of internal memory resources for the storage of instructions and data. These resources correspond to registers and tightly coupled memories. They simplify the memory hierarchy of the processor because it is not necessary to consider cache memories.

4.3.2 Test operations

A general purpose processor executes test operations by means of standard instruction sequences. If these sequences are composed of many instructions, they will consume numerous memory resources and will require a considerable amount of time to execute. In order to improve processor execution time, it is necessary to identify frequently used functions performed by layers L2 to L4, which, if necessary, can be included in the instruction set of the ROBSY processor.

4.3.2.1 Pattern generation (L2)

One way to reduce processor execution time is by equipping the instruction set with single instructions capable of generating patterns and analyzing test responses.

Section 3.4.3 classified test patterns used in the FBTS into algorithmic, pseudo-random, exhaustive/pseudo-exhaustive, and deterministic patterns. Algorithmic, pseudo-random, exhaustive, and pseudo-exhaustive patterns are generated by means of arithmetic and logic operations, in which every new pattern is computed based on previous patterns. Deterministic patterns are defined during the pre-test phase, and therefore do not suffer any modifications during test execution. They are typically stored in memory.

For static and dynamic fault testing, the ROBSY processor should generate algorithmic patterns based on walking-1/0, modified counting, interleaved true/complement, and maximum aggressor fault model sequences. The first four sequences provide high fault coverage values with a reduced number of patterns [41]. The last sequence represents an alternative to detect signal integrity faults at the interconnections [22, 23, 121].

Table 4.1 presents walking-1/0, modified counting, and interleaved true/complement sequences for 4-bit patterns. The pattern column shows the pattern value, and the

operation column shows the operation used to generate the pattern of the same row. The value in parentheses identifies the number of the previous pattern used to generate the new one. Initial patterns have an empty operation field.

#	Walking-1		Walking-0		Modified counting		Interleaved true/complement	
	Pattern	Operation	Pattern	Operation	Pattern	Operation	Pattern	Operation
1	0001		1110		110010		110010	
2	0010	Shift left(1)	1101	Rotate left(1)	100101	Rotate left(1)	001101	Negate(1)
3	0100	Shift left(2)	1011	Rotate left(2)	001011	Rotate left(2)	100101	Rotate left(1)
4	1000	Shift left(3)	0111	Rotate left(3)			011010	Negate(3)
5							001011	Rotate left(3)
6							110100	Negate(5)

Table 4.1: Walking one/zero, modified counting, and interleaved true/complement sequences

Table 4.1 shows that a walking-1/0 sequence is obtained based on an initial pattern with just one bit set to 1/0 and shift or rotate operations. A modified counting or interleaved true/complement sequence is obtained based on a special initial pattern and rotate or negation operation. The initial pattern is computed before the test execution by means of a maximal length LFSR sequence that starts with the all ones value. The low significant bit of each pattern of the sequence (without considering the all ones value) corresponds to the initial pattern [41]. Although not mentioned in [41], it is necessary to work with all bits of the sequence independently of the number of interconnections under test. In the example presented in Table 4.1, six bits are used to generate the sequence for four interconnections. The pattern corresponds to the underlined bits.

The maximum aggressor fault model also uses algorithmic sequences. Table 4.2 shows the sequences for the detection of negative/positive glitches and falling/rising delays. The table shows that three initial patterns and shift, rotation, and negation operations are used. The initial patterns have a single bit set to 1/0 or all bits set to 1/0.

#	Negative glitch		Positive glitch		Falling delay		Rising delay	
	Pattern	Operation	Pattern	Operation	Pattern	Operation	Pattern	Operation
1	1111		0000		0001		1110	
2	0001		1110		1110	Negate(1)	0001	Negate(1)
3	1111		0000		0010	Shift left(1)	1101	Rotate left(1)
4	0010	Shift left(2)	1101	Rotate left(2)	1101	Negate(3)	0010	Negate(3)
5	1111		0000		0100	Shift left(3)	1011	Rotate left(3)
6	0100	Shift left(4)	1011	Rotate left(4)	1011	Negate(5)	0100	Negate(5)
7	1111		0000		1000	Shift left(5)	0111	Rotate left(5)
8	1000	Shift left(6)	0111	Rotate left(6)	0111	Negate(7)	1000	Negate(7)

Table 4.2: Maximum aggressor fault sequences for glitches and delays

Table 4.1 and Table 4.2 show that the algorithmic test sequences do not require any special instructions for their generation. Each pattern is obtained based on one or more initial patterns and standard shift, rotate, and negation instructions.

In the same way, the generation of exhaustive or pseudo-exhaustive pattern sequences does not require the use of special operations. The sequences are generated by standard increment or decrement operations that emulate the function of an accumulator or counter. These operations are performed by addition and subtraction instructions.

Pseudo-random pattern sequences are typically generated by LFSRs, which use at least one rotation and XOR operation per pattern. Therefore, they require multiple standard instructions per pattern. The use of special instructions for the generation of pseudo-random patterns is a good alternative for the ROBSY processor because it reduces the number of instructions required per pattern.

Deterministic patterns do not suffer any transformations during test execution. Therefore, the ROBSY processor does not include special instructions for this kind of pattern sequences. However, it is necessary to include proper support for loading patterns from memory and transferring them to the co-processor. Loading deterministic patterns can be realized in two different ways. If the patterns are stored in the program memory (coded in the instructions), then loading patterns becomes fetching an instruction from program memory. But if the patterns are located in data memory, loading deterministic patterns requires the use of load instructions. On the other hand, transferring deterministic patterns (as well as all generated patterns) to the co-processor can be performed by means of store instructions. Standard load and store instructions for accessing patterns from memory and transferring them to the co-processor should be able to access large memory arrays and execute in few clock cycles to reduce their impact on processor execution time.

To summarize, the generation of algorithmic, exhaustive, and pseudo exhaustive patterns is performed based on single instructions. For this purpose, it is necessary to include standard addition, subtraction, shift, rotate, and negation instructions as part of the ROBSY processor. On the other hand, the generation of pseudo-random patterns requires the use of special instructions based on LFSRs. The use of deterministic patterns is realized by coding patterns in the instructions and supporting standard load and store instructions that execute in few clock cycles and are able to access large memory arrays.

4.3.2.2 Test response analysis (L2)

Section 3.4.3 showed that there are two different options for the analysis of test responses. The first option compares each test response with an expected one. The second option compares signatures instead of test responses, which are obtained by compacting

expected test responses during the pre-test phase and the acquired test responses during test execution.

The comparison of test responses can be performed based on standard instructions that compare values. This is usually performed by instructions that subtract the two values to compare and set a processor general purpose register or group of condition flags (zero, carry, and overflow) to a corresponding value. In this case, it is necessary to store the expected responses in memory or generate them during the test. The comparison of signatures reduces the amount of comparisons that have to be performed and the memory required to store expected responses. However, it is necessary to spend time in the compaction of the acquired test responses, and consider the occurrence of aliasing. The compaction of test responses is typically performed by a MISR, whose operation can be emulated by the processor based on at least two XOR and one rotate instruction per test response. In order to reduce the number of standard instructions, a single special instruction that emulates a MISR can be implemented. An alternative is to reutilize the special instruction used for the generation of pseudo-random patterns. This is performed by an LFSR instruction and an XOR instruction, and avoids the implementation of the MISR instruction.

The analysis of test responses in the ROBSY processor is realized based on standard comparison instructions and the two instruction approach that emulates a MISR for the compaction of test responses.

4.3.2.3 Control flow operations (L2, L3, L4)

Control flow operations are required for all layers. L2 relies on loops in order to describe instruction sequences for pattern generation and analysis. L3 is mainly composed of control flow operations that describe the structure of test algorithms, in which branches and loops are executed based on conditions typically derived from the L2 results. L4 uses control flow operations for the coordination of test algorithms, and diagnosis purposes. Additionally, the layer concept requires a mechanism to encapsulate functions related to a given layer. This is accomplished by supporting procedures and procedure calls.

Control flow operations are described by constructs supported by the RTDL language such as for, while, if, else, switch, call, and return statements. These high level statements are transformed to unconditional and conditional branches by the compiler (at least in the RISC design philosophy). Unconditional branches are executed without evaluating any conditions, while conditional branches require the evaluation of conditions typically set by arithmetic and logic instructions.

The ROBSY processor requires both kinds of branching mechanisms. In this case, the instructions considered are:

- Unconditional branch instructions with absolute addressing mode.
- Conditional branch instructions with absolute addressing mode.
- Conditional branch instructions based on the *set then jump* methodology.
- Conditional branch instructions based on a condition code register.
- Call and return instructions that save/restore the return address to/from memory.

Branching instructions use an absolute addressing mode instead of a relative addressing mode. The use of absolute addresses is enough for the ROBSY processor, because the location of instructions in memory is defined during compilation. Therefore, it is not necessary to support code reallocation, simplifying the processor instruction set. The *set then jump* methodology [130] was selected over the *set and jump* methodology. The former uses separate instructions to set conditions and perform the branch, while the latter uses a single instruction for both operations. The *set then jump* methodology was selected in order to maintain the complexity of each instruction as low as possible.

A condition code register represents a collection of status flags such as zero, carry, sign, and overflow. The flags are set by means of arithmetic and logic instructions and allow the implementation of equality or relation operators. This approach was chosen over general purpose registers because flags also provide an efficient way to concatenate arithmetic and logic operations for operands larger than the processor data width (Section 5.2.3).

Call and return instructions are necessary in order to support layer procedures and interrupts. The call instruction uses an absolute addressing mode and stores the return address in the stack. The return instruction is in charge of restoring the return address from the stack and branching to this address. For the ROBSY processor, the use of conditional call instructions is not considered.

4.3.2.4 Diagnosis (L2, L3 and L4)

Diagnosis can be performed in order to locate the device with faulty interconnections, or to locate the device and faulty interconnections.

Diagnosis of the device with faulty interconnections does not require the use of specific diagnosis algorithms because the identification of the processor/co-processor pair that detects the fault is sufficient to recognize the DUT with the faulty interconnection. On the other hand, diagnosis of the faulty interconnections requires the utilization of test

algorithms developed for this purpose. These algorithms rely on sequential and combinational diagnosis methods as presented in Section 2.1.5.

Sequential diagnosis methods require the implementation of diagnosis operations as part of layers L2 and L3. L2 generates patterns and analyzes the obtained test responses. L3 decides the next step to perform based on the information delivered by L2. For this purpose, it is sufficient to use the instructions already discussed in Sections 4.3.2.1, 4.3.2.2, and 4.3.2.3. Combinational diagnosis methods require the implementation of diagnosis operations as part of layers L2, L3, and L4. L2 compacts the test responses in a signature, which requires the use of diverse standard arithmetic and logic operations depending on the type of fault dictionary. L3 describes the algorithm steps carried out for diagnosis based on the control flow instructions already discussed in Section 4.3.2.3. L4 performs the look-up process that matches the computed signature against the signatures stored in the fault dictionary. For this purpose, standard load, store, and comparison instructions are sufficient.

The ROBSY processor does not have to be equipped with special instructions for diagnosis purposes. Instructions used for the description of control flow, pattern generation, and test response analysis are sufficient.

4.3.3 Interfaces I2 and I3

Layer functions implemented in ESW are bound by interfaces I2 and I3. As presented in Section 4.2.1, the ROBSY processor should be equipped with the proper features for the implementation of both interfaces.

4.3.3.1 Interface I2

Interface I2 represents the interface between the ATE and FBTS, or more exactly, between the ATE and processor forming part of each FBTS domain. It comprises the ATE test controller, PCB communication infrastructure, FBTS communication infrastructure, and the processor debug-interface.

The PCB communication infrastructure usually corresponds to a test interface. The reason for this is that test interfaces use a low number of interconnections and can be easily tested. The de facto test interface considered for ROBSY is the JTAG interface used for BScan testing (Section 2.2.3.2), which is tested by means of basic infrastructure tests. The JTAG interface is compatible with ATEs used for BScan testing, which makes it not necessary to develop a special ATE and test controller for the ROBSY approach.

Additionally, FPGAs provide JTAG support for configuration and communication purposes [43].

The FBTS communication infrastructure relies on special DRs, which are connected to the TAP controller through a JTAG primitive. The use of DRs provides the scalability necessary to couple an arbitrary number of FBTS domains, forming a network of DRs. The network can be described based on the IEEE 1149.1-2013 BScan standard [1] or the IEEE 1687 IJTAG standard [137–139]. Figure 4.1 presents the data chain of a flat network of DRs, in which each DR belongs to one of the four FBTS domains. Other network configurations are possible, such as hierarchical structures.

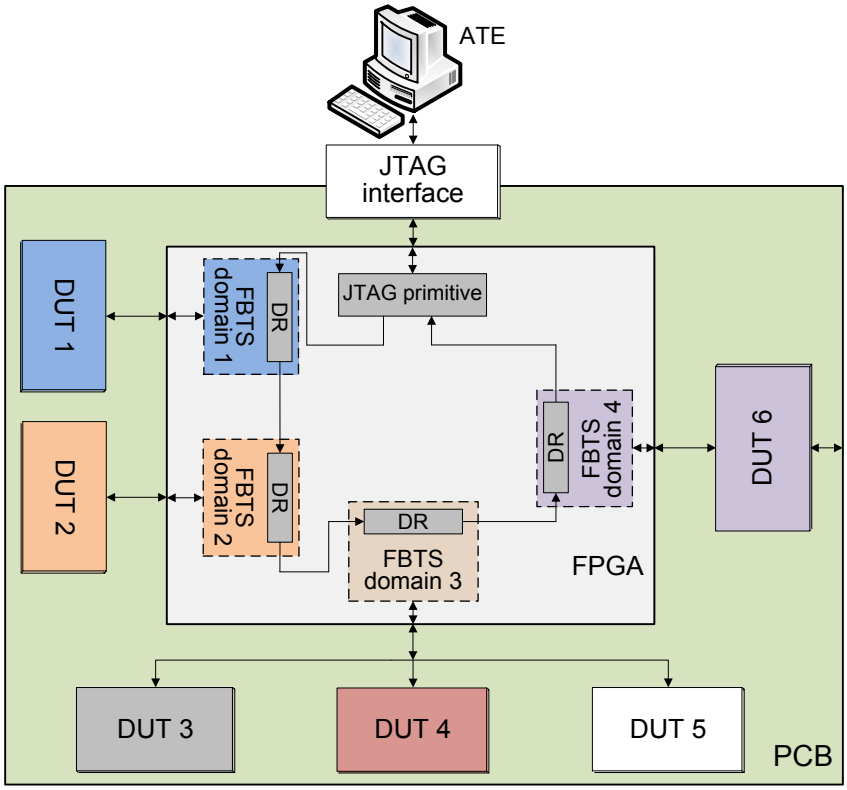


Figure 4.1 FBTS domains with DRs used for communication with ATE

As already discussed in Section 3.3.2, each FBTS domain has a single processor, which is the component involved in the ATE/FBTS communication. The ROBSY processor is equipped with a debug-interface implemented using the DRs located in each FBTS domain. The use of debug-interfaces is the same approach of PBT, which has been proven to be very successful [45, 46]. A debug-interface provides a mechanism to control and observe the state of the processor during the test execution. Moreover, the debug-interface is a standard communication interface that does not change its properties independently of the FPGA. This means that problems related to the support for different debug-interfaces (one of the main disadvantages of PBT) are not considered in ROBSY.

4.3.3.2 Interface I3

Interface I3 is the communication link between the main components of each FBTS domain, namely the processor and co-processors. As discussed in Section 3.4.6, the structure of each FBTS domain represents a loosely-coupled system that comprises a single processor and at least one co-processor. The processor acts as a master with total control over the bus transfers, whereas the co-processor acts as a slave responding to petitions made by the processor.

From the point of view of the ROBSY approach, the amount and type of data transferred through interface I3 varies depending on the location of the interface in the layer concept. Therefore, an on-chip bus should provide the necessary flexibility and scalability. The on-chip bus is based on the Wishbone interconnection architecture supplied by the Open Cores organization [140]. This architecture fits very well in the ROBSY approach because it is open source, scalable, and provides a simple, flexible, and portable way to connect different components. Additionally, it is very popular and many Wishbone compatible IPs are freely available. For a comparison of Wishbone with other on-chip interconnection architectures, refer to [141].

4.4 Adaptation to the test scenario

As presented in Section 3.1.2, a test scenario differentiates itself from others based on three main aspects: DUTs, FPGA, and test requirements. These aspects vary strongly from case to case, making it necessary to include adaptation mechanisms as part of the ROBSY approach.

The adaptation of the ROBSY processor makes it possible to ensure an efficient usage of FPGA resources or to improve the processor execution time. The efficient usage of resources facilitates the inclusion of additional layers and domains in the FBTS that otherwise could not be included in the FPGA. An improvement of the processor execution time facilitates the fulfillment of critical test requirements related to test time. Equipping the ROBSY processor with an adaptation mechanism allows:

- Fine tuning the processor ISA and microarchitecture to the constraints of the specific test scenario.
- Including the adaptation mechanism in the FBTS automatic generation flow.

Sections 4.4.1 and 4.4.2 present available adaptation mechanisms and the one selected for the ROBSY processor.

4.4.1 General adaptation mechanisms

There are two general mechanisms that can be used for the adaptation of a processor to a specific scenario, whether the scenario refers to a test scenario or not. The two mechanisms are:

- Development of a family of processors.
- Development of a configurable processor.

The first mechanism adapts the processor by selecting the proper one from a family of processors depending on the scenario. Every processor is developed independently, but it might share some properties with other family members. In this case, the achievable level of adaptation depends on the number of available processors and on the way each processor fits in the given scenario. Typically, this approach is used for complex processors without configuration options.

In the second mechanism, the adaptation of the processor is performed by selecting the constellation of configuration parameter values that is more appropriate for a given scenario. In this case, it is necessary to develop a single configurable processor in order to generate different processor variants. This is the mechanism typically used in soft-core processors.

It is possible to develop a family of processors or a configurable processor that is grouped in one of the following classes:

- Same ISA and different microarchitectures.
- Different ISAs and microarchitectures.

In the first class, every member of the family or every processor variant derived from the configurable processor has the same ISA. This means that each processor is capable of executing the same programs. The differences are located at the microarchitecture level, providing different trade-offs in terms of resource utilization, energy consumption, CPU time, etc.

A processor family belonging to this class uses the 32-bit ISA of Intel (IA-32), whose members are the Pentium, Celeron (lower cost), and Xeon (higher performance) processors [130]. A configurable processor belonging to this class is the Nios II soft-core processor, which is equipped with configuration parameters that mainly affect its microarchitecture [61].

In the second class, there is at least one processor developed based on a different ISA. In this case, each processor is not able to run the same programs given that instructions, instruction formats, or other properties related to the ISA are not compatible. The advantage of this class of processors is the potential to achieve a higher adaptation.

A processor family belonging to this class of processors is formed by extending the processor family based on the IA-32 with the processor family based on the IA-64. It was not possible to find a configurable processor with well-defined configuration parameters at the ISA level. At a first glance, it might seem that the Proteus configurable soft core processor [75] is equipped with configuration parameters at the ISA level. These options permit changing the data width, number of supported registers, and activating or deactivating some instructions. However, by looking at its configuration mechanism, it was found that these configuration parameters do not influence the instruction set. The instruction width and coding formats remain the same although it is possible to code registers using fewer bits.

4.4.2 Adaptation mechanism of the ROBSY processor

The ROBSY processor should have the highest possible level of adaptation. For this purpose, a first alternative is to develop a family of processors. However, the development and maintenance effort required to support a family of processors is very high, making this alternative inappropriate.

The development of a configurable processor represents a more suitable alternative. Configuration parameters represent a natural adaptation mechanism of soft-core processors. In comparison to an adaptation mechanism based on a family of processors, the development and maintenance of a single configurable processor is a more manageable task, given that it is only necessary to consider a unique source code base.

The challenge is to design a configurable processor with the level of configurability required for the ROBSY approach. For this purpose, the ROBSY processor should be equipped with configuration parameters at the ISA and microarchitecture level. This is a very interesting approach because it provides a very flexible processor with a high adaptability potential. Nevertheless, it should be considered that configuration parameters at the ISA level require configurable processor development tools (compiler, assembler, linker, instruction set simulator, etc.).

4.5 Summary

Chapter 4 presented the concept of the ROBSY processor, which is based on three main pillars:

- Analysis of general design aspects.
- Processor specialization based on the properties of L4-L2 and I2 and I3.
- Processor adaptation mechanism.

During the analysis of general design aspects, the functional and non-functional requirements that the ROBSY processor has to fulfill were presented. The functional requirements define the support for operations used in layers L2 to L4 and interfaces I2 and I3. The non-functional requirements highlight the adaptation, FPGA-based implementation, code portability, and the focus on the reduction of test time and an efficient utilization of resources.

At that point, it was established that none of the general soft-core or test processors available in the literature is suitable for ROBSY. After that, main design decisions were performed as part of the analysis of general design aspects. The decisions include the use of VHDL as the design language and inference for the adaptation to FPGAs from different families and vendors. Furthermore, the processor was defined as a scalar single-issue in-order RISC processor with pipelining.

The discussion about the processor specialization was carried out based on three main aspects:

- Tailoring the processor functionality for testing purposes.
- Defining standard and special test operations for layers L2 to L4.
- Specifying the support required for interfaces I2 and I3.

It was shown that unsigned fixed-point arithmetic and logic operations as well as standard load and store operations are sufficient for testing purposes. The processor data width was limited to a single configurable value, and operating system related functions do not have to be considered. Additionally, it was shown that there is no need for a hierarchical memory organization because embedded FPGA memory blocks are the only memory resources available for instructions and data storage.

The implementation of layers L2 to L4 in ESW requires equipping the ROBSY processor with standard arithmetic and logic instructions, such as addition, subtraction, comparison, shift, rotate, negation, and XOR. In the same way, the processor should support standard load and store instructions, which are necessary to transfer information between the

processor memory, I/O, and general purpose registers. For control flow operations, the processor should be equipped with conditional and unconditional branching, call, and return instructions. These instructions should be implemented using an absolute addressing mode and a conditional code register with the flags.

The ROBSY processor should provide the option to code a complete pattern in an instruction in order to efficiently store deterministic patterns in program memory. In the same way, it should be equipped with special LFSR instructions for the emulation of LFSRs and MISRs. Multiplication or division operations are not required.

For interfaces I2 and I3, the ROBSY processor should be equipped with a debug-interface and an on-chip bus system, respectively. The debug-interface communicates to the ATE based on DRs, and the on-chip bus system is implemented as a shared bus based on the Wishbone interconnection architecture.

Finally, it was shown that the proper adaptation mechanism of the ROBSY processor consists of the development of a highly flexible configurable processor. The configuration parameters located at the ISA and microarchitecture level provide high flexibility for the adaptation of the processor to the test scenario.

5 ROBSY processor

5.1 Introduction

Chapter 4 defined the ROBSY processor as a scalar single-issue in-order RISC processor with support for standard arithmetic, logic, control flow, and data transfers instructions, as well as test instructions for the emulation of LFSRs. RTL was chosen as the design abstraction level, VHDL as the design language, and configuration parameters at the ISA and microarchitecture level as the adaptation mechanism to different test scenarios. This chapter presents the processor implementation in more detail, with main focus on:

- Processor ISA.
- Processor microarchitecture.
- Debug-interface properties.
- Configuration parameters.

Sections 5.2 and 5.3 present the ISA and microarchitecture of the ROBSY processor, respectively. Section 5.4 presents the debug-interface, including the support for JTAG. Section 5.5 provides information about the configuration parameters. This chapter concludes with a summary in Section 5.6.

5.2 Instruction set architecture

The instruction set architecture (ISA) —also known as the processor architecture— represents the processor attributes that are visible to the programmer [142]. It defines the processor's behavior and properties that are visible from the programmer's point of view without revealing any implementation details. Therefore, it provides information about the supported data types, visible state (registers and memory), I/O, operations (instruction set and format), interrupts, and exceptions.

5.2.1 Native data types

As already discussed in Section 4.3.1.1, the support for unsigned fixed-point arithmetic is sufficient for testing purposes. The processor uses a fixed-point value with a unique width (*data_width*), which represents the width of a general purpose register (GPR). The *data_width* is also a configurable parameter of the processor, which influences the properties of its ISA and microarchitecture.

The *data_width* changes the instruction width because it defines the number of bits required to code an immediate value. An additional configuration parameter known as *short_imm_set* provides the option to code an immediate value using a smaller number of bits. The width of an immediate data value (*imm_data_width*) is described as follows:

$$imm\_data\_width = \begin{cases} data\_width, & when\ short\_imm\_set = false \\ \left\lceil \frac{data\_width}{2} \right\rceil, & when\ short\_imm\_set = true \end{cases} \quad (5.1)$$

If the size of variables found in the DUT-M differs from the processor *data_width*, it is the responsibility of the compiler to handle these variables using the appropriate instruction sequences.

5.2.2 Programmer visible state and I/O

5.2.2.1 Registers

The ISA is composed of a diverse set of registers: general purpose registers (GPRs), special function registers (SFRs), instruction register (IR), program counter (PC), and stack pointer (SP).

GPRs are used as the storage mechanism for operands and results of arithmetic, logic, test, and data transfer instructions. The number (*GPRs_num*) and width (*data_width*) of GPRs are configuration parameters. *GPRs_num* has the following restrictions:

$$GPRs\_num = |GPRs|, \quad with \begin{cases} |GPRs| > 2 \\ |GPRs| > \left\lceil \frac{data\_space\_depth}{data\_width} \right\rceil \end{cases} \quad (5.2)$$

Equation (5.2) shows that there are two restrictions for *GPRs_num*. The first restriction states that there should be at least three GPRs as part of the processor. This is necessary given that GPR R0 is hardwired to zero, and at least two additional GPRs are required for the execution of an arithmetic, logic, test, or data transfer instruction. The second restriction states that there should be enough GPRs for addressing the data address space of the processor (Section 5.2.2.2).

SFRs are used for specific functions. The latest processor version is equipped with seven SFRs (two of them are optional). The configuration parameter *SFRs_num* defines the number of SFRs, and it enables the inclusion of optional SFRs and addition of new SFRs for future use. In the current processor version, *SFRs_num* is at least five.

$$SFRs\_num = |SFRs|, \qquad with\ |SFRs| > 4 \tag{5.3}$$

The first SFR is the condition code register, which is composed of the zero, sign, carry, and overflow condition flags. The zero flag is set every time the result of an arithmetic, logic, or test instruction is zero. The sign flag is set every time the most significant bit (MSB) of an arithmetic, logic, or test instruction is one. The carry and overflow flags are properly set during the execution of arithmetic and test instructions. The second SFR is a shadow register of the condition code register. This shadow register is used for restoring the original state of the flags after the execution of an interrupt service routine. The restoration procedure is described in ESW and requires access to this SFR.

The third SFR is the exception code register. This register automatically stores a value that identifies the cause of an exception activated during normal program execution. The current processor version uses five exception bits (Section 5.2.4). The fourth and fifth SFRs are the interrupt mask and interrupt register. The interrupt mask is used to enable a specific interrupt. The interrupt register is automatically set every time an interrupt is triggered in order to identify the interrupt source (Section 5.2.4). Two optional SFRs are used to define an address range of the data memory reserved for communication with the ATE. Both registers are used in conjunction with the VarioTap test approach of Göpel electronics [45]. This test approach is not used in this dissertation, and therefore the two optional SFRs are not required.

The maximum width of an SFR is *data_width,* which enables the transfer of information between SFRs and GPRs in a single instruction. The minimum width of an SFR is five, which corresponds to the number of bits required to code the exception cause. Therefore, the SFRs impose the following restriction to the processor *data_width*:

$$data\_width > 4 \tag{5.4}$$

The IR is used to store the instruction fetched from program memory. Its width depends on the number of bits required to code an instruction, and is known as *instruction_width* (Section 5.2.3). The PC is used to store the address of the next instruction to fetch from program memory. Its width depends on the program memory depth, known as *prog_mem_depth* (Section 5.2.2.2). The SP is used to point to the last value stored in the stack. The width of the SP depends on the stack memory depth, which is known as *stack_mem_depth* (Section 5.2.2.2). The SP is managed by the processor, incrementing or decrementing it based on push, pop, and procedure call instructions.

Figure 2.15 shows a graphical representation of the GPRs, SFRs, IR, PC, and SP.

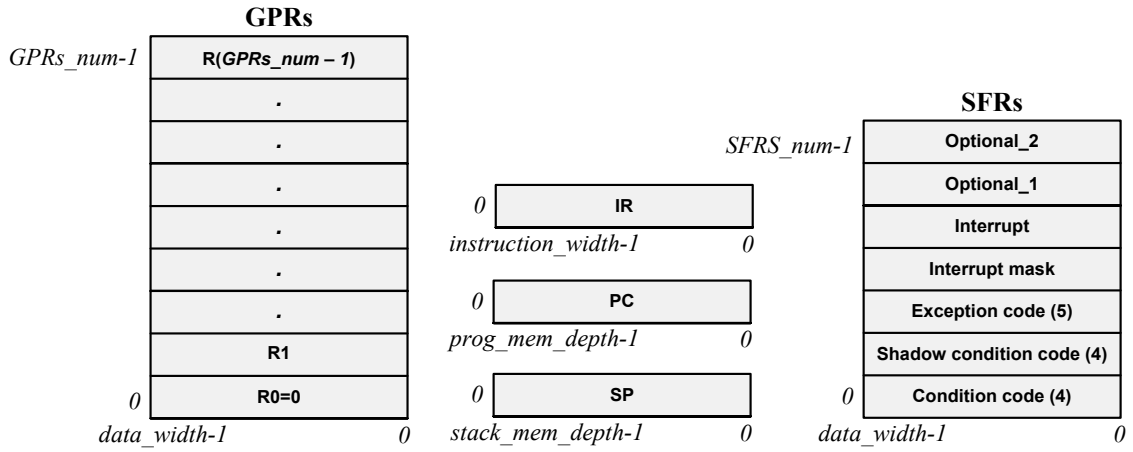


Figure 5.1: ROBSY processor registers

5.2.2.2 Address spaces

The processor has three independent address spaces, which are known as the program, data, and stack address spaces.

The program address space represents the program memory, and is defined by the configuration parameter *prog_mem_depth*. *prog_mem_depth* should provide enough capacity to store all instructions constituting the program. Therefore, it is computed as presented in Equation (5.5). The program memory width (*instruction_width*) is equal to the number of bits required to code an instruction (Section 5.2.3).

$$prog\_mem\_depth = \lceil \log_2(|program\ instructions|) \rceil \tag{5.5}$$

The data address space arranges data memory, SFRs, and I/O in a single address space. In this way, the same load and store instructions are used to access variables stored in data memory, SFR values, and I/O. The width of the data space is *data_width* and its depth is *data_space_depth*. The latter is calculated based on the depth of the data memory (*data_mem_depth)*, *SFRs_num*, and the number of I/O addresses (*IO_addr_num*) (Equation (5.8)). *data_mem_depth* and *IO_addr_num* are configurable, making it possible to change the data storage capacity and the number of I/O addresses of the processor.

$$data\_mem\_depth = \lceil \log_2(|program\ variables|) \rceil \tag{5.6}$$

$$IO\_addr\_num = |co - processor\ addresses| \tag{5.7}$$

$$data\_space\_depth = \lceil \log_2(2^{data\_mem\_depth} + SFRS\_num + IO\_addr\_num) \rceil \tag{5.8}$$

The stack address space represents the stack memory. The depth of the stack memory (*stack_mem_depth*) is a configurable parameter, providing the option to change the storage capacity of the stack. The stack memory width (*stack_width*) depends on *data_width* and *prog_mem_depth*, given that it should be possible to store a program address or a data value. The computation of *stack_mem_depth* and *stack_width* is performed as follows:

$$stack\_mem\_depth = \lceil \log_2(|stack\ locations|) \rceil \tag{5.9}$$

$$stack\_width = \max\begin{pmatrix} data\_width \\ prog\_mem\_depth \end{pmatrix} \tag{5.10}$$

Figure 5.2 shows a graphical representation of the three address spaces with their corresponding depth and width.

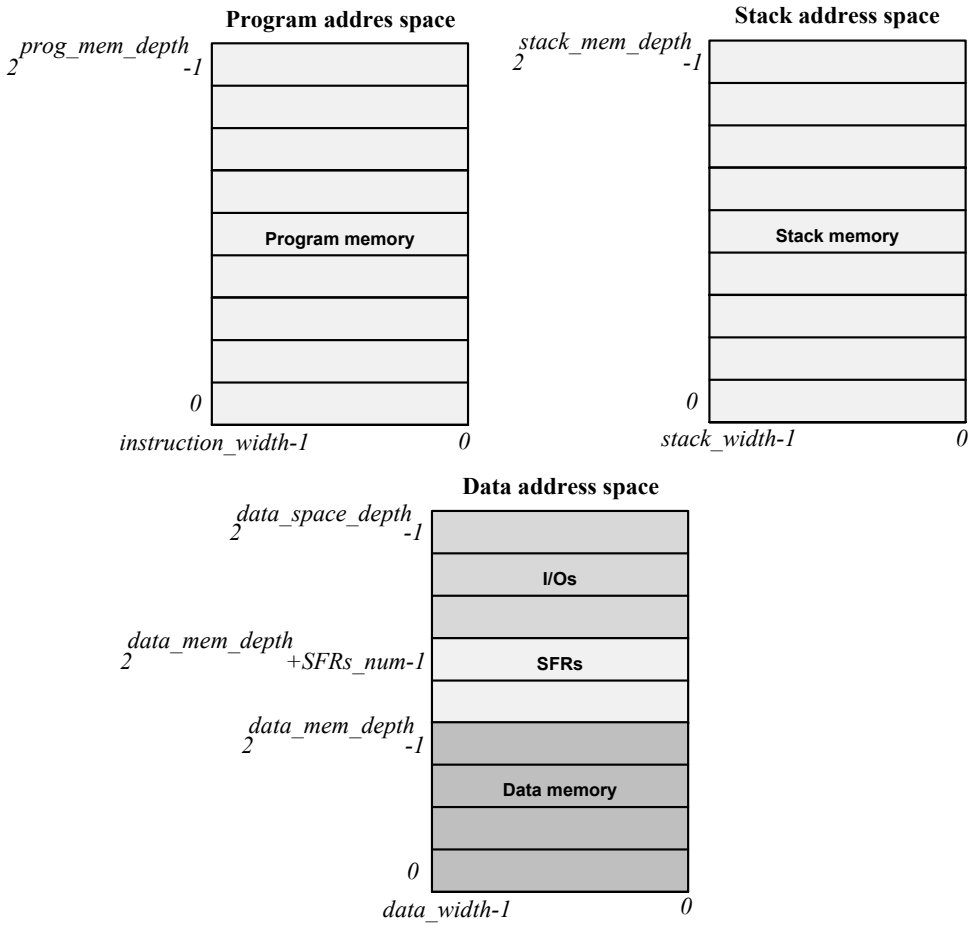


Figure 5.2: ROBSY processor address spaces

The main reason to support three independent address spaces is the high configurability of the ROBSY processor, which makes it possible to obtain processor variants with different values for *instruction_width*, *stack_width*, and *data_width*.

The program address space is addressed by the PC. The value of the PC is incremented automatically or changes due to call, return, and conditional or unconditional branching instructions. The stack address space is addressed by the SP. The value of the SP changes based on push, pop, and procedure call instructions.

The data address space is addressed by GPRs. It requires special attention because it is not possible to address the complete data address space based on a single GPR or immediate value if *data_space_depth* is greater than *data_width*. In order to deal with this problem, the ROBSY processor concatenates GPRs to form the most significant bits (MSBs) of the address. These GPRs are known as page pointers, and the number of page pointers (*page_pointers_num*) is defined as follows:

$$page\_pointers\_num = |page\ pointers| = \left\lceil \frac{data\_space\_depth}{data\_width} - 1 \right\rceil \qquad (5.11)$$

Page pointers are concatenated in such a way that GPR R1 corresponds to the MSBs of the address, followed by R2, R3, and so on.

5.2.3 Instruction set

Table 4.1 shows the instructions of the ROBSY processor. These instructions are divided into two main classes: class-0 and class-1. Class-0 instructions are conditional and unconditional branches, procedure calls, data transfers, and push/pop instructions. Class-1 instructions are arithmetic, logic, and test instructions. The type and number of operands change depending on the specific instruction.

Class	Instructions	Operands
0	**JMP, JC, JNC, JZ, JNZ, JS, JNS, JO, JNO**	Immediate address value
	CALL, CALL_C, RET, RET_C	Immediate address value, -
	STOP, NOP	-
	LOAD, STORE	GPRs + immediate data value
	PUSH, POP	GPR
1	**ADD, SUB**	GPRs + immediate data value
	AND, OR, XOR, NOR	GPRs + immediate data value
	SHR, SHL, ROTR, ROTL	GPRs + immediate data value
	LFSR	GPRs + immediate data value

Table 5.1: ROBSY processor instructions

The processor has variations of the same instruction. This is necessary in order to support different operands (GPR-GPR or GPR-immediate), the *short_imm_set* configuration parameter, and the execution of arithmetic, logic, and test instructions for variables larger

than *data_width*. The processor has 50 instructions in total, 18 of which are class-0 and 32 are class-1 instructions.

5.2.3.1 Class-0 instructions

Table 5.2 shows the class-0 branching instructions (Section 4.3.2.3). The argument of these instructions is an immediate value (*imm_address_value*), which represents the target address of the branch. The size of *imm_address_value* is *prog_mem_depth* in order to support absolute addressing mode. Table 5.2 shows the branching instructions, in which the first column shows the instruction name, the second column shows the operation performed, and the last four columns shows the condition code register flags that are affected. C stands for carry, Z for zero, S for sign, and O for overflow.

Instruction	Description	C	Z	S	O
JMP	PC ← *imm_address_value*				
JC	PC ← *imm_address_value* if C is set				
JNC	PC ← *imm_address_value* if C is unset				
JZ	PC ← *imm_address_value* if Z is set				
JNZ	PC ← *imm_address_value* if Z is unset				
JS	PC ← *imm_address_value* if S is set				
JNS	PC ← *imm_address_value* if S is unset				
JO	PC ← *imm_address_value* if O is set				
JNO	PC ← *imm_address_value* if O is unset				

Table 5.2: Branching instructions

Table 5.3 shows class-0 instructions used for procedures. CALL and CALL_C branch to the start address of the procedure and store the return address (*return_address*) in the stack. The *imm_address_value* coded in the instruction defines the start address of the procedure. If CALL_C is executed, the processor does not respond to interrupt requests during the execution of the procedure. CALL_C allows the execution of critical portions of code, such as interrupt and exception service routines. RET and RET_C are used to exit the procedure and resume normal program execution. For this purpose, the processor pops the return address from the stack and performs a branch. RET_C is used to reactivate interrupts. It allows using CALL instructions inside a CALL_C subroutine without reactivating the interrupts.

Instruction	Description	C	Z	S	O
CALL	PC ← *imm_address_value*; stack[SP+1] ← *return_address*				
CALL_C	PC ← *imm_address_value*; stack[SP+1] ← *return_address*; disable interrupts				
RET	PC ← stack[SP]				
RET_C	PC ← stack[SP]; enable interrupts				

Table 5.3: Procedure call instructions

Table 5.4 shows the STOP and NOP instructions. STOP halts program execution by deactivating the PC. The only way to continue program execution is by means of the debug-interface (Section 5.4). NOP does not execute any operation. It increments the PC value without changing the state of any other register or memory location.

Instruction	Description	C	Z	S	O
STOP	PC ← PC				
NOP	PC ← PC + 1				

Table 5.4: STOP and NOP instructions

Table 5.5 shows the LOAD and STORE instructions, which perform data transfers between GPRs and the processor data space. Both instructions use a page pointer and two operands to describe the address of the data address space. The two operands are the GPR R_{SI} (register source 1) and the immediate data value (*imm_data_value*). R_{DS} (register destination source) is the GPR used as destination for load instructions or source for store instructions. The addressing mode is base plus displacement. It uses the concatenation of page pointers and R_{S1} (*page_pointers*.R_{SI}) as the base address and the *imm_data_value* as a constant displacement. This addressing mode makes it possible to implement the register indirect addressing mode (*imm_data_value* is zero) and the immediate indirect addressing mode (R_{SI} is R0), and allows efficient handling of composite data types.

Instruction	Description	C	Z	S	O
LOAD	R_{DS} ←data_address_space[*page_pointers*.R_{SI} + *imm_data_value*]				
STORE	data_address_space[*page_pointers*.R_{SI} + *imm_data_value*] ← R_{DS}				

Table 5.5: LOAD and STORE instructions

The following example clarifies the use of page pointers. A ROBSY processor variant with *data_space_depth* equal to 20 and *data_width* equal to 8 requires a page pointer with two GPRs *R1* and *R2* (Equation (5.11)). *R1* corresponds to address bits 19-16, *R2* to address bits 15-8, and R_{SI} to address bits 7-0. The *imm_data_value* is added to the address formed by the page pointer and R_{S1} in order to obtain the effective address value.

Table 5.6 shows the class-0 PUSH and POP instructions. They push or pop the content of a GPR into the stack, using R_{DS} as source or destination. The SP is incremented due to a PUSH instruction, and decremented due to a POP instruction.

Instruction	Description	C	Z	S	O
PUSH	Stack_address_space[SP+1] ← R_{DS}, SP ←SP+1				
POP	R_{DS} ←stack_address_space[SP] , SP ←SP-1				

Table 5.6: PUSH and POP instructions

5.2.3.2 Class-1 instructions

Class-1 instructions are arithmetic, logic, and test instructions. For the execution of class-1 instructions, the ROBSY processor operates as a three address machine. This means that each class-1 instruction has one result and two operands. The result is always a GPR represented by R_{DS}, while the two operands are defined as a pair of GPRs (R_{S1} and R_{S2}) or as a GPR and an immediate value (R_{S1} and *imm_data_value*).

Table 5.7 presents the class-1 addition and subtraction instructions. The processor supports six instruction variants for addition and subtraction. ADD and SUB use R_{S1} and R_{S2}, whereas ADDI and SUBI use R_{S1} and *imm_data_value*. If the processor configuration parameter *short_imm_set* is true, the *imm_data_value* of ADDI and SUBI is sign extended. ADDIU and SUBIU are used if no sign extension is required. The instructions with an ending C use the carry flag as a third operand. These instructions are necessary to perform additions or subtractions for variables larger than *data_width*.

Instruction	Description	C	Z	S	O
ADD	$R_{DS} \leftarrow R_{S1} + R_{S2}$	X	X	X	X
ADDI	$R_{DS} \leftarrow R_{S1}$ + *imm_data_value* (sign extended)	X	X	X	X
ADDIU	$R_{DS} \leftarrow R_{S1}$ + *imm_data_value* (zero extended)	X	X	X	X
ADDC	$R_{DS} \leftarrow R_{S1} + R_{S2}$ + C	X	X	X	X
ADDIC	$R_{DS} \leftarrow R_{S1}$ + *imm_data_value* + C (sign extended)	X	X	X	X
ADDIUC	$R_{DS} \leftarrow R_{S1}$ + *imm_data_value* + C (zero extended)	X	X	X	X
SUB	$R_{DS} \leftarrow R_{S1} - R_{S2}$	X	X	X	X
SUBI	$R_{DS} \leftarrow R_{S1}$ - *imm_data_value* (sign extended)	X	X	X	X
SUBIU	$R_{DS} \leftarrow R_{S1}$ - *imm_data_value* (zero extended)	X	X	X	X
SUBC	$R_{DS} \leftarrow R_{S1} - R_{S2}$ - C	X	X	X	X
SUBIC	$R_{DS} \leftarrow R_{S1}$ - *imm_data_value* - C (sign extended)	X	X	X	X
SUBIUC	$R_{DS} \leftarrow R_{S1}$ - *imm_data_value* - C (zero extended)	X	X	X	X

Table 5.7: ADD and SUB instructions

Table 5.8 shows the class-1 logic instructions. The processor supports two instruction variants for logic operations. An instruction variant uses two GPRs for the source operands, whereas the other instruction variant uses a GPR and an *imm_data_value*. The last four columns of Table 5.8 show that the carry and overflow flag are always set to zero, while the zero and sign flags are set depending on the result.

The transfer of data between GPRs or between a GPR and an immediate data value is performed using OR and ORI instructions and R0 as R_{S1}. Negation operations are performed using NOR instructions and R0 as R_{S1}. If the processor configuration parameter *short_imm_set* is true, the *imm_data_value* is zero extended, and ORH with R0 as R_{S1} is used to move an *imm_data_value* to the MSBs of R_{DS}.

Instruction	Description	C	Z	S	O
AND	$R_{DS} \leftarrow R_{S1}$ & R_{S2}	0	X	X	0
ANDI	$R_{DS} \leftarrow R_{S1}$ & *imm_data_value* (zero extended)	0	X	X	0
OR	$R_{DS} \leftarrow R_{S1} \mid R_{S2}$	0	X	X	0
ORI	$R_{DS} \leftarrow R_{S1} \mid$ *imm_data_value* (zero extended)	0	X	X	0
ORH	$R_{DS} \leftarrow R_{S1} \mid$ *imm_data_value* (MSBs)	0	X	X	0
XOR	$R_{DS} \leftarrow R_{S1} \oplus R_{S2}$	0	X	X	0
XORI	$R_{DS} \leftarrow R_{S1} \oplus$ *imm_data_value* (zero extended)	0	X	X	0
NOR	$R_{DS} \leftarrow \sim(R_{S1} \mid R_{S2})$	0	X	X	0
NORI	$R_{DS} \leftarrow \sim(R_{S1} \mid$ *imm_data_value*) (zero extended)	0	X	X	0

Table 5.8: Logic instructions

Table 5.9 shows the class-1 shift and rotation instructions. Every instruction uses R_{S1} as operand and *imm_data_value* to define the number of bits to shift or rotate the content of the GPR represented by R_{S1}. SHL and SHR perform logical shift operations with zeros replacing the discarded bits. SHLC and SHRC use the carry flag to replace discarded bits. SHRA performs an arithmetic shift, using the sign of R_{S1} to replace discarded bits. ROL and ROR perform circular shifts or bit rotations. ROLC and RORC consider the carry flag as part of the rotation operand.

Instruction	Description	C	Z	S	O
SHL	$R_{DS} \leftarrow R_{S1} <<$ *imm_data_value* (0-fill)	X	X	X	X
SHLC	$R_{DS} \leftarrow R_{S1} <<$ *imm_data_value* (C-fill)	X	X	X	X
SHR	$R_{DS} \leftarrow R_{S1} >>$ *imm_data_value* (0-fill)	X	X	X	X
SHRC	$R_{DS} \leftarrow R_{S1} >>$ *imm_data_value* (C-fill)	X	X	X	X
SHRA	$R_{DS} \leftarrow R_{S1} >>$ *imm_data_value* (sign-fill)	X	X	X	X
ROL	$R_{DS} \leftarrow$ rol(R_{S1}, *imm_data_value*)	X	X	X	X
ROLC	$R_{DS} \leftarrow$rol(R_{S1}.C, *imm_data_value*)	X	X	X	X
ROR	$R_{DS} \leftarrow$ ror(R_{S1}, *imm_data_value*)	X	X	X	X
RORC	$R_{DS} \leftarrow$ ror(R_{S1}.C, *imm_data_value*)	X	X	X	X

Table 5.9: Shift and rotate instructions

Table 5.10 shows the class-1 instructions used to emulate an LFSR. Every instruction uses R_{S1} for the implementation of the LFSR state, and R_{S2} to define the feedback polynomial. The LFSR is emulated using internal XORs between each state flip-flop.

Instruction	Description	C	Z	S	O
LFSR	$R_{DS} \leftarrow$lfsr(R_{S1}, R_{S2})	X	X	X	X
LFSRL	$R_{DS} \leftarrow$ lfsr(R_{S1}, R_{S2}) (C-concatenation)	0	X	X	X
LFSRH	$R_{DS} \leftarrow$ lfsr(R_{S1}, R_{S2}) (C-concatenation, O-feedback)	X	X	X	

Table 5.10: Test instructions

The LFSR instruction emulates a LFSR of *data_width* bits. For example, the LFSR in Figure 5.3 with feedback polynomial of Equation (5.12) is emulated by a processor with

data_width equal to 8, R_{S1} equal to one (seed), and R_{S2} equal to 0xB8 (polynomial). Table 5.11 shows the sequence of values obtained by executing the instruction five times.

$$x^8 + x^6 + x^5 + x^4 + 1 \tag{5.12}$$

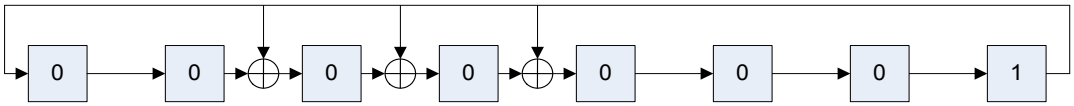


Figure 5.3: 8-bit LFSR with 0xB8 feedback polynomial

Pattern number	Pattern value
1	00000001
2	10111000
3	01011100
4	00101110
5	00010111
6	10110011

Table 5.11: Sequence of patterns 8-bit LFSR with 0xB8 feedback polynomial

LFSRL and LFSRH emulate an LFSR wider than *data_width*. For this purpose, LFSRL computes the *data_width* LSBs of the LFSR, and LFSRH computes the MSBs of the LFSR. The feedback and concatenation between registers is carried out through the carry and overflow flags. LFSRL uses the carry flag as input for concatenation, and it sets the overflow flag to the LSB of R_{S1} (feedback bit) and the carry flag to 0. LFSRH uses the overflow flag as feedback input value and the carry flag as concatenation input value. It sets the carry with the LSB of R_{S1} (concatenation bit) and leaves the overflow flag unchanged.

$$x^{24} + x^{12} + x^{10} + x^9 + x^6 + x^5 + x^3 + x^1 + 1 \tag{5.13}$$

```
1:   ; initialize GPRs
2:   OR R1, R0, 0x01; LFSR state bits 7-0
3:   OR R2, R0, 0x00; LFSR state bits 15-8
4:   OR R3, R0, 0x00; LFSR state bits 23-16
5:   OR R4, R0, 0x35; LFSR feedback polynomial bits 7-0
6:   OR R5, R0, 0x0B; LFSR feedback polynomial bits 15-8
7:   OR R6, R0, 0x80; LFSR feedback polynomial bits 23-16
8:   ; compute first value
9:   SHR R0, R2, 0x01; initialize carry with LFSR state bit 8
10:  LFSRL R1, R1, R4; compute LFSR state bits 7-0. Set carry an overflow
11:  LFSRH R3, R3, R6; compute LFSR state bits 23-16. Set carry
12:  LFSRH R2, R2, R5; compute LFSR state bits 15-8. Set carry
```

Figure 5.4: Code for LFSR with 0x800B35 feedback polynomial and 8-bit data_width

A 24-bit LFSR with the feedback polynomial presented in Equation (5.13) is emulated by a processor with *data_width* equal to eight using one LFSRL instruction and two LFSRH instructions. In this case, it is necessary to use three GPRs to represent the feedback polynomial (R_{S2}), and other three GPRs to represent the LFSR state (R_{S1}). Figure 5.4 shows the corresponding assembly code with R4-R6 representing the feedback polynomial, R1-R3 the LFSR state, and a seed equal to 0x000001.

5.2.3.3 Instruction formats

All instructions of the ROBSY processor are coded using the same length and a uniform format. The reason for this is that fetching and decoding fixed-length instructions is a more efficient task in comparison to fetching and decoding variable-length instructions.

The MSB of each instruction is used to code the instruction class and is followed by a fixed number of bits that represent the instruction command. The command width (*command_width*) varies depending on the processor configuration given that each instruction can be enabled or disabled. Equation (5.14) presents the formal definition of the *command_width*. The number of enabled instructions belonging to class-0 and class-1 are represented by *instrs_enable_c0* and *instrs_enable_c1*.

$$command\_width = \left\lceil \log_2\left(max(|instrs\_enable\_c0|, |instrs\_enable\_c1|)\right)\right\rceil \quad (5.14)$$

The remaining instruction bits are used to code the operands of each instruction. Figure 5.5 shows the four resulting instruction formats. The bits marked with P represent optional padding bits, which might be necessary depending on the processor configuration. The instruction class CL requires one bit. The c*ommand_width* is calculated based on Equation (5.14), *prog_mem_depth* based on Equation (5.5), and *imm_data_width* based on Equation (5.1).

The instruction format *f1* is used for branch and call instructions (Table 5.2 and Table 5.3). The operand is the program memory address *imm_address_value*. The instruction format *f2* is used to code procedure return as well as STOP and NOP instructions (Table 5.3 and Table 5.4). These instructions do not use operands.

The instruction formats *f3* and *f4* are used for arithmetic, logic, test, PUSH/POP, and LOAD/STORE instructions (Table 5.5, Table 5.6, Table 5.7, Table 5.8, Table 5.9, and Table 5.10). *f3* uses three operands in order to code the address of the three GPRs: R_{DS}, R_{S1}, and R_{S2}. The PUSH and POP instructions use just one of the three arguments. R_{S1} is used for PUSH and R_{DS} for POP. In both cases, the GPR R0 is used for the remaining operands. The instruction format *f4* also uses three operands. Two of them are used to

code GPRs and the other one is used to code an immediate value. The operands are R_{DS}, R_{S1}, and *imm_data_value*. For LOAD/STORE instructions, *imm_data_value* represents the address displacement.

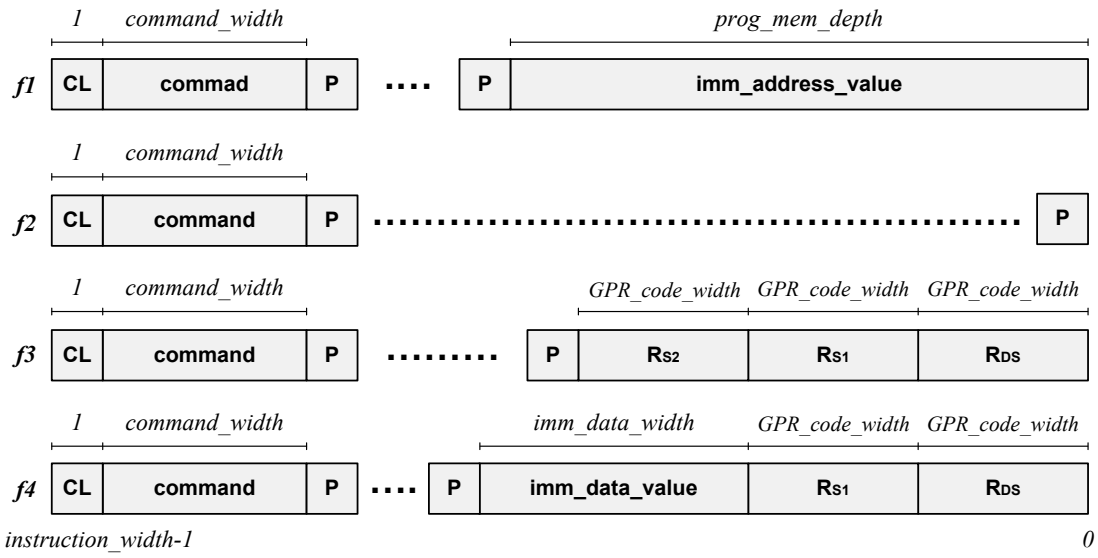


Figure 5.5: Instruction formats

GPR_code_width is the number of bits required to code a GPR address. It is calculated as follows:

$$GPR\_code\_width = \lceil \log_2(GPRs\_num) \rceil \tag{5.15}$$

The *instruction_width* is calculated depending on the maximum width of the operands. This value is known as *max_op_width* and is calculated based on the arguments of *f1*, *f3*, and *f4*. Equations (5.16) and (5.17) show the formulas.

$$max\_op\_width = max\begin{pmatrix} prog\_mem\_depth, \\ 3 \cdot GPR\_code\_width, \\ 2 \cdot GPR\_code\_width + imm\_data\_width \end{pmatrix} \tag{5.16}$$

$$instruction\_width = 1 + command\_width + max\_op\_width \tag{5.17}$$

5.2.4 Interrupts and exceptions

Interrupts and exceptions are unexpected events that required the disruption of the normal flow of program execution. In the ROBSY processor, an interrupt is defined as an unexpected event from outside the processor, while an exception is defined as an

unexpected event from within the processor. An exception is triggered due to an anomalous condition that appears during program execution.

Although interrupts are not considered for the ROBSY approach, they are included for future use. When an interrupt is triggered, the ROBSY processor suspends normal program execution and starts the interrupt service routine (ISR). Nested interrupts are not supported. If an exception is activated the processor automatically stores a code representing the exception cause in the exception code SFR. Depending on the type of exception, it stops program execution or starts the exception service routine (ESR). There are two program memory addresses reserved for the ISR and ESR, which are defined as follows:

$$ISR\_address = 2^{prog\_mem\_depth} - 2 \tag{5.18}$$

$$ESR\_address = 2^{prog\_mem\_depth} - 1 \tag{5.19}$$

The number of interrupts (*interrupts_num*) is configurable but limited by the processor *data_width*. This limitation is necessary in order to use a single SFR for the implementation of the interrupt register and mask. It is the job of the ESW to identify the triggered interrupt. The value of *interrupts_num* is:

$$interrupts\_num = |interrupts|, \qquad with\ |interrupts| \leq data\_width \tag{5.20}$$

The ROBSY processor is able to identify five different exception causes. The exception code SFR is five bits wide. Three bits are used to code the exception, while two additional bits are used to signal if the exception is currently active and if the ESR is being executed. Table 5.12 presents the exception cause and the corresponding exception code. The first four exceptions are treated as fatal errors, causing the processor to stop execution. The I/O transaction error causes the execution of the ESR.

Cause	Exception code
Normal operation	000
Stack overflow	001
Stack underflow	010
Undefined class 0 instruction	011
Undefined class 1 instruction	100
I/O transaction error	101

Table 5.12: Exception codes

5.3 Microarchitecture

The microarchitecture —also known as the processor organization— represents the way a given ISA is implemented. Microarchitecture attributes include those hardware details that are transparent to the programmer, such as operational units and interconnections [142]. This section presents the main features of the microarchitecture of the ROBSY processor considering the configuration parameters.

5.3.1 Top level view

The ROBSY processor is classified as a single instruction single data (SISD) processor with separate address spaces (Harvard machine). Figure 5.6 shows the top level view together with the debug-interface, I/O bus, and co-processors.

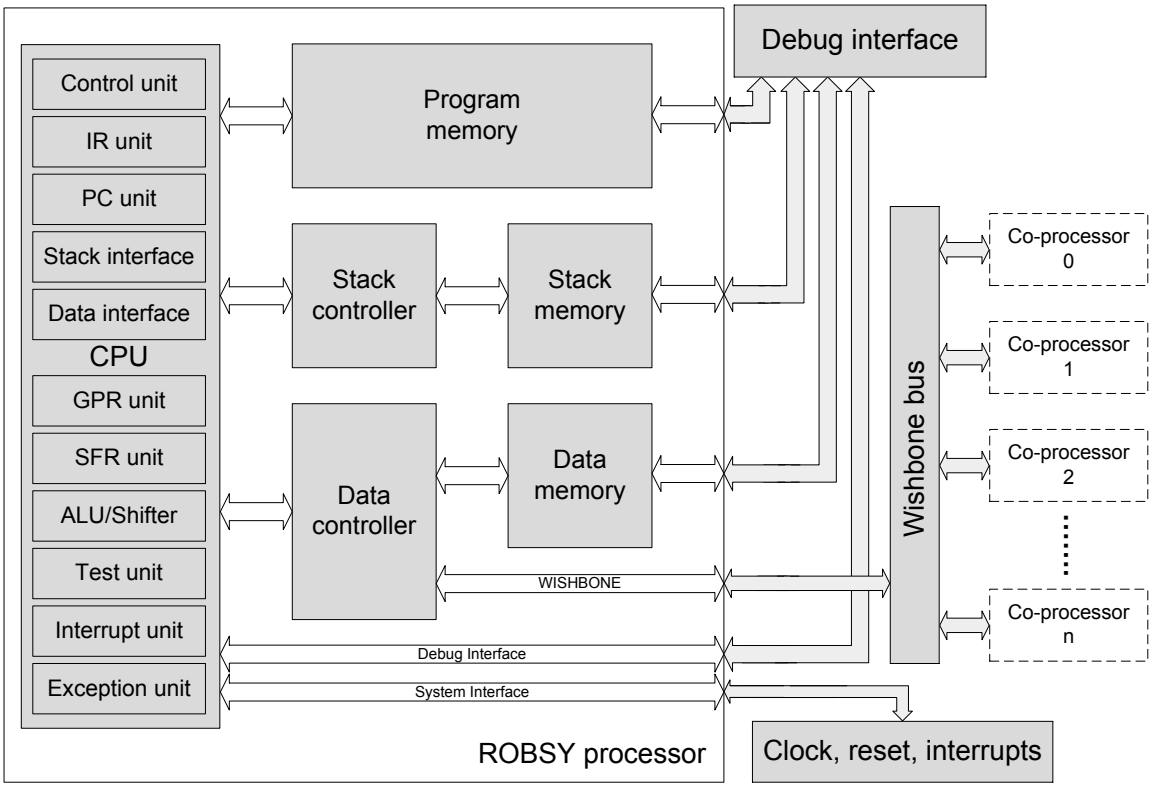


Figure 5.6: ROBSY processor top level view

The processor is composed of six main modules, three physically separate memories (program, stack, and data memories), a stack controller, a data controller, and a central processing unit (CPU). It has a single clock and reset port, and an interrupt input port. The I/O bus is the physical link between the processor and co-processors, and is implemented based on the Wishbone standard (Section 4.3.3.2). The debug-interface has access to the processor memories and the CPU through dedicated ports. Section 5.4 presents detailed information about the debug-interface.

5.3.2 Program, data, and stack memories

The ROBSY processor has a program memory to store instructions, a data memory to store data such as variables, and a stack memory to store data and return addresses. These memories have different sizes, and therefore are physically separate in order to store a value in a single address location using a single clock cycle. Otherwise, multiple memory accesses would be necessary to access them, resulting in longer processing time and a higher utilization of resources. The size of the stack memory is defined based on *stack_mem_depth* and *stack_width* (Equations (5.9) and (5.10)), the size of the program memory based on *prog_mem_depth* and *instruction_width* (Equations (5.5) and (5.17)), and the size of the data memory based on *data_width* and *data_mem_depth* (Equations (5.4) and (5.6)).

They are implemented using the synchronous dual port memory blocks found in the FPGA. One port is connected to the processor, and the other one is connected to the debug-interface. In this way, it is possible to access the memory content through the debug-interface without stopping the processor execution. The memory blocks are inferred based on a generic VHDL description [129], which allows using the same code for FPGAs of different families and vendors.

5.3.3 Data controller

The data controller is in charge of multiplexing the access to the Wishbone bus and data memory, depending on the effective address sent by the CPU through the data interface unit. In the case that the address corresponds to a data memory location, the CPU signals are forwarded to data memory. Otherwise, the Wishbone bus starts an I/O transaction. The width of the Wishbone data and address buses is *data_width* and *data_space_depth*, respectively.

The data controller implements the Wishbone protocol by means of the finite state machine (FSM) presented in Figure 5.7. During an I/O transaction, the state machine leaves the *idle* state, performing the state transitions represented by the green arrows. If an I/O transaction error occurs, the state machine reaches the error state (red arrows). In the error state, the data controller signalizes the CPU of an I/O transaction error, causing the triggering of an exception. The implementation of the I/O bus protocol supports a configurable number of wait cycles (*wait* state), a single master as part of the on-chip bus, and single write and read transactions.

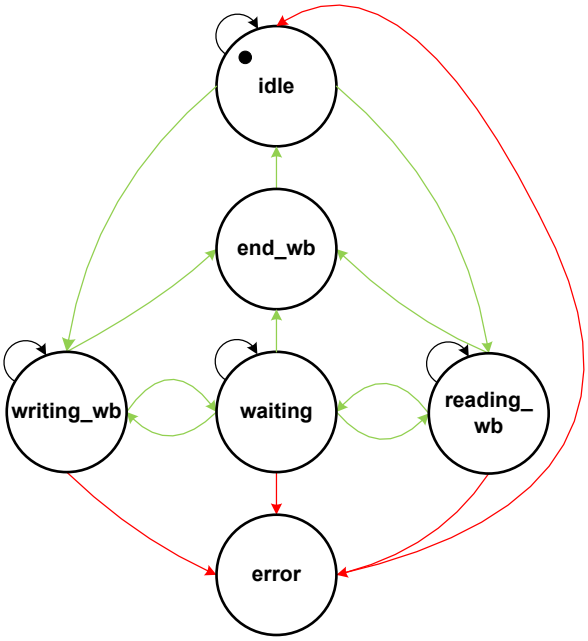


Figure 5.7: Data controller finite state machine

5.3.4 Stack controller

The stack controller manages the access to the stack memory. It is in charge of accessing the stack, setting the empty and full status flags based on the status of the stack memory, and incrementing or decrementing the stack pointer (SP). The SP points out to the memory address in which the last value was stored. In this way, it is possible to read the stack during a POP or return instruction in a single clock cycle.

The stack controller is a state machine that handles the status flags and errors caused by pushing to a full stack (stack overflow) or popping from an empty stack (stack underflow). In the case of an error, the stack controller signalizes the error to the CPU. Figure 5.8 shows the FSM.

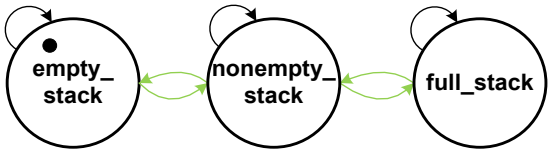


Figure 5.8: Stack controller finite state machine

5.3.5 Central processing unit

The CPU represents the core of the processor. It is composed of 11 units: control, IR, PC, GPR, SFR, test, interrupt, exception, ALU/Shifter, stack interface, and data interface units. Additionally, it can be configured as a multicycle or three stage pipeline unit.

5.3.5.1 Pipelining

If the configuration parameter *pipeline_set* is set to true, the ROBSY processor is implemented as a pipelined processor. Otherwise, it is implemented as a multicycle processor. The multicycle processor executes each instruction one at a time using multiple clock cycles. On the other hand, the pipelined processor is equipped with a three stage pipeline that allows processing three instructions at the same time. The three stages correspond to instruction fetch (IF), instruction decode (ID), and execute (EX). Additionally, the processor supports operand forwarding multiplexers that avoid stalling the pipeline due to data dependencies. Figure 5.9 presents a simplified illustration of the three pipeline stages.

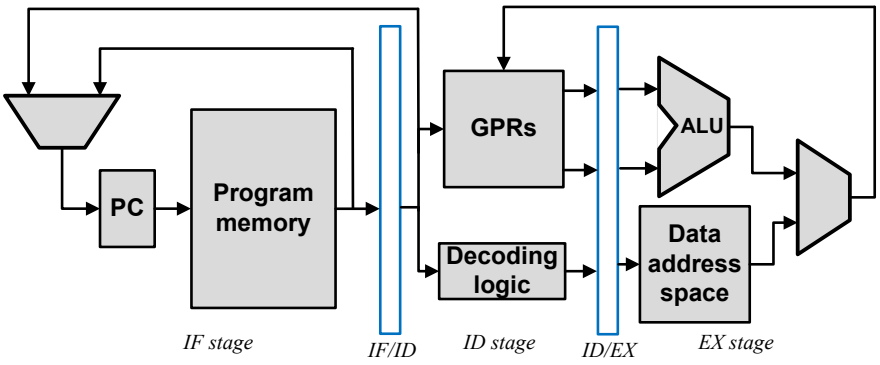


Figure 5.9: Three-stage pipelined processor

A memory access stage was not implemented in the ROBSY processor. The reasons for this are the absence of a data cache and the use of a small dedicated arithmetic module that computes effective addresses during the EX stage. This module is located in the data interface unit. In the same way, a write back stage was not included in the processor. Experiments were carried out with a three stage (no write back) and four stage (with write back) ROBSY processor implementations [143]. It was shown that the four stage processor requires approximately 10% more resources in comparison to the three stage processor. The additional resources and the design complexity of a four stage pipeline were the determinant factors for the implementation of a three-stage pipelined processor.

5.3.5.2 Control unit

The CPU control unit is in charge of controlling all other units. It is implemented using a FSM, which operates based on the instruction being processed, interrupt and exception triggers, and control signals coming from the debug-interface.

The implementation of the FSM depends on *pipeline_set* because the multicycle and pipeline processors require different FSMs. Figure 5.10 presents the state diagrams of both FSMs, in which the black arrows are state transitions performed during normal

instruction execution flow. Blue arrows correspond to state transitions performed in case of an interrupt or exception trigger, and red arrows correspond to state transitions performed in case of a fatal error (Section 5.2.4).

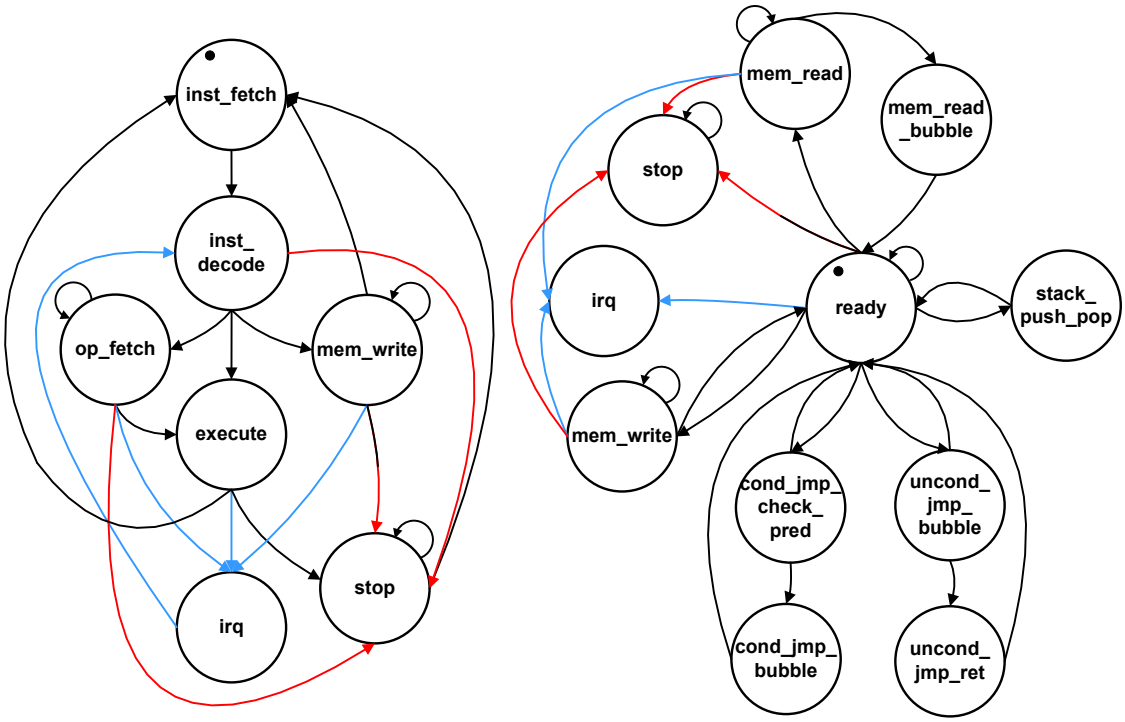


Figure 5.10: FSM of multicycle (left) and pipeline (right) processor

The FSM of the multicycle processor (Figure 5.10 left) has seven states. The five states that describe the normal instruction execution flow are instruction fetch (*inst_fetch*), instruction decode (*inst_decode*), operand fetch (*op_fetch*), memory write (*mem_write*), and execute (*execute*). *inst_fetch* and *inst_decode* represent the instruction fetch and decode actions. Transitions to *op_fetch* or *mem_write* are performed during the execution of a LOAD or STORE instruction, respectively. The *execute* state is reached during the execution of any other instruction (e.g. arithmetic, logic, and test instructions), or after an operand fetch (LOAD instruction). The interrupt request state (*irq*) is used to start the execution of an ISR or ESR. In the case of an interrupt, the transition to the *irq* state takes place after the execution of the actual instruction. A transition to the *stop* state takes place in case of a fatal error, a STOP instruction, or a stop signal from the debug-interface.

The FSM of the pipelined processor (Figure 5.10 right) has eleven states. The processor remains in the *ready* state during the execution of arithmetic, logic, and test instructions. The state machine leaves the *ready* state in order to attend interrupt or exception requests, stop the processor, or insert bubbles (delays) in the pipeline. The latter might be necessary during the execution of branching instructions, procedure calls, LOAD, STORE, PUSH,

and POP instructions. The transition to *irq* and *stop* states works in a similar way as in the FSM of the multicycle processor.

5.3.5.3 Program counter and instruction register units

The program counter unit (PC unit) fetches an instruction from program memory. It computes the address of the next instruction to fetch either by incrementing the PC or by updating the PC with an absolute address value. The absolute address value is obtained from a branch or procedure call instruction.

The instruction register unit (IR unit) stores the instruction or instructions (pipeline implementation) that are being executed. In the case of the pipeline processor, there is an IR for every pipeline stage. When an interrupt or exception is triggered, the IR unit is responsible of loading a CALL_C in the IR with the program address of the service routine. This is performed when the state machine of the control unit reaches the *irq* state.

5.3.5.4 Stack and data interfaces

The stack and data interfaces implement the link between the CPU and the stack and data controllers. They are used to start a data transfer to/from the stack memory, SFRs, data memory, or I/O.

The stack interface is in charge of managing data transfers to/from stack memory. For this purpose, a mechanism based on push and pop control signals is implemented between the stack interface and the stack controller. The data interface computes the effective data space address based on the page pointers, the base register, and the displacement. Depending on the region pointed out by the address, the data interface starts an external data transfer through the data controller unit or an internal data transfer with the SFR unit.

5.3.5.5 GPR and SFR units

GPRs and SFRs are found in the GPR and SFR units, respectively. GPRs are implemented based on FPGA registers or memory blocks, while SFRs are implemented based on FPGA registers.

The configuration parameter *GPR_mem* defines the implementation mechanism of the GPRs. When this parameter is true, GPRs are implemented using the FPGA memory blocks. Otherwise, they are implemented using FPGA registers. In the case that memory blocks are used for the pipeline processor implementation, it is necessary to use two true dual port memory instances in order to provide enough read and write ports for the pipelined processor.

5.3.5.6 ALU/Shifter and test unit

ALU, shifter, and test units perform the arithmetic, logic, shifting, and test instructions supported by the processor. Additionally, these units are responsible of properly setting the carry, zero, sign, and overflow flags.

The ALU performs addition, subtraction, and bitwise logic functions. It is implemented by means of FPGA look-up tables (LUTs). The shifter carries out shifting and rotation instructions. It is implemented as a barrel shifter or multiplier based on the value of the configuration parameter *shifter_mult*. If *shifter_mult* is false, the barrel shifter is implemented in a multiplexer topology using FPGA LUTs. If *shifter_mult* is true, the shifter is implemented using the multipliers available in the FPGA. The test unit is implemented as a combinational function that performs a LFSR operation and sets the flags properly. For this purpose, the FPGA LUTs are used.

5.3.5.7 Interrupt and exception units

Interrupt and exception units manage interrupt requests and occurrence of exceptions. They generate the signals required by the control, SFR, PC, and IR units in order to start the execution of an ISR or ESR.

The interrupt unit uses a global interrupt mask that prevents nested interrupts, and that is also controlled by the CALL_C and RET_C instructions. The interrupt unit reacts to external interrupt requests and signalizes the CPU that the ISR has to be executed.

The exception unit is equipped with an FSM with three states (Figure 5.11). Each state represents a different execution mode: normal execution (*normal_ex*), exception execution (*exception_ex*), and fatal error (*fatal_error*). The state machine remains in *normal_ex* until an exception is triggered. If the exception trigger is produced by an I/O transaction error, the FSM goes to the *exception_ex* state, and signalizes the CPU that the ESR has to be executed. The FSM remains in this state until the exception service routine ends (blue arrows). The FSM performs a transition to *fatal_error* if the exception cause is not an I/O transaction error, or if a new exception is triggered during the ESR (red arrows). The arrival to the *fatal_error* state informs the CPU state machine that it has to stop program execution.

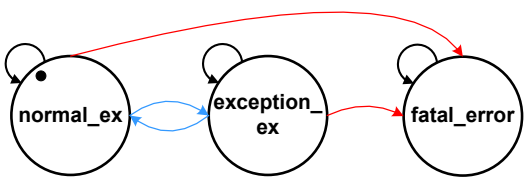


Figure 5.11: FSM of exception module

5.4 Debug-Interface

As discussed in Section 4.3.3, the debug-interface is the communication link between the processor and the ATE. Therefore, it is used for communication and debugging purposes. In the same way as the ROBSY processor, it is equipped with configuration parameters that allow its adaptation to the processor and test scenario. It is described in VHDL using standard language constructs. However, an FPGA-dependent JTAG primitive is necessary to access JTAG signals and TAP state information. This primitive cannot be inferred using a generic VHDL description.

5.4.1 Debug-commands

The debug-interface supports 23 debug-commands used with the multicycle or pipelined processor. Table 5.13 presents a description of each command. The first column is the command name, the second column shows commands executed only in the stop state (Figure 5.10), and the last column provides a brief description of each command.

Debug-command	Stop	Description
SINGLE_STEP	X	Processor executes a single instruction. CPU control unit returns to stop state.
HALT		Processor stops program execution. CPU control unit goes to stop state.
CONTINUE	X	Processor resumes program execution. CPU control unit leaves stop state.
RESET		Processor is reset. Program execution is restarted from program address 0.
PROG_MEM_READ		Content of a single program memory cell is read.
PROG_MEM_WRITE		Content of a single program memory cell is written.
DATA_MEM_READ		Content of a single data memory cell is read.
DATA_MEM_WRITE		Content of a single data memory cell is written.
STACK_MEM_READ		Content of a single stack memory cell is read.
STACK_MEM_WRITE		Content of a single stack memory cell is written.
CPU_STATE_READ		State of the CPU control unit is read.
GPR_READ		Content of a single GPR is read.
GPR_WRITE	X	Content of a single GPR is written.
SFR_READ		Content of a single SFR is read.
SFR_WRITE	X	Content of a single SFR is written.
PC_READ	X	Content of the PC is read. Value read is address of last executed instruction.
PC_WRITE	X	Content of the PC is written.
SP_READ		Content of the SP is read.
IR_READ		Content of the IR is read.
BREAK_POINT_READ		Address and activation value of a single break point is read.
BREAK_POINT_WRITE		Address and activation value of a single break point is written.
INTERRUPT_READ		Information about the execution of an interrupt service routine is read.
DEBUG_ID_READ		Identification value of the debug-interface is read.

Table 5.13: Debug commands

The debug-commands HALT, CONTINUE, RESET, DATA_MEM_READ, DATA_MEM_WRITE, CPU_STATE_READ, and DEBUG_ID_READ are necessary for the ATE/FBTS communication. DEBUG_ID_READ represents an alternative to test the communication link. HALT, CONTINUE, and RESET are used for synchronization purposes, and DATA_MEM_READ and DATA_MEM_WRITE are used to exchange information between ATE and processor.

Loading a new program to the processor program memory is performed by means of the PROG_MEM_READ and PROG_MEM_WRITE debug-commands. The other commands are used for debugging, and are not necessary during test execution.

5.4.2 Access to the JTAG port

The debug-interface has access to the JTAG port based on the JTAG primitive. In order to understand how this primitive works, it is necessary to take a look at the architecture of an IEEE 1149.1 compliant FPGA. For this purpose, Figure 5.12 illustrates the JTAG components and interconnections included in an FPGA. The components are TAP controller, JTAG IR and DRs, and at least one private (or user) DR. The latter is accessed through the JTAG primitive.

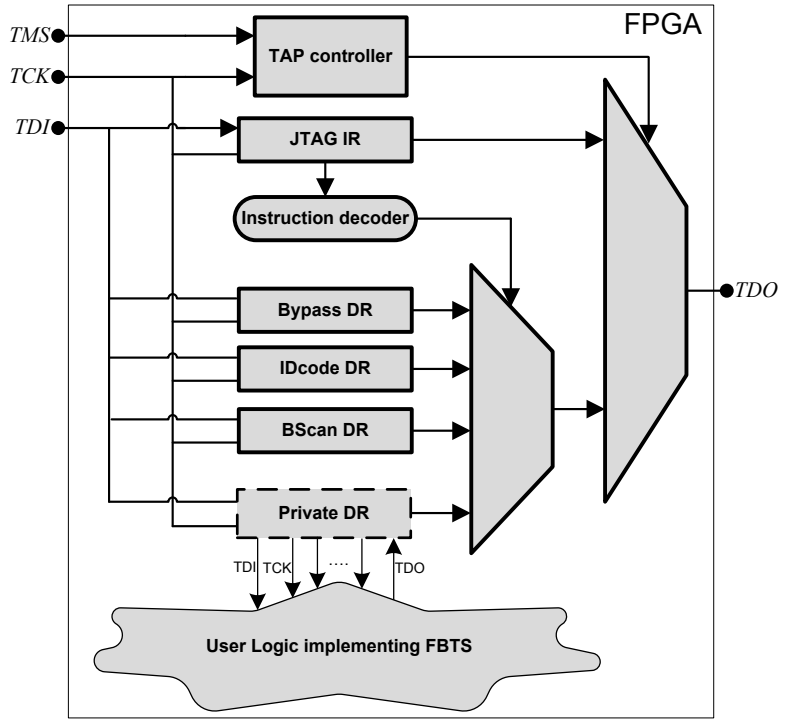


Figure 5.12: JTAG components and interconnections in an FPGA device

The JTAG primitive provides access to the JTAG ports TDI, TDO, TCK, and additional signals that inform the actual state of the TAP controller. The interface to the user logic is FPGA dependent and is considered by the debug-interface VHDL description.

5.4.3 Structure of the debug-interface

The debug-interface uses two debug DRs, which are accessed through the FPGA private DR. One of the debug DRs is the debug command register, while the other one is the debug data register. The former is used to shift in the debug-command, and the latter is used to shift in and out data values associated to the command.

Figure 5.13 presents the block diagram of the debug-interface. The structure is very similar to the IEEE 1149.1 architecture given that the debug command and data register are connected between the TDI and TDO signals in the same way as any other JTAG DR. Additionally, the debug-interface has a sub-module that implements the functions required for every command.

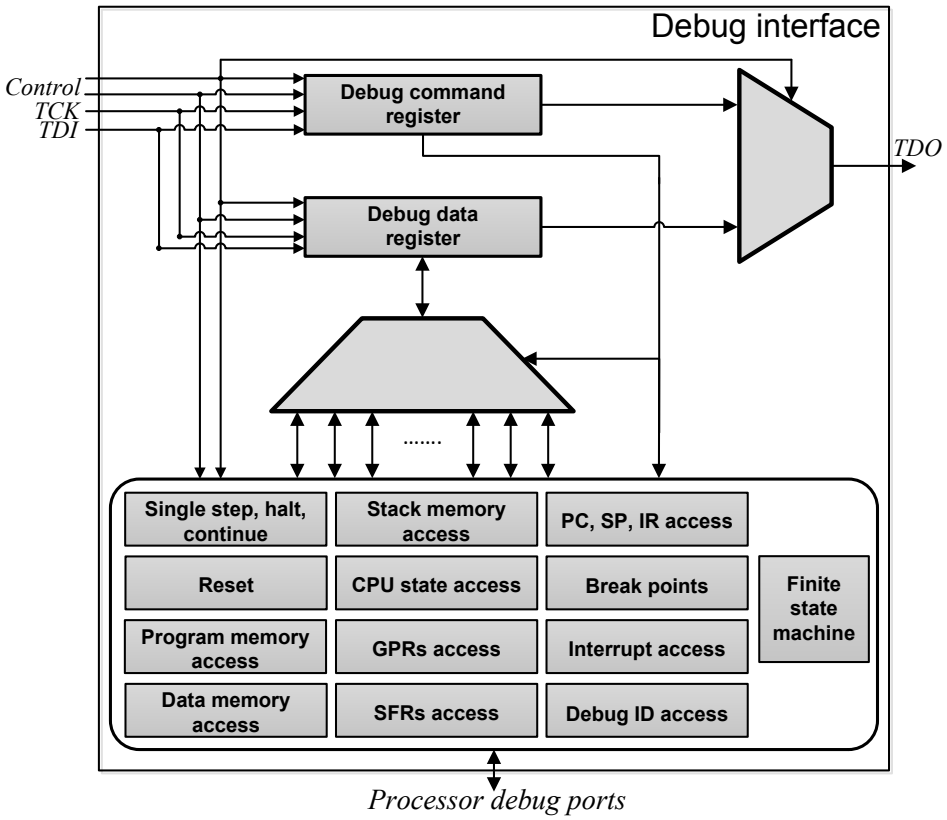


Figure 5.13: Block diagram of the debug-interface

The debug-interface has two groups of ports. The first group comprises ports accessed through the FPGA private DR. These ports are TCK, TDI, TDO, and control signals representing the actual state of the TAP controller. The second group are processor debug ports that connect the debug-interface to the processor.

5.4.3.1 Debug command register and debug data register

The width of the debug command and data registers depends on the configuration options of the debug-interface and processor. The number of bits required to code a debug

command (*deb_command_reg_width*) depends on the commands that are enabled. In this case, the maximal width is five bits, which is obtained when the 23 commands are enabled. The debug command register width is described as follows:

$$deb\_command\_reg\_width = \lceil \log_2(debug\_commands\_enable) \rceil \tag{5.21}$$

The width of the debug data register (*deb_data_reg_width*) depends on the processor configuration and debug commands enabled. It is computed as follows:

$$deb\_data\_reg\_width = \max \begin{pmatrix} instruction\_width, \\ prog\_mem\_depth, \\ data\_width, \\ data\_mem\_depth, \\ stack\_width, \\ stack\_mem\_depth, \\ GPR\_code\_width, \\ \lceil \log_2(SFRs\_num) \rceil, \\ prog\_mem\_depth + 1 \end{pmatrix} \tag{5.22}$$

It seems that some factors in Equation (5.22) are redundant. However, depending on the debug-commands that are disabled, some of the factors are not considered. For example, the last factor of the equation is not necessary if the break point commands are disabled.

Figure 5.14 shows the switching FSM included in the debug-interface. It is used to synchronize shifting operations to both debug registers based on a single FPGA private DR. In this way, additional private DRs are available for other purposes.

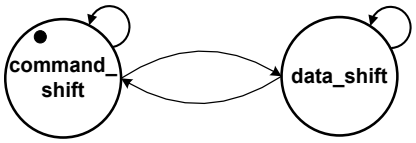


Figure 5.14: Switching state machine

The FSM toggles between *command_shift* and *data_shift* states at the end of every data shift cycle of the TAP controller. There is an exchange of information between the ATE and debug command register when the state machine is in the *command_shift* state, whereas the exchange of information between the ATE and debug data register takes place when the state machine is in the *data_shift* state.

5.4.3.2 Finite state machine

The FSM of the debug-interface is used to coordinate all steps necessary to process each debug-command. There is one FSM for the multicycle processor and another FSM for the pipelined processor. Figure 5.15 shows the state diagrams of both FSMs.

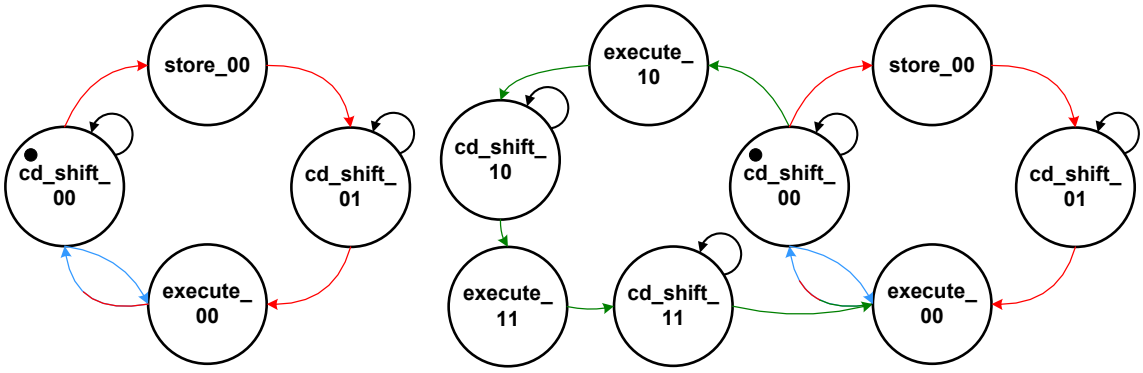


Figure 5.15: FSM of debug-interface for multicycle (left) and pipeline(right) processor

Inputs to the FSMs are the debug-command shifted in the debug command register and the switching signal from the switching FSM. The *cd_shift_#* states are states, in which the command and data values are shifted in the debug registers. A transition to a *store_#* or *execute_#* state is performed after the end of command and data shifts. The *store_00* state is used to store the information shifted in the debug data register for a later used. The *execute_#* states are used to interact with the processor.

Both FSMs have two basic transition paths: *cd_shift_00*, *store_00*, *cd_shift_01*, *execute_00* (red arrows), and *cd_shift_00*, *execute_00* (blue arrows). The blue transition path represents the execution of debug commands that shift in/out a single data value to/from the debug data register. This is the case of SINGLE_STEP, HALT, CONTINUE, RESET, CPU_STATE_READ, PC_WRITE, SP_READ, BREAK_POINT_READ, BREAK_POINT_WRITE, INTERRUPT_READ, and DEBUG_ID_READ commands. The red transition path represents the execution of debug commands that shift in/out two values to/from the debug data register. This is the case of the remaining commands. For example, PROG_MEM_READ requires shifting in an address and shifting out the value read from program memory.

The FSM of the pipelined processor includes a third transition path represented by the green arrows (*cd_shift_00*, *execute_10*, *cd_shift_10*, *execute_11*, *cd_shift_11*, *execute_00*). This path is used for the PC_READ and IR_READ commands in order to read the PC and IR values of fetch, decode, and execute pipeline stages.

5.5 Configuration parameters

Configuration parameters are used to adapt ROBSY processor and debug-interface to the test scenario. The assignment of values to these parameters is performed before synthesis. In this way, it is possible to save resources because processor functions not required for a given test scenario are not implemented.

Table 5.14 presents the configuration parameters, which are grouped in three classes: ISA, microarchitecture, and debug-interface. The first column defines the class, the second columns the parameter name, and the third column the data type of the parameter. The last column presents a short description of the configuration parameter.

Class	Parameter	Type	Description
ISA	*instruction_enable*	Boolean	Enables/disables each of the 50 instructions.
	instruction_code	Natural	Sets instruction command code value.
	data_width	Natural	Sets processor data width. Equation (5.4).
	short_imm_set	Boolean	Sets value of *imm_data_width*. Equation (5.1).
	GPRs_num	Natural	Sets number of GPRs. Equation (5.2).
	SFRs_num	Natural	Sets number of SFRs. Equation (5.3).
	prog_mem_depth	Natural	Sets depth of program memory. Equation (5.5).
	stack_mem_depth	Natural	Sets depth of stack memory. Equation (5.9).
	data_mem_depth	Natural	Sets depth of data memory. Equation (5.6).
	IO_addr_num	Natural	Sets number of I/O addresses. Equation (5.7).
	interrupts_num	Natural	Sets number of interrupts. Equation (5.20).
Micro-architecture	*GPR_mem*	Boolean	GPRs with FPGA memory blocks or registers.
	shifter_mult	Boolean	Shifter with FPGA multipliers or LUTs.
	pipeline_set	Boolean	Enables/disables the processor pipeline.
	wb_retries_num	Natural	Sets number of retry cycles of Wishbone bus.
	wb_waits_num	Natural	Sets number of wait cycles of Wishbone bus.
Debug-interface	*deb_command_enable*	Boolean	Enables/disables each of the 23 debug-commands.
	deb_comand_code	Natural	Sets debug command code value.

Table 5.14: Configuration parameters

The ISA configuration parameters influence the instruction set, native data type, registers, memory, I/O, and interrupts. The first two parameters of Table 5.14 disable/enable each of the 50 instructions supported by the processor and define the value of the instruction command code. The other ISA configuration parameters define the processor native data type width, depth of memories, and number of GPRs, SFRs, I/O, and interrupts.

Based on the ISA parameters, other properties of the processor are automatically computed. They are the instruction width (Equation (5.17)), the width of the program, data, and stack memories (Equations (5.17), (5.4), and (5.10)), and the number of page

pointer registers (Equation (5.11)). The width of the debug data register is also computed as presented in Equation (5.22).

The microarchitecture configuration parameters affect properties of the processor that are not visible to the programmer. Configurability at this level allows adding, removing, or substituting functions and components of the processor without affecting the ISA. In the ROBSY processor, the microarchitecture configuration parameters are used to define the implementation mechanism of the barrel shifter and GPRs, implementation of the pipeline stages, and properties of the Wishbone bus. The latter are required to delimit the maximal number of wait and retry cycles allowed during an I/O transaction.

The last configuration parameter class corresponds to the debug-interface parameters. It is possible to enable/disable debug-commands and define the command code. Based on these parameters, the width of the debug command register is computed as presented in Equation (5.21).

The ISA configuration parameters impact the resource utilization and test time. They also make necessary to adjust the processor development tools (complier, assembler, instruction set simulator, etc.). Additionally, the ATE has to know the value of some of the ISA configuration parameters for communication purposes. For example, in order to write to or read from data memory, the ATE has to know the *data_width* and *data_mem_depth* values. The microarchitecture configuration parameters impact the resource utilization and test time. On the other hand, the debug-interface configuration parameters only affect the resource utilization.

5.6 Summary

This chapter presented the design and implementation details of the ROBSY processor and debug-interface. The instruction set comprises 50 instructions used for branches, procedure calls, load and store transactions, and arithmetic, logic, and test operations. The test instructions are based on the emulation of an LFSR.

The processor supports fixed-point values with a configurable width, and is equipped with GPRs, SFRs, and three separate address spaces known as program, data, and stack spaces. Program and stack address spaces represent the program and stack memories, while the data address space represents the data memory, SFRs, and I/O. PC and SP address the program and stack spaces. A page pointer implemented based on GPRs addresses the complete data space. The processor supports a configurable number of

interrupts and six exceptions to manage error conditions triggered during program execution.

The processor microarchitecture has six main modules, three physically separate memories (program, stack, and data), a stack controller, a data controller, and a central processing unit (CPU). The processor works with a single clock and reset signals, and it has an interrupt input port. It supports a multicycle operation mode as well as a three stage pipeline mode. The I/O bus is implemented based on the Wishbone standard.

The debug-interface supports 23 debug-commands. These commands make it possible to control the test program execution and exchange test information with the ATE. The debug-interface is described in VHDL using standard language constructs, but it is necessary to include an FPGA dependent JTAG primitive in order to access JTAG signals and TAP state information.

The adaptation mechanism of the ROBSY processor and debug-interface is based on configuration parameters, whose values are defined before synthesis. The configuration parameters are grouped in three classes: ISA, microarchitecture, and debug-interface. They influence the way the processor and debug-interface are implemented, altering the processor resource utilization and execution time. Chapter 6 presents the mechanism used to define the value of the configuration parameters automatically, and Chapter 7 presents the effect that these parameters have on the resource utilization and test time.

6 Automatic generation process

6.1 Introduction

The automatic generation of the test system is essential for the viability of any FBT approach. It reduces costs, accelerates test development, and minimizes errors caused by the human factor.

In the ROBSY approach, the automatic generation process is in charge of the generation of software (SW) for the ATE, embedded software (ESW) for the ROBSY processor, and hardware (HW) descriptions for co-processors and the complete FBTS. Additionally, the automatic generation process is in charge of determining proper values for the processor and debug-interface configuration parameters.

This chapter presents the automatic generation process of the test system for an FBTS with a single processor/co-processor pair, in which the main emphasis lies on the ESW generation and the adaptation of the processor. Section 6.2 presents an overview of the automatic generation process. Section 6.3 deals with the automatic generation of SW for the ATE. Section 6.4 deals with automatic generation of ESW and HW and the assignment of values for the processor and debug-interface configuration parameters. Finally, section 6.5 presents a summary of the chapter.

6.2 Overview of the automatic generation process

Figure 6.1 shows an overview of the automatic generation process for a single DUT-M. The DUT-M is the main source of information for the generation process because it contains the description of the DUT I/Os, timing, and electrical properties, as well as the layer procedures specifying the test algorithms. Based on the DUT-M description, the automatic generation process produces SW for the ATE, ESW for a processor, and HW for a co-processor.

The first step of the generation process is the layer partitioning, which defines the location of interfaces I2 and I3, and therefore the layers that should be implemented in SW, ESW, and HW. As already presented in Section 3.4.6, there are 6 possible layer partitions considering that L1 and L5 are always implemented in HW and SW, respectively.

After the layer partitioning is performed, layers that should be implemented in SW, ESW, and HW are the input to the ATE program generator and FBTS generator. In this case, DUT I/Os, timing, and electrical properties described in the DUT-M are grouped together with the HW layer procedures because they are required for the implementation of L1 and the generation of additional files used by the synthesis tool (e.g. pin assignment file).

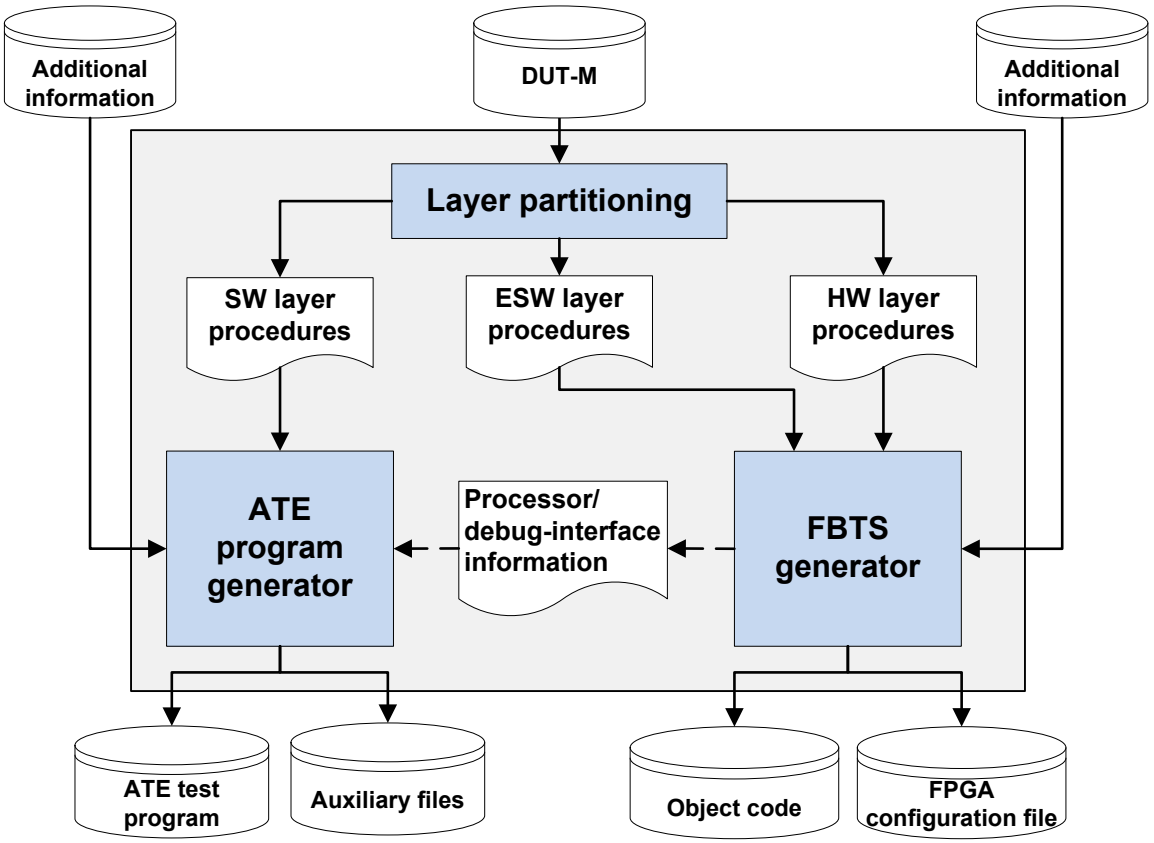


Figure 6.1: Overview of automatic generation process for a single DUT-Model

The ATE program generator is in charge of producing the test program and auxiliary files required by the ATE. For this purpose, additional information is passed to the generator, which corresponds to the ATE software tool and the structure of the BScan chain. Apart of the SW layer procedures, processor/debug-interface information generated by the FBTS generator depending on the value of the processor and debug-interface configuration parameters is also passed to the ATE program generator. Section 6.3 presents the ATE program generator in more detail.

The FBTS generator produces the FPGA configuration file and the object code executed by the ROBSY processor. Additionally, the FBTS generator is responsible of assigning values to the processor and debug-interface configuration parameters, compiling and assembling the ESW layer procedures, and generating the co-processor and FBTS hardware descriptions. Furthermore, it is responsible of synthesizing the hardware

descriptions based on the corresponding FPGA synthesis tool. Section 6.4 presents the FBTS generator in more detail.

6.3 ATE program generator

Figure 6.2 shows the ATE program generator. It consists of a SW compiler and an auxiliary SW generator. The SW compiler generates the ATE test program, while the auxiliary SW generator generates auxiliary files required by the ATE.

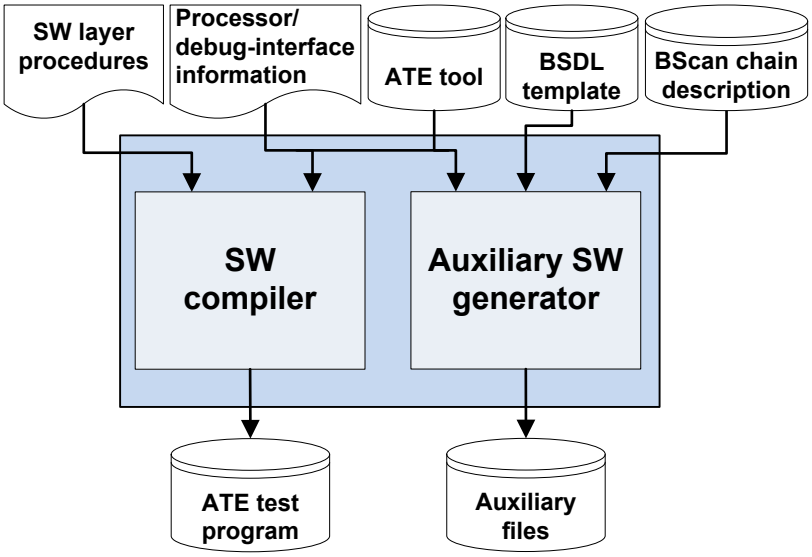


Figure 6.2: ATE program generator

6.3.1 ATE/FBTS communication

Before presenting details of the SW compiler and auxiliary SW generator, it is necessary to explain the ATE/FBTS communication mechanism. The data exchange between ATE and FBTS takes place through the JTAG interface and the debug command and data registers of the debug-interface. In this case, the ATE takes the role of the active communication participant, while the processor takes the role of the passive participant.

The ATE sends an identification tag in order to inform the processor which procedure of the highest layer implemented in ESW to execute. After that, it sends the value of the procedure arguments of type IN and INOUT. After the arguments are sent, the ATE sends a start command that indicates the processor to start execution. At this point, the ATE waits until the FBTS finishes the execution of the ESW and HW layer procedures by polling the corresponding status flag. When the FBTS finishes execution, the ATE acquires the value of the procedure arguments of type OUT and INOUT. An example of DUT-M procedures is presented in Table 7.2.

In the case that the FBTS is equipped with two performance counters (active time and test time counter) to measure execution time, the ATE has to send the proper information to initialize them and receive their content. Section 7.2.4 provides more information about the performance counters.

The steps carried out during the ATE/FBTS communication considering the amount of data transfers (DTs) are as follows:

1. Initialization of test time counter (3 DTs).
2. Initialization of ESW procedure.
    a. Send of procedure identification tag (1 DT).
    b. Send value of arguments of type IN and INOUT (*x* DTs).
    c. Acquisition of processor status (1 DT).
    d. Send start command (1 DT).
3. Poll processor status (*p* DTs).
4. End ESW procedure.
    a. Acquisition of argument values of type OUT and INOUT (*y* DTs).
    b. Acquisition of the active time counter content (1+*c* DTs).
5. Acquisition of the test time counter content (3+*c* DTs).

Steps 1 and 5 are performed at the beginning and end of test execution, respectively. Steps 2 to 4 are performed each time that a procedure belonging to the highest layer implemented in ESW is called by a SW layer procedure. A DT is a single access to the debug-interface, which typically comprises four independent JTAG shifting operations. The amount of DTs required for sending or acquiring argument values (*x* and *y*) depends on the processor *data_width* and the number and width of arguments. The amount of DTs required to acquire the counter values (*c*) depends on the processor *data_width* and the counters width. The amount of data-transfers performed during polling (*p)* depends on the time that the ATE has to wait until the processor executes the ESW procedure. Due to the layer concept, the execution of the ESW procedure also involves the execution of procedures belonging to lower layers.

A region of the processor data memory is reserved for the ATE/FBTS communication. This region is used to share procedure tags, argument values, performance counter values, and start and finish conditions. During the test execution, there is no need to use all the features of the debug-interface. Therefore, only the following debug-interface commands are required:

- HALT, CONTINUE, RESET, DATA_MEM_READ, DATA_MEM_WRITE, CPU_STATE_READ, SFR_READ, SFR_WRITE, DEBUG_ID_READ.

6.3.2 Software compiler

The SW compiler is in charge of parsing SW layer procedures for the generation of the ATE test program. For this purpose, it needs to know the tool used by the ATE to execute the test program. In Section 6.3.3 more information about the supported tools is provided.

The ATE/FBTS communication is realized based on wrapper procedures. These wrappers are added automatically to the test program, and they describe the steps explained in Section 6.3.1. The SW compiler makes use of the processor/debug-interface information delivered by the FBTS generator for this purpose. This information corresponds to the processor and debug-interface configuration parameters *data_width*, *data_mem_depth*, *deb_comand_code*, and *deb_comand_enable*. In the same way, the start address of the region in the processor data memory used for communication purposes is part of the processor/debug-interface information.

The SW compiler is composed of a lexical analyzer and a parser that translates SW layer procedures to the language supported by the ATE tool. The GNU tools FLEX (lexical analyzer generator) and Byson (parser generator) were used for the development of the SW compiler. For this purpose, the Cygwin Unix-like environment was used.

6.3.3 Auxiliary software generator and supported ATE tools

The auxiliary SW generator generates additional auxiliary files required by the ATE tool. For this purpose, the auxiliary SW generator makes use of most of the processor/debug-interface information delivered by the FBTS generator. This information corresponds to the *deb_comand_code*, and *deb_comand_enable* configuration parameter values, as well as the processor configuration parameter values necessary to compute the size of the debug-interface command and data registers (Equations (5.21) and (5.22)).

The current version of the SW generator supports two ATE tools. The first one is Nebula from Intellitech [144], which interprets test programs written on a procedural description language (PDL) that is not 100% in compliance with the PDL specified in the IEEE 1149.1-2013 standard [1]. The main differences between the PDL supported by Nebula and the standard are as follows:

- Standardized level-0 instructions iProc and iProcGroup are not supported by Nebula.
- Level-0 instruction iScope is supported by Nebula but it is not included in the standard.

The second tool is quartus_stp from Altera [145], which interprets scripts written in the Tool Command Language (TCL). This tool is compatible with Altera FPGAs only.

If Nebula is used, the SW compiler transforms the SW layer procedures to PDL level-1. In this case, the auxiliary SW generator produces a BScan chain file, a BScan description language (BSDL) file, and an auxiliary PDL file. These files are generated based on a set of templates or descriptions that are either developed by the test engineer or are part of a library. The BScan chain file describes the structure of the BScan chain on the PCB, and it is generated based on a BScan chain description file. The BSDL file corresponds to the FPGA BSDL description. This file is generated based on a BSDL template of the FPGA and updated with the debug command and data registers of the debug-interface, the JTAG instructions required to access them, and mnemonics of the debug-interface command codes. The auxiliary PDL file describes all shifting operations required to execute each of the debug-interface commands.

If instead of Nebula, the quartus_stp tool is used, the SW compiler transforms the SW layer procedures to a TCL script. In this case, the auxiliary SW generator produces a single auxiliary file, which includes the description of the debug-interface command and data registers, the debug-interface command codes, and the TCL procedures that describe all shifting operations required to execute each of the debug-interface commands. In this case, there is no need for a BScan chain or BSDL description file because the quartus_stp tool is able to identify the Altera FPGA found on the PCB automatically.

The auxiliary SW generator is implemented in TCL. This script reads the required information and generates the corresponding auxiliary files.

6.4 FPGA based test system generator

Figure 6.3 presents the FBTS generator, which is composed of the ESW generator, HW generator, and synthesis tool. The ESW generator produces object code and processor/debug-interface information required by the SW and HW generators. The inputs are the ESW layer procedures and a subset of configuration parameters known as the independent configuration parameters (Section 6.4.2). The HW generator produces the FBTS description and additional files required by the synthesis tool such as FPGA constraints and pin assignment files. This is carried out based on the HW layer procedures, PCB netlist and constraints, and the outcome of the ESW generator. The synthesis tool is in charge of producing the FPGA configuration file.

The dashed arrow in Figure 6.3 indicates that it is not mandatory to use the object code as input to the HW generator. In the case that the object code is used as input, the content of the processor's program and data memories is initialized with the object code and becomes part of the FPGA configuration file. Otherwise, the ATE has to load the object code to the processor memories after the FPGA is configured. Using the ATE to write to program memory provides a way to change the functionality of the FBTS without reconfiguring the FPGA.

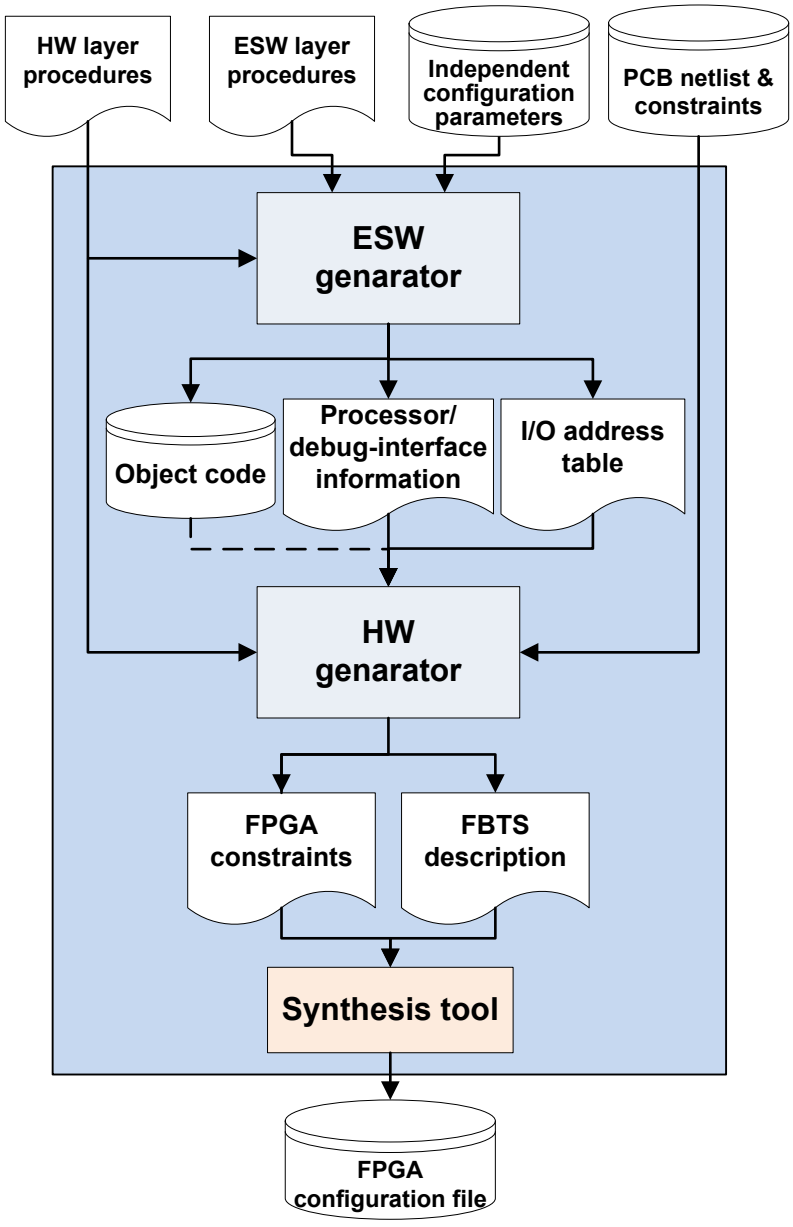


Figure 6.3: FBTS generator

6.4.1 Processor/co-processor communication

Before presenting the details of the FBTS generator, it is necessary to explain the processor/co-processor communication mechanism. The communication mechanism is very similar to the ATE/FBTS communication mechanism presented in Section 6.3.1. However, instead of the JTAG interface and debug command and data registers, the data exchanged is performed based on the on-chip Wishbone bus and Wishbone registers included in each co-processor. In this case, the processor takes the role of the active communication participant, while the co-processor takes the role of the passive participant.

The steps performed during the communication are the same steps performed for the ATE/FBTS communication with the exception that it is not necessary to exchange procedure identification tags. In this case, each co-processor is equipped with unique control and status Wishbone registers per procedure that are addressed depending on the procedure that should be executed. Procedure arguments are represented by Wishbone registers, and their number depends on the number and width of arguments and processor *data_width*.

The Wishbone registers are mapped to the I/O address space of the processor. As shown in Figure 6.3, the ESW generator produces an I/O address table that contains the addresses assigned to all co-processor Wishbone registers. This information is used by the HW generator for the generation of the co-processor hardware description.

6.4.2 Reorganization of the configuration parameters

One of the main tasks of the FBTS generator is to determine the value of a subset of the processor and debug-interface configuration parameters. In this way, the ROBSY processor is adapted to a given test scenario and there is still some room for the test engineer to evaluate different test time and resource utilization trade-offs.

For this purpose, the processor and debug-interface configuration parameters are reorganized in four groups. Table 6.1 presents the four groups, which are known as constant (independent), variable (independent), DUT-M (dependent), and ASM (dependent). The first column shows the name of the group, whereas the second, third and fourth columns show the name, type, and class of a given configuration parameter.

The adaptation of the processor is performed by determining the proper value of the dependent configuration parameters. The value of the DUT-M (dependent) configuration parameters is computed by analyzing the ESW layer procedures of the DUT-M, whereas

the value of the ASM (dependent) configuration parameters is computed by analyzing the assembly program generated during the ESW compilation (Section 6.4.3.1). This is performed as follows:

i. *data_mem_depth* is computed by looking at the total number of global and local variables found in ESW layer procedures. For this purpose, it is necessary to consider the width of each variable and the relations between callers and callees.
ii. *stack_mem_depth* is computed by looking at the relations between callers and callees found in ESW layer procedures. The stack is used to store memory pointers and return addresses of the procedures.
iii. *IO_addr_num* is computed by looking at the arguments of procedures belonging to the highest layer implemented in HW. For this purpose, it is necessary to compute the total number of Wishbone registers required by the co-processor.
iv. *instruction_enable* and *instruction_code* are computed by listing the type of instructions used in the assembly program.
v. *GPRs_num* is computed by listing the GPRs used in the assembly program.
vi. *prog_mem_depth* is computed by counting the total number of instructions in the assembly program.

Group	Parameter	Type	Class
Constant (independent)	*SFRs_num*	Natural	ISA
	interrupts_num	Natural	ISA
	wb_retries_num	Natural	Microarchitecture
	wb_waits_num	Natural	Microarchitecture
	deb_command_enable	Boolean	Debug-interface
	deb_command_code	Natural	Debug-interface
Variable (independent)	*data_width*	Natural	ISA
	short_imm_set	Boolean	ISA
	GPR_mem	Boolean	Microarchitecture
	pipeline_set	Boolean	Microarchitecture
	shifter_mult	Boolean	Microarchitecture
DUT-M (dependent)	*data_mem_depth*	Natural	ISA
	stack_mem_depth	Natural	ISA
	IO_addr_num	Natural	ISA
ASM (dependent)	*instruction_enable*	Boolean	ISA
	instruction_code	Natural	ISA
	GPRs_num	Natural	ISA
	prog_mem_depth	Natural	ISA

Table 6.1: Reorganization of processor and debug-interface configuration parameters

The number of GPRs, the depth of the stack, program, and data memories, and the I/O address space are tailored by determining the proper value of the dependent configuration

parameters. Additionally, processor instructions and functional units that are not necessary for the execution of the ESW layer procedures are not implemented as part of the processor. In this way, it is possible to obtain significant reduction of the amount of resources necessary to implement the processor. Similar approaches for removing instructions and processor features that are not used in a given scenario have proven to be very effective in the reduction of resources and power consumption [126, 146, 147].

At this point, it is necessary to define the value of the constant and variable (independent) configuration parameters. As the name implies, the same value is assigned to the constant (independent) configuration parameters independently of the test scenario. This is possible due to the following reasons:

- *SFRs_num* is adjusted if new SFRs are included in the ROBSY processor or if the two optional SFRs are used. Given that both circumstances do not apply for the ROBSY approach, *SFRs_num* remains constant for all test scenarios.
- *wb_retries_num* and *wb_waits_num* are used to define configuration options of the Wishbone bus. Given that the value of both parameters does not depend on the test scenario, it remains constant for all test scenarios.
- *deb_command_enable* and *deb_command_code* are known in advance as already discussed in Section 6.3.1. Therefore, they remain constant for all test scenarios.
- *interrupts_num* is 0 for all test scenarios because they are not used in the ROBSY approach. As already mentioned in Section 6.4.1, polling is used for the processor/co-processor communication.

The number of processor variants that are available for the test engineer to evaluate different test time and resource utilization trade-offs is defined by the five variable (independent) configuration parameters. This facilitates the evaluation of trade-offs because the number of processor variants that have to be considered is much lower in comparison to the case in which all configuration parameters have to be defined by the test engineer.

Besides, it is not possible to compute the value of the variable (independent) configuration parameters in an exact way just by analyzing the DUT-M and assembly program. *GPR_mem*, *pipeline_set,* and *shifter_mult* belong to the microarchitecture class, which means that they affect properties related to the processor microarchitecture only. Therefore, they do not have any influence on the generation of the object code and are not influenced by the DUT-M or assembly program.

On the other hand, *data_width* and *short_imm_set* belong to the ISA class. However, their value cannot be computed in an exact way based on the properties of the ESW layer

procedures. The reason for this is that any value assigned to these parameters produces a processor variant that can execute the ESW layer procedures. Besides, they are ideal candidates to let the test engineer evaluate different test time and resource utilization trade-offs because they have a significant impact on the properties of the processor ISA, affecting not only the resource utilization but also test time.

If new configuration parameters are included in the processor, it is necessary to order them in one of the four groups. Typically, microarchitecture configuration parameters can be ordered in the variable (independent) group and debug-interface configuration parameters are ordered in the constant (independent) group. In the case of ISA configuration parameters, it is necessary to evaluate if they can be ordered in the variable (independent) or dependent configuration parameters groups.

6.4.3 Embedded software generator

The ESW generator computes the value of the dependent configuration parameters and generates the object code and additional information required by ATE and HW generators. The object code is the numerical machine code stored in the processor program and data memories that is obtained by compiling the ESW layer procedures of the DUT-M. The processor/debug-interface information includes the value of all configuration parameters, the data memory start address for the ATE/FBTS communication, and the I/O address table for the processor/co-processor communication.

Figure 6.4 shows the ESW generator in more detail. It is divided into three main phases that are executed sequentially starting with phase 1.

6.4.3.1 Phase 1

Phase 1 comprises the DUT-M analyzer and ESW compiler. The DUT-M analyzer examines the ESW layer procedures and the procedures of the highest layer implemented in HW in order to compute the I/O address table, the start address for the ATE/FBTS communication, and the DUT-M (dependent) configuration parameters. After this step is carried out, the ESW compiler generates the assembly program based on the ESW layer procedures, independent configuration parameters, and the information produced by the DUT-M analyzer.

The computation of *data_mem_depth* is realized based on the amount of memory required to store global and local variables of ESW layers. For local variables, it is necessary to analyze the interactions between callers and callees by means of a procedure tree that models these interactions. The computation of *stack_mem_depth* also makes use of the

procedure tree in order to define the maximum amount of stack memory locations necessary to store return addresses and data pushed in and popped from the stack.

The computation of *data_mem_depth* and *stack_mem_depth* is performed based on the following assumptions:

- There are no dynamic memory allocations.
- A caller in layer *n* is allowed to call procedures located in layers *n* and *n-1*.
- Recursion is not allowed.
- The callee is not allowed to call the caller.

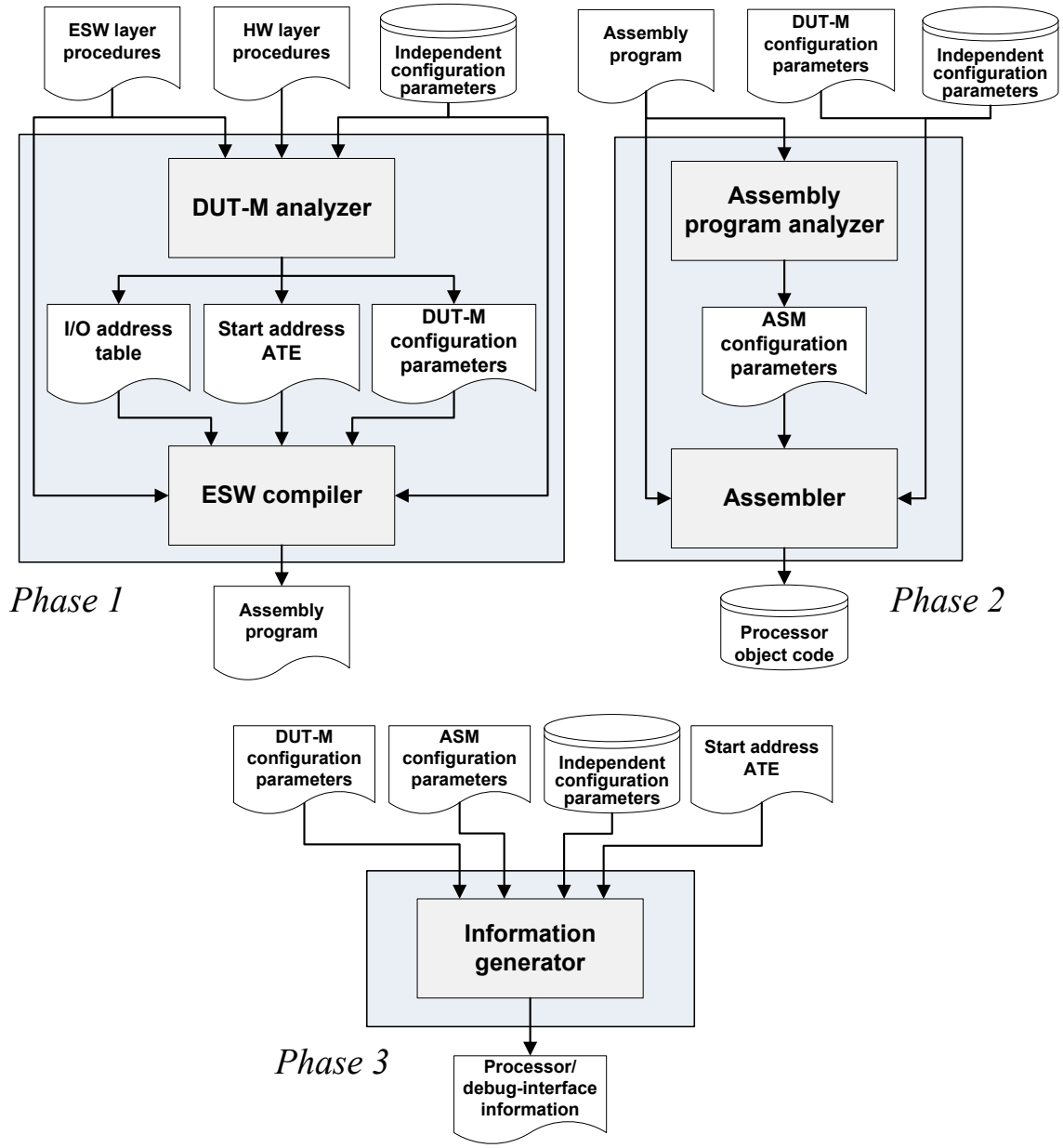


Figure 6.4: Embedded software generator. Phase 1 (top left), 2 (top right), 3 (bottom)

These assumptions do not represent any limitations for the ROBSY approach. The first assumption is compatible with RTDL because the language does not support dynamic memory allocation. This would become a big issue for the generation of hardware. The other three assumptions are compatible with the layer concept.

The computation of the I/O address table, the start address for the ATE/FBTS communication, and *IO_addr_num* is realized by analyzing the arguments of the highest layer implemented in HW and taking into consideration the value of *data_width*, *data_mem_depth*, and *SFRs_num*. The DUT-M analyzer also takes into consideration the control and status Wishbone registers.

After the analysis step, the ESW compiler parses ESW layer procedures and generates the assembly program with the corresponding wrappers for the ATE/FBTS and processor/co-processor communication. For this purpose, the ESW compiler considers the value of independent configuration parameters, the address of co-processor Wishbone registers (I/O address table), and the data memory start address.

The DUT-M analyzer and ESW compiler are implemented based on a lexical analyzer and a parser. The GNU tools FLEX (lexical analyzer generator) and Byson (parser generator) of the Cygwin environment were used for the development of the lexical analyzer and parser. The DUT-M analyzer computes DUT-M configuration parameters, start address for ATE/FBTS communication, and I/O address table. The parser of the ESW compiler translates ESW layer procedures to an intermediate low-level representation. TCL scripts transform the intermediate representation into the processor specific assembly program.

6.4.3.2 Phase 2

Phase 2 comprises the analysis of the assembly program and generation of object code. The assembly program analyzer examines the assembly program in order to compute the ASM configuration parameters. After that, the assembler generates the object code and value of the independent and dependent configuration parameters.

prog_mem_depth is computed by counting the number of instructions in the assembly program. *GPRs_num* is computed by looking for the registers used in the assembly program. This is possible because the ESW compiler was built in such a way that it maps all local and global variables into memory locations using as many registers as necessary. *instruction_enable* and *instruction_code* are computed by collecting information about instructions used in the assembly program. In this way, unused instructions are not implemented as part of the processor ISA. This is known as ISA subsetting [126].

The assembly program analyzer was developed in TCL. The assembler was developed in C as a two-pass assembler. For the assembly process, the assembler has to know the value of all the ISA configuration parameters.

6.4.3.3 Phase 3

Phase 3 comprises the generation of the processor/debug-interface information. For this purpose, the information generator encapsulates the outcome of phases 1 and 2 into a single file. This file is built in such a way that it can be parsed by the ATE program generator. The information generator was developed in TCL.

6.4.4 Hardware generator

The HW generator produces hardware descriptions for the co-processor and FBTS, and additional information required by the synthesis tool, such as a project description and FPGA constraints (timing and pin assignments). For this purpose, it makes use of the HW layer procedures, processor/debug-interface information, I/O address table, PCB netlist, and hardware constraints.

The PCB netlist helps determining the FPGA pins connected to the DUT as well as the clock and reset signals. The hardware constraints provide information about the timing properties of clock and reset signals, as well as additional information necessary for the generation of FPGA constraints. The development of the HW generator is not part of this dissertation. For more information about this subject refer to [98, 99].

6.4.5 Synthesis tool

The synthesis tool outputs the FPGA configuration file based on the FBTS description, processor description, FPGA constraints, and project files delivered by the HW generator. This tool is FPGA dependent, and the actual generation process supports Xilinx as well as Altera FPGAs. In the case of Altera, Quartus II tools [145] quartus_sh, quartus_map, quartus_fit, quartus_asm, quartus_sta, and quartus_pgm are used. In the case of Xilinx, ISE tools [148] xst, ngdbuild, map, par, trce, and bitgen are used.

6.5 Summary

This chapter presented the automatic generation process of the ROBSY approach with emphasis in the ESW generation. The automatic generation process supports a single processor co-processor pair and comprises three generators: the SW generator, FBTS

generator, and HW generator. The generators are responsible of producing the software procedures for the ATE, the object code for the processor, and the FBTS hardware description based on a specific layer partition. The analyzers, compilers, and additional tools found in the generators were developed by means of TCL scripts, C programs, and GNU tools FLEX and Byson.

The ESW generation process is also in charge of determining the value of a subset of the processor and debug-interface configuration parameters. The main goal is to adapt the processor to the test scenario and reduce the number of processor variants that are available for the test engineer to evaluate different test time and resource utilization trade-offs. For this purpose, the configuration parameters are reorganized in four groups:

- Constant (independent).
- Variable (independent).
- DUT-M (dependent).
- ASM (dependent).

The same values are assigned to the constant (independent) configuration parameters independently of the test scenario. The DUT-M (dependent) configuration parameters are computed based on the ESW layers of the DUT-M, and the ASM (dependent) configuration parameters are computed based on the assembly program generated by the ESW compiler. The computations are performed in a deterministic way based on the equations described in Chapter 6. The independent configuration parameters basically tailor the resource utilization of the ROBSY processor depending on the properties of the ESW layer procedures.

The number of processor variants that can be generated in order to evaluate different test time and resource utilization trade-offs is defined by the five variable (independent) configuration parameters. In contrast to the dependent configuration parameters, the value of the variable (independent) configuration parameters cannot be computed in a deterministic way due to their properties.

7 Experimental phase

7.1 Introduction

This chapter presents the experimental results obtained with the ROBSY test system. For this purpose, the automatic generation flow discussed in Chapter 6 generates the SW, ESW, and computes the processor dependent configuration parameters based on the DUT-M and layer partition. The results provide an answer to the following questions:

- ROBSY processor:
  - What is the effect of the adaptation mechanism and independent configuration parameters on resource utilization and test time?
  - Is there a way to find an efficient processor variant in terms of resource utilization and test time for a given test scenario?
  - Which layers are implemented more efficiently in ESW?
- ROBSY test system:
  - How does the layer partition influence the resulting processor and FBTS in terms of resource utilization and performance?
  - What is the speed-up achieved in comparison to other test techniques?

Sections 7.2 and 7.3 describe the experimental setup and DUTs. Section 7.4 presents an analysis of resource utilization results, while Sections 7.5 and 7.6 present an analysis of obtained performance. Section 7.7 provides a comparison of ROBSY against Nios II and a virtual length shift register. The chapter concludes with a summary in Section 7.8.

7.2 Experimental setup

7.2.1 Hardware setup

The experiments were performed using the DE2-115 development board from Terasic [149] and the VarioTap Coach Board from Göpel Electronic [150]. The ATE is a personal computer with an Intel core i7-960 (3.2 GHz) processor and 6 GB RAM. Figure 7.1 shows the hardware setup without including the ATE.

The DE2-115 is equipped with a Cyclone IV-E FPGA (EP4CE115F29C7) (1). This FPGA has the highest capacity of the Cyclone IV-E family [151] and it provides following features:

- 114.480 logic elements (LEs). A LE comprises a flip flop and a 4-input look-up table (LUT).
- 432 9-Kbits memory blocks (M9K blocks) that can be configured as 8192x1, 4096x2, 2048x4, 1024x8/9, 512x16/18, or 256x32/36 memory arrays.
- 266 18x18 multipliers that can be configured as two 9x9 multipliers.

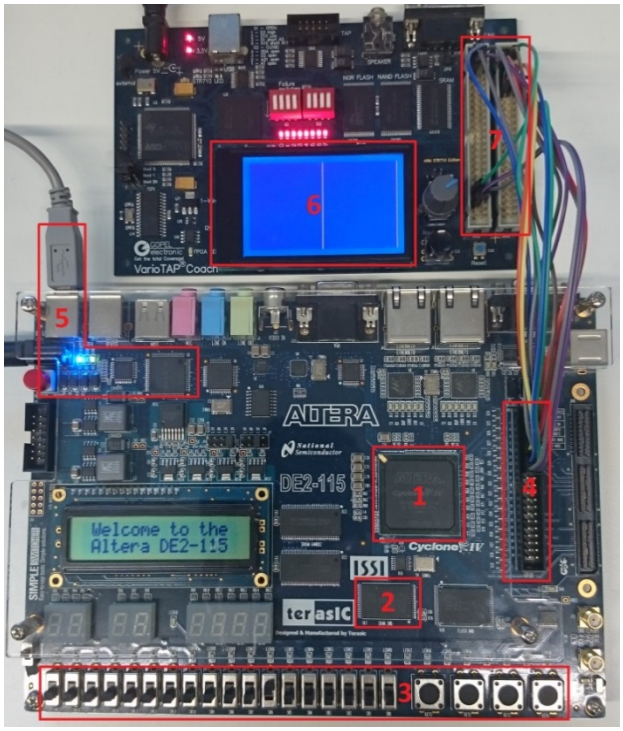


Figure 7.1: Hardware setup (no ATE). VarioTap coach (top) and DE2-115 (bottom)

The DE2-115 includes an IS61WV102416 SRAM (2), which is one of the DUTs (Section 7.3.1). Additionally, it includes a 50 MHz oscillator used as FPGA clock source, four push buttons and eighteen switches (3), and a 40-pin expansion header (4). The board is equipped with an embedded USB Blaster (5) used in JTAG mode for the configuration of the FPGA and communication between ATE and FBTS. The TCK frequency is 6 MHz.

The VarioTap Coach is equipped with a DIP128-6 LCD (Electronic Assembly) [152] and corresponding LCD controller (6). This is the second DUT (Section 7.3.2), which is accessed through the 50-pin XH2 and XH3 I/O connectors (7). A bridge between DE2-115 and VarioTap Coach connects the Cyclone IV-E FPGA with the LCD controller via 14 female/female jumper wires. This setup was necessary because the Spartan 3 FPGA (XC3S50A) located on the VarioTap Coach does not have enough capacity for implementing all FBTS variants.

7.2.2 Software setup

7.2.2.1 Software tools

The ATE is equipped with Windows 7, Quartus II 15.1, TCL/TK 8.6, and Codeblocks 13.12. Quartus II performs the synthesis of the FBTS, configures the FPGA, and is used for the ATE/FBTS communication. TCL/TK and Codeblocks are used during the automatic generation process to run scripts and generate the assembler, respectively.

The Quartus II tools quartus_sh, quartus_map, quartus_fit, quartus_asm, and quartus_sta [145] run in batch mode for the FBTS synthesis. quartus_pgm is in charge of configuring the FPGA and quartus_stp of executing SW layer procedures. In this way, it is possible to automate the generation of the ROBSY test system, the synthesis and configuration of the FBTS, and the test execution. Nebula was not utilized because it does not support a batch execution mode. In any case, similar test time results are expected with Nebula.

It was not possible to make use of the HW generator. The reason for this is that the actual HW generator only supports layer L1. Therefore, it was necessary to work with hand-written co-processors and an automatic generated processor/co-processor interface. The interface adapts the co-processor to different *data_width* and *IO_addr_num* values. A TCL script generates the interface with the proper amount of Wishbone registers.

7.2.2.2 Quartus II settings

For the synthesis of the FBTS, the following compiler settings were selected:

- ALLOW_ANY_RAM_SIZE_FOR_RECOGNITION: ON.
- OPTIMIZATION_MODE: AGGRESSIVE PERFORMANCE.

The first setting allows the synthesis tool to implement GPRs using M9K blocks, even if the number of GPRs is low. The second setting tells the synthesis tool to make an aggressive effort to optimize the FBTS for speed. The side effect of this option is an increase in the synthesis time and the use of more resources in comparison to a balanced optimization mode. The other compiler settings have their default values.

The FPGA Synopsys design constraints file has a 1 ns clock period constraint. The use of an over-constrained clock period forces the synthesis tool to achieve high operation frequency values. Working with a constraint of 20 ns (required value for the FBTS in the DE2-115) would deliver solutions that fulfill this constraint, but that are not necessarily close to the maximum achievable operation frequency of the FBTS.

7.2.3 Processor and FBTS variants

The automatic generation process computes the value of the DUT-M and ASM (dependent) configuration parameters (Table 6.1). On the other hand, the same values are assigned to the constant (independent) configuration parameters. These values are:

- *SFRs_num*: 5. *interrupts_num*: 0. *wb_retries_num*: 3. *wb_waits_num*: 3.
- *deb_command_enable/deb_command_code*: HALT/1, CONTINUE/2, RESET/3, DATA_MEM_READ/4, DATA_MEM_WRITE/5, CPU_STATE_READ/6, SFR_READ/7, SFR_WRITE/8, DEBUG_ID_READ/9.

The variable (independent) configuration parameters generate multiple processor variants with different resource utilization and performance trade-offs. Boolean values true or false are assigned to *short_imm_set*, *GPR_mem*, *pipeline_set*, and *shifter_mult*. Natural values 8, 12, 16, 20, 24, 28, and 32 are assigned to *data_width*. Table 7.1 presents the way to identify a processor variant. The 'X' in index I specifies a *data_width* value, while .0-.15 values represent a 4-bit-vector formed by ordering *short_imm_set*, *pipeline_set*, *GPR_mem*, and *shifter_mult* from the most to the lowest significant bit, respectively. For example, $I_{20.7}$ is a 20-bit processor variant with the 4-bit vector [0111] (*short_imm_set* false, and *pipeline_set*, *GPR_mem*, and *shifter_mult* true).

I	*short_imm_set*	*pipeline_set*	*GPR_mem*	*shifter_mult*
$I_{X.0}$	false	false	false	false
$I_{X.1}$	false	false	false	true
$I_{X.2}$	false	false	true	false
$I_{X.3}$	false	false	true	true
$I_{X.4}$	false	true	false	false
$I_{X.5}$	false	true	false	true
$I_{X.6}$	false	true	true	false
$I_{X.7}$	false	true	true	true
$I_{X.8}$	true	false	false	false
$I_{X.9}$	true	false	false	true
$I_{X.10}$	true	false	true	false
$I_{X.11}$	true	false	true	true
$I_{X.12}$	true	true	false	false
$I_{X.13}$	true	true	false	true
$I_{X.14}$	true	true	true	false
$I_{X.15}$	true	true	true	true

Table 7.1: 112 possible processor variants

The reason to limit *data_width* to a subset of values is to maintain the number of processor variants low. The use of untypical values (12, 20, 24, and 28) provides the option to work with processor variants highly adapted to the DUT-M. For example, a 20-

bit processor provides a more efficient utilization of resources in comparison to a 32-bit processor if the maximum size of variables declared in ESW procedures is 20 bits.

7.2.4 Resource utilization and performance metrics

The *resource utilization* is measured based on *LEs*, *M9K blocks*, and *9x9 multipliers* used for implementing the FBTS and processor. The results are delivered by Quartus II after place & route.

The performance of the FBTS is measured based on the *FBTS active time* and *test time*, whose values are obtained by performance counters included in the FBTS.

FBTS active time is the accumulated time that the FBTS is active during the complete test execution without considering the execution of SW layers or time required for the ATE/FBTS communication. This is an ideal metric to analyze the effect that processor variants and layer partitions have on the test execution because it only considers the time required for the layers implemented in HW and ESW. The *FBTS active time* is measured using a 32-bit performance counter known as *active time counter*.

Two values are calculated based on the content of the *active time counter*. The first value is the *real FBTS active time*. It is obtained by dividing the content of the counter by the operation frequency of the FBTS (50 MHz). The second value is the *minimum FBTS active time*. It is obtained by dividing the content of the counter by the maximal operation frequency (F_{max}) achieved with the FBTS. F_{max} is the value dumped by the TimeQuest timing analyzer (quartus_sta).

Test time is the execution time of the complete test. It includes the *FBTS active time* and time required for the execution of SW layers and for the ATE/FBTS communication. It is measured using a second 32-bit performance counter known as the test time counter. It counts the total number of clock cycles during the test. The test time is calculated by dividing the content of the counter by the operation frequency of the FBTS (50 MHz).

Counters are used instead of theoretical models because they are easy to implement and provide real measurements difficult to obtain with theoretical models. Models would have to consider the execution time of the ATE, processor, and co-processor, as well as the timing behavior of the DUT and ATE/FBTS communication.

7.3 Devices under test

The two DUTs used for the experiments are an SRAM and LCD. As mentioned in Section 3.1.2, the application field of FBT is the detection and diagnosis of defects located at the PCB interconnections and solder joints. For this purpose, a DUT-Ms was developed for each DUT.

The FPGA is also used to inject stuck-at faults, bridging-and faults, and dominant faults (dominant address) in the data and address interconnections. This was necessary in order to validate the correct functionality of the ROBSY test system and DUT-Ms. The activation and selection of a fault is performed through the switches of the DE2-115.

7.3.1 SRAM

The IS61WV102416 SRAM (Integrated Silicon Solution) [109] populating the DE2-115 has the following features:

- 1024Kx16 (16Mbit) high-speed asynchronous static RAM.
- 20 address lines, 16 bidirectional data lines, and active low chip select, output enable, write enable, and upper/lower byte control lines.
- 10 ns write and read cycle times.

The DUT-M describes the interface of the IS61WV102416, write and read access functions (L1), and the procedures for the detection and diagnosis of stuck-at, bridging, and address-dominant faults (L5-L2). The DUT-M includes four global variables that represent two-dimensional pass/fail fault dictionaries (Section 2.1.5). These variables are declared as vectors, in which the first index represents the width of each element and the second index the number of elements. The vectors fd_dataBus_bridge[32][120] and fd_addrBus_bridge[42][190] store pass/fail signatures for diagnosis of bridging-and faults at the data and address buses, respectively. fd_addrBus_dominant_stuckat[42][20] and fd_addrBus_dominant[32][16] store pass/fail signatures for diagnosis of stuck-at faults at the address bus and dominant faults between the address and data buses.

Table 7.2 shows a summary of the procedures found in the DUT-M. The first column represents the layer to which the procedure belongs. The second and third columns show name of the procedure and arguments with their type (IN, OUT, INOUT), respectively. The fourth column indicates the local variables of each procedure.

Layer	Procedure	Arguments	Local variables
L1	write_pattern()	IN data[16], IN addr[20]	-
	read_pattern()	OUT data[16], IN addr[20]	-
L2	single_pattern()	IN wr_rd[2], IN data[16], IN addr[20], OUT data_read[16], OUT error[1]	vData_exp[16]
	seq_pattern()	IN wr_rd[2], IN data_op[3], IN data[16], IN addr_op[2], IN addr[20], OUT error[1]	vError[1], vError_tmp[1], vData_val[16], vAddr_val[20], vIndex[5], vData_dummy[16]
	seq_pattern_d42()	IN wr_rd[2], IN data_op[2], IN data[16], IN addr_op[2], IN addr[20], OUT signa[42]	vError[1], vData_val[16], vAddr_val[20], vIndex[5], vData_dummy[16]
	seq_pattern_d32()	IN wr_rd[2], IN data_op[3], IN data[16], IN addr_op[2], IN addr[20], OUT signa[32]	vError[1], vData_val[16], vAddr_val[20], vIndex[5], vData_dummy[16]
	single_pattern_d42()	IN wr_rd[2], IN data[16], IN addr[20], OUT signa_data[42]	vError[1], vData_dummy[16]
L3	test_detection()	OUT error[1]	vError[1], vError_tmp[1], vData[16], vAddr[20]
	test_diagnosis()	OUT fault_code[3], OUT data[16], OUT signa32[32], OUT signa42[42]	vError[1], vSigna32[32], vSigna42[42], vSigna_data[16]
	dataBus_sutckat()	OUT error[1], OUT signa[16]	vError[1], vData[16], vAddr[20], vData_read[16]
	addrBus_dominant()	OUT signa[42]	vError[1], vData[16], vAddr[20], vSigna[42]
	dataBus_bridge()	OUT signa[32]	vError[1], vData[16], vAddr[20], vSigna[32]
	addrBus_bridge_ stuckat()	OUT signa[42]	vError[1], vData[16], vAddr[20], vSigna[42]
L4	coordination()	OUT error_code[3], OUT signa_data[16], OUT index_fdict1[8], OUT index_fdict2[8]	vError[2], vError_code[3], vSigna32[32], vSigna42[42], vSigna_data[16], vIndex_fdict1[8], vIndex_fdict2[8]
	addrBus_stuckat_ bridge_analysis()	IN signa[42], OUT error_code[3], OUT index_fdict[8]	vError:code[3], vIndex_fdict[8], vIndex_result[8], vData[16], vAddr[20], vSigna[42]
	dataBus_bridge_ analysis()	IN signa[32], OUT index_fdict[8]	vIndex_fdict[8], vIndex_result[8], vSigna[32]
	addrBus_dominant_ analysis()	IN signa42[42], IN signa32[32], OUT index_fdict1[8], OUT index_fdict2[8]	vIndex_fdict[8], vIndex_result1[8], vIndex_result2[8], vSigna42[42], vSigna32[32]
L5	sram_test	-	vError_code[3], vSigna_data[16], vSigna32[32], vSigna42[42], vIndex_fdict1[8], vIndex_fdict2[8]

Table 7.2: SRAM DUT-M (fault diagnosis) properties

L1 procedures write_pattern() and read_pattern() describe write and read access functions taking into account the timing properties of the DUT. The two single_pattern procedures of L2 forward deterministic patterns between L3 and L1, and the three seq_pattern procedures generate walking-1/walking-0/increment/decrement test patterns. Procedures without a _d# suffix are used for fault detection, whereas procedures with a _d# suffix are used for fault diagnosis.

L3 procedure test_detection() describes the structure of the fault detection algorithm. On the other hand, test_diagnosis() and remaining L3 procedures describe the structure of the fault diagnosis algorithm. L4 procedure coordination() manages the detection and diagnosis algorithms. In the case that the detection algorithm does not find any fault, there is no need to execute the diagnosis algorithm. The remaining L4 procedures are called by coordination() to match the obtained pass/fail signatures with signatures stored in the fault dictionaries. L5 procedure sram_test() starts the test execution and reports test results. The report informs if the test was successful or if a fault was detected and the location of the fault.

7.3.2 LCD

The DIP128-6 LCD (Electronic Assembly) [152] and LCD controller (6) populate the VarioTap Coach. The LCD controller is implemented in a complex programmable logic device that receives commands and data from the FPGA and sends control and image information to the LCD. In comparison to the SRAM, the LCD is a slower device and has fewer pins. The features of the LCD and LCD controller are:

- 128x64 dots.
- 3 address lines, 8 data lines, and active low chip select, output enable, and write enable control lines.
- 2 µs write cycle time.

The DUT-M describes the interface to the LCD controller, write access functions (L1), and procedures required to initialize the LCD and generate a square pattern that moves through the screen (L5-L2). This DUT-M does not use any global variables.

Table 7.3 shows the summary of procedures found in the DUT-M. In comparison to the SRAM DUT-M, it has a lower number of procedures variables and arguments. Additionally, the size of procedures and variables is smaller.

Layer	Procedure	Arguments	Local variables
L1	write_pattern()	IN data[8], IN addr[3]	-
L2	single_pattern()	IN data[8], IN addr[3]	-
	scan_pattern()	IN screen_side[1], IN pattern_val[8], IN empty_val[8]	vScreen_side_command[1], vScreen_side_addr[3], vIndex_1[8], vIndex_2[8], vIndex_3[8], vPage_value[8]
L3	test_pattern()	-	vData[8], vAddress[3]
L4	coordination()	-	-
L5	lcd_test	-	-

Table 7.3: LCD DUT-M (LCD test) properties

L1 procedure write_pattern() describes the write access to the LCD controller taking into account timing properties of the DUT. There are no read procedures because it is not possible to acquire data from the LCD controller. As a consequence, the only alternative to evaluate the test is by inspecting the screen during test execution. L2 procedures single_pattern() and scan_pattern() forward deterministic patterns between L3 and L1 and generate all patterns necessary for displaying a square shape that moves through half of the screen. scan_pattern() is the most complex procedure of the DUT-M with three index variables used for loops and arithmetic operations that compute the actual location of the active dot. L3 procedure test_pattern() describes the structure of the test algorithm. It initializes the LCD, configures the LCD controller, and calls scan_pattern() twice in order to test the left and right sides of the screen. L4 procedure coordination() does not provide any additional functionality. L5 procedure lcd_test() starts the test and reports start and end of test execution.

7.4 Resource utilization

7.4.1 Effect of the dependent configuration parameters

The automatic generation process computes the DUT-M and ASM (dependent) configuration parameters (Table 6.1). In this way, the instruction set, number of GPRs, size of memories, and size of the I/O address space are tailored to the layers implemented in ESW and procedure arguments belonging to the highest layer implemented in HW. The effect of automatically assigning values to these parameters is:

- The processor's instruction set, GPRs, memories, and Wishbone interface are implemented using a minimum amount of resources.
- The dependent configuration parameters are computed without affecting the CPI values of the processor.
- The number of configuration parameters that have to be considered by the test engineer is reduced to the variable (independent) configuration parameters.

Table 7.4 and Table 7.5 show the dependent configuration parameter values computed for both DUT-Ms if layers L4-L2 are implemented in ESW. The first column is the index I representing the processor variant (Table 7.1), and the remaining columns show the value of the dependent configuration parameters. For the sake of clarity, values of the first row are considered reference values. Differences between this row and other rows are highlighted in bold font. The *instruction_enable* column shows only the differences. A '-'

prefix represents an instruction found in the first row but not included in the processor variants of the given row. A '+' prefix represents the opposite case.

I	*data_ mem_ depth*	*stack_ mem_ depth*	*prog_ mem_ depth*	*IO_ addr_ num*	*GPRs _num*	*instruction_enable*
$I_{8.0}$-$I_{8.7}$	11	5	12	2040	10	JMP, JC, JZ, JNZ, CALL, CALL_C, RET, RET_C, STOP, NOP, LOAD, STORE, PUSH, POP, ADD, ADDI, ADDC, ADDIC, SUB, SUBI, SUBIC, AND, ANDI, OR, ORI, NOR, SHL, SHR, SHLC
$I_{12.0}$-$I_{12.7}$	11	**4**	12	2040	**7**	**-ADDC**
$I_{16.0}$-$I_{16.7,}$ $I_{20.0}$-$I_{20.7}$	**10**	**4**	12	**1016**	**6**	**-ADDC, -ADDIC, -SUBIC, +ROL**
$I_{24.0}$-$I_{24.7,}$ $I_{28.0}$-$I_{28.7,}$ $I_{32.0}$-$I_{32.7}$	**10**	**4**	12	**1016**	**5**	**-ADDC, -ADDIC, -SUBIC**
$I_{8.8}$-$I_{8.15}$	11	5	**14**	2040	10	**+ADDIU, +SUBIU, +SUBC, +ORH**
$I_{12.8}$-$I_{12.15}$	11	**4**	12	2040	**7**	**-ADDC, +ADDIU, +ORH**
$I_{16.8}$-$I_{16.15,}$ $I_{20.8}$-$I_{20.15}$	**10**	**4**	12	**1016**	**6**	**-ADDC, -ADDIC, -SUBIC, +ADDIU, +ORH, +ROL**
$I_{24.8}$-$I_{24.15,}$ $I_{28.8}$-$I_{28.15,}$ $I_{32.8}$-$I_{32.15}$	**10**	**4**	12	**1016**	**5**	**-ADDC, -ADDIC, -SUBIC, +ADDIU, +ORH**

Table 7.4: SRAM DUT. Dependent configuration parameters (L4-L2)

I	*data_ mem_ depth*	*stack_ mem_ depth*	*prog_ mem_ depth*	*IO_ addr_ num*	*GPRs _num*	*instruction_enable*
$I_{8.0}$-$I_{8.7}$, $I_{12.0}$-$I_{12.7}$, $I_{16.0}$-$I_{16.7,}$ $I_{20.0}$-$I_{20.7}$, $I_{24.0}$-$I_{24.7,}$ $I_{28.0}$-$I_{28.7,}$ $I_{32.0}$-$I_{32.7}$	5	4	9	24	4	JMP, JC, JZ, JNZ, CALL, CALL_C, RET, RET_C, STOP, NOP, LOAD, STORE, PUSH, POP, ADD, ADDI, SUB, SUBI, SUBIC, ANDI, ORI
$I_{8.8}$-$I_{8.15}$, $I_{16.8}$-$I_{16.15}$, $I_{20.8}$-$I_{20.15}$, $I_{24.8}$-$I_{24.15,}$ $I_{28.8}$-$I_{28.15,}$ $I_{32.8}$-$I_{32.15}$	5	4	9	24	4	**+ADDIU, -SUBIC, +OR, +ORH**
$I_{12.8}$-$I_{12.15}$	5	4	9	24	4	**+ADDIU, -SUBIC, +AND, +OR, +ORH**

Table 7.5: LCD DUT. Dependent configuration parameters (L4-L2)

The comparison of Table 7.4 and Table 7.5 show that processor variants generated based on the SRAM DUT-M have larger memories, a more variable instruction set, and higher number of GPRS and I/O addresses. For example, the SRAM processor variants use 26-33 from the 50 instructions available, whereas the LCD processor variants use 21-24 instructions. This is due to the higher diversity of operations found in the SRAM DUT-M in comparison to the LCD DUT-M.

Both tables also show that variable (independent) configuration parameters *data_width* and *short_imm_set* are responsible of changes of the dependent configuration parameter values. The reason for this is that these two parameters affect the properties of the ISA, while the other variable (independent) configuration parameters only affect the microarchitecture. This dependency is more visible in Table 7.4 due to the higher complexity of the SRAM DUT-M.

In Table 7.4, there is a decrease of dependent configuration parameter values *data_mem_depth*, *stack_mem_depth*, *prog_mem_depth*, *IO_addr_num*, and *GPRs_num* as *data_width* increases. The reason of this is that a higher *data_width* makes it possible to manage variables and procedure arguments in a more efficient way. For example, a 32-bit addition with an 8-bit processor requires at least four instructions and four memory locations per operand, whereas the same addition with a 32-bit processor requires a single instruction and a single memory location per operand. This effect is not observed in Table 7.5 because the size of the largest variables and arguments of the LCD DUT-M is 8-bit.

The instruction set also suffers changes based on *data_width*. Table 7.4 shows that addition and subtraction instructions with carry are used only for $I_{8.x}$-$I_{12.x}$. These instructions are necessary to concatenate 16-bit counting sequences in processor variants with a native data type smaller than 16. The appearance of the ROL instruction for $I_{16.x}$-$I_{20.x}$ is due to the way the compiler implements 16-bit and 20-bit walking-0 sequences for the data and address buses. Instead of using a combination of shift and negation instructions, the compiler uses a single ROL instruction for this purpose.

The independent configuration parameter *short_imm_set* produces changes in the instruction set and program memory depth (*prog_mem_depth*). Instructions such as ORH, ADDIU, and SUBIU are used to handle immediate data and address values when *short_imm_set* is true (Equation (5.1)). Table 7.4 shows that the value of *prog_mem_depth* is different for the 8-bit processor variants $I_{8.0}$-$I_{8.7}$ and $I_{8.8}$-$I_{8.15.}$ The reason of this is explained in Section 7.4.2.2.

To conclude, Table 7.4 and Table 7.5 show a proof of the adaptation of the ROBSY processor to a given layer partition, and that the variable (independent) configuration parameters *data_width* and *short_imm_set* play an important role.

7.4.2 Processor variants

The combination of values for variable (independent) configuration parameters G*PR_mem*, *pipeline_set*, *shifter_mult*, *short_imm_set*, and *data_width* produces 112 processor variants that require a different amount of resources. Their effect is explained in the following sections. The exact quantification of the effect of these parameters is not realized, because it depends on the DUT-M, FPGA, and synthesis tool.

7.4.2.1 Effect of G*PR_mem*, *pipeline_set*, and *shifter_mult*

G*PR_mem*, *pipeline_set*, and *shifter_mult* produce changes in the processor microarchitecture only. Therefore, processor variants that differentiate themselves in the

value of these parameters share the same DUT-M and ASM (dependent) configuration parameters. This is observed in Table 7.4 and Table 7.5, in which each row represents groups of processor variants (e.g. $I_{8.0}$-$I_{8.7}$, $I_{8.7}$-$I_{8.15}$, $I_{12.0}$-$I_{12.7}$, etc.).

Figure 7.2 and Figure 7.3 show the resource utilization[5] of all 16-bit processor variants with debug-interfaces ($I_{16.0}$-$I_{16.15}$) if layers L4-L2 are implemented in ESW. The vertical axes represent LEs, M9K blocks, and 9x9 multipliers. The horizontal axis represents the 4-bit vector 'Y' of each processor variant ($I_{16.Y}$). Vertical lines divide the results in four main sectors. Each sector includes four processor variants with same *short_imm_set* and *pipeline_set* values but different *GPR_mem* and *shifter_mult* values.

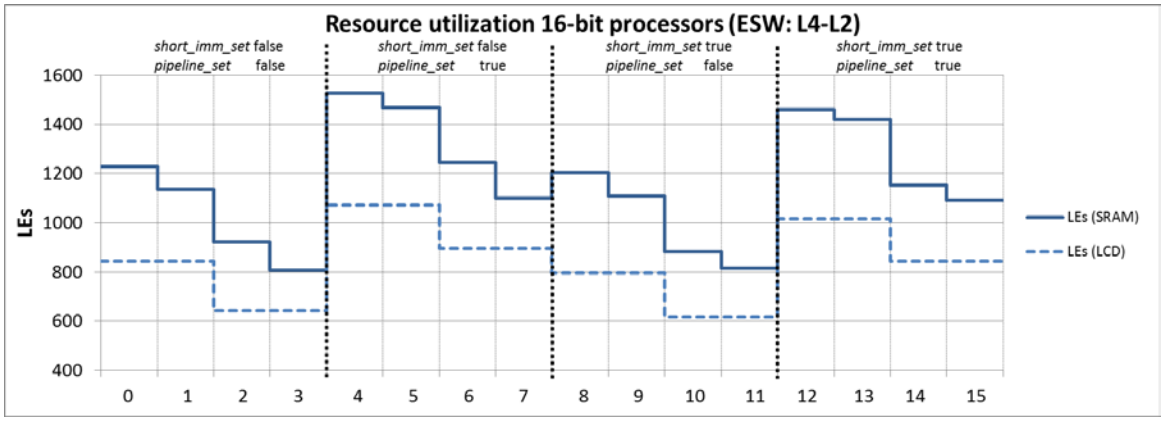


Figure 7.2: Processor logic elements $I_{16.0}$-$I_{16.15}$. SRAM and LCD DUTs (L4-L2)

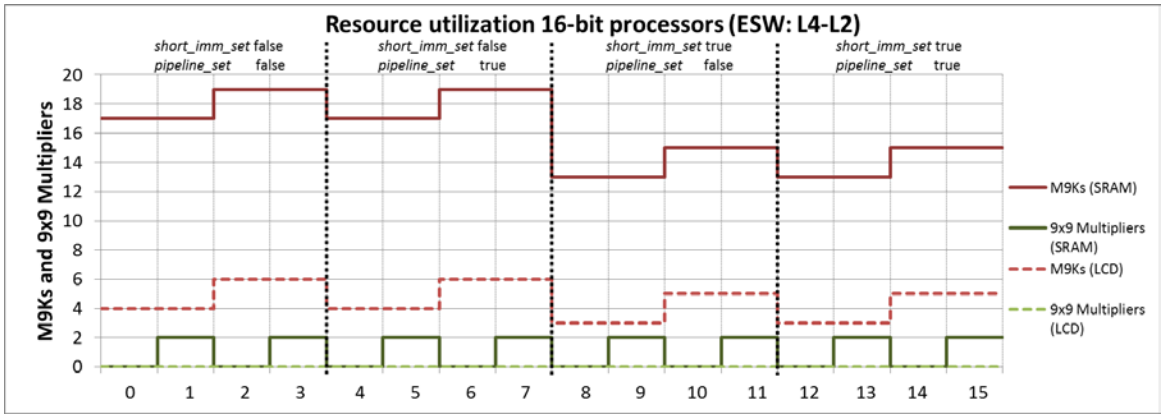


Figure 7.3: Processor M9K blocks and 9x9 multipliers $I_{16.0}$-$I_{16.15}$. SRAM and LCD DUTs (L4-L2)

Figure 7.2 and Figure 7.3 show that processor variants generated based on the SRAM DUT-M require more resources (~400 LEs, ~11 M9K blocks, and ~2 9x9 multipliers). This is due to higher dependent configuration parameter values (Section 7.4.1).

Each sector shows that the activation of *GPR_mem* and *shifter_mult* produces a reduction of LEs and an increase of M9K blocks and 9x9 multipliers. This is the expected behavior

[5] The results are discrete quantities. Connection lines between points are used only for the sake of better clarity. This applies to all results presented in this chapter.

because *GPR_mem* replaces LEs with M9K blocks for the implementation of GPRs, whereas *shifter_mult* replaces LEs with 9x9 multipliers for the implementation of the barrel shifter. The activation of *GPR_mem* produces processor variants with two additional M9K blocks, which are necessary for the implementation of three independent memory ports that can be accessed at the same time. The activation of *shifter_mult* produces processor variants with two 9x9 multipliers that perform 16-bit shifting and rotation operations. An interesting observation is that *shifter_mult* does not produce any changes on the processor variants generated for the LCD DUT. The reason is that the LCD DUT-M does not use any shift or rotation instructions.

The analysis of *pipeline_set* is performed by comparing results of sectors 1 and 2 ($I_{16.0}$-$I_{16.3}$ and $I_{16.4}$-$I_{16.7}$) or sectors 3 and 4 ($I_{16.8}$-$I_{16.11}$ and $I_{16.12}$-$I_{16.15}$). The activation of *pipeline_set* produces a significant increase of LEs and does not affect the M9K blocks or 9x9 multipliers. The growth of LEs is due to the implementation of the pipeline stages, result forwarding and hazard avoidance logic.

Parameters *GPR_mem* and *pipeline_set* produce the higher variation of LEs. The number of M9K blocks varies depending on *GPR_mem*, and 9x9 multipliers are only necessary if *shifter_mult* is true and shift or rotation operations are used in the DUT-M. This behavior persists for all *data_width* values and layer partitions.

7.4.2.2 Effect of *short_imm_set*

The main goal of *short_imm_set* is to reduce the number of M9K blocks. Its activation produces processor variants that code an *imm_data_value* using half of the *data_width* bits (Equation (5.1)). This decreases the number of bits required to code an instruction, and therefore the size of the program memory. This effect is observed in Figure 7.3 by comparing sectors 1 and 3 ($I_{16.0}$-$I_{16.3}$ and $I_{16.8}$-$I_{16.11}$) or sectors 2 and 4 ($I_{16.4}$-$I_{16.7}$ and $I_{16.12}$-$I_{16.15}$). The number of M9Ks blocks is reduced from 17 and 19 to 13 and 15 for the SRAM DUT and from 4 and 6 to 3 and 5 for the LCD DUT.

Table 7.6 shows the memory utilization of the $I_{16.0}$-$I_{16.1}$ (*short_imm_set* false) and $I_{16.8}$-$I_{16.9}$ (*short_imm_set* true) processor variants. Differences are highlighted in bold font.

DUT-M	*short_imm_set*	*Stack memory*		*Data memory*		*Program memory*	
		size	*M9K blocks*	*size*	*M9K blocks*	*size*	*M9K blocks*
SRAM	**false**	16x16	1 (256x32)	1024x16	2 (512x16)	**4096x27**	**14** (4096x2)
	true	16x16	1 (256x32)	1024x16	2 (512x16)	**4096x19**	**10** (4096x2)
LCD	**false**	16x16	1 (256x32)	32x16	1 (512x16)	**512x25**	**2** (512x**16**)
	true	16x16	1 (256x32)	32x16	1 (512x16)	**512x17**	**1** (512x**18**)

Table 7.6: M9K blocks for stack, data, and program memories $I_{16.0}$-$I_{16.1}$/$I_{16.8}$-$I_{16.9}$ (L4-L2)

The size of the stack and data memories remains constant independently of *short_imm_set*. The width of the data memory is *data_width*, and Equation (5.10) delivers the same value for the stack memory. The depth of both memories remains constant (Table 7.4 and Table 7.5). In this case, the synthesis tool uses M9K blocks configured as a 256x32 and 512x16 for the implementation of both memories.

The last column of Table 7.6 shows the dependency of the program memory size based on *short_imm_set*. The depth remains constant because the value computed by the SW generator (Equation (5.5)) is the same for the processor variants $I_{16.0}$-$I_{16.1}$ and $I_{16.8}$-$I_{16.9}$ (Table 7.4 and Table 7.5). On the other hand, there is an eight bit reduction of the memory width if *short_imm_set* changes from false to true (Equation (5.17)). In this case, this reduction is translated to a decrease of four (SRAM) and one (LCD) M9K blocks.

However, a reduction of M9K blocks cannot be guaranteed every time that *short_imm_set* is true. Figure 7.4 presents the M9K blocks utilization of all 8-bit processor variants for the SRAM and LCD DUTs. In this case, the number of M9K blocks increases from 14 and 16 to 43 and 45 for the SRAM DUT-M and remains constant for the LCD DUT-M.

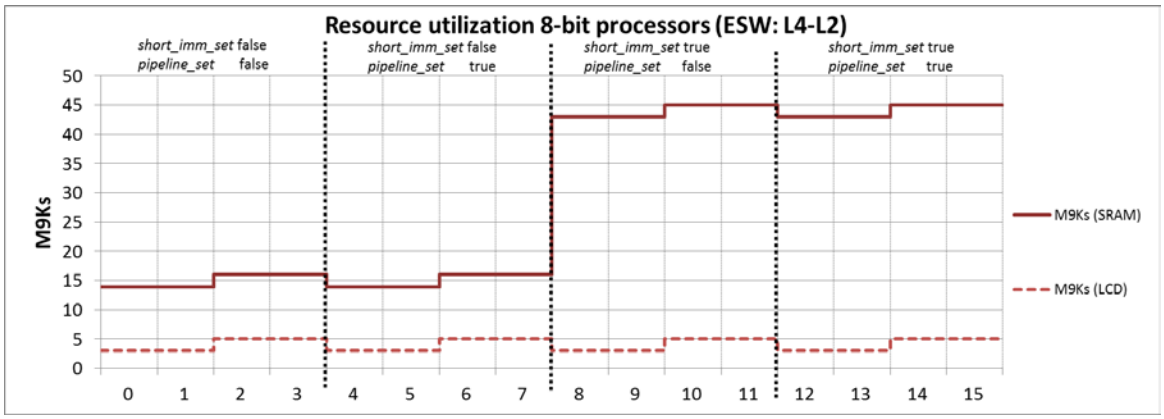


Figure 7.4: Processor M9K blocks $I_{8.0}$-$I_{8.15}$. SRAM and LCD DUTs (L4-L2)

Table 7.7 shows the memory utilization of the $I_{8.0}$-$I_{8.1}$ (*short_imm_set* false) and $I_{8.8}$-$I_{8.9}$ (*short_imm_set* true) processor variants. Differences are highlighted in bold font.

DUT-M	short_imm_set	Stack memory		Data memory		Program memory	
		size	M9K blocks	size	M9K blocks	size	M9K blocks
SRAM	**false**	32x12	1 (256x32)	2048x8	2 (2048x4)	**4096x21**	**11 (4096x2)**
	true	32x14	1 (256x32)	2048x8	2 (2048x4)	**16384x20**	**40 (8192x1)**
LCD	**false**	16x9	1 (256x32)	32x8	1 (512x16)	512x**17**	1 (512x18)
	true	16x9	1 (256x32)	32x8	1 (512x16)	512x**14**	1 (512x18)

Table 7.7: M9K blocks for stack, data, and program memories $I_{8.0}$-$I_{8.1}$/$I_{8.8}$-$I_{8.9}$ (L4-L2)

In this case, the last column of Table 7.7 shows that the program memory width of processor variants generated for the SRAM-DUT-M decreases from 21 to 20 bits, while

the number of memory locations increases from 4096 to 16384. The one-bit instead of the expected four-bit reduction is due to an increase of the program memory depth, which makes it necessary to use more bits to code a program address (Equation (5.17)). The increase of the program memory depth is caused by a high amount of additional instructions produced by the ESW compiler when *short_imm_set* is true. These instructions are necessary to compute and handle addresses of local and global variables given that the 4 bits available to code an *imm_data_value* are not sufficient.

On the other hand, it is not possible to further reduce the number of M9K blocks for processor variants generated based on the LCD-DUT-M.

7.4.2.3 Effect of *data_width*

The parameter *data_width* has the main influence on the resource utilization. Higher *data_width* values produce a steady growth in the number of LEs, M9K blocks, and 9x9 multipliers. This is observed in Figure 7.5 and Figure 7.6, which show the resource utilization of the 112 processor variants including debug-interfaces for the SRAM and LCD DUTs. Vertical lines separate processor variants based on the value of *data_width*.

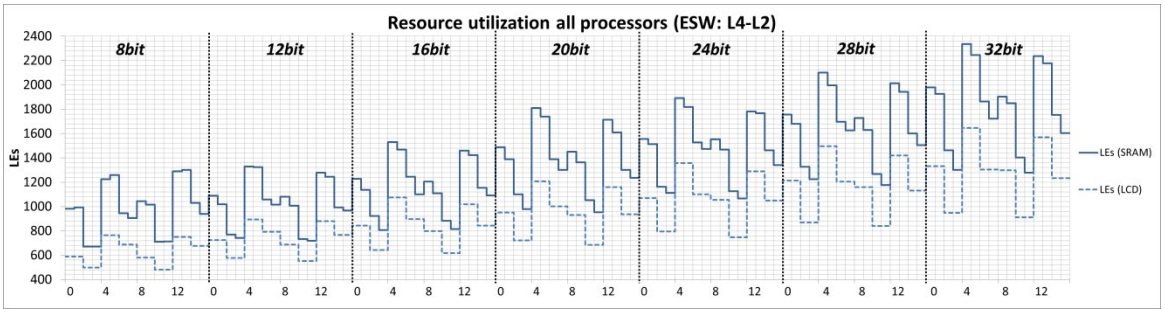


Figure 7.5: Processor logic elements $I_{8,0}$-$I_{32,15}$. SRAM and LCD DUTs (L4-L2)

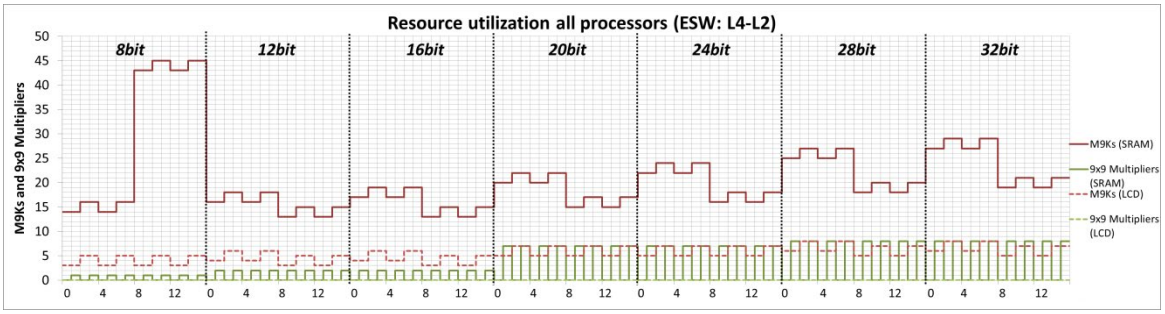


Figure 7.6: Processor M9K blocks and 9x9 multipliers $I_{8,0}$-$I_{32,15}$. LCD DUT (L4-L2)

Figure 7.5 and Figure 7.6 show that an increase in *data_width* produces a steady growth of LEs, M9K blocks, and multipliers. The increase of LEs and M9K blocks takes place although the depth of memories and number of GPRs decreases with higher *data_width* values (Table 7.4). This is caused by the need for wider memories and GPRs. The width of GPRs and data memory is *data_width*, while the width of the stack and program

memories is computed using Equations (5.10) and (5.17). The only exception is a reduction of M9K blocks and LEs of processor variants generated for the SRAM DUT-M if *data_width* changes from 8 to 12. This is caused by *short_imm_set* as already explained in Section 7.4.2.2.

7.4.3 Layer partition

The layer partition defines layers that are implemented in SW, ESW, and HW. It influences the resource utilization of the processor and FBTS because it affects the value of dependent configuration parameters and functionality implemented in a co-processor.

7.4.3.1 Effect of the layer partition on the processor

Layers implemented in ESW affect the computation of DUT-M and ASM (dependent) configuration parameters, producing processor variants that use a different amount of resources. The layer partition with L4-L2 in ESW produces processor variants with the highest resource utilization because other partitions have fewer layers implemented in ESW.

Table 7.8 presents the dependent configuration parameter values computed for the SRAM DUT-M if L3 is implemented in ESW. The comparison of Table 7.8 and Table 7.4 (L4-L2 in ESW) shows a significant reduction in the number of instructions, I/O addresses, number of GPRs, and depth of the data, stack, and program memories. Additionally, the processor variants of Table 7.8 do not include shift or rotation instructions, and therefore they do not use 9x9 multipliers independently of the *shifter_mult* value.

I	*data_ mem_ depth*	*stack_ mem_ depth*	*IO_ addr_ num*	*GPRs _num*	*prog_ mem_ depth*	*instruction_enable*
$I_{8.0}$-$I_{8.7}$ $I_{12.0}$-$I_{12.7}$	6	3	184	5	12	JMP, JZ, JNZ, CALL, CALL_C, RET, RET_C, STOP, NOP, LOAD, STORE, PUSH, POP, ADDI, SUBI, ANDI, OR, ORI, NOR
$I_{16.0}$-$I_{16.7}$, $I_{20.0}$-$I_{20.7}$	6	3	**56**	**4**	12	
$I_{24.0}$-$I_{24.7}$, $I_{28.0}$-$I_{28.7}$, $I_{32.0}$-$I_{32.7}$	**5**	3	**88**	**4**	**11**	
$I_{8.7}$-$I_{8.15}$	6	3	184	5	13	**-ADDI, +ADDIU, +SUBIU, +ORH**
$I_{12.7}$-$I_{12.15}$	6	3	184	5	12	**-ADDI, +ADDIU, +ORH**
$I_{16.7}$-$I_{16.15}$	6	3	**56**	**4**	12	**-ADDI, +ADDIU, +ORH**
$I_{20.7}$-$I_{20.15}$	6	3	**56**	**4**	12	**-ADDI, +ADDIU, +ORH**
$I_{24.7}$-$I_{24.15}$, $I_{28.7}$-$I_{28.15}$	**5**	3	**88**	**4**	12	**-ADDI, +ADDIU, +ORH**
$I_{32.7}$-$I_{32.15}$	**5**	3	**88**	**4**	12	**-ADDI, +ADDIU, +ORH**

Table 7.8: Dependent configuration parameters for all processor variants (L3)

Differences in the dependent configuration parameters are directly translated to a lower utilization of LEs, M9K blocks, and 9x9 multipliers as observed in Figure 7.7, Figure 7.8, and Figure 7.9. These results show the power of the adaptation mechanism.

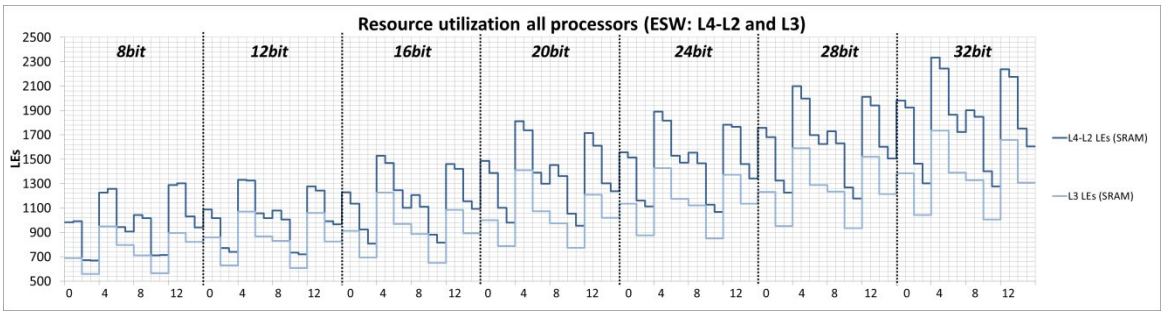


Figure 7.7: Processors logic elements $I_{8.0}$-$I_{32.15}$. SRAM DUT (L4-L2 and L3)

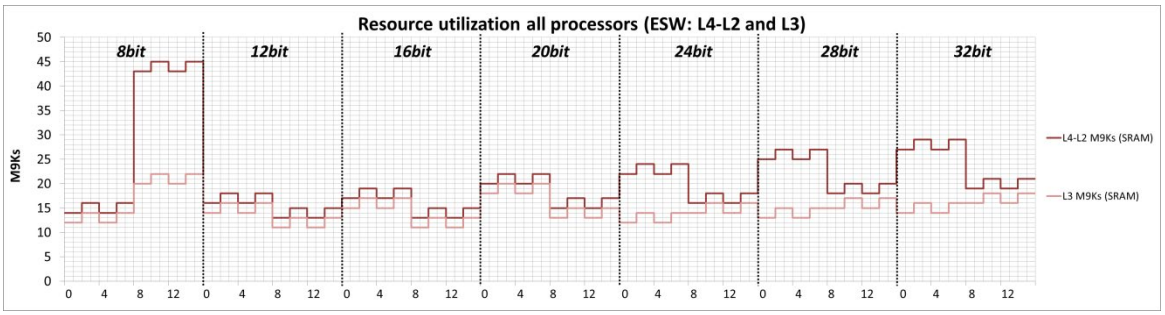


Figure 7.8: Processors M9K blocks $I_{8.0}$-$I_{32.15}$. SRAM DUT (L4-L2 and L3)

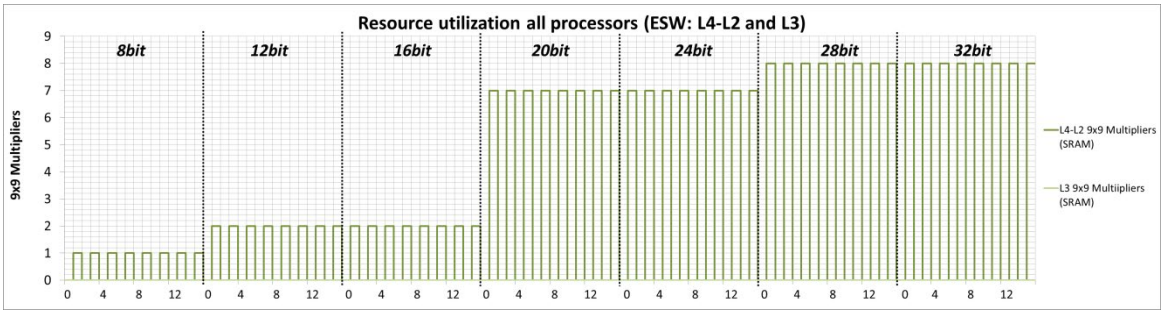


Figure 7.9: Processors 9x9 Multipliers $I_{8.0}$-$I_{32.15}$. SRAM DUT (L4-L2 and L3)

7.4.3.2 Effect of the layer partition on FBTS

The FBTS comprises the processor, co-processor, and performance counters. In this dissertation, co-processors and performance counters do not affect the utilization of M9K blocks or 9x9 multipliers. They are implemented using LEs and represent a positive offset added to the LEs of each processor variant. This offset remains relatively constant unless the processor *data_width* or the layer partition is modified. Changes in *data_width* makes it necessary to adapt the processor/co-processor interface (size and number of Wishbone registers), and changes in the layer partition might alter layers implemented in HW.

Figure 7.10 shows the average LE utilization of the FBTS for the SRAM and LCD DUTs. The average is calculated based on the LE utilization of the 112 FBTS variants. The horizontal axis represents layers implemented in ESW.

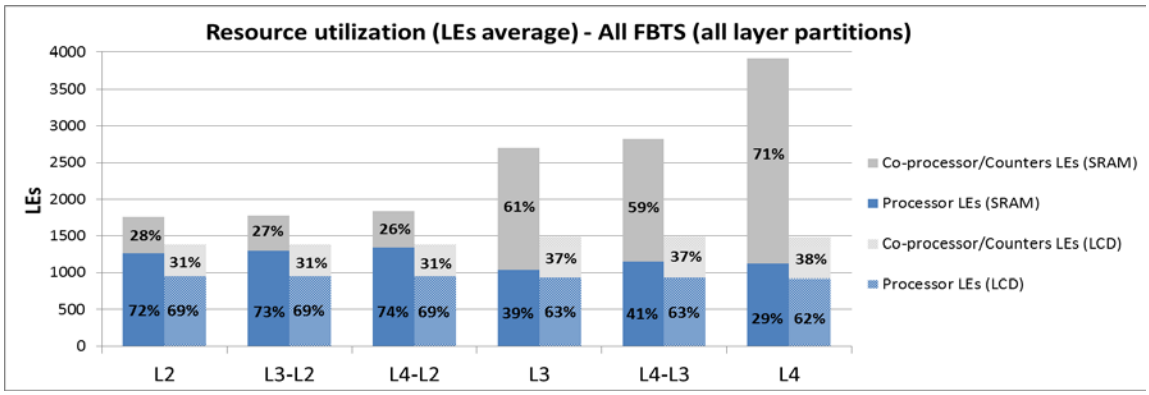


Figure 7.10: FBTS logic elements (average). SRAM and LCD DUTs (all layer partitions)

The main observations for the SRAM DUT based on Figure 7.10 are:

i. The implementation of additional layers in HW produces a significant increase of LEs, which are required for the implementation of the co-processor.
ii. The implementation of additional layers in ESW produces a lower increase of LEs in comparison to the implementation of additional layers in HW.
iii. The total LEs used for layer partitions that implement the same layers in the FBTS (same location of I2) is smaller when more layers are implemented in ESW.

Observation (i) is based on the comparison of columns L2, L3-L2, L4-L2 (L1 in HW) to columns L3, L4-L3 (L2-L1 in HW) or to column L4 (L3-L1 in HW) for the SRAM. The complexity of layers L1, L2, and L3 of the SRAM DUT-M in terms of number and size of procedures, variables, and operations makes it necessary to use a high amount of LEs when these layers are implemented in HW.

Observation (ii) shows that the LE utilization of the processor increases at a lower rate in comparison to the LE utilization of the co-processor as additional layers are implemented in ESW or HW. This is evident by looking at the LE utilization of the processor for layer partitions L2, L3-L2, and L4-L2, or for layer partitions L3 and L4-L3. Therefore, in terms of LEs the processor is an efficient mechanism to implement additional layers of the SRAM DUT-M in the FBTS.

Observation (iii) is a consequence of observation (ii). This is evident when comparing the LE utilization of the FBTS for layer partitions L4, L4-L3, and L4-L2, or of L3 and L3-L2. These layer partitions implement the same layers in the FBTS, and show that implementing more layers in ESW requires less LEs.

In contrast to the SRAM DUT-M, the LE utilization of the FBTS generated for the LCD DUT-M presents a different behavior. There is a lower overall utilization of LEs with small changes between layer partitions. This is caused by the lower complexity of the LCD DUT-M, in which only L1 and L2 describe most of the test functions. In this case, it can even make sense to implement all layers in HW. The reason of this is that the processor consumes more than 60% of the LEs even if L4 is the only layer implemented in ESW.

7.4.4 FPGA capacity

All FBTS variants used for the experiments fit in any FPGA from the Altera Cyclone IV-E family. This is observed in Figure 7.11, which presents the utilization percentage of LEs, M9K blocks, and multipliers of the FBTS and processor variants for different FPGAs. The percentages are obtained based on the resource utilization average of all FBTS variants generated for the SRAM DUT-M if layers L4-L2 are implemented in ESW. This layer partition produces processor variants with the highest utilization of resources.

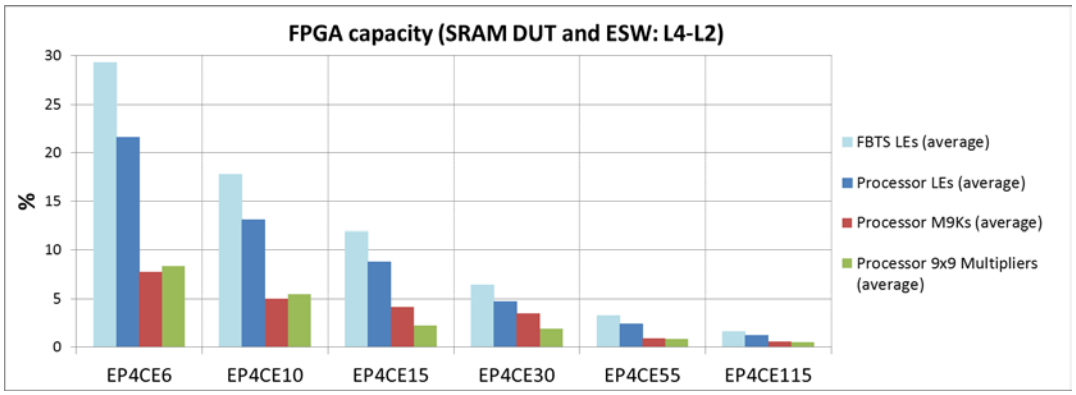


Figure 7.11: FPGA capacity. SRAM DUT (L4-L2)

The EP4CE6 and EP4CE115 are the FPGAs with the lowest and highest capacity, respectively. The capacity of Cyclone IV-E FPGAs is [151]:

FPGA	LEs	M9K blocks	9x9 Multipliers
EP4CE6	6272	270	30
EP4CE10	10320	414	46
EP4CE15	15408	504	112
EP4CE30	28848	594	132
EP4CE55	55856	2340	308
EP4CE115	114480	3888	532

Figure 7.11 shows that it is possible to include at least three similar FBTS variants in the EP4CE6 and at least 20 similar FBTS variants in the EP4CE115. This makes it possible to implement a multi-co-processor or multi-domain FBTS even in Cyclone-IV E FPGAs

of low capacity. Of course, the exact number of processors and co-processors included in the FPGA depends on the properties of the PCB, DUT-Ms, and selected layer partitions. Figure 7.11 also shows that LEs are the FPGA resources with the highest. Therefore, configuration parameters such as *GPR_mem* and *shifter_mult* are very useful to reduce their use.

7.5 FBTS active time

FBTS active time represents the total test time if delays caused by the execution of SW layers and ATE/FBTS communication are not considered. Therefore, it is an ideal mean to show the effect of the processor configuration and ESW/HW partition. It is measured considering the longest execution path of test algorithms. For this purpose, it was necessary to inject a stuck-at fault at address line 0 of the SRAM. For the LCD, it was not necessary to inject any faults because execution time does not change in the presence of faults.

7.5.1 Processor variants

The effect of the variable (independent) configuration parameters (G*PR_mem*, *pipeline_set*, *shifter_mult*, *short_imm_set*, and *data_width*) on the FBTS active time is presented in the following sections.

7.5.1.1 Effect of G*PR_mem*, *pipeline_set*, *shifter_mult,* and *short_imm_set*

Figure 7.12 and Figure 7.13 show real and minimum FBTS active time values for both DUT-Ms if L4-L2 are implemented in ESW and *data_width* is equal to 16. In the same way as the resource utilization, the results present a repetitive shape in each sector.

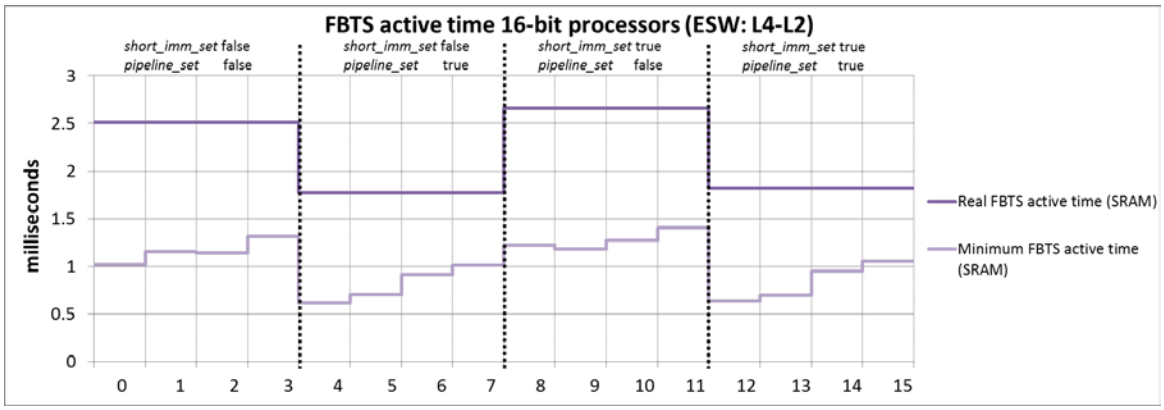


Figure 7.12: FBTS active time $I_{16.0}$-$I_{16.15}$. SRAM DUT (L4-L2)

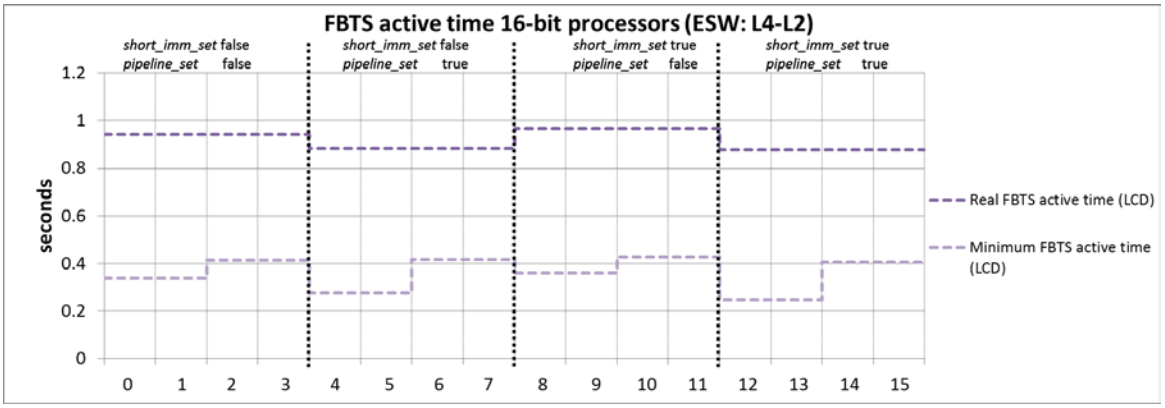


Figure 7.13: FBTS active time $I_{16.0}$-$I_{16.15}$. LCD DUT (L4-L2)

Figure 7.12 and Figure 7.13 show that FBTS active time for the LCD DUT-M is longer than FBTS active time for the SRAM DUT-M. This is due to the longer access time of the LCD controller (µs range) in comparison to the SRAM (ns range).

Both figures also show that real FBTS active time remains constant in each sector. The reason of this is that independent configuration parameters *GPR_mem* and *shifter_mult* only alter the implementation of GPRs and shifter without changing their clock cycle behavior. Therefore, real FBTS active time values exclusively depend on *short_imm_set* and *pipeline_set*.

The activation of *pipeline_set* produces a reduction of real FBTS active time typically in the range of 25%-35%. The activation of *short_imm_set* produces a slight increase, which is typically in the range of 0%-5%. This increase is caused by additional instructions used to handle immediate data and address values (*imm_data_value*). However, *short_imm_set* also can produce a significant increment of FBTS active time in certain cases. If there is a significant increase of instructions produced by the compiler when *short_imm_set* is true (Section 7.4.2.2), the processor has to execute more instructions, and therefore FBTS active time values increase.

Minimum FBTS active time is influenced by all four independent configuration parameters. In this case, the activation of *GPR_mem* or *shifter_mult* produces an increase of this metric. This means that the processor critical path gets affected when embedded blocks are used for the implementation of the GPRs and shifter. For the LCD DUT, *shifter_mult* does not produce any changes given that shift or rotate instructions are not used. There is a reduction of the minimum FBTS active time when *pipeline_set* is true, whereas *short_imm_set* does not produce any significant changes.

The behavior of the minimum FBTS active time metric is not further discussed in the following sections because it behaves in a similar way for all *data_width* values and layer partitions.

7.5.1.2 Effect of *data_width*

Figure 7.14 and Figure 7.15 show real FBTS active time values for both DUTs and the 112 processor variants.

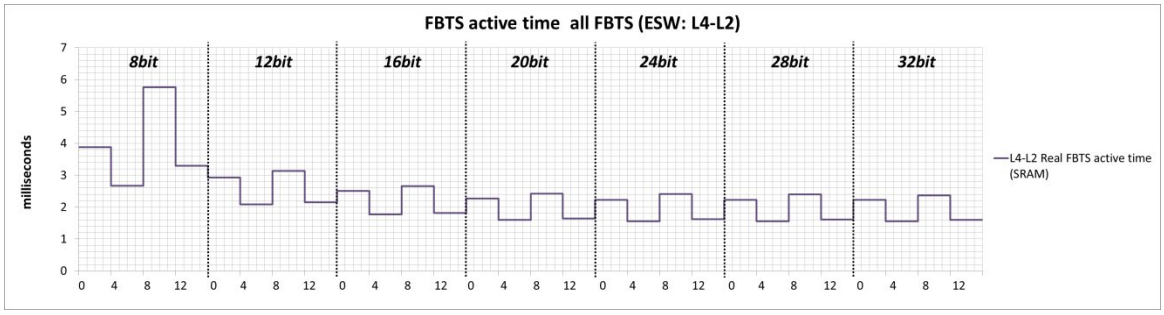


Figure 7.14: FBTS active time $I_{8.0}$-$I_{32.15}$. SRAM DUT (L4-L2)

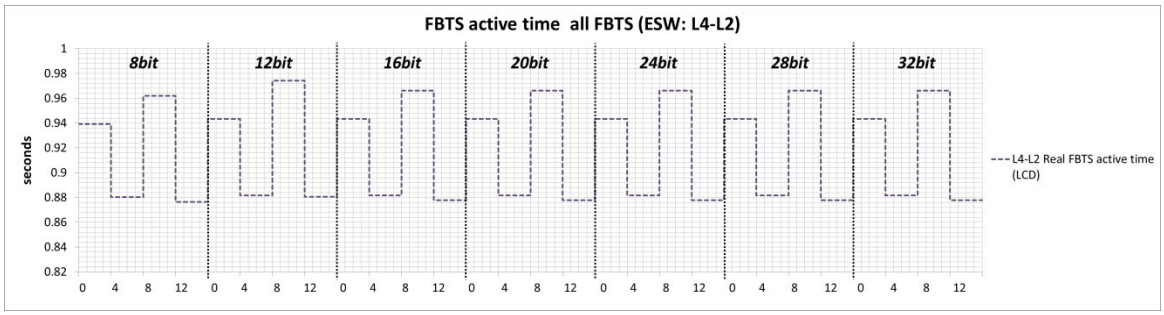


Figure 7.15: FBTS active time $I_{8.0}$-$I_{32.15}$. LCD DUT (L4-L2)

They show that 8-bit, 12-bit, 20-bit, 24-bit, 28-bit, and 32-bit processor variants behave in the same way as the 16-bit processor variants. Parameters *pipeline_set* and *short_imm_set* affect real FBTS active time values, while *GPR_mem* and *shifter_mult* do not.

The SRAM results show that an increase of *data_width* produces a gradual reduction of FBTS active time until a certain *data_width* value, which is known as the *break value*. After the break value is reached, FBTS active time remains relatively constant in comparison to changes obtained before the break value. For the SRAM DUT, the break value is 20-bits. The reductions before the break value are in the range of 9%-45%. These reductions are caused by a more efficient mapping of the DUT-M variables and procedure arguments in the processor ISA. Variables larger than *data_width* require multiple instructions for the storage and execution of arithmetic or logic instructions. Therefore, increasing *data_width* reduces the number of instructions necessary to handle these variables. For example, 8-bit processor variants require three GPRs and three memory locations for the storage of vAddr_shift[20] (Table 7.2). This means that at least three instructions are necessary for the execution of load, store and shift operations. On the other hand, 20-bit processor variants require a single instruction for this variable.

Reductions obtained after *data_width* surpasses the break value are smaller (1%-2%). This is due to a low use of variables larger than 20 bits. In the SRAM DUT-M, these

variables are rarely found inside loops and are not found in L2 procedures, which are the more frequently executed procedures.

The LCD results show no gradual reduction of real FBTS active time if *data_width* increases. This means that the break value is equal to 8. This is as expected because variables and procedure arguments of the LCD DUT-M have a maximum width of 8 bits (Table 7.3). The 1% increase of the FBTS active time is due to additional masking instructions necessary to handle variables if *data_width* is larger than the variable width.

7.5.2 Layer partition

The layer partition influences FBTS active time because it defines the layers that are implemented in ESW, and HW. Sections 7.5.2.1 and 7.5.2.2 present the real FBTS active time results for all layer partitions. Both sections discuss differences between layer partitions and the behavior of the break value.

7.5.2.1 FBTS active time for SRAM DUT-M

Figure 7.16 shows real FBTS active time values for the SRAM DUT-M and all layer partitions. Results show strong differences between layer partitions basically due to the layers implemented in ESW.

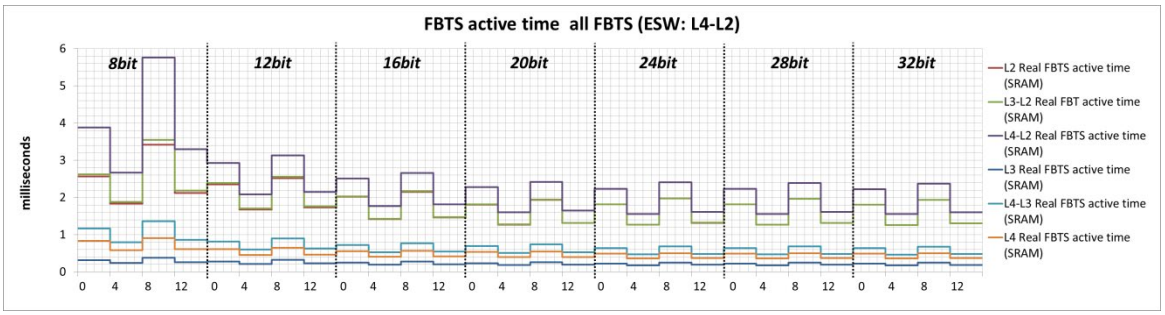


Figure 7.16: FBTS active time $I_{8.0}$-$I_{32.15}$. SRAM DUT (all layer partitions)

The longest real FBTS active time values are obtained for layer partitions that implement L2 in ESW (L2, L3-L2, and L4-L2). This is due to the fact that L1 is the only layer implemented in HW. The overlapping of L2 and L3-L2 results means that the implementation of L3 in ESW does not affect FBTS active time considerably. On the other hand, the implementation of L4 in ESW increases the FBTS active time. The reason is that the signature matching in L4 is a time consuming task.

The results present a significant reduction if L2 is implemented in HW (L3, L4-L3, or L4 in ESW). However, the implementation of L3 in HW does not produce the shortest FBTS active time. In this case, the implementation of L3 in ESW produces the shortest FBTS

active time. The reason is that L4 is implemented in SW, and therefore its execution time is not considered.

The break value of layer partitions that implement L2, L3-L2, L4-L2, and L3 in ESW is *data_width* equal to 20. Reductions are in the range of 5% to 45% before reaching the break value. After that, FBTS active time values remain relatively constant with small reductions in the range of -1% to 2%. The -1% value represents an increase caused by additional instructions required to mask variables.

On the other hand, the break value of layer partitions that implement L4-L3 and L4 in ESW is 24. In this case, reductions are in the range of 3% to 34% before reaching the break value. After that, FBTS active time values remain relatively constant with changes in the range of 0% to 0.5%. The translation of the break value from 20 to 24 is due to the fact that the execution time required to handle L4 variables becomes more relevant given that L2 is not implemented in ESW. L4 variables are handled more efficiently by 24-bit processor variants because they use two instead of the three instructions required by a 20-bit processor for variables such as vSigna42[42] and vSigna32[32].

7.5.2.2 FBTS active time for LCD DUT-M

Figure 7.17 shows real FBTS active time values for the LCD DUT-M and all layer partitions. The results present a complete different behavior in comparison to the SRAM. There is an overlapping of layer partitions that divides the results into two main groups, and the break value is always eight independently of layers implemented in ESW.

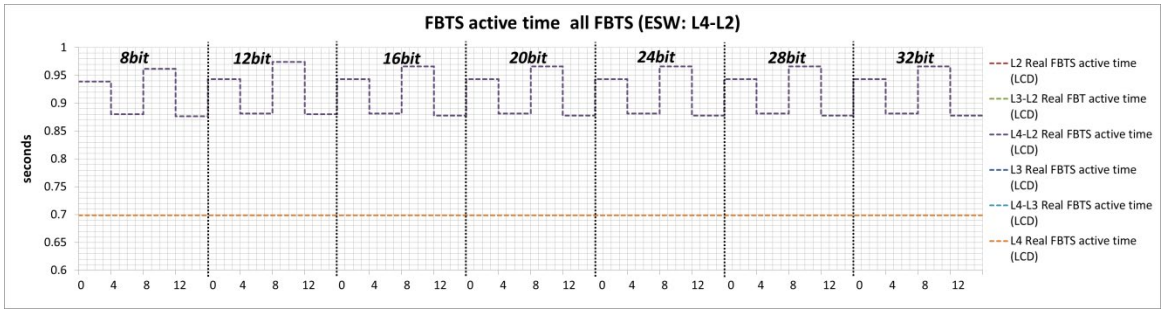


Figure 7.17: FBTS active time $I_{8.0}$-$I_{32.15}$. LCD DUT (all layer partitions)

Layer partitions that implement L2 in ESW (L2, L3-L2, and L4-L2) have the longest FBTS active time values and are overlapped. This is due to a major complexity in terms of number of procedures, operations, and variables of L2 in comparison to L3 and L4. This complexity makes it necessary to spend more time in the execution of L2.

The group with the lowest FBTS active time comprises layer partitions that implement L2 in HW. The results seem to require the same time independently of the independent configuration parameters of the processor. This is not true and is due to the scale of the

figure and the fact that the fraction of time required for the execution of ESW is very short in comparison to the time required for the execution of L1 and L2. These layers are not affected by the independent configuration parameters because they are implemented in HW.

7.6 Test time

Test time includes the real FBTS active time, execution time of SW layers, and time required for the ATE/FBTS communication. This section shows that the results are extremely dependent on the ATE/FBTS communication delays.

7.6.1 ATE/FBTS communication delays

ATE/FBTS communication time depends on the amount of data transfers (DTs) and time required for the execution of each DT. As mentioned in Section 6.3.1, the ATE FBTS communication is performed in six steps that transfer procedure arguments of type IN, OUT and INOUT, perform polling, and read the content of performance counters. Equation (7.1) describes the total number of DTs performed during test execution. x_i and y_i represent DTs required for procedure arguments. p_i represents DTs performed during polling, and c DTs necessary to read the content of a performance counter. The constant value 6 is obtained by adding the constant number of DTs of steps 1 and 5, and the constant value 4 is obtained by adding the constant number of DTs of steps 2 to 4. The summation describes the total DTs performed during test execution, in which m represents the number of procedures of the highest layer implemented in ESW and n_i the number of times that the i-th ESW procedure is called by the ATE.

$$DTs = 6 + c + \sum_{i=1}^{m} n_i \cdot (4 + x_i + y_i + p_i + c) \tag{7.1}$$

A DT typically requires four independent JTAG DR-shift operations. Two DR-shifts exchange data with the debug data register, one DR-shift exchanges data with the debug command register, and one 1-bit DR-shift is used to skip the command shift cycle (Section 5.4.3). For the experiments, the debug data and command registers have the same size (*deb_reg_width*), which is computed as the maximum value of Equations (5.21) and (5.22). Equation (7.2) describes the time necessary to perform a single DT (T_{DT}) based on the test access time (T_{ACC}) of a JTAG transaction (Equation (2.2)) and assuming that there is a single device on the JTAG chain.

$$T_{DT} = 3 \cdot \left(\frac{deb\_reg\_width + 6}{F_{TCK}} + \delta_s + \delta_h\right) + 1 \cdot \left(\frac{1 + 6}{F_{TCK}} + \delta_s + \delta_h\right) \quad (7.2)$$

The first factor of Equation (7.2) represents the T_{ACC} of the three DR-shifts used to exchange information with the debug command and data registers. The second factor represents the 1-bit DR-shift used to skip the command cycle. F_{TCK} is the operation frequency of TCK, and δ_s and δ_h are software and hardware delays that appear during the data transmission. Based on Equations (7.1) and (7.2), the test time (T_{test_time}) is:

$$T_{test\_time} = DTs \cdot T_{DT} + T_{SW} \quad (7.3)$$

T_{SW} is the time required for the execution of the SW layers. In this case, FBTS active time is part of the $DTs \cdot T_{DT}$ factor (polling time).

7.6.1.1 Experimental δ_s and δ_h delays

δ_s and δ_h are the software and hardware delays produced during the transmission of data through the USB-Blaster, respectively. In [30] they are approximated to $1/F_{TCK}$. However, this approximation is not realistic for the USB-Blaster and the quartus_stp tool. Equation (7.4) presents a way to calculate $\delta_s + \delta_h$. It is obtained by substituting Equation (7.2) into Equation (7.3) and isolating $\delta_s + \delta_h$.

$$\delta_s + \delta_h = \frac{1}{4} \cdot \left(\frac{T_{test\_time} - T_{SW}}{DTs} - \frac{3 \cdot debug\_reg\_width + 25}{F_{TCK}}\right) \quad (7.4)$$

T_{SW} is the time needed by quartus_stp to execute SW layer procedures, and is measured using the TCL instruction *time* without considering the ATE/FBTS communication. T_{test_time} is computed based on the test time performance counter and the operation frequency of the FBTS (50MHz). F_{TCK} is the operation frequency of the USB-Blaster (6 MHz). Additional BScan devices are not considered because the JTAG chain of the DE2-115 comprises the Cyclone IV-E FPGA only.

Delays computed for the SRAM and LCD tests vary between *1 and 3 milliseconds*. These values are approximately 10.000 times higher than $1/F_{TCK}$ and confirm that the approximation presented in [30] is not realistic for the USB-Blaster and quartus_stp. The consequence is that the ATE/FBTS communication has a great influence on test time given that $\delta_s + \delta_h$ delays can be longer than the total FBTS active time.

7.6.2 Experimental results

Table 7.9 shows experimental results for the SRAM DUT-M. The processor configuration selected for each layer partition corresponds to a processor variant in the break value ($I_{20.7}$ or $I_{24.7}$). T_{FBTS} and T_{SW} are the real FBTS active time and execution time of SW layers, respectively. The number of DR-shifts is computed by multiplying by four the *DTs* obtained with Equation (7.1). Test time (T_{test}) is calculated based on the content of the test time performance counter and FBTS working frequency (50 MHz).

Layers in ESW	Processor variant	T_{FBTS} (ms)	T_{SW} (ms)	DR-shifts	T_{test} (ms)
L4-L2	$I_{20.7}$	1.599	0.6	76	234.01
L4-L3	$I_{24.7}$	0.469	0.7	76	228.00
L4	$I_{24.7}$	0.362	0.7	76	225.99
L3-L2	$I_{20.7}$	1.274	1.3	120	318.06
L3	$I_{20.7}$	0.183	1.3	120	332.16
L2	$I_{20.7}$	1.270	35.5	1976	4032.34

Table 7.9: Test time of ROBSY test system. SRAM DUT

Table 7.9 shows that T_{FBTS} varies between 0.1 and 1.6 milliseconds depending on the layer partition. However, T_{test} varies between 225 and 4032 milliseconds. This difference is not caused by T_{SW}, it is caused by $\delta_s + \delta_h$ and the amount of DR-shifts. In this case, this means that test time is defined by the number of DR-shifts, and moreover by the location of interface I2 (ATE interface). On the other hand, the location of interface I3 (processor/co-processor interface) or the processor configuration parameters do not have a strong influence on test time.

The longest test time in Table 7.9 corresponds to L2 in ESW given that this partition requires the highest number of DR-shifts. The reason of this is that L2 procedures have a high number of arguments and are called multiple times by L3.

Table 7.10 shows experimental results for the LCD DUT-M. The processor configuration selected for each layer partition corresponds to a processor variant in the break value ($I_{8.4}$). In this case, T_{FBTS} varies between 699 and 880 milliseconds, while T_{test} varies between 784 and 1792 milliseconds.

Layers in ESW	Processor variant	T_{FBTS} (ms)	T_{SW} (ms)	DR-shifts	T_{test} (ms)
L4-L2	$I_{8.4}$	880.4	0.4	512	968.0
L4-L3	$I_{8.4}$	699.0	0.4	424	784.0
L4	$I_{8.4}$	699.0	0.4	424	784.0
L3-L2	$I_{8.4}$	880.4	0.4	512	967.9
L3	$I_{8.4}$	699.0	0.4	424	784.0
L2	$I_{8.4}$	880.4	4.7	920	1792.1

Table 7.10: Test time of ROBSY test system. LCD DUT

In comparison to the SRAM DUT-M, the differences between T_{FBTS} and T_{test} are smaller. In this case, $\delta_s + \delta_h$ and the number of DR-shifts do not influence the results in the same way as for the SRAM. This is due to the long access time of the LCD controller, which produces T_{FBTS} values in the range of hundreds of milliseconds. Additionally, most of the DR-shifts are executed for polling purposes. This explains why the number of DR-shifts is different for layer partitions with the same SW layers (e.g. L3-L2 and L3). In this case, the dependency of test time on the layer partition and the processor configuration parameters is more evident.

7.6.3 FPGA configuration time

The configuration time of the Cyclone IV-E FPGA (EP4CE115F29C7) in JTAG mode is 9 seconds. It remains constant independently of the DUT-M, processor variant, or layer partition. The reason for this is that the size of the configuration file is always the same for a given FPGA if the FPGA configuration is performed in JTAG mode.

The configuration time was not considered as part of the test time analysis because it depends on the specific PCB, ATE, and available configuration methods. Options to reduce the configuration time are increasing the TCK frequency or using data compression methods and other configuration modes [151]. The latter can be used to obtain configuration times below 100 milliseconds. If short test time is the main test requirement, it is necessary to consider high speed configuration methods during the design of the PCB.

7.7 Comparison to other approaches

7.7.1 ROBSY vs Nios II

The Nios II was used to implement three FBTS variants. As already mentioned in Section 2.4.1, the Nios II is a 32-bit proprietary and configurable soft-core processor with 32 GPRs developed and optimized for Altera FPGAs [61]. Three Nios II cores (/e, /s, and /f) are available and they support a common ISA. Table 7.11 shows main properties of the cores used for the experiments.

	Pipeline stages	*Shared memory*	*Instruction cache*	*Data cache*	*Branch prediction*	*JTAG debug interface*
Nios II /e	multicycle	16384 bytes	-	-	-	level 1
Nios II /s	5	16384 bytes	512 bytes	-	static	level 1
Nios II /f	6	16384 bytes	512 bytes	512 bytes	dynamic	level 1

Table 7.11: Properties of the Nios II cores

In the experimental setup, the three cores are connected to a shared memory used for the storage of instructions and data. This memory is configured with the minimum size required for the experiments (Table 7.11). The three FBTS (each one including a Nios II core) are developed based on the SRAM DUT-M. The layers L4-L2 are implemented in ESW, whereas L1 and L5 are implemented in HW and SW, respectively. The SRAM DUT-M and respective layer partition were selected because they produce ROBSY processor variants with the highest resource utilization and longest FBTS active time.

Layers L4-L2 are manually described in C using the same structure of the SRAM DUT-M and compiled with the Nios II software build tools. In order to reduce memory footprint, compilation options *enable_reduced_device_drivers*, *enable_small_c_library*, and *enable_lightweight_device_driver_api* are activated, whereas *enable_c_plus_plus* is deactivated. L1 is implemented with the same co-processor used for the 32-bit ROBSY processor variants (including performance counters). For this purpose, the Wishbone bus is replaced with the Avalon bus of Nios II. L5 is coded manually using the script capabilities of the GNU debugger supported by Nios II.

The ROBSY processor variants selected for the comparison to the Nios II cores are $I_{8.0}$, $I_{32.7}$, and $I_{20.7}$. $I_{8.0}$ is an 8-bit processor without pipeline or embedded blocks (*short_imm_set* false, *pipeline_set* false, G*PR_mem* false, and *shifter_mult* false). $I_{32.7}$ and $I_{20.7}$ are 32-bit and 20-bit processors with pipeline and embedded blocks (*short_imm_set* false, *pipeline_set* true, G*PR_mem* true, and *shifter_mult* true). $I_{8.0}$ and $I_{32.7}$ have the minimum and maximum *data_width* values, and $I_{20.7}$ is located at the break value.

7.7.1.1 Resource utilization

Figure 7.18 and Figure 7.19 show the resource utilization of ROBSY processor and Nios II processor. $I_{8.0}$ consumes less LEs and M9K blocks than the Nios II cores, and it does not make use of any multipliers (same as Nios II/e). However, the LE utilization of $I_{8.0}$ is closed to the LE utilization of Nios II /e, although $I_{8.0}$ is an 8-bit processor and Nios II/e is a 32-bit processor. This is due to the fact that the ISA and microarchitecture of the Nios II cores are optimized for 32 bits and Altera FPGAs. On the other hand, the ROBSY processor is a generic processor designed for different FPGAs (families and vendors) and with data width used as a configuration parameter (*data_width*). This means that it does not have an ISA or microarchitecture optimized for a given data width or FPGA.

Nevertheless, the configurability provided by ROBSY makes it possible to have 32-bit processor variants such as $I_{32.7}$ that consume less LEs than the Nios II/s or /f cores. Still, $I_{32.7}$ consumes four extra M9K blocks and two extra multipliers in comparison to the Nios II/f core. This is also a consequence of the universality of the ROBSY processor. $I_{20.7}$

shows that ROBSY processor can produce results that required even less resources than the Nios II/s and /f and still provide good FBTS active time results (Section 7.7.1.2).

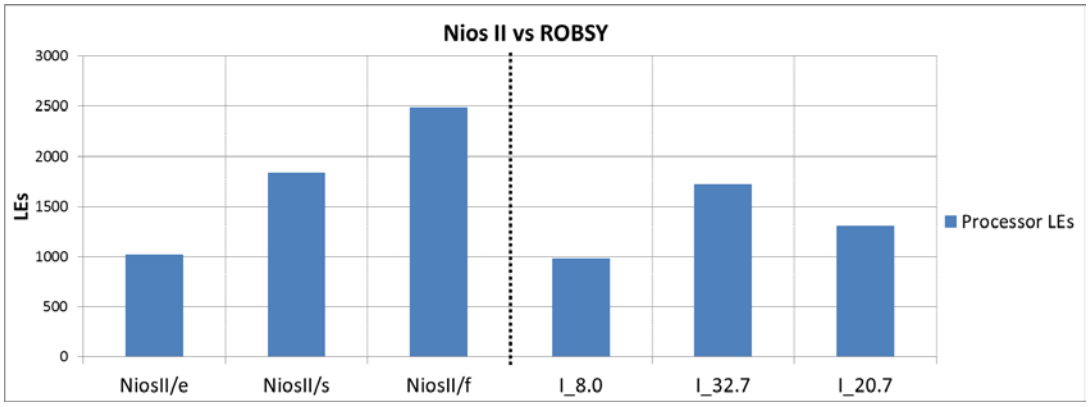


Figure 7.18: Nios II vs ROBSY logic elements. SRAM DUT (L4-L2)

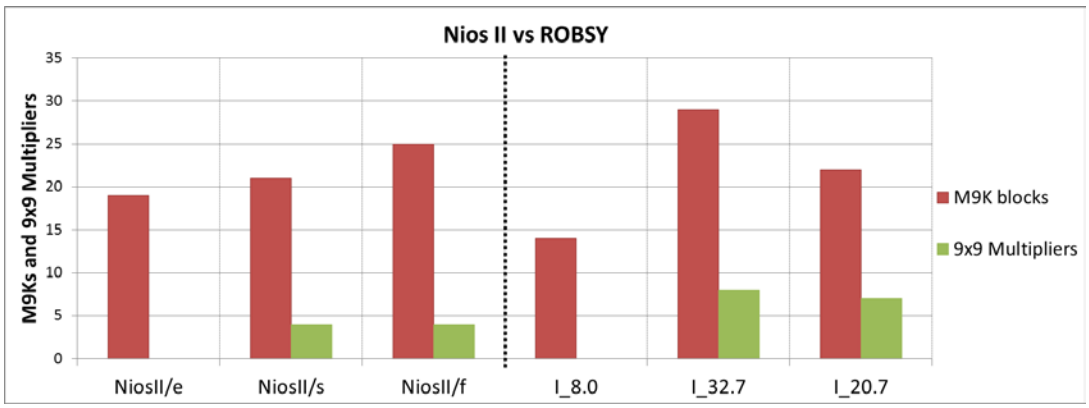


Figure 7.19: Nios II vs ROBSY M9Ks and 9x9 multipliers. SRAM DUT (L4-L2)

To summarize, the resource utilization of the ROBSY processor is in the same range of Nios II. The configuration options provided by ROBSY make it possible to obtain processor variants that are adapted to the given test scenario, and therefore provide a low utilization of resources. This is possible even if the ROBSY processor is not optimized for Altera FPGAs.

7.7.1.2 Performance

Figure 7.20 shows FBTS active time results for ROBSY and Nios II cores. The comparison between these values has to be done considering that different source files (DUT-M for ROBSY and C program for Nios II) and compilers are used to generate the object code. The comparison shows that results obtained with ROBSY are in the same range as results obtained with the Nios II. This takes place even if Nios II cores have more pipeline stages or additional performance mechanisms such as dynamic branch prediction.

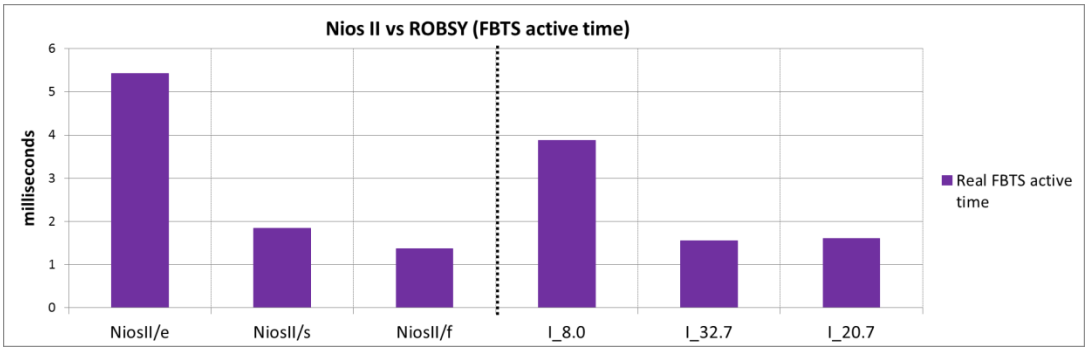


Figure 7.20: Nios II vs ROBSY FBTS active time. SRAM DUT (L4-L2)

Table 7.12 shows test time results of ROBSY and Nios II. In this case, the ROBSY test system requires half of the time to execute the test. It is important to mention that in both cases the results are highly dependent on the number of DTs (Section 7.6). In the case of ROBSY, the number of DTs is mainly determined by the arguments of the DUT-M procedures belonging to the highest layer implemented in ESW. In the case of the Nios II, the number of DTs depends on procedure arguments but also on the operation of the GNU debugger.

Test approach	T_{test} (ms)
ROBSY $I_{8.0}$	261.97
ROBSY $I_{32.7}$	240.00
ROBSY $I_{20.7}$	234.01
Nios II/e	550.02
Nios II/s	560.13
Nios II/f	430.00

Table 7.12: ROBSY vs Nios II test time. SRAM DUT (L4-L2)

7.7.2 ROBSY vs VLSR

In order to compare the ROBSY test system against other FBT techniques, the same test algorithms of SRAM and LCD DUT-Ms were implemented using a virtual length shift register (VLSR) (Section 3.2.2). A VLSR is a virtual BScan register implemented using the FPGA that is dynamically configured to reduce its length based on the number of DUT interconnections. The VLSR used for the comparison to the ROBSY test system relies on the D architecture [30], which uses separate data and mask shift registers. The data register comprises control and I/O cell pairs to drive test patterns and get responses. The mask register dynamically configures the length of the data register. This architecture was chosen because it provides the shortest test time in comparison to other architectures.

The VLSR implemented for the experiments supports 373 I/Os, and therefore consists of a mask register with 373 cells and a data register with 746 cells. A comparison of ROBSY and BScan was not carried out because it would produce results similar to the

VLSR. The resource utilization would be zero and longer test time values would be expected due to a longer BScan chain.

7.7.2.1 Resource utilization

Figure 7.21 shows the LE utilization of the ROBSY FBTS and VLSR. The utilization of M9K blocks and multipliers is not shown because the VLSR does not make use of them.

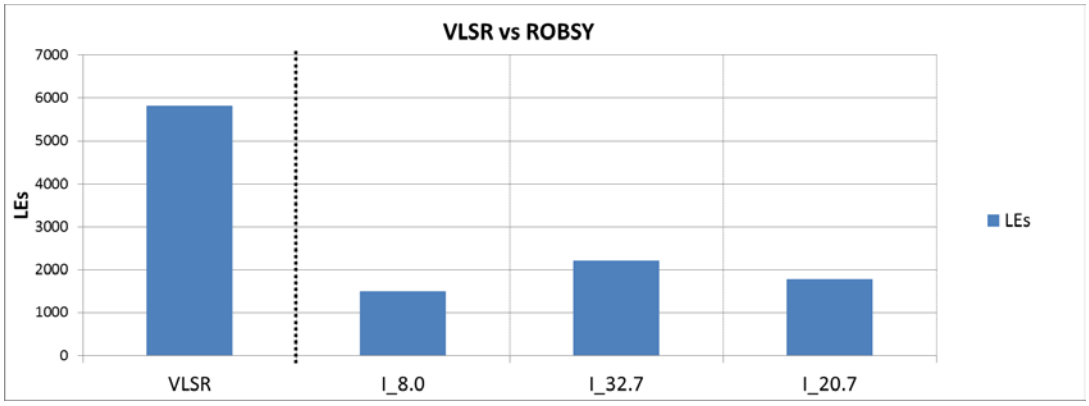


Figure 7.21: VLSR vs ROBSY logic elements. SRAM DUT (L4-L2)

The VLSR requires three times more LEs than the ROBSY FBTS. The reason of this is that the size of the VLSR only depends on the number of I/Os. For each I/O, it is necessary to use approximately 6 LEs to implement a mask and data register cell. The advantage is that the same VLSR can be used to test different DUTs connected to the FPGA.

7.7.2.2 Performance

Table 7.13 presents test time results of the ROBSY test system and VLSR for the SRAM DUT-M. ROBSY DR-shifts and T_{test} results are obtained from Table 7.9. The data register of the VLSR is dynamically configured with 82 cells in order to access all SRAM pins. The quartus_stp and USB-Blaster tools are used in order to obtain a fair comparison.

Test approach		DR-shifts	T_{test} (ms)	Speed-up
ROBSY	***L4-L2***	76	234.01	5.8
	L4-L3	76	228.00	5.9
	L4	76	225.99	6.0
	L3-L2	120	318.06	4.2
	L3	120	332.16	4.1
	L2	1976	4032.34	0.3
VLSR		564	1352.09	1

Table 7.13: ROBSY vs VLSR test time. SRAM DUT-M (all layer partitions)

Table 7.13 shows that T_{test} values of ROBSY are four to six times shorter than VLSR values for all layer partitions (excluding L2 in ESW). The T_{test} value for L2 in ESW is approximately three times longer than the VLSR. The differences are mainly due to

dissimilar number of DR-shifts. In this case, the high number of DR-shifts required for ROBSY if L2 is implemented in ESW is a consequence of the great amount of DTs that have to be performed. This shows that the FBTS for the SRAM DUT-M that only includes L1 and L2 in the FPGA is not appropriate in terms of test time.

Table 7.14 shows test time results for the LCD DUT-M. ROBSY DR-shifts and T_{test} results are obtained from Table 7.10. The data register of the VLSR is dynamically configured with 30 cells in order to access all pins of the LCD controller.

Test approach		DR-shifts	T_{test} (ms)	Speed-up VLSR
ROBSY	***L4-L2***	512	968.0	275
	L4-L3	424	784.0	340
	L4	424	784.0	340
	L3-L2	512	967.9	275
	L3	424	784.0	340
	L2	920	1792.1	149
VLSR		131088	266320	1

Table 7.14: ROBSY vs. VLSR test time. LCD DUT-M (all layer partitions)

In comparison to the SRAM DUT-M, results for the LCD DUT-M show higher speed-ups (149-340). This is due to the higher differences in the number of DR-shifts necessary for ROBSY and the VLSR.

The comparison of ROBSY to the VLSR shows that ROBSY is able to improve test time results obtained with a VLSR. This is possible because the implementation of test functions in the FBTS reduces the amount of information that has to be transferred between the ATE and FPGA. However, the implementation of L2-L1 in the FBTS tends to produce high ATE/FBTS traffic. Therefore, it is convenient to at least implement layers L3-L1 in the FBTS.

Although it is not shown in Table 7.13 and Table 7.14, it is important to remember that the ROBSY test system applies test patterns at-speed, while the application speed of the VLSR depends on TCK and the length of the data register.

7.8 Summary

Chapter 7 presented experimental results of the ROBSY test system for two different DUTs. The first DUT is an SRAM, in which the DUT-M describes two algorithms for the detection and diagnosis of faults. The second DUT is a LCD, in which the DUT-M describes a square pattern that moves through the screen.

The experiments are used to analyze the processor and FBTS resource utilization (LEs, M9K blocks, and 9x9 multipliers), FBTS active time, and test time based on the configuration parameters of the processor and layer partitions. For comparison purposes, same tests were performed with Nios II cores and a VLSR.

7.8.1 Resource utilization

7.8.1.1 ROBSY processor

It was demonstrated that the automatic generation process is able to adapt the ROBSY processor to the DUT-M. It computes values of the dependent configuration parameters, defining proper instructions, number of GPRs, and size of memories. The dependent configuration parameters allow producing efficient ROBSY processor variants in terms of resources without affecting the performance of the processor (execution time). In this way, it is possible to implement the ROBSY processor in FPGAs of low capacity or a multi-domain FBTS. Additionally, the number of configuration parameters that have to be defined manually is reduced to five variable (independent) configuration parameters. These parameters cannot be exactly calculated by analyzing the DUT-M or assembly program properties.

The variable (independent) configuration parameters provide a way to manually influence the resource utilization of the processor. This is done based on trade-offs between type of FPGA resources and trade-offs between FPGA resources and performance. *GPR_mem* and *shifter_mult* produce trade-offs between LEs and M9K blocks and between LEs and 9x9 multipliers, respectively. *pipeline_set* produces trade-offs between LEs and performance, and *short_imm_set* between M9K blocks and performance. The activation of *short_imm_set* typically reduces the number of M9K blocks at the expense of additional instructions necessary for handling immediate data and address values. However, it was shown that *short_imm_set* can produce the opposite effect. For low *data_width* values, the compiler might produce a very high number of additional instructions (e.g. SRAM DUT-M and 8-bit processors). Finally, *data_width* produces trade-offs between all FPGA resources and performance. The increase of *data_width* goes with an increase of LEs, M9K blocks, and 9x9 multipliers, but it provides the option to perform operations using fewer instructions.

The constellation *short_imm_set* true, *pipeline_set* false, *GPR_mem* true, and *shifter_mult* true produces the lowest utilization of LEs at the expense of additional embedded blocks, while the opposite constellation produces the contrary effect. However, resource utilization differences between processor variants vary depending on the value of the

dependent configuration parameters as well as on the synthesis tool and selected synthesis options.

7.8.1.2 FPGA-based test system

The resource utilization of the complete FBTS is affected by the processor variant but most importantly by the layer partition, which defines layers implemented in SW, ESW, and HW. The experiments showed that the layer partition offers flexibility in terms of resource utilization, but results are highly dependent on the properties of the specific DUT-M (number and size of procedures, variables, and operations).

Experiments carried out with the SRAM and LCD DUT-Ms showed that the layer partition has a greater influence on resources than the processor independent configuration parameters. However, the layer partition can produce significant or no variations of the results. This depends on the properties of each layer of the DUT-M. For example, the implementation of an additional layer in HW for the SRAM DUT-M produced a significant increase of LEs. On the other hand, the implementation of an additional layer in HW for the LCD DUT-M produced low variations of LEs.

The results also showed that including a processor as part of the FBTS might not always contribute to an efficient utilization of resources. If the DUT-M presents a low complexity, with a small number and size of procedures, variables, and operations (e.g. LCD DUT-M), the resource utilization of the processor might be very high in comparison to the resource utilization of a co-processor for the same layers. In such a case, it is necessary to evaluate if other advantages of the processor (reusability, standard communication mechanism, pre-verified component, debugging) or if the ROBSY approach itself is appropriate for testing the interconnections to the specific DUT.

Finally, the use of layer partitions and configuration parameters provide a way to make an efficient use of resources, facilitating the implementation of a multi-co-processor or multi-domain FBTS in FPGAs with different capacities. It was shown that the FBTS generated for the experiments fit in any FPGA of the Cyclone IV-E family, and there are even additional resources available to extend the FBTS.

7.8.2 FBTS active time

FBTS active time represents the total test time if delays caused by the execution of SW layers and the ATE/FBTS communication are not considered. It is an ideal mean to show the effect of the processor configuration and ESW/HW partition because the execution of SW and the ATE/FBTS communication delays are not considered.

The analysis of FBTS active time was performed by means of the real and minimum FBTS active time metrics. The former is calculated based on the real operation frequency of the FBTS (50 MHz), and the latter is calculated based on the maximum operation frequency of the FBTS (TimeQuest timing analyzer results).

7.8.2.1 ROBSY processor

The real FBTS active time is not affected by the dependent configuration parameters or by all variable (independent) configuration parameters. It is only affected by the variable (independent) configuration parameters *short_imm_set*, *pipeline_set*, and *data_width*. The activation of *short_imm_set* typically results in longer values due to the execution of additional instructions required to handle short immediate data and address values. The activation of *pipeline_set* results in shorter values due to the pipeline. This means that for a given *data_width*, the shortest real FBTS active time is obtained if *pipeline_set* is true and *short_imm_set* is false (at the expense of more LEs and M9K blocks).

An increase of *data_width* produces a reduction of the real FBTS active time until a certain *data_width*, which is known as the break value. As soon as this value is exceeded, the FBTS active time remains relatively constant. The break value depends on properties of the layer procedures implemented in ESW, specifically on the size and utilization rate of variables and arguments. It signalizes processor variants with the most efficient utilization of resources because processor variants with higher *data_width* values consume more resources without significantly improving the real FBTS active time. The break values obtained for the SRAM and LCD DUT-Ms and all layer partitions are 20-24 and 8, respectively. They show that the selection of untypical *data_width* values can lead to a better adaptation of the processor.

The minimum FBTS active time is influenced by all the configuration parameters. It was shown that the configuration parameters that increase the number of embedded blocks (e.g. *GPR_mem* and *shifter_mult*) reduce the maximum operation frequency achieved with the FBTS, increasing the minimum FBTS active time values.

7.8.2.2 Layer partition

The layer partition plays an essential role in the FBTS active time, and produces more noticeable changes in comparison to the processor configuration parameters. Depending on the properties of the DUT-M (layers implemented in ESW and HW), it can either reduce the FBTS active time in a significant way or produce unnoticeable changes. The implementation of additional layers in HW typically results in shorter real FBTS active time values. However, there are cases in which the implementation of additional layers in HW does not produce any significant changes (e.g. L3-L1 in comparison to L2-L1 in HW

for the LCD DUT-M). The same is true for the implementation of additional layers in ESW. The reason of this is that the FBTS active time (in the same way as the resource utilization) also depends on the properties of the DUT-M.

7.8.3 Test time

Test time includes the real FBTS active time, execution time of SW layers, and time required for the ATE/FBTS communication. It was shown that the ATE/FBTS communication can significantly influence the results depending on the value of $\delta_s + \delta_h$ delays and number of DR-shifts.

In the experimental setup presented in this chapter, $\delta_s + \delta_h$ delays produced by the USB-Blaster and quatus_stp are in the range of 1-3 milliseconds. For the SRAM test, it was shown that test time depends on the number of DR-shifts because the real FBTS active time is shorter than the $\delta_s + \delta_h$ delays. For the LCD test, it was shown that test time does not show such a strong dependency on the ATE/FBTS communication as in the case of the SRAM test. This is due to the low number of DR-shifts that has to be executed (without considering polling) and the fact that the real FBTS active time is in the same range or longer than $\delta_s + \delta_h$ delays.

The consequence is that processor configuration parameters as well as the location of interface I3 (processor/co-processor interface) do not affect test time results for FBTS active time values shorter than $\delta_s + \delta_h$. In such a case, test time depends on the location of interface I2 (ATE interface) given that it defines the number of DR-shifts. The influence of the ATE/FBTS communication can be reduced by selecting a test controller with shorter $\delta_s + \delta_h$ delays, or by reducing the number of DR-shifts. Section 8.6 presents alternatives for reducing the number of DR-shifts.

The FPGA configuration time also plays an important role in the test time. However, the configuration time can be reduced to values below 100 milliseconds. This is achieved by using high speed configuration methods that have to be considered during the design of the PCB. However, the FPGA configuration time will become less relevant as the implementation of multiple FBTS domains takes place.

7.8.4 Comparison to Nios II and VLSR

In terms of resource utilization, it was shown that the resource utilization of the ROBSY processor is in the same range as the resource utilization of Nios II cores. It is even possible to obtain a more efficient utilization of resources with the ROBSY processor by

selecting proper values for the configuration parameters. The reason of this is that the configuration options provided by ROBSY make it possible to obtain processor variants that are better adapted to the given test scenario even if the ROBSY processor is not optimized for a specific FPGA. The comparison to the VLSR shows a much higher utilization of LEs in the case of the VLSR. This is due to the high number of FPGA I/Os and that for each I/O it is necessary to used approximately 6 LEs.

In terms of FBTS active time, the comparison to Nios II cores for a single layer partition shows that ROBSY is able to deliver results in the same range of the Nios II. This is achieved even if the Nios II cores are optimized for Altera FPGAs, have more pipeline stages, and support dynamic branch prediction.

In terms of test time, the comparison of ROBSY to the Nios II test systems showed that results obtained with ROBSY are half as long in average. The reason of this is that the GNU debugger tool supported by Nios II requires more DR-shifts than the approach developed for the ROBSY test system. The comparison to the VLSR showed that ROBSY is also able to achieve shorter test time values. In the case of the SRAM, the speed-ups achieved are in the range of 4-6 depending on the layer partition. However, the implementation of L2-L1 in the FBTS can produce longer test times if the amount of DR-shifts is higher with ROBSY than with the VLSR. In the case of the LCD, the speed-ups are even higher (149-339).

To conclude, the comparison to the Nios II shows that ROBSY can provide better resource utilization results and similar performance even if the processor is not optimized for a specific FPGA or for a given data width. This is due to the processor's adaptation mechanism and specialization for testing. The comparison to VLSR shows that the implementation of test functionality in the FPGA reduces the number of DR-shifts. Additionally, it is important to remember that ROBSY applies test patterns at-speed, while the application speed of the VLSR depends on TCK and the VLSR length.

7.8.5 Synthesis tool

The FPGA synthesis tool use heuristics to map the functionality described at the RTL to the FPGA blocks. In this way, it is able to provide optimal results in a short amount of time. Drawbacks are that it is quite difficult to predict the time required for synthesis and that the heuristics might influence the results without an apparent reason. For example, the tool might perform differently during the implementation of similar processor variants, duplicating registers that were not duplicated in other cases, using a different

amount of resources for routing, including more M9K blocks or 9x9 multipliers than necessary or producing solutions with unrelated maximum operation frequencies.

These uncertainties produced by the synthesis tool make it very difficult to predict what will be the final effect of a given layer partition or processor variant. Therefore, it is necessary to keep in mind that the results presented in this chapter represent typical behaviors. However, it is possible to obtain different results in certain cases or if other synthesis tools, tool versions, or FPGAs are used.

7.8.6 Selection of a processor variant

Sections 7.4 and 7.5 show that the resource utilization and FBTS active time can be modified based on the independent processor configuration parameters. In order to select a proper processor variant, there are some guiding principles that can be used by the test engineer after the definition of the layer partition.

The first step is the selection of a proper *data_width* value considering the size of variables and arguments of procedures implemented in ESW and the size of procedure arguments belonging to the highest layer implemented in HW. The goal is to select a *data_width* value that matches the break value. After that, the test engineer can start experimenting with the remaining independent configuration parameters. For this purpose, the following guiding principles should be considered:

- *short_imm_set* reduces the number of M9K blocks at the expense of extra instructions that can increase FBTS active time. However, the test engineer has to consider the possible counter effect of this parameter for low *data_width* values (Section 7.4.2.2).
- *pipeline_set* reduces FBTS time at the expense of additional LEs for the implementation of pipeline related logic.
- *GPR_mem* reduces the number of LEs at the expense of two additional M9K blocks. It does not affect the real FBTS active time, but it increases the minimum FBTS active time (reduction of the maximum operation frequency).
- *shifter_mult* reduces the number of LEs at the expense of additional multipliers. The effect of *shifter_mult* takes place only if shift or rotation instructions are required. It does not affect the real FBTS active time, but it increases the minimum FBTS active time.

If the obtained processor and FBTS variant do not achieve the required FBTS active time based on the independent configuration parameters *short_imm_set*, *pipeline_set*, *GPR_mem*, and *shifter_mult*, the test engineer can increase *data_width* or select a

different layer partition at the expense of higher utilization of resources. On the other hand, if the resource utilization is above a maximum constraint, the test engineer can reduce *data_width* or select a different layer partition at the expense of longer real FBTS active time values.

These guiding principles cannot be considered general rules that stay valid for all test scenarios. The properties of the DUT-M, FPGA, and heuristics of the synthesis tool can lead to different results. They provide a fast mechanism to select a proper processor variant, and can be used for the development of heuristics that automatically compute the value of the independent configuration parameters.

8 Conclusions

8.1 Introduction

This chapter summarizes this dissertation. Section 8.2 presents a summary of the work, and Section 8.3 lists the major achievements. Sections 8.4 and 8.5 present a discussion of the impact and limitations of the provided solution, respectively. Finally, Section 8.6 provides an outlook of possible extensions and improvements as part of the future work.

8.2 Summary of the dissertation

This dissertation deals with the development of a novel FPGA-based test (FBT) approach used during the manufacturing of printed circuit boards. In this context, the general issue is related to the development of a test system with the following features:

- Architecture adapted to the specific test scenario.
- Low design effort for test engineer.
- Fault coverage of static and dynamic faults.
- At-speed test.

The dissertation proposes the development of a test system, whose main components are the external automatic test equipment (ATE) and the FPGA located on the printed circuit board. The test system is automatically generated based on a high-level description of the device under test (DUT) and test algorithms, which is known as device under test model (DUT-M). The DUT-M relies on a layer concept that divides the test functionality into five layers (L1-L5) and three interfaces (I1-I3). The implementation of test functions in the ATE is performed by transforming layers of the DUT-M to software (SW) routines, and the implementation of test functions in the FPGA is performed by transforming layers of the DUT-M to an FPGA-based test system (FBTS). The FBTS is composed of two fundamental components: a programmable test processor and a hardwired co-processor. This means that the implementation of test functions in the FBTS is performed by transforming layers of the DUT-M to embedded software (ESW) routines executed by the test processor or to hardware (HW) descriptions representing the hardwired co-processors. The selection of layers that are implemented in SW, ESW, and HW is known as the layer partition, and it is performed by defining the location of interfaces I2 and I3.

The adaptation of the test system to different test scenarios relies on the flexibility of the layer concept and the option to implement layers in the ATE or FBTS. The low design effort required to build the test system is due to the DUT-M, which is the only description that has to be developed (or selected from a library) by the test engineer. Based on this description, the SW, ESW, and HW are automatically generated.

The test system executes tests at-speed in order to guarantee the detection of dynamic faults. This is possible because the access to DUTs is implemented by co-processors, which are clocked at the operation frequency of the printed circuit board. The use of FPGAs already located on the printed circuit board, the automatic generation of the test system, the layer concept, and the processor/co-processor architecture of the FBTS provide the test engineer with a powerful test approach.

The main contribution of this dissertation is the analysis, development, and evaluation of the test processor included in the FBTS. The key features of the processor are:

- Tailored and specialized for testing purposes.
- Adaptation mechanism based on configuration parameters defined at the ISA and microarchitecture level.
- Debug-interface for ATE/FBTS communication and Wishbone bus for processor/co-processor communication.

The test processor is known as ROBSY processor. It is a scalar soft-core processor described at the register transfer level in VHDL. The VHDL description is portable to FPGAs from different families and vendors, and it was used in Altera as well as Xilinx FPGAs. The ROBSY processor is developed based on the RISC design philosophy with pipelining support and configuration options that change properties of its ISA and microarchitecture. It supports a single data width, test operations such as instructions for the generation of pseudo-random sequences and walking 1/0 sequences, and includes a debug-interface with JTAG support.

Apart from the ROBSY processor, this dissertation proposes a concept for the automatic generation flow of the complete test system. One essential part of the concept describes the way that the ROBSY processor is automatically adapted to the layers implemented in ESW and the transformation steps necessary to generate the object code. The generation flow analyses the DUT-M and generated assembly program in order to compute the value of the configuration parameters known as dependent configuration parameters. These parameters are used to adapt the processor to the DUT-M, and only affect the resource utilization of the processor. There is a second group of configuration parameters known as independent configuration parameters, whose values are not computed by the

automatic generation flow. The value of these parameters is defined by the test engineer and they provide trade-offs between resource utilization and performance.

In order to show the feasibility of the approach, a practical implementation of the automatic generation flow for a single processor/co-processor pair was developed. The flow includes the adaptation of the ROBSY processor, the generation of the object code, and the generation of SW for the ATE based on TCL and the IEEE standards 1149.1-2013 and IJTAG. The VHDL description of the co-processors was manually developed.

The practical implementation of the automatic generation flow is used to generate the ROBSY test system for two DUT-Ms. The first DUT-M is used to test the interconnections between an FPGA and an IS61WV102416 SRAM device. It describes fault detection and fault diagnosis algorithms based on deterministic patterns, algorithmic patterns, and fault dictionaries. The second DUT-M is used to test the interconnections between the same FPGA and an LCD DIP128-6 controller. The LCD DUT-M presents a lower complexity in terms of number and size of procedures, variables, and operations. It describes a square pattern that moves through the screen, which has to be visually inspected in order to detect any fault. The experiments were performed using the DE2-115 board from Terasic and the VarioTap Coach Board from Göpel Electronics. The Cyclone IV-E FPGA (EP4CE115F29C7) on the DE2-115 was used for the implementation of the FBTS.

The experimental results provide a way to evaluate the resource utilization and performance for different processor variants and layer partitions. The performance measurements were done based on the accumulated time that the FPGA-based test system is active (FBTS active time) and total test execution time (test time). Results show that the concept works and that the processor adapts itself to the DUT-M based on the automatic generation flow. Additionally, it was possible to analyze the behavior of the dependent and independent configuration parameters and propose guiding principles for assigning proper values to the independent configuration parameters. An important conclusion is that results significantly depend on the properties and complexity of the DUT-M and that the processor represents a good alternative to implement functions of layers L4-L2 if these layers present a complexity similar to the SRAM DUT-M.

For comparison purposes with state of the art soft-core processors, the test functionality of the SRAM DUT-M was implemented in three FBTS based on the Nios II/e, /s, and /f cores. The comparison showed that the resource utilization of the ROBSY processors is in the same range or below the Nios II depending on which ROBSY processor variant is used for the comparison. Additionally, FBTS active time and test time values obtained with ROBSY are 20% and 50% shorter in average, respectively. This shows that ROBSY

can provide an efficient utilization of resources and better performance even if the processor is not optimized for a given FPGA or data width as is the case of the Nios II.

For comparison purposes with state of the art FPGA-based test approaches, the test functionality of both DUT-Ms was also implemented using a generic FPGA test instrument (virtual length shift register, VLSR). Results for SRAM and LCD show that the ROBSY test system is 4 to 6 and 149 to 340 times faster than the VLSR depending on the layer partition. Additionally, the proposed test system is able to execute tests at-speed, which is not possible with the VLSR. This shows the advantages of ROBSY and that it is a suitable alternative for testing printed circuit boards.

8.3 Achievements

The theoretical part of the work includes the following achievements:

- A concept of a novel test system for printed circuit board testing.
- An FPGA-based test system composed of configurable test processors and hardwired co-processors organized in domains.
- A test processor tailored and specialized for board-level testing with configuration options defined at the ISA and microarchitecture level.
- An automatic generation flow for the generation of the test system and adaptation of the processor.

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

- The implementation of the ROBSY processor in VHDL.
    - ISA and microarchitecture tailored and specialized for FPGA-based tests.
    - JTAG-based debug-interface and a Wishbone bus for communication and debugging purposes.
    - ISA and microarchitecture configuration options defined in VHDL packages.
- The implementation of an automatic generation flow:
    - Automatic generation of VHDL packages based on the analysis of DUT-M and assembly program.
    - A compilation tool for translating the DUT-model high level description to the ROBSY processor object code and to the TCL or PDL/BSDL descriptions executed by the external ATE.

8.4 Impact

This dissertation introduced a novel and powerful FPGA-based test concept for performing tests of printed circuit boards. This concept overcomes the most limiting aspects of other FPGA-based test approaches, which correspond to the inability to execute tests at-speed, long test times, and a high development effort. The test processor is the first processor developed for printed circuit board testing, and it provides a standard interface to the ATE and the option to efficiently embed more test functions in the FPGA. It was shown that it is an ideal mechanism to control and observe the test execution and that it can provide resource utilization and performance results in the same range or better than Nios II cores, which are optimized for Altera FPGAs.

Furthermore, with the automatic generation of the test system, it is possible to implement the FPGA-based test system without the need for test engineers with a high FPGA design expertise. This makes the concept attractive and applicable to industry.

8.5 Limitations

The limitations of the FPGA-based test approach deal with the need for an FPGA on the printed circuit board, synthesis of the FPGA-based test system on the field, FPGA configuration time, and use of a test processor in low complexity test scenarios.

The main limitation of the approach presented in this dissertation is that it can be only used to test printed circuit boards that are already equipped with an FPGA for their normal operation. The addition of an FPGA to the board just for testing purposes makes no sense at all because the tested interconnections (between FPGA and DUTs) will not be used during the normal operation of the board.

The need to synthesize the FPGA-based test system on the field makes it necessary to consider the synthesis tool, synthesis time, resource utilization, and timing results. As a consequence, the automatic generation flow should support mechanisms to deal with FPGA-based test systems that do not fit in the FPGA or that do not reach the required operation frequency. This is a great challenge for the development of the automatic generation flow because it should be aware of the synthesis options offered by the tools, and it should include mechanism to give some feedback or guide test engineers in the case of problems.

This dissertation did not consider FPGA configuration time as part of test time. This is due to the fact that the configuration time depends on the capacity of the FPGA and the

selected configuration mechanism. The experiments presented show that the FPGA configuration time can take much longer than test time. Therefore, it is necessary to counteract its effect by using high performance configuration mechanisms (when possible) and by implementing FPGA-based test systems with multiple domains. The latter makes it possible to test interconnections to multiple DUTs at the same time and to configure the FPGA only once.

The use of the test processor might result in an inefficient utilization of resources for low complexity DUT-Ms. As presented in the LCD test, the processor might consume more than 60% of the resources even if layers L1-L3 are implemented in hardware and only a very basic L4 functionality is implemented in embedded software. In such a case, the use of a processor does not offer any advantages and better results would be likely obtained with a completely hardware implementation. However, this effect tends to disappear when multiple co-processors are connected to the processor in the same domain.

8.6 Future work

There are multiple research directions concerning the test system as well as the ROBSY processor. They are necessary in order to explore other alternatives, enhance the FPGA-based test approach, or develop parts of the concept that are not already developed.

- Partitioning and automatic generation process
    - Development of a high level DUT-M analyzer in order to evaluate the results of different layer partitions. In this way, it is possible to accelerate the selection of a proper partition based on the properties of the DUT-M.
    - Extension of the automatic generation process in order to enable the generation of a multi-domain multi-co-processor FBTS. For this purpose, it is necessary to analyze and compile multiple DUT-Ms at the same time.
    - Research and development of mechanisms used as part of the automatic generation process to deal with FBTS variants that do not fit in the FPGA or do not reach the required operation frequency.
    - Research and evaluation of heuristics that automatically compute the independent configuration parameters values of the ROBSY processor (e.g. design of experiments [153]).
- ATE/FBTS and processor/co-processor communication
    - Optimization of the debug-interface in order to reduce the number of DR-shifts required for exchanging information between the ATE and FBTS.

  - Research of compression and decompression schemes that can be applied to the ATE/FBTS communication. This is necessary in order to reduce test time and support high volume data transfers (FLASH programming).
  - Support for burst transfers as part of the Wishbone bus and evaluation of an interrupt-based processor/co-processor communication.
- FBTS architecture
  - Utilization of accumulation buffers between layers L1 and L2. This is necessary to guarantee the application of pattern sequences at-speed independently of the layer partition.
  - Support for hard-core controllers already available in the FPGA for the implementation of L1 functions.
  - Development and evaluation of different structures for the ATE/FBTS communication in a multi-domain FBTS. This can be realized considering flat and hierarchical structures proposed in the IEEE 1149.1-2013 and IJTAG standards.
  - Research of layer partitions that implement L5 in the FPGA. This includes the analysis and definition of embedded visualization mechanisms for test results.
- Processor and compiler development
  - Optimization of the compiler used to generate the object code. For this purpose, it is necessary to efficiently map variables to registers and perform code optimizations. The goal is to further reduce the processor execution time and the size of the processor memories.
  - Research of instruction set extension mechanisms based on the DUT-M.
- Experiments with additional DUT-Ms
  - Perform additional experiments with other DUTs such as FLASH memories in order to evaluate the test approach when high amount of data is exchanged between FPGA and DUT.
  - Evaluation of the FPGA-based test system for the execution of bit error rate tests (BERT).

Bibliography

[1] *1149.1 IEEE standard for test access port and boundary-scan architecture*, 2013.

[2] J. Sachsse, J. H. Meza Escobar, S. Ostendorff, and H.-D. Wuttke, "A Holistic Approach of an Architecture for Tests of FPGA Based Systems with Boundary Scan," in *Zuverlässigkeit und Entwurf (ZuE 2010): 4. GMM/GI/ITG-Fachtagung*, Berlin: VDE publisher, 2010, pp. 51–52.

[3] J. Sachße, H.-D. Wuttke, S. Ostendorff, and J.-H. Meza Escobar, "Architecture of an Adaptive Test System Built on FPGAs," in *24th Architecture of Computing systems (ARCS 2011)*, Berlin, New York: Springer, 2011, pp. 86–97.

[4] J. H. Meza Escobar, J. Sachsse, S. Ostendorff, and H.-D. Wuttke, "Automatic generation of an FPGA based embedded test system for printed circuit board testing," in *13th Latin American Test Workshop (LATW 2012)*: IEEE Proceedings, 2012, pp. 75–80.

[5] J. H. Meza Escobar, J. Sachsse, S. Ostendorff, and H.-D. Wuttke, "ISA configurability of an FPGA test-processor used for board-level interconnection testing," in *14th Latin American Test Workshop (LATW 2013)*: IEEE Proceedings, 2013.

[6] M. Jenihhin *et al.,* "Automated Design Error Localization in RTL Designs," (English), *IEEE Design & Test*, vol. 31, no. 1, pp. 83–92, 2014.

[7] J. H. Meza Escobar, S. Ostendorff, and H.-D. Wuttke, "A configurable test processor for board-level testing," in *19th Euromicro Conference on Digital System Design (DSD 2016)*: Proceedings, 2016.

[8] A. Mitschele-Thiel, *Systems engineering with SDL: Developing performance-critical communications systems*. Chichester, New York: J. Wiley, 2001.

[9] C. Haubelt and J. Teich, *Digitale Hardware/Software-Systeme: Spezifikation und Verifikation*. Berlin, Heidelberg: Springer, 2010.

[10] J. Teich, *Digitale Hardware/Software-Systeme: Synthese und Optimierung,* 2nd ed. Berlin: Springer, 2007.

[11] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital systems testing and testable design*. Piscataway, NJ: IEEE Press, 1990.

[12] M. L. Bushnell and V. D. Agrawal, *Essentials of electronic testing for digital, memory and mixed-signal VLSI circuits*. New York: Kluwer Academic, 2002.

[13] O. Novák, E. Gramatová, and R. Ubar, *Handbook of testing electronic systems,* 1st ed. Praha: Czech Technical University Publishing House, 2005.

[14] J. Bergeron, *Writing testbenches: Functional verification of HDL models,* 2nd ed. Boston: Kluwer Acad. Publ., 2003.

[15] N. K. Jha and S. Gupta, *Testing of digital systems*. Cambridge: Cambridge University Press, 2003.

[16] J. T. de-Sousa and P. Cheung, *Boundary-scan interconnect diagnosis*. Boston: Kluwer Academic Publishers, 2001.

[17] C. E. Stroud, *A designer's guide to built-in self-test*. Boston: Kluwer Academic Publishers, 2002.

[18] Y. C. Kim, V. D. Agrawal, and K. K. Saluja, "Multiple faults: modeling simulation and test," in *7th Asia and South Pacific Design Automation Conference (ASP-DAC 2002)*, 2002, pp. 592–597.

[19] R. Ubar, J. Raik, and H. T. Vierhaus, *Design and test technology for dependable systems-on-chip*. Hershey, PA: Information Science Reference, 2010.

[20] K. Heragu, J. H. Patel, and V. D. Agrawal, "Segment delay faults: a new fault model," in *14th VLSI Test Symposium (VTS 1996)*, 1996, pp. 32–39.

[21] V. P. Kodali, *Engineering electromagnetic compatibility: Principles, measurements, technologies, and computer models,* 2nd ed. New York: IEEE, 2001.

[22] V. Meyer, A. Palit, W. Anheier, A. Sticht, and J. Schoeffel, "Can Signal Integrity Faults be Detected by Delay Tests?," in *12th IEEE North Atlantic Test Workshop (NATW 2003)*, 2003, pp. 131–136.

[23] M. Tehranipour, N. Ahmed, and M. Nourani, "Testing SoC interconnects for signal integrity using boundary scan," in *21st VLSI Test Symposium (VTS 2003)*: IEEE Comput. Soc, 2003, pp. 158–163.

[24] M. Nourani and A. Attarha, "Signal integrity: Fault Modeling and Testing in HighSpeed SoCs," (English), *Journal of Electronic Testing*, vol. 18, no. 4/5, pp. 539–554, 2002.

[25] M. Cuviello, S. Dey, Xiaoliang Bai, and Yi Zhao, "Fault modeling and simulation for crosstalk in system-on-chip interconnects," in *IEEE/ACM International Conference on Computer-Aided Design. (ICCAD 1999)*: Digest of Technical Papers, 1999, pp. 297–303.

[26] I. Pomeranz and S. M. Reddy, "On dictionary-based fault location in digital logic circuits," (English), *IEEE Transactions on Computers*, vol. 46, pp. 48–59, 1997.

[27] T. Wenzel and H. Ehrenberg, "Embedded system access (ESA)," White paper, Göpel Electronic, 2012.

[28] W. J. Dally and J. W. Poulton, *Digital systems engineering*. Cambridge, U.K., New York, NY, USA: Cambridge University Press, 1998.

[29] R. S. Khandpur, *Printed circuit boards: Design, fabrication, assembly and testing*. New York: McGraw-Hill, 2006.

[30] I. Aleksejev, "FPGA-based Embedded Virtual Instrumentation," Ph. D, Department of Computer Engineering, Tallinn Univ. of Technology, Tallinn, Estonia, 2013.

[31] D. Maliniak, *Test is a matter of life or death in the automotive industry.* [Online] Available: http://electronicdesign.com/test-amp-measurement/test-matter-life-or-death-automotive-industry. Accessed on: Mar. 16 2015.

[32] M. Berger, *Test- und Prüfverfahren in der Elektronikfertigung: Vom Arbeitsprinzip bis Design-for-Test-Regeln*. Heidelberg: Hüthig, 2012.

[33] S. F. Scheiber, *Building a successful board-test strategy,* 2nd ed. Boston: Butterworth-Heinemann, 2001.

[34] D. Edwards, *PCB design and its impact on device reliability.* [Online] Available: http://electronicdesign.com/boards/pcb-design-and-its-impact-device-reliability. Accessed on: Mar. 16 2015.

[35] S. Oresjo, "A New Test Strategy for Complex Printed Circuit Board Assemblies," in *National Electronic Packaging and Production Conference (Nepcon West 1999)*: Proceedings, 1999.

[36] J. Kirschling, *Improved Fault Coverage in a Combined X-Ray and In-Circuit Test Environment.* [Online] Available: http://www.home.agilent.com/upload/cmc_upload/All/kirschling_etronix_59882291en.pdf?&cc=DE&lc=ger. Accessed on: Jul. 03 2013.

[37] K. Parker, *The boundary-scan handbook: Analog and digital,* 2nd ed. Boston, MA: Kluwer, 1998.

[38] H. Goepel, "Reducing the Cost of Test with Boundary Scan," (English), *EE-Evaluation Engineering,* http://www.evaluationengineering.com/articles/200401/reducing-the-cost-of-test-with-boundary-scan.php, 2004.

[39] H. Ehrenberg, *Why should you care about JTAG/Boundary Scan /IEEE 1149.x*. Presentation. [Online] Available: http://www.ieee.li/pdf/viewgraphs/jtag_boundary_scan.pdf. Accessed on: Jan. 22 2016.

[40] B. Nadeau-Dostie, J. Cote, H. Hulvershorn, and S. Pateras, "An embedded technique for at-speed interconnect testing," in *IEEE International Test Conference (ITC 1999)*: Proceedings, 1999, pp. 431–438.

[41] A. Jutman, "At-speed on-chip diagnosis of board-level interconnect faults," in *9th IEEE European Test Symposium (ETS 2004)*: IEEE Proceedings, 2004, pp. 2–7.

[42] R. Folea, "Programming Flash Memory from FPGAs and CPLDs Using the JTAG Port," (English), *XCell Journal*, no. 49, pp. 77–79, 2004.

[43] D. Wallace, *Using the JTAG Interface as a General-Purpose Communication Port*. XCell Online. [Online] Available: http://matthieu.benoit.free.fr/pdf/xc_jtag53.pdf. Accessed on: Jan. 22 2016.

[44] B. Nadeau-Dostie, *Design for at-speed test, diagnosis, and measurement*. Boston: Kluwer Academic, 2000.

[45] T. Wenzel and H. Ehrenberg, "Combining Boundary Scan and JTAG Emulation for Advanced Structural Test and Diagnostics," (English), *Göpel Electronic*, 2009.

[46] A. Tšertov, "System modeling for processor-centric test automation," Ph. D, Department of Computer Engineering, Tallinn Univ. of Technology, Tallinn, Estonia, 2012.

[47] S. Park and T. Kim, "A new IEEE 1149.1 boundary scan design for the detection of delay defects," in *Design, Automation and Test in Europe Conference & Exhibition (DATE 2000)*, Los Alamitos, California: IEEE Comput. Soc, 2000, pp. 458–462.

[48] J. Nejedlo and R. Khanna, "Intel® IBIST, the full vision realized," in *IEEE International Test Conference (ITC 2009)*, Washington, D.C: IEEE Proceedings, 2009, pp. 1–11.

[49] D. Bonnett, "Combine Boundary Scan with CPU Emulation to Extend Test Coverage," (English), *Asset Intertech*, 2004.

[50] J. Webster, B. Fenton, D. Stringer, and B. Bennetts, "On the synergy of boundary scan and emulation board test: a case study," in *Board Test Workshop (BTW 2003)*: Proceedings, 2003.

[51] S. Devadze, A. Jutman, A. Tsertov, M. Instenberg, and R. Ubar, "Microprocessor-based System Test using Debug Interface," in *IEEE Norchip Conference (Norchip 2008)*: IEEE Proceedings, 2008, pp. 98–101.

[52] A. Tsertov, A. Jutman, and S. Devadze, "Testing beyond the SoCs in a lego style," in *East-West Design and Test Symposium (EWDTS 2010)*: IEEE Proceedings, 2010, pp. 334–338.

[53] A. Tsertov, R. Ubar, A. Jutman, and S. Devadze, "SoC and Board Modeling for Processor-Centric Board Testing," in *14th Euromicro Conference on Digital System Design (DSD 2011)*: IEEE Proceedings, 2011, pp. 575–582.

[54] S. Devadze, A. Jutman, and R. Ubar, "Turning JTAG Inside Out for Fast Extended Test Access," in *10th Latin American Test Workshop (LATW 2009)*, Piscataway, N.J.: IEEE Proceedings, 2009.

[55] S. Devadze, A. Jutman, I. Aleksejev, and R. Ubar, "Fast Extended Test Access via JTAG and FPGAs," in *IEEE International Test Conference (ITC 2009)*, Washington, D.C: IEEE Proceedings, 2009.

[56] S. Ostendorff, H.-D. Wuttke, J. Sachsse, and S. Kohler, "A new approach for adaptive failure diagnostics based on emulation test," in *Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*: IEEE Proceedings, 2010, pp. 327–330.

[57] I. Aleksejev, A. Jutman, S. Devadze, S. Odintsov, and T. Wenzel, “FPGA-based synthetic instrumentation for board test,” in *IEEE International Test Conference (ITC 2012)*: IEEE Proceedings, 2012, pp. 1–10.

[58] A. Jutman, S. Devadze, I. Aleksejev, and T. Wenzel, “Embedded synthetic instruments for Board-Level testing,” in *17th IEEE EUROPEAN TEST SYMPOSIUM (ETS 2012)*: IEEE Proceedings, 2012.

[59] Göpel Electronic, *ChipVorx basics and applications.* [Online] Available: http://www.goepel.com. Accessed on: Jul. 12 2015.

[60] A. Crouch, *FPGA-Controlled Test (FCT): What it is and why is it needed?* [Online] Available: http://www.asset-intertech.com/eresources/fpga-controlled-test-fct-what-it-and-why-it-needed. Accessed on: Jan. 22 2016.

[61] Altera, *Nios II classic Processor Reference Guide.* [Online] Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/nios2/n2cpu_nii5v1.pdf. Accessed on: Dec. 14 2015.

[62] Xilinx, *MicroBlaze Processor Reference Guide: UG081.* [Online] Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/mb_ref_guide.pdf. Accessed on: Jul. 11 2014.

[63] Xilinx, *PicoBlaze 8bit Embedded Microcontroller User Guide.* [Online] Available: http://www.xilinx.com/support/documentation/ip_documentation/ug129.pdf. Accessed on: Dec. 14 2015.

[64] Ensilica, *esi-RISC cores: Configurable Embedded Processor IP.* Accessed on: Jul. 11 2014.

[65] Cadence, *Xtensa LX6 Customizable DPU.* [Online] Available: http://ip.cadence.com/uploads/533/Cadence_Tensillica_Xtensa_LX6_ds-pdf. Accessed on: Dec. 14 2015.

[66] A. Läuger, *T48 µController Integration Manual.* [Online] Available: http://opencores.org/project,t48,overview. Accessed on: Jul. 11 2015.

[67] S. Rhoads, *Plasma Processor.* [Online] Available: http://opencores.org/project,plasma,overview. Accessed on: Jul. 11 2014.

[68] A. Meziti ElIbrahimi, *copyBlaze.* [Online] Available: http://opencores.org/project,copyblaze,overview. Accessed on: Jul. 11 2014.

[69] Aeroflex Microelectronic Solutions, *Leon 3 Processor.* [Online] Available: http://www.gaisler.com/index.php/products/processors/leon3. Accessed on: Dec. 14 2015.

[70] S. Tan, *AEMB 32-bit Microprocessor Core Datasheet.* [Online] Available: http://opencores.org/project,aemb,overview. Accessed on: Jul. 11 2015.

[71] C. Santifort, *Amber Project User Guide.* [Online] Available: http://opencores.org/project,amber. Accessed on: Jul. 11 2015.

[72] OpenCores, *OpenRISC 1200 IP Core Specification: (Preliminary Draft).* [Online] Available: http://opencores.org/or1k/OR1K:Community_Portal#Get_source_code. Accessed on: Dec. 14 2015.

[73] Lattice semiconductor, *LatticeMico8 Processor Reference Manual.* [Online] Available: http://www.latticesemi.com/en/Products/DesignSoftwareAndIP/IntellectualProperty/IPCore/IPCores02/Mico8.aspx. Accessed on: Jul. 11 2015.

[74] Lattice semiconductor, *LatticeMico32 Processor Reference Manual.* [Online] Available: http://www.latticesemi.com/en/Products/DesignSoftwareAndIP/IntellectualProperty/IPCore/IPCores02/LatticeMico32.aspx. Accessed on: Dec. 14 2015.

[75] O. Bründler, *Proteus Prozessor Architektur.* [Online] Available: http://www.logicsolutions.ch/Download.htm. Accessed on: Jul. 11 2014.

[76] A. Strole, H.-J. Wunderlich, and O. Haberl, "TESTCHIP: A Chip for Weighted Random Pattern Generation, Evaluation, and Test Control," in *16th European Solid-State Circuits Conference (ESSCIRC 1990)*: Proceedings, 1990, pp. 101–104.

[77] L. Ali, Z. Darus, M. Ali, and I. Ahmed, "Test processor ASIC design," in *IEEE International Conference on Semiconductor Electronics (ICSE 1996)*: IEEE Proceedings, 1996, pp. 261–265.

[78] Z. Darus, I. Ahmed, and L. Ali, "A test processor chip implementing multiple seed, multiple polynomial linear feedback shift register," in *6th Asian Test Symposium (ATS 1997)*: IEEE Proceedings, 1997, pp. 155–160.

[79] M. Ali, S. Islam, and M. Ali, "Test processor chip design with complete simulation result including reseeding technique," in *IEEE International Conference on Semiconductor Electronics (ICSE 2002)*: IEEE Proceedings, 2002, pp. 218–221.

[80] M. Kabir and L. Ali, "Design of GLFSR based test processor chip," in *IEEE Student Conference on Research and Development (SCOReD 2009)*: IEEE Proceedings, 2009, pp. 234–237.

[81] C. Galke, M. Pflanz, and H. Vierhaus, "A test processor concept for systems-on-a-chip," in *IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD 2002)*: IEEE Proceedings, 2002, pp. 210–212.

[82] C. Kretzschmar, C. Galke, and H. Vierhaus, "A hierarchical self test scheme for SoCs," in *10th IEEE International On-Line Testing Symposium (IOLTS 2004)*: IEEE Proceedings, 2004, pp. 37–42.

[83] R. Kothe, C. Galke, and H. Vierhaus, “A multi-purpose concept for SoC self test including diagnostic features,” in *11th IEEE International On-Line Testing Symposium (IOLTS 2005)*: IEEE Proceedings, 2005, pp. 241–246.

[84] C. Galke, T. Koal, and H. Vierhaus, “Möglichkeiten und Grenzen der automatischen SBST Generierung für einfache Prozessoren: Fallstudie des Testprozessors T5016tp,” in *Dresdner Arbeitstagung Schaltungs- und Systementwurf (DASS 2007)*, Dresden: TUDpress, 2007.

[85] R. Frost, D. Rudolph, C. Galke, R. Kothe, and H. Vierhaus, “A Configurable Modular Test Processor and Scan Controller Architecture,” in *13th IEEE International On-Line Testing Symposium (IOLTS 07)*: IEEE Proceedings, 2007, pp. 277–284.

[86] T. Koal, R. Kothe, and H. Vierhaus, “Der erste Mikroprozessor der BTU Cottbus: die Entwicklung des Testprozessors,” in *Forum der Forschung*, BTU Cottbus: Eigenverlag, 2009, pp. 127–132.

[87] S. Zeidler, C. Wolf, M. Krstic, F. Vater, and R. Kraemer, “Design of a Test Processor for Asynchronous Chip Test,” in *20th Asian Test Symposium (ATS 2011)*: IEEE Proceedings, 2011, pp. 244–250.

[88] S. Zeidler, C. Wolf, M. Krstic, and R. Kraemer, “Entwurf einer neuen Testprozessorlösung für den Funktionaltest asynchroner Schaltungen,” in *24. GI/GMM/ITG - Workshop Testmethoden und Zuverlässigkeit von Schaltungen und Systemen (TuZ 2012)*: Proceedings, 2012.

[89] S. Zeidler, “Enabling Functional Tests of Asynchronous Circuits Using a Test Processor Solution,” Ph.D., Brandenburgische Technische Universität Cottbus-Senftenberg, Cottbus, 2013.

[90] Göpel Electronic, *Systemsoftware JTAG/Boundary Scan.* [Online] Available: http://www.goepel.com/jtag-boundary-scan/boundary-scan-instrumente/software.html. Accessed on: Nov. 20 2015.

[91] C. Giaconia, A. Di Stefano, and G. Capponi, “Reconfigurable digital instrumentation based on FPGA,” in *3rd IEEE International Workshop on System-on-Chip for Real-Time Applications (2003)*: IEEE Comput. Soc, 2003, pp. 120–122.

[92] S. Di Carlo *et al.,* “A Low-Cost FPGA-Based Test and Diagnosis Architecture for SRAMs,” in *IEEE 1st International Conference on Advances in System Testing and Validation Lifecycle (VALID 2009)*: IEEE Proceedings, 2009, pp. 141–146.

[93] L. Mostardini *et al.,* “FPGA-based low-cost automatic test equipment for digital integrated circuits,” in *IEEE International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS 2015)*: IEEE Proceedings, 2009, pp. 32–37.

[94] C. T. Nadovich, *Synthetic instruments: Concepts and applications*. Amsterdam: Elsevier/Newnes, 2004.

[95] National Instruments, *Creating a Synthetic Instrument with Virtual Instrumentation Technology.* [Online] Available: http://www.ni.com/white-paper/3183/en/. Accessed on: Oct. 25 2013.

[96] J. Ferry, J. Scesnak, and S. Shaikh, "A strategy for board level in-system programmable built-in assisted test and built-in self test," in *IEEE International Test Conference (ITC 2005)*: IEEE Proceedings, 2005, pp. 798–807.

[97] J. Ferry, "FPGA-based universal embedded digital instrument," in *IEEE International Test Conference (ITC 2013)*: IEEE Proceedings, 2013, pp. 1–9.

[98] S. Ostendorff, J. H. Meza Escobar, H.-D. Wuttke, T. Sasse, and S. Richter, "Modeling timing constraints for automatic generation of embedded test instruments," in *17th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS 2014)*: IEEE Proceedings, 2014, pp. 201–206.

[99] S. Ostendorff, "Methode zur Modellierung und automatischen Generierung von FPGA-basierten Testinstrumenten," Ph. D, Department of Computer Science and Automation, TU Ilmenau, Ilmenau, Germany, 2016.

[100] R. Saleh *et al.,* "System-on-Chip: Reuse and Integration," (English), *Proceedings of the IEEE*, vol. 94, pp. 1050–1069, 2006.

[101] CK Components, "6 mm Tact Switches," PTS645 Series Datasheet, 2013.

[102] Avago Technologies, "Single digit surface mount LED Display," HDSM-281x/283x Datasheet, 2011.

[103] P. P. Chu, *FGPA Prototyping by VHDL examples: Xilinx SpatanTM-3 version*. Hoboken N.J: J. Wiley, 2008.

[104] Texas Instruments, "Low-power I/O Expander with Interrupt output and configuration Registers," TCA9535 Datasheet, 2009.

[105] Maxim, "Thermistor-to-Digital Converter," MAX6682 Datasheet, 2002.

[106] Analog Devices, "Digital Gyroscope Sensor," ADIS16266 Datasheet, 2014.

[107] Cristalfontz America, "Parallel Character LCD," CFAH1602B Datasheet, 2010.

[108] SanDisk, "SanDisk SD Card Version 2.2," Product Manual, 2004.

[109] Integrated Silicon Solution Inc ISSI, "1M x 16 High-speed asynchronous CMOS static RAM," IS61WV102416ALL Datasheet, 2009.

[110] ISSI, "18Mb Pipeline no wait state bus SRAM," IS61NLP25672/IS61NVP25672 Datasheet, 2011.

[111] Microchip, "32k I2C Serial EEPROM," 24AA32A Datasheet, 2005.

[112] Spansion, "MirrorBit Flash Family," S29GL-N Datasheet, 2007.

[113] STMicroelectronics, "NAND Flash Memory," NAND01G-B2B Datasheet, 2006.

[114] Wolfson microelectronics, “Audio CODEC with Headphone Driver and programmable Sample Rates,” WM8731 Datasheet, 2012.

[115] Analog Devices, “10-Bit, 4× Oversampling SDTV Video Decoder,” ADV7180 Datasheet, 2009.

[116] S. Vogel, “Analyse und Konzeption von FPGA-basierten Leiterplattenstrukturtests für high-speed Schnittstellen am Beispiel moderner HD-Bildsensoren,” Master Thesis, TU Ilmenau, Ilmenau, 2013.

[117] Marvell, “Integrated 10/100/1000 Ultra Gigabit Ethernet Transceiver,” 88E1111 Product Brief, 2009.

[118] Analog Devices, “Triple 10-Bit high Speed Video DAC,” ADV7123 Datasheet, 2009.

[119] Integrated Silicon Solution Inc ISSI, “512Mb SYNCHRONOUS DRAM,” IS42S86400B Datasheet, 2009.

[120] MIRA Deutron electronics, “1G bits DDR2 SDRAM,” P3R1GE3JGF Datasheet, 2009.

[121] C. Leong *et al.,* “Using FPGA Technology for static and dynamic Fault Detection in multi-bus, multi-FPGA, multi-Board Electronic Systems,” in *V Jornadas sobre Sistemas Reconfiguráveis (REC 2009)*: Proceedings, 2009.

[122] G. Hempel, C. Hochberger, and A. Koch, “A Comparison of Hardware Acceleration Interfaces in a Customizable Soft Core Processor,” in *International Conference on Field Programmable Logic and Applications (FPL 2010)*: IEEE Proceedings, 2010, pp. 469–474.

[123] J. L. Hennessy, D. A. Patterson, and K. Asanović, *Computer architecture: A quantitative approach,* 5th ed. Waltham, MA: Morgan Kaufmann, 2012.

[124] P. Mishra and N. Dutt, “Architecture description languages for programmable embedded systems,” (English), *IEEE Proceedings - Computers and Digital Techniques*, vol. 152, no. 3, 2005.

[125] P. Mishra and N. Dutt, *Processor description languages: Applications and methodologies*. Amsterdam, Boston: Morgan Kaufmann Publishers/Elsevier, 2008.

[126] P. Yiannacouras, J. G. Steffan, and J. Rose, “Exploration and Customization of FPGA-Based Soft Processors,” (English), *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 266–277, 2007.

[127] J. P. Langlois, G. Bois, and S. Vakili, “Customised soft processor design: a compromise between architecture description languages and parameterisable processors,” (English), *IET Computers & Digital Techniques*, vol. 7, no. 3, pp. 122–131, 2013.

[128] P. Mishra, A. Kejariwal, and N. Dutt, “Synthesis-driven exploration of pipelined embedded processors,” in *17th International Conference on VLSI Design (VLSID 2004)*: IEEE Proceedings, 2004, pp. 921–926.

[129] A. Krahn and E. Wolf, “Portierung eines vorgegebenen Soft-Core Prozessors auf unterschiedliche FPGA Umgebungen und Vergleich unterschiedlicher Soft-Core Prozessorrealisierungen,” Projektseminar, TU Ilmenau, Ilmenau, 2011.

[130] S. P. Dandamudi, *Guide to RISC processors: For programmers and engineers*. New York: Springer, 2005.

[131] J. Nurmi, *Processor design: System-on-chip computing for ASICs and FPGAs*. Dordrecht, London: Springer, 2007.

[132] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded computing: A VLIW approach to architecture, compilers and tools*. San Francisco, Calif: Morgan Kaufmann, 2005.

[133] C. E. LaForest and J. G. Steffan, “Efficient Multi-Ported Memories for FPGAs,” in *18th ACM/SIGDA international symposium on Field programmable gate arrays (FPGA 2010)*, New York, USA: Proceedings, 2010.

[134] H. Wong, V. Betz, and J. Rose, “Comparing FPGA vs. custom CMOS and the impact on processor microarchitecture,” in *19th ACM/SIGDA International Symposium on Field programmable gate arrays (FPGA 2011)*: Proceedings, 2011.

[135] H. Wong, V. Betz, and J. Rose, “Quantifying the gap between FPGA and custom CMOS to aid microarchitectural design,” (English), *IEEE Transactions on VLSI Systems*, 2013.

[136] A. S. Tanenbaum, *Modern operating systems,* 3rd ed. Upper Saddle River, NJ: Pearson/Prentice Hall, 2008.

[137] *IEEE standard for access and control of instrumentation embedded within a semiconductor device*, 2014.

[138] F. G. Zadegan, E. Larsson, A. Jutman, S. Devadze, and R. Krenz-Baath, “Design, Verification, and Application of IEEE 1687,” in *23rd Asian Test Symposium (ATS 2014)*: IEEE Proceedings, 2014, pp. 93–100.

[139] A. Ibrahim and H. G. Kerkhoff, “iJTAG integration of complex digital embedded instruments,” in *9th International Design & Test Symposium (IDT 2014)*: Proceedings, 2014, pp. 18–23.

[140] OpenCores, “Wishbone B4: Wishbone SoC interconnection architecture for portable IP cores,” opencores.org, 2010. [Online] Available: http://cdn.opencores.org/downloads/wbspec_b4.pdf. Accessed on: Jun. 30 2015.

[141] M. Sharma and D. Kumar, “Wishbone bus architecture: A survey and comparison,” (English), *International Journal of VLSI design & Communication Systems*, 2012.

[142] W. Stallings, *Computer organization and architecture: Designing for performance,* 6th ed. Upper Saddle River, NJ: Prentice Hall Pearson Education International, 2003.

[143] P. Harig, "Untersuchung und Implementierung einer Pipeline-Struktur für den ROBSY Prozessor zur Ausführung von Transfer und Arithmetisch-logischen Befehlen," Bachelor Arbeit, TU Ilmenau, 2013.

[144] Intellitech, *Intellitech supports Silicon Instruments through the new IEEE 1149.1-2013 JTAG standard: Presentation.* [Online] Available: http://www.intellitech.com/ijtag-sib/ijtag-instruments.pdf. Accessed on: Jul. 31 2015.

[145] Altera, *Quartus II Scripting Reference Manual.* [Online] Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/tclscriptrefmnl.pdf. Accessed on: Jul. 31 2015.

[146] B. Le Gal and C. Jego, "Softcore Processor optimization according to real-application requirements," (English), *IEEE Embedded Systems Letters*, vol. 5, no. 1, pp. 4–7, 2013.

[147] S. Rajotte, D. Carolina Gil, and J. P. Langlois, "Combining ISA extensions and subsetting for improved ASIP performance and cost," in *IEEE International Symposium on Circuits and Systems (ISCAS 2011)*: IEEE Proceedings, 2011, pp. 653–656.

[148] Xilinx, *Command line tools user guide.* [Online] Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/devref.pdf. Accessed on: Jan. 26 2016.

[149] Terasic, *DE2-115 User Manual.* [Online] Available: http://de2-115.terasic.com. Accessed on: Oct. 05 2015.

[150] Göpel Electronic, *VarioTap Coach Technical Description.* [Online] Available: http://genesis.goepel.com/. Accessed on: Oct. 05 2015.

[151] Altera, *Cyclone IV Device Handbook.* [Online] Available: https://www.altera.com/products/fpga/cyclone-series/cyclone-iv/support.html. Accessed on: Oct. 05 2015.

[152] Electronic Assembly, "EA DIP128-6 LCD Graphic Module 128x64 Dots," Datasheet, Electronic Assembly, 2013. [Online] Available: http://www.lcd-module.de/datenblaetter.html. Accessed on: Oct. 05 2015.

[153] D. Sheldon, F. Vahid, and S. Lonardi, "Soft-core processor customization using the Design of Experiments paradigm," in *Design, Automation & Test in Europe Conference (DATE 2007)*, 2007, pp. 1–6.

Erklärung

Ich versichere, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Bei der Auswahl und Auswertung folgenden Materials haben mir die nachstehend aufgeführten Personen in der jeweils beschriebenen Weise unentgeltlich geholfen:

1. Struktur des DUT-Modells. Dr.-Ing. Heinz-Dietrich Wuttke, Jörg Sachsse, und Steffen Ostendorff (TU Ilmenau).
2. Entwicklung der RTDL Sprache für die DUT-Modelle. Jörg Sachße und Steffen Ostendorff (TU Ilmenau).

Weitere Personen waren an der inhaltlich-materiellen Erstellung der vorliegenden Arbeit nicht beteiligt. Insbesondere habe ich hierfür nicht die entgeltliche Hilfe von Vermittlungs- bzw. Beratungsdiensten (Promotionsberater oder anderer Personen) in Anspruch genommen. Niemand hat von mir unmittelbar oder mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer Prüfungsbehörde vorgelegt.

Ich bin darauf hingewiesen worden, dass die Unrichtigkeit der vorstehenden Erklärung als Täuschungsversuch bewertet wird und gemäß § 7 Abs. 10 der Promotionsordnung den Abbruch des Promotionsverfahrens zur Folge hat.

Ilmenau, 01.06.2016 Jorge Hernan Meza Escobar