Test-Access Planning and Test Scheduling for Embedded Core-Based System Chips

Sandeep Kumar Goel

Promotiecommissie:

Voorzitter:	prof.dr.ir. A.J. Mouthaan	University of Twente, NL
Secretaris:	prof.dr.ir. A.J. Mouthaan	University of Twente, NL
Promotor:	prof.dr.ir. T. Krol	University of Twente, NL
Asst. Promotor:	dr.ir. H.G. Kerkhoff	University of Twente, NL
Leden:	prof.dr.ir. C.H. Slump	University of Twente, NL
	prof.dr. H. Wallinga	University of Twente, NL
	prof.dr. B.M. Al-Hashimi	University of Southampton, UK
	prof.ir. R. Segers	Technical University of Eindhoven, NL

Cover design:
Hennie Alblas and Sandeep Kumar Goel

Explanation of the cover:
The cover shows an example of an access infrastructure for a part of the Philips High Tech Campus, Eindhoven. In this thesis, test-access planning for embedded-core based system chips is addressed. The buildings in the cover can be considered as various cores in a system chip, while the roads correspond to test-access mechanisms.

Publisher:
University Press,
P.O. Box 513, 5600MB, Eindhoven, The Netherlands.

Goel, Sandeep Kumar

Test-access planning and test scheduling for embedded core-based system chips / by Sandeep Kumar Goel. - Enschede : University of Twente, 2005.
Proefschrift. - ISBN 90-74445-65-9
NUR 959
Trefw: system-on-chip / test scheduling / test architecture design / wrapper / test-access mechanism.

The work described in this thesis has been carried out at the Philips Research Laboratories in Eindhoven, The Netherlands, as part of the Philips Research programme.

©Royal Philips Electronics N.V. 2005
All rights reserved. Reproduction in whole or in part is
prohibited without the written consent of the copyright owner.

TEST-ACCESS PLANNING AND TEST SCHEDULING FOR EMBEDDED CORE-BASED SYSTEM CHIPS

PROEFSCHIRFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof.dr. W.H.M. Zijm
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op woensdag 2 februari 2005 om 13:15 uur

door

Sandeep Kumar Goel
geboren op 31 Augustus 1976
Meerut, India

Dit proefschrift is goedgekeurd door de promotor
prof.dr.ir. T. Krol

en de assistent-promotor
dr.ir. H.G. Kerkhoff

Preface

This thesis completes the research work that I have been doing in the field of testing of embedded-core based SOCs for past four years. It all started with my master's project here in Philips Research Laboratories, Eindhoven on the same topic. After finishing my master's project, I had several ideas in my mind in which I could easily extend the work described in the master's thesis. In 2000, when I joined the group Digital Design & Test at Philips Research Laboratories, Eindhoven, my research topic was *silicon debug*. Thanks to the Friday afternoon research concept, I started working on the testing of embedded-core based SOCs with in a year. Soon the ideas came down from my mind to paper and I published a couple of papers.

At the European Test Workshop (ETW'02) in Corsica, Greece, Prof. Bashir Al-Hashimi from the University of Southampton, UK, asked me whether I would be interested in doing a Ph.D. on the same topic. At that time, due to personal reasons I did not want to move to UK and leave Philips Research. Therefore, I decided to do a Ph.D. at one of the dutch universities. Later, my group leader Ad ten Berg introduced me to my promotor Prof. Thijs Krol and assistant promotor Associate Prof. Hans Kerkhoff from the University of Twente, Enschede. I am extremely happy for having the opportunity to pursue a Ph.D. while working in a industrial research environment. Four years of continuous work resulted in this thesis. During the course of work, I had numerous pleasant moments and experiences with several friends and colleagues. Therefore, I would like to thank everybody who has supported me over the last four years. I would specially like to thank the following people.

- My colleague Erik Jan Marinissen for teaching me the fundamentals of core-based testing. I also thank for his constant knowledge support and encouragement throughout this research work.

- Prof. Bashir Al-Hashimi from the University of Southampton, UK for giving me an idea to do a Ph.D. on the research work I was doing on this topic.

- My group leader Ad ten Berg for his support and introducing me to my promotors at the University of Twente, Enschede.

- My promotors Prof. Thijs Krol and assistant promotor Hans Kerkhoff for their support and critical review of this work.

- M.Sc. student Ludovic Krundel from ISIM, University of Montpellier II, France, and his supervisors Marie-Lise Flottes and Bruno Rouzeyre of LIRMM, France for their contribution to the work related to user constraints.
- Ph.D. student Anuja Sehgal and her supervisor Krishnendu Chakrabarty of Duke University, Durham, NC, USA for fruitful discussions during the conception of the work related to hierarchical constraints.
- My colleagues Jose Pineda de Gyvez, Maurice Meijer, Bart Vermeulen, and Harald Vranken for their useful comments on early draft versions of various papers published in several conferences based on the work described in this thesis.
- Kuoshu Chiu, Toan Nguyen, and Steven Oostdijk of Philips Semiconductors, San Jose, CA, USA for their contributions regarding the first use of TR-ARCHITECT on the Philips PNX8550 SOC.

Finally, I would like to thank my parents and parents-in-law for always believing in me and encouraging me for higher studies. Last but not the least, I would like to thank my wife Rachna for her true love, understanding, and patience through so many years. Especially, for always giving me the opportunity to get away from the daily work at home and focus on my thesis. Without her support, this thesis would not have been possible.

Eindhoven. Sandeep Kumar Goel
January, 2005

Table of Contents

Preface v

1 Introduction 1

1.1 Core-Based Design Paradigm . . . 1
1.2 Manufacturing Test . . . 3
1.3 Challenges in Testing Core-based SOCs . . . 5
1.4 Motivation . . . 8
1.4.1 Test Access Planning . . . 8
1.4.2 Test Scheduling . . . 8
1.5 Objectives . . . 9
1.6 Original Contributions . . . 9
1.7 Thesis Outline . . . 10

2 Core Test-Wrapper Design 13

2.1 Introduction . . . 13
2.2 Prior Work . . . 14
2.3 Wrapper Architecture . . . 16
2.4 Problem Definition . . . 17
2.5 TAM Chain Design . . . 18
2.5.1 Ordering of TAM Chain Items . . . 18
2.5.2 Partitioning of TAM Chain Items . . . 20
2.6 Proposed Algorithms . . . 21
2.7 Experimental Results . . . 24
2.8 Summary . . . 29

3 Test Architecture Design **31**
3.1 Introduction 31
3.2 Prior Work 32
3.2.1 Test Architecture Design 32
3.2.2 Test Architecture Optimization 34
3.3 Problem Definition 34
3.4 Lower Bound on Test Time 35
3.5 Test Bandwidth Utilization 38
3.5.1 Type-1 Idle Bits: Imbalanced Test Completion Times 38
3.5.2 Type-2 Idle Bits: Assigned to Non Pareto-Optimal Width . . . 38
3.5.3 Type-3 Idle Bits: Imbalanced Scan Chains 40
3.6 Test Architecture Design Algorithm 42
3.6.1 Creating a Start Solution 42
3.6.2 Optimize Bottom Up 43
3.6.3 Optimize Top Down 45
3.6.4 Reshuffle 47
3.6.5 Checking Empty Wire 48
3.6.6 TR-Architect Computational Complexity 49
3.7 Test Time Calculation 49
3.7.1 Test Bus Architecture Test Time 50
3.7.2 TestRail Architecture Test Time 50
3.8 Experimental Results 52
3.9 Summary 59

4 Layout-Driven Test Architecture Design **61**
4.1 Introduction 61
4.2 Prior Work 62
4.3 Wire-Length Cost Model 63
4.4 Optimal Ordering of Cores 65
4.5 Layout-Driven Test Architecture Design 68
4.5.1 Layout-Driven Creating a Start Solution 70
4.5.2 Layout-Driven Optimize BottomUp 71
4.5.3 Layout-Driven Optimize TopDown 72
4.5.4 Layout-Driven Reshuffle 73
4.6 Experimental Results 75
4.7 Summary 78

5 Control-Aware Test Architecture Design 79
5.1 Introduction . . . 79
5.2 Prior Work . . . 80
5.3 Test-Control Classification . . . 80
5.4 Pseudo-Static Test Control . . . 81
5.4.1 One WIR Chain per SOC . . . 84
5.4.2 One WIR Chain per TAM . . . 85
5.5 Dynamic Test Control . . . 86
5.5.1 On-Chip Generation . . . 87
5.5.2 Shift-Register Implementation . . . 88
5.5.3 Dedicated Chip Pins . . . 88
5.6 Experimental Results . . . 91
5.7 Summary . . . 95

6 User-Constrained Test Architecture Design 97
6.1 Introduction . . . 97
6.2 Prior Work . . . 98
6.3 Test Architecture Specification . . . 98
6.3.1 Keywords . . . 99
6.3.2 Example . . . 100
6.4 Test Architecture Design . . . 101
6.4.1 User Constraints . . . 102
6.4.2 User-Constrained Test Architecture Design . . . 104
6.5 Experimental Results . . . 107
6.6 Summary . . . 110

7 Test Architecture Design for SOCs with Hierarchical Cores 111
7.1 Introduction . . . 111
7.2 Prior Work . . . 112
7.3 Hierarchical Core Model . . . 113
7.4 Testing of Hierarchical Cores . . . 114
7.5 Improved Wrapper Architecture . . . 119
7.5.1 Testability of the Proposed Wrapper Cells . . . 122
7.5.2 Ordering of Elements in a TAM . . . 123
7.6 Experimental Results . . . 125
7.7 Summary . . . 128

8 Conclusions **131**

A List of Parameters **145**

B Computational Complexity **147**

B.1 Introduction 147

B.2 Creating a Start Solution 148

B.3 Optimize Bottom Up 148

B.4 Optimize Top Down 148

B.5 Reshuffle 149

B.6 Checking Empty Wire 149

Summary **151**

Samenvatting **153**

About the Author **155**

Chapter 1

Introduction

The market-driven electronics industry continuously requires products with greater functionality, higher reliability, lower costs and shorter time-to-market. These are realized by the unprecedented advancements in the semiconductor process technology. Semiconductor chips (ICs) are considered the foundation of modern electronic products. The need of faster and smaller products has driven the semiconductor industry to introduce a new generation of complex chips. These chips allow integration of a wide range of complex functions, which used to comprise a system, into a single die, called *System-On-Chip* (SOC).

Being able to rapidly develop, manufacture, test, and verify SOCs and products using such SOCs is very crucial for the continued success of our economy at-large. One of the major issues in making SOC production practical and cost effective is the complexity of creating a multi-million gate SOC from scratch using conventional methods and design flow. To overcome this problem, the design community has created a new chip design paradigm based on design reuse [KB99], in which an IC consists of multiple large pre-designed and pre-verified reusable building blocks, and only a few IC-specific modules. These large reusable building blocks are called *cores* and the design paradigm that utilizes these cores is called *core-based design paradigm*. The use of embedded cores reduces the design-development time through design reuse and allows import of external design expertise.

1.1 Core-Based Design Paradigm

Modern SOC designs often contain one or multiple programmable CPUs, DSP cores, application-specific hardware blocks, embedded memories of different types, and some analog modules [GZ97]. Therefore, embedded cores not only cover a wide range of system functions, but also contain an unprecedented range of design styles, from logic to DRAM to analog. Furthermore, they may come in hierarchical composition also. For instance, a complex core may embed one or more simple cores. Figure 1.1 shows

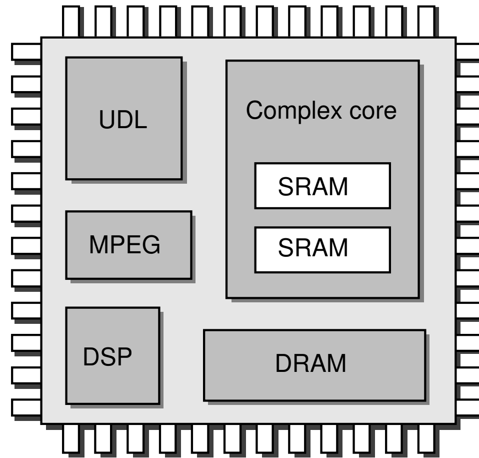


Figure 1.1: An example of a core-based SOC design.

an example of a core-based SOC design. The successful design of such complex SOCs requires expertise in diverse technical areas, which are increasingly hard to find in a single design house. Next, it is not economical for a typical organization to create all the required intellectual property (IP) functions by itself. Hence, IP cores are being delivered from multiple sources. The variety of sources provides a diversity of trade-off and formats, and opens the door for *plug-and-play* requirements in chip design. Dataquest [Smi97] reported that mainstream ASIC designers fill 90% of their silicon area with embedded cores, 40-60% of which using external cores and the rest from internally developed ones, while leaving only 10% of the chip for the application-specific user-defined logic (UDL).

The embedded cores come in a wide range of hardware-description levels. They spread from fully optimized layouts in GDSII format to widely flexible RTL (register-transfer level) codes. Based on the hardware description level, embedded cores can be classified into the following three major categories [VSI96]:

1. *Soft Cores*: a soft core is a synthesizable RTL code.

2. *Firm Cores*: a firm core is a synthesized RTL code or a generic -library gate-level netlist. A firm core that is not placed and routed, may be optimized with respect to performance and size for a few target technologies during placement.

3. *Hard Cores*: a hard core exists in layout and is already optimized for one or more key parameters, such as speed, size or power for a specific target technology. Hard cores come as placed and routed netlists, physical-layout files (GDSII format), or in a netlist/layout combination.

The three types of cores offer various trade-offs. Soft cores leave much of the implementation to their users, but are flexible and process-independent. Hard cores have been optimized for predictable performance, but lack flexibility. Firm cores offer

a compromise between the two. The core-based design paradigm divides the IC design community into two groups: *core providers* and *core users*. A core provider designs and maintains a library of reusable IP cores, while a core user is responsible for design and manufacturing of system chips using various cores and user-defined logic modules. Both core providers and core users can exist within a single company, as well as in different companies.

For many large semiconductor and system companies, the core-based design is not a totally new phenomenon, as they have practiced design reuse for years by integrating simple macros [BBT95] and in-house cores into ASICs. However, these cores were not mixed and matched from multiple external sources. The practical implementation of the core-based design paradigm with cores from diverse sources faces numerous challenges in the areas of SOC design, integration and test. The list of challenges is typically headed by the complexity of test and diagnosis. As the semiconductor industry keeps moving towards the creation of large and complex SOCs, testing of these chips is becoming more and more difficult. If no attention is paid to the specific issues that are related to design for testability (DfT) and test generation for complex SOCs, testing may become the bottleneck in their overall development trajectory. In the next sections, first a brief introduction to manufacturing test is given and then the main challenges of testing system chips are described [Zor97].

1.2 Manufacturing Test

Manufacturing test is a critical step in the IC manufacturing process. Due to imperfections in the manufacturing process, all chips need to be tested for manufacturing defects and system chips are no exception to that. IC test is performed multiple times during volume production to screen ICs upon their manufacturing. IC testing ensures that bad chips are not shipped to the customer and hence helps in meeting customer's quality requirements. Furthermore, IC test plays a key role in analyzing defects in the semiconductor manufacturing process. The feedback derived from the test is the only way to analyze and isolate many of the defects in today's processes. Time-to-yield, time-to-market, and time-to-quality are all controlled by test.

To test a circuit, test stimuli need to be applied to the circuit and test responses need to be observed from the circuit. The observed responses are then compared to the expected responses and if a mismatch is found, the circuit is considered to be defective. Test stimuli can be generated on-chip or off-chip. Similarly, test responses can be observed on-chip or off-chip. To model the physical effect of a defect, abstract fault models have been developed. The most popular is the *stuck-at* fault model [Eld59]. This model considers that a defect will cause an input or an output of a gate to be stuck at a constant '1' or '0' value. Some defects even causes propagation delays along paths in the circuit to fall outside the desired limits. To model these faults, two very commonly used fault models are the *gate-delay fault* [BR83,Wag85] and the *path-delay fault* [Smi85]. Gate-delay faults model those defects that occur at inputs or output of a gate and cause the gate delay to be outside its specified range. Path-delay faults model those defects that cause cumulative propagation delays along circuit paths.

Various test methods such as functional test, BIST, Iddq, and scan test, etc., are described in literature [ABF94, BA00]. Some of the test methods require modifications to the circuit-under-test. The main goal of all these test methods is to detect as many faults as possible under minimum test time and area overhead. A brief description of main test techniques are described below.

Functional Test

In case of a functional test, no fault model is assumed and no modifications are required in the circuit. The circuit is tested in the normal operating mode. Test stimuli are applied at the primary input terminals and the test responses are observed at the primary output terminals of the circuit-under-test. In a functional test, the functional behavior of the circuit is tested and therefore unless tested exhaustively, measuring the quality of the test is very difficult. Exhaustive testing requires a very large test-application time and hence is not feasible for large complex SOCs.

Built-In-Self-Test (BIST)

In Built-In-Self-Test (BIST), test stimuli are generated on-chip by means of a linear feedback shift register (LFSR), while the test responses are compacted into a digital signature by means of a multiple-input signature register (MISR). Therefore, BIST requires modifications to the circuit-under-test. One of the problems with BIST is the fault coverage and long test-application time due to the large number of test patterns. Typically, a LFSR generates pseudo-random patterns and hence it is very hard to detect all faults. In order to increase fault coverage, advanced techniques such as test points, deterministic BIST are required that results in a large area overhead.

Scan Test

Scan test is a well-known and often used test technique. Here, various scan paths are created in the circuit-under-test by means of shift-registers. These shift registers are commonly referred to as *scan chains*. Therefore, this method also requires modifications to the circuit-under-test. Test patterns are generated off-chip by using Automatic Test Pattern Generator (ATPG) tools. To test the circuit, test stimuli are shifted into the scan chains and applied to the primary input terminals. Then the circuit is run in the normal operation mode and finally, test responses are shifted-out from the scan chains and observed from the primary output terminals. The main advantages of scan testing are a very high fault coverage and low test-pattern count.

Iddq Test

Iddq testing [CT97] is a test technique based on measuring the quiescent supply current of the circuit-under-test. The decision criterion is based on the fact that a CMOS circuit

does not draw any significant current when in a stable state. In a quiescent state, only the leakage current flows, which in most cases can be neglected. The fact that under certain conditions a significant current flows when the circuit-under-test is in a quiescent state, indicates the presence of a manufacturing defect in the circuit. Application of test stimuli can be done by using scan chains inside the circuit. Unfortunately, for very deep-submicron processes, Iddq testing is becoming increasingly limited in use, due to increase in the standby current for these processes.

Delay Fault Test

Delay fault testing is used to detect defects that cause propagation delays along the paths in the circuit-under-test to fall outside the specified limits. To detect a delay fault, two-pattern technique is commonly used. In this technique, two test patterns are applied to the circuit in two consecutive clock cycles. The first test pattern is used to set a proper value at the input end of the path under test, while the second test pattern causes a rising or falling transition at the same end. By sampling the output end of the path under test after the desired interval, one can check the occurrence of a delay fault. Similar to the Iddq testing, once again, the application of test stimuli and observation of test responses can be done by using scan chains inside the circuit-under-test. For modern chips, where clock speed is in the range of GHz, delay fault testing is becoming more and more important.

1.3 Challenges in Testing Core-based SOCs

The use of pre-designed cores in an SOC design looks conceptually similar to the use of standard IC components in the design of traditional system-on-boards (SOBs) or printed-circuit-boards (PCBs). However, the manufacturing test processes in both cases are quite different. In the traditional system-on-board design, a board designer plugs in a number of stand-alone *pre-manufactured* and *pre-tested* IC components into a PCB. Whereas in a core-based system chip, an SOC designer needs to embed cores which are not yet manufactured and hence untested. Therefore, the SOC designer is responsible for manufacturing and testing of not only the interconnect between the cores, but also the cores themselves. Another key difference between SOB and SOC is the accessibility of component peripheries. In a SOB, direct physical access to every input and output terminal of the component is available for probing during manufacturing test for the SOB. Whereas, in case of an SOC, cores are deeply embedded in the SOC and direct physical access to peripheries of the cores is not available by default.

Apart from testing issues of the traditional deep-submicron chips, such as fault coverage, overall test cost and time-to-market, testing of system chips has three main testing challenges. These challenges are described below.

1. **Core Internal Test**

   A core is typically the hardware description of today's standard ICs, e.g. DSP, RISC processor, and DRAM. The internal test of a core is typically composed

of internal design-for-testability (DfT) structures (e.g. scan chains, test points, or BIST), if any, and the required set of test patterns to be applied and observed at the core peripheries. Generating a test for a core requires in-depth knowledge about the internals of the core. In most cases, except for soft cores, the core users have very limited knowledge about the internal structure of the used core. Therefore, it is very hard for a core user to prepare the test for it, especially if a core is hard one or is an encrypted intellectual property block.

This necessitates that core providers who own their cores and know about the internals of the cores also develop test solutions for their cores. This means that together with the description of the core, a core provider also delivers its test information, i.e. the DfT structures and the test patterns.

However, here a major issue for a core provider is to determine the type of DfT and the quality of the test without even knowing the environment in which the core will be used. Furthermore, the core internal test prepared by the core provider should be adequately described in a widely accepted and ready-to-use format such as the *IEEE Standard Test Interface Language* (STIL) [Soc99] or the proposed *IEEE Core Test Language* (CTL) [KKK+99, KLT+01].

2. **Core Test Access and Core Isolation**

   The core tests developed by the core providers are originally described at the input/output terminals of the core. In SOCs, cores are deeply embedded in the environment and usually their terminals are not directly accessible from the SOC pins. This necessitates the existence of test-access paths from the SOC primary pins to the embedded core and vice versa with sufficient bandwidth to fulfill the test requirement of the core. Furthermore, in order to apply the given set of tests to a core, the core must be isolated from its environment. The test access to embedded cores and isolation of cores are obviously the responsibilities of the SOC designer, who owns the system design and therefore knows the environment of an embedded core in the particular SOC.

3. **Test Integration and Scheduling**

   Once the tests for all cores have been developed, the SOC designer needs to develop tests for the interconnect wiring and logic between the cores. Furthermore, all these tests need to be translated from the core terminals to the SOC pins [ML99]. Finally, one needs to integrate the test facilities of all embedded cores and the interconnect circuitry under an SOC-level test-control mechanism. The SOC-level test-control mechanism is required to execute various tests and apply/capture the necessary test data. In addition to test integration, SOC test requires efficient test scheduling. The tests of various cores should be scheduled such that there are no conflicts, while the chip-level requirements like test time and power dissipation during test are satisfied.

Zorian et al. [ZMD98] presented a conceptual test architecture for testing embedded core-based SOCs that meets the test requirements described above. Figure 1.2 shows the conceptual test architecture for an example SOC. The architecture consists of the following three elements:

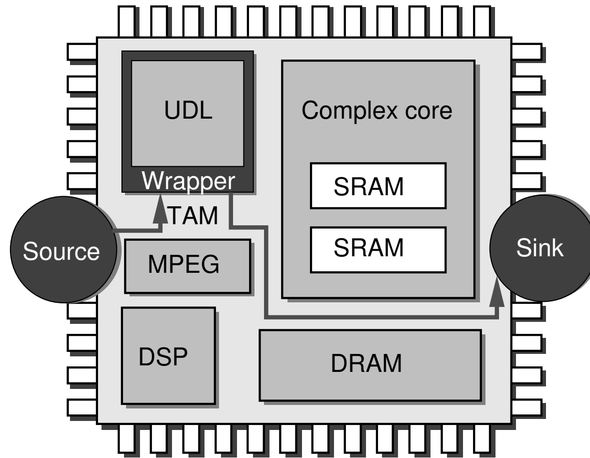


Figure 1.2: An example of the conceptual test architecture [ZMD98].

1. *Test pattern source and sink*

   A test pattern source generates test stimuli, whereas the test pattern sink receives test responses. Source and sink for a core can be implemented either off-chip by means of external Automatic Test Equipment (ATE), on-chip by means of Built-In Self-Test (BIST) or as a combination of both. Furthermore, source and sink do not need to be of the same type.

2. *Test access mechanism*

   A test-access mechanism (TAM) takes care of on-chip test data transport from the source to the core-under-test and from the core-under-test to the sink. Basically, a TAM contains a number of wires that bridges the physical distance between the source and a core, as well as between the core and the sink. TAMs can be implemented in various ways. For example, existing functional buses on the chip can be used to transport test data or a set of transparent paths can be created in the design.

3. *Core test-wrapper*

   A core test-wrapper is a thin shell around the core. The core test-wrapper forms an interface between the embedded core and its environment. It connects the terminals of the embedded core to the rest of the chip and to the TAM. In case of any mismatch between the number of core terminals and TAM width, the wrapper also provides *width-adaptation* by means of parallel to serial conversion and vice versa.

All three previously discussed elements can be implemented in various ways, such that a whole palette of possible approaches for testing embedded core emerges. Different implementations have their specific advantages and disadvantages, especially with respect to silicon area and test-application time.

1.4 Motivation

For the core-based design methodology to be successful, it is important to have proper tool support for insertion of design-for-test hardware such as wrappers and TAMs, test development and test application for complex SOCs. To design a core-based test architecture for an SOC, the SOC designer needs to design wrappers around all cores and needs to provide one or more TAMs to every core. Various cores in an architecture can also share the same TAM. For the core test-wrapper, the IEEE P1500 Standard for Embedded Core Test [HM] is an IEEE standard under development that defines a standard but scalable core wrapper architecture [DZW+03] to enhance test interoperability between multiple cores. However, the IEEE P1500 does *not* provide rules and algorithms to automatically design an optimal wrapper for a core. Therefore, efficient algorithms are required to generate an optimal wrapper around a core. Furthermore, the standard does not standardize TAM design and optimization, as this is exclusively in the domain of the SOC designer and depends on many SOC-specific parameters. Many tasks in this domain involve optimization of complex problems or elaborate bookkeeping and hence are very suited for automation. Two such tasks are *test-access planning* and *test scheduling*.

1.4.1 Test Access Planning

The term *test-access planning* refers to an activity that involves analysis of the chip-level resources and evaluation of the consequences of the usage of various TAM and wrapper types. It also involves trade off of configurations in terms of cost factors such as area, test time, performance impact, and test quality. This activity is also referred to as *test architecture design*. To design a test architecture for an SOC with a given set of cores and a given number of test pins, the SOC designer has to determine (1) the number of TAMs, (2) the TAM widths, (3) the assignment of cores to TAMs, and (4) the wrapper design for each core. For a small SOC, having only a few cores and a few test pins, a good test architecture can be designed manually. However, the complexity of designing an architecture increases with the increase in the number of cores and test pins. In fact, the problem of designing an optimal test architecture is $\mathcal{NP}$ hard [ICM01], indicating that the required computing time increases exponentially with the problem instance size. Therefore, techniques are required, which can efficiently search the solution space of feasible architectures and yield an optimal or near-optimal test architecture.

1.4.2 Test Scheduling

The size of the SOC-level test data set is an important cost factor, since it determines both the required storage capacity for test patterns of the automatic test equipment (ATE) and the test-application time. The test-data volume for modern SOCs is increasing dramatically and even faster than the number of transistors in SOCs. The increase in test-data volume can lead to a situation where an SOC has a large test-application

time and requires large and expensive automatic test equipment (ATE) with very deep test-vector memory per channel (pin) to store all the test data. At SOC level, there is at least one test per core, in addition to one or more tests for the interconnect wiring and logic circuitry. All these tests need to be executed such that there are no resource conflicts. The time required to execute all tests in an SOC architecture is the overall SOC test-application time. The term *test scheduling* refers to an activity that determines start times of the various core tests, such that no resource conflicts occur and the overall SOC test-application time is minimized or the power dissipation during test does not exceed a threshold level. There is a need for efficient test scheduling algorithms which minimize the overall SOC test time and help in fitting the test-data volume for the SOC on the target ATE.

1.5 Objectives

This thesis describes parts of the research that has been carried out at Philips Research Laboratories, Eindhoven, in the domain of testing embedded-core based system chips. As mentioned in the motivation, many tasks in this domain involve optimization of complex problems or elaborate bookkeeping and hence are very suited for automation. Two such tasks are *test-access planning* (or test architecture design) and *test scheduling*. Therefore, the objective of this research was to develop an automated tool [GM04b] that can assist SOC designers in selecting cost-effective test architectures and test schedules for their embedded-core based system chips. Such a tool will lead to better design decisions and high productivity. Consequently, it will also reduce time-to-market.

The basic requirements for such a tool are as follows:

- For a given SOC, the tool should design a test architecture such that the overall SOC test time is minimized. This objective will also help in fitting the test-data volume for the SOC on the target ATE.
- In order to have better acceptance by designers, the tool should be easy-to-use and must be able to satisfy the preferences of the user based on practical constraints such as layout, design hierarchy and other design-specific constraints.
- As both the test architecture design as well as test scheduling problems are $\mathcal{NP}$ hard, the tool should be efficient in terms of computing time or run time.

1.6 Original Contributions

The following original contributions have been described in this thesis:

- For the core-test wrapper, the problem of wrapper design is shown to be equivalent to the well-known *Multi-Processor Scheduling* (MPS) problem and hence

it is $\mathcal{NP}$ hard. Therefore, it is shown that various heuristic algorithms available for MPS problem can be used to design an optimal wrapper around a core.

- A lower bound on the overall SOC test time is defined and three different types of idle-bits are identified that make the lower bound unachievable in most of the practical cases.
- For test architecture design, a novel efficient and effective heuristic algorithm TR-ARCHITECT is presented. For a given SOC, TR-ARCHITECT computes an optimized test architecture with respect to SOC test time in a negligible computational time.
- The basic version of TR-ARCHITECT is extended to include layout constraints and a SOC-level test-control mechanism.
- To provide full control to the SOC designer over the test architecture design, a Test Architecture Specification (TAS) language, which can be used to specify a full or partial test architecture in a concise way, is presented.
- TR-ARCHITECT is extended in order to accommodate a wide range of hard-to-model user constraints specified by means of a TAS file.
- An improved wrapper architecture for efficient testing of hierarchical cores is presented. It is shown that by using the proposed wrapper architecture, optimal test schedules can be obtained for SOCs with hierarchical cores.

1.7 Thesis Outline

The rest of the thesis is organized as follows.

Chapter 2 addresses the issue of wrapper design for embedded cores. The relationship between wrapper design and test time is addressed. The problem of wrapper design is shown to be equivalent to the well-known $\mathcal{NP}$-hard problem of Multi-Processor Scheduling (MPS), and various heuristic algorithms to solve this problem are described. Finally, for a set of cores with a range of TAM widths, the test times obtained from the various heuristic algorithms are compared.

Chapter 3 addresses the issue of effective and efficient design of SOC test architectures consisting of wrappers and TAMs with respect to overall SOC test time. First, the problem of test architecture optimization is defined both for SOCs with hard and soft cores. Next, an architecture-independent theoretical lower bound on the test time of a given SOC is derived. Furthermore, three types of idle bits are classified and analyzed, which increase the test time beyond the theoretical lower-bound value. Subsequently, a novel test architecture optimization algorithm named TR-ARCHITECT is presented. Finally, experimental results are presented for the *ITC'02 SOC Test Benchmarks* [MIC]. It is shown that in negligible computational time, TR-ARCHITECT drastically outperforms manual best-effort engineering results, and on average also outperforms other test- architecture design algorithms.

In Chapter 4, to take layout constraints into account, a simple yet effective TAM wire-length cost model is presented. The wire length of a TAM depends on its width and the ordering of cores connected to the TAM. It is shown that the problem of determining an optimal ordering of cores connected to a TAM is equivalent to the well-known *Traveling Salesman Problem* (TSP) [GJ79] and a greedy algorithm is presented to solve it. Subsequently, a layout-driven version of TR-ARCHITECT is presented that takes into account the layout positions of all cores in the SOC and combines two costs i.e. SOC test time and TAM wire length into one cost function. Depending on the weight associated with each cost, TR-ARCHITECT computes an optimized test architecture.

Chapter 5 deals with the SOC-level test-control mechanism required for the SOC test architecture. Here, both the time required to set test modes between tests and the number of dedicated test control pins required for the execution of various tests are considered. The SOC-level test control is classified into two categories: (1) pseudo-static test control and (2) dynamic test control. To deal with pseudo-static test control, two test strategies are presented and their impact on the SOC test schedule are discussed. For dynamic test-control, a pin-constrained design of test architecture is presented.

Chapter 6 presents a novel Test Architecture Specification (TAS) language that can be used to specify a full or partial test architecture in a concise way. It is described how TR-ARCHITECT has been extended in order to accommodate a wide range of user constraints. The modified TR-ARCHITECT reads a TAS file as an input. All architecture parameters specified by the user are considered as constraints, while everything which is not specified, is left for the tool to optimize for minimal test time, TAM wire length, etc.

In Chapter 7, the problem of test architecture design for SOCs with hierarchical cores is addressed. First, a generic hierarchical core model is presented, and four different practical design scenarios that occur between two adjacent hierarchy levels are identified. Next, the testing requirements for a hierarchical core are discussed and an improved wrapper architecture that allows efficient testing of hierarchical cores is presented. By means of experimental results, it is shown that by using the proposed wrapper architecture, optimal test schedules can be obtained for SOCs with hierarchical cores.

Chapter 8 concludes this thesis and presents recommendations for future work. In Appendix A, a list of all relevant parameters used in this thesis is described. Appendix B presents the computational complexity analysis for the basic test architecture design algorithm TR-ARCHITECT.

Chapter 2

Core Test-Wrapper Design

2.1 Introduction

A core test-wrapper forms an interface between the core and its system-on-chip (SOC) environment. The wrapper connects the core terminals both to the rest of the SOC as well as to the test-access mechanism (TAM). A core test-wrapper provides the switching between the following three mandatory modes of operation:

1. Normal mode
2. Inward-facing or In-test mode
3. Outward-facing or Ex-test mode.

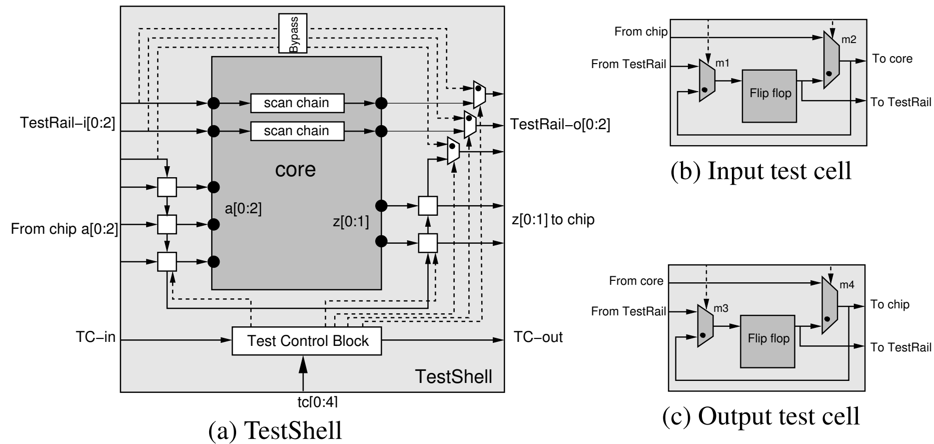
In the *normal* mode, the wrapper is transparent and the core is connected to its system environment. The *inward-facing* mode is used to test the circuitry inside the core itself. In this mode, the core wrapper is configured in such a way that the test stimuli can be applied at the core's input terminals and the test responses can be observed at the core's output terminals. In the *outward-facing* mode, the circuitry and wiring outside the core (i.e. interconnect logic) is tested. In this mode, the core wrapper is configured in such a way that the test stimuli for the interconnect logic after the outputs of this core can be applied at the core's output terminals, while the test responses coming from the interconnect logic before the inputs of this core can be observed at the core's input terminals. Apart from these mandatory modes, a core test-wrapper can provide several optional modes depending upon the requirements. Furthermore, the wrapper may provide width adaptation in case of a mismatch between the number of core's terminals and the TAM width, e.g., by means of serial-parallel and parallel-serial conversion. This will often be required in practice, since large cores typically have hundreds of core terminals, while the total TAM width is limited by the number of SOC pins.

This chapter addresses the issue of wrapper design for embedded cores. The wrapper is designed in such a way that the access requirements for normal mode, inward-facing mode, and outward-facing mode are met. The relationship between wrapper

design and test time will be addressed. Furthermore, it will be shown that the wrapper design problem is equivalent to the well-known $\mathcal{NP}$-hard problem of Multi-Processor Scheduling (MPS), and various heuristic algorithms to solve this problem will be provided. Finally, for a set of cores with a range of TAM widths, the test times obtained from the various heuristic algorithms will be compared.

2.2 Prior Work

Various wrapper architectures have been described in literature. Marinissen and others [MAB$^+$98] proposed a core test-wrapper named *TestShell* that is currently used within Philips. An example of this TestShell is shown in Figure 2.1(a). The example core shown in figure has two scan chains, three functional input terminals `a[0:2]`, and two functional output terminals `z[0:1]`. In this approach, TAMs are called *TestRails*. In principle, a TestShell is connected to the same TestRail at both input and output. Therefore, for a TestShell, the number of incoming and outgoing TAM wires are equal.



(a) TestShell

(b) Input test cell

(c) Output test cell

Figure 2.1: Conceptual view of the Philips TestShell [MAB$^+$98].

The TestShell consists of multiple test cells, an optional bypass register, a *Test Control Block* (TCB), and multiplexers to select various wrapper modes. Test cells provide controllability and observability at the core terminals. In principle there is one test cell for every core terminal, although some core terminals do not have a test cell associated with them. There are multiple types of test cells, e.g., depending on the direction of core terminals, such as input, output, and bi-directional. Examples of input and output test cells are shown in Figure 2.1(b) and Figure 2.1(c). The bypass register allows a TAM to bypass a core and its wrapper, in order to test another core that is connected to the same TAM. The Test Control Block (TCB) controls the operation of the wrapper and consists of a shift and an update register. The TCB operation is controlled from a few direct control inputs (`tc [0:4]`) to the TCB. The TestShell supports all three mandatory modes. i.e. normal, inward-facing, and outward-facing.

Apart from these three, TestShell also supports an additional mode called bypass mode. In the bypass mode, a core is bypassed in order to test another core that is connected to the same TAM.

Varma and Bhatia described a very similar wrapper, called *Test Collar* [VB98]. Apart from different naming for basically similar features, the main difference between this and the previously described approach is that the Test Collars do not have a bypass feature. The IEEE P1500 Standard for Embedded Core Test [HM] is an IEEE standard under development that defines a standard but scalable core test-wrapper architecture [MKL+02, DZW+03]. This wrapper (as shown in Figure 2.2) is very similar to the previously described TestShell and Test Collar.

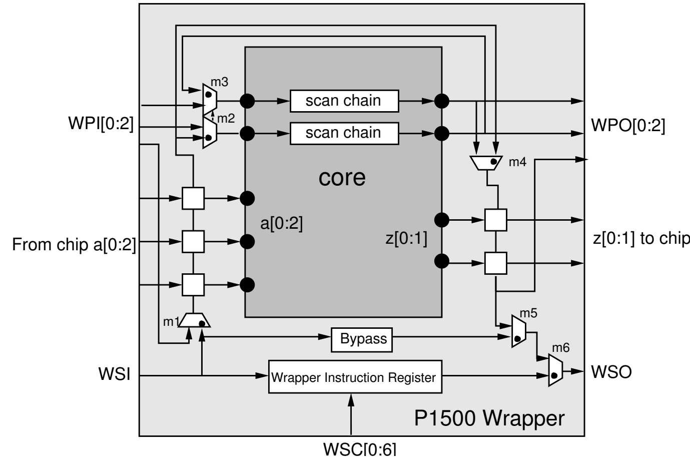


Figure 2.2: Conceptual view of the IEEE P1500 wrapper [DZW+03].

The P1500 wrapper has *Wrapper Boundary Cells* and a *Wrapper Instruction Register* (WIR) with similar functionality to respectively the test cells and the Test Control Block (TCB) in TestShell. Nevertheless, there are some remarkable differences between TestShell and the current P1500 proposal, which are as follows.

- *Number of TAMs*. The P1500 wrapper connects to one mandatory single-bit wide TAM with input and output terminals as `WSI` and `WSO` respectively. Furthermore, the P1500 wrapper also connects to zero or more multi-bit wide parallel TAMs with input and output terminals as `WPI` and `WPO`. A minimal compliant implementation has only the single-bit TAM, along which test control values for the WIR, as well as test stimuli and responses are transported. Envisaged typical usage has one multi-bit wide TAM next to the mandatory single-bit wide TAM. In that case, the bulk test-data access is performed along the multi-bit TAM, while the single-bit wide TAM is used to program the WIR. Multiple multi-bit wide TAMs are also allowed.

- *TAM widths*. For a multi-bit wide TAM, the number of incoming and outgoing wires do not need to be equal.

- *Bypasses*. The two wrappers allow for different types of bypasses. The TestShell has a TAM-wide bypass. The P1500 wrapper has a bypass for the single-bit wide TAM (enabled by multiplexer `m5` in Figure 2.2), next to the possibility to bypass the core-internal scan chains while accessing the wrapper boundary register (enabled by multiplexer `m4`).

The above publications on core test-wrappers provide general concepts for the wrapper architecture. All approaches seem to assume that automated generation of their particular wrapper architecture is possible. However, none of the above publications details the rules and algorithms required for such wrapper generator tools.

2.3 Wrapper Architecture

The wrapper architecture proposed in this chapter is very similar to the ones described in the section above. The proposed architecture basically unites the features of both the Philips TestShell [MAB+98] and the IEEE P1500 Wrapper [DZW+03]. An example of the proposed wrapper architecture is shown in Figure 2.3. For test-data access, the proposed architecture has one or more multi-bit wide TAMs. For each of the multi-bit wide TAMs, number of wires at the input and the output end of the wrapper are equal. For174 test-control access, the proposed architecture has one single-bit wide TAM, through which instructions are loaded into the WIR. This TAM can also be used to transport test stimuli and responses. A minimal implementation of the above equals the minimal implementation as proposed by the IEEE P1500.

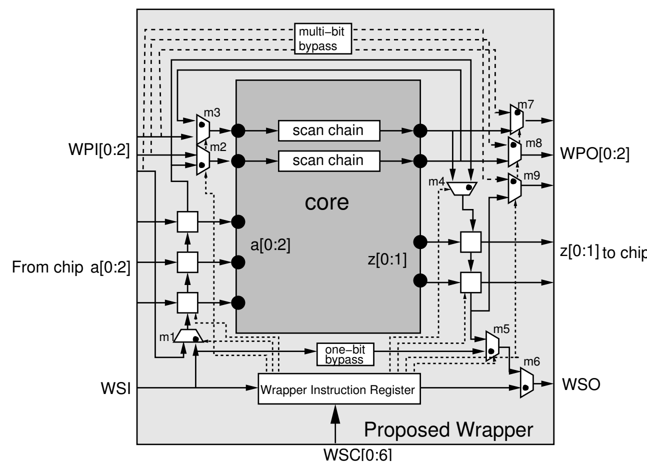


Figure 2.3: An example of the proposed wrapper architecture.

The wrapper contains a wrapper cell per core terminal. The type of wrapper cell required for a core terminal depends primarily on the type of core terminal and its corresponding (test) access requirements. In the wrapper, only digital synchronous terminals are connected to wrapper cells. For both the analogue and asynchronous signals, both the Philips [MAB+98] as well as the IEEE P1500 [DZW+03] have proposed

'direct test access', i.e. these signals pass the wrapper un-hindered and can for example be directly connected to IC pins. Furthermore, for bi-directional digital synchronous terminals, one can split a bi-directional terminal into a separate input, output, and direction control terminals. For the direction control terminal, direct test-access is assumed, while for input and output terminals wrapper cells are required. Example of input and output wrapper cells are shown in Figure 2.1(b) and Figure 2.1(c) respectively.

The wrapper cells and core-internal scan chains are connected into TAM chains in between the TAM inputs and outputs in order to meet the access requirements. In the proposed architecture, an optional bypass is allowed for every single wire in a TAM. This includes the optional bypass for the multi-bit wide TAM in the Philips TestShell, as well as the bypass for the single-bit TAM, which is mandatory in the IEEE P1500. In Figure 2.3, the top multi-bit bypass represents the bypass for the multi-bit wide TAM, while the bottom one-bit bypass represents the bypass for the single-bit wide TAM. Furthermore, in the proposed wrapper architecture optional bypasses of core-internal scan chains (not shown in Figure 2.3) are allowed. Following the Philips and P1500 wrapper examples, our bypasses contain not just wires and/or buffers, but also a register, which allows for concatenating an arbitrarily large number of cores in the same TAM chain.

A *Wrapper Instruction Register* (WIR), equivalent to the WIR in P1500 and the TCB of Philips, provides pseudo-static control signals to the wrapper itself. These signals control the mode of the wrapper by setting control signals of wrapper cells and various multiplexers. The WIR is implemented using a shift and update register. Via a single-bit interface, a new test control instruction is shifted into the WIR, which becomes active only after clocking it into the update register. The update register prevents invalid instructions from being given to wrapper and core while shifting in a new instruction.

2.4 Problem Definition

Wrappers are used both for core internal and core external testing. Optimizing the wrapper with respect to the test time for the core internal testing might lead to conflicting requirements with respect to optimization of the test time for the core external testing. In the typical case, the core internal circuitry is much larger than the circuitry that is used to interconnect the cores. Therefore, the test-data volume involved in core internal testing is much larger than the test-data volume for core external testing. Moreover, in many cases, the wrapper is designed by the core provider to whom the circuit environment in which the core will be used is not known. Hence, data about the core external test is not available at the time of the wrapper design. Therefore, priority is given to optimizing the test time for the core internal testing.

One can distinguish between the problems of wrapper design for SOCs with hard cores versus soft cores. Hard cores are those, for which the *scan insertion* has been already carried out. For hard cores, the number and length of the core-internal scan chains are fixed and cannot be changed while designing the wrapper. Examples of such cores are third-party black-boxed (layout) IP cores and encrypted cores, for which the

implementation, including the core-internal scan chain design, is fixed. Soft cores are those for which the *scan insertion* is yet to be done. For these cores, the number and length of the core-internal scan chains are not decided yet, or can still be changed while creating the the wrapper design. Examples of such cores are in-house design cores and third-party firm cores before scan insertion.

The problem of designing a wrapper around a hard core can be formally defined as follows:

[CTWD] **CORE TEST WRAPPER DESIGN**

Instance: Given a core c with a number of functional input terminals i_c, a number of functional output terminals o_c, a number of functional bi-directional terminals b_c, a number of test patterns p_c, a set of scan chains $\mathcal{S}$ and for each scan chain $S_i \in \mathcal{S}$, its length $l(S_i)$. Furthermore, a number $w_{\max}$ given is that represents the maximum number of TAM wires allowed to connect to the core.
Objective: Determine a wrapper design for the core such that the overall core internal test time t_c (in clock cycle) is minimized. □

Once the appropriate wrapper cells are selected for a given core, the remaining task in order to complete the wrapper design is to make the interconnections between the wrapper cells, the core internal scan chains, and the TAM wires. This activity is referred to as *TAM chain design*, and the elements (i.e. wrapper cells and scan chains) that make up a TAM chain are called *TAM chain items*. Test access is already guaranteed if all TAM chain items are accessible from the TAM wires.

In case of the wrapper design for a soft core, the wrapper design problem reduces to a simple balanced distribution of wrapper input cells, scan flip flops and wrapper output cells over the available TAM wires. As all the TAM chain items are of one-bit size, this problem is very trivial and can be solved optimally. Therefore, this problem will not be addressed here.

2.5 TAM Chain Design

The activity of TAM chain design consists of two parts:

1. Ordering the TAM chain items within TAM chains,
2. Partitioning the TAM chain items over the given TAM chains.

In this section, it will be shown that the partitioning over and ordering within TAM chains of the items has a large impact on the size of the resulting test time.

2.5.1 Ordering of TAM Chain Items

First, lets start with the test time t_c of a core c. Per test pattern, test stimuli need to be loaded into the wrapper input cells as well as into the core internal scan chains. The

time required to load the stimuli for a pattern is called scan-in time si_c. Similarly, test responses need to be unloaded from the core internal scan chains as well as from the wrapper output cells. The time required to unload the responses for a pattern is called scan-out time so_c. In practice, the scan-out time for a pattern is pipelined (in time) with the scan-in time for the next pattern. This reduces test time.

Considering the pipelining as mentioned above, the test time t_c for a core c can be derived as [GM01]:

$$t_c = \{1 + \max(si_c, so_c)\} \cdot p_c + \min(si_c, so_c) \tag{2.1}$$

where p_c represents the total number of test patterns for core c and should be greater than zero. Note that this formula is valid even for non-scan-testable cores, for which $si_c = so_c = 0$.

From the set of all TAM chain items, two non-disjunctive subsets are involved in the loading and unloading of test patterns. The wrapper input cells and the core internal scan chains (referred to as *input items*) participate in the loading of test patterns. The wrapper output cells and the core internal scan chains (referred to as *output items*) participate in the unloading of test patterns. In order to reduce si_c and so_c, it is best to order the items in any TAM chain such that the input items are at the head and the output items are at the tail of the TAM chain. Given the fact that core internal scan chains are in both sets, they should be in the middle of a TAM chain.

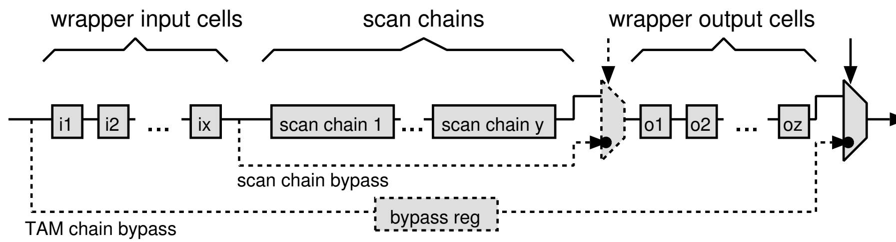


Figure 2.4: Ordering of TAM chain items (optional items are dashed).

Figure 2.4 shows a generic template for a single TAM chain. The items are ordered such that the TAM chain contains subsequently (1) wrapper input cells, (2) core internal scan chains, and (3) wrapper output cells. Optionally one can provide a *bypass* for the core internal scan chains. These scan chains do not take part in the core external testing. At the cost of a multiplexer and an additional control wire, one can reduce the length of the access chain by bypassing them during the core external tests.

Also optionally, one can provide a bypass for the entire TAM chain in the wrapper. Such a bypass is particularly useful if multiple cores are concatenated into a single TAM, such as is the case in the daisychain architecture as described in [AM98]. Cores which are not tested, can be bypassed in order to reduce the access length to cores which are tested. As multiple cores are concatenated into one TAM, this might lead to long TAM wires and hence long propagation delays. In order to prevent propagation delays from becoming too long and to contribute to the *plug-and-play* character of the proposed wrapper, it is proposed to equip the wrapper bypass with a register.

2.5.2 Partitioning of TAM Chain Items

The TAM width $w_{\max}$ is the result of a trade-off between its transport capacity and associated costs w.r.t. additional IC pins, silicon area, etc. [MAB+98]. Therefore, in many practical cases, the total number of TAM chain items is much larger than the TAM width. If this is the case, it is required that the set of TAM chain items is partitioned into a number of subsets equal to the number of available TAM wires.

The partitioning of TAM chain items over TAM wires determines the scan-in time si_c and scan-out time so_c for core c, and hence determines its test time t_c. As can be derived from Equation (2.1), the test time is minimal if the maximum of si_c and so_c is minimal. Hence, one should look for a partition of the TAM items that achieves this minimal test time. The partitioning problem can be formulated as finding an assignment of all TAM chain items to one of the available TAM chains such that the maximum of scan-in and scan-out test times is minimized. This problem can be formalized as follows:

[PTCI] **PARTITIONING OF TAM CHAIN ITEMS**
Instance: Given a set $\mathcal{WI} = \{WI_1, WI_2, \ldots, WI_x\}$ of *wrapper input cells*, each wrapper input cell having a *length* $l(WI_i) = 1$. Given a set $\mathcal{S} = \{S_1, S_2, \ldots, S_{|\mathcal{S}|}\}$ of *core internal scan chains*, where scan chain S_i has *length* $l(S_i)$. Given a set $\mathcal{WO} = \{WO_1, WO_2, \ldots, WO_z\}$ of *wrapper output cells*, each wrapper cell having a *length* $l(WO_i) = 1$. Furthermore is given a set of $w_{\max}$ identical TAM chains. It is defined that for any $X \subseteq \mathcal{WI} \cup \mathcal{S} \cup \mathcal{WO}$, $l(X) = \sum_{x \in X} l(x)$. A *TAM partition* is a partition $\mathcal{P} = \{P_1, P_2, \ldots, P_{w_{\max}}\}$ of $\mathcal{WI} \cup \mathcal{S} \cup \mathcal{WO}$ into $w_{\max}$ disjoint sets, one for each TAM chain. The *input set* is defined as $IN_i = P_i \backslash \mathcal{WO}$. Likewise, the *output set* is defined as $OUT_i = P_i \backslash \mathcal{WI}$. The *scan-in length* for TAM partition $\mathcal{P}$ is defined by $si(\mathcal{P}) = \max_{1 \leq i \leq w_{\max}} l(IN_i)$. The *scan-out length* for TAM partition $\mathcal{P}$ is defined by $so(\mathcal{P}) = \max_{1 \leq i \leq w_{\max}} l(OUT_i)$.
Objective: Find an *optimal* TAM partition $\mathcal{P}^*$ such that $\max(si(\mathcal{P}^*), so(\mathcal{P}^*)) \leq \max(si(\mathcal{P}), so(\mathcal{P}))$ for all partitions $\mathcal{P}$ of $\mathcal{WI} \cup \mathcal{S} \cup \mathcal{WO}$ into $w_{\max}$ subsets. □

To solve the PTCI problem, a three-step approach is proposed.

1. Assign the core internal scan chains in $\mathcal{S}$ to TAM chains, such that the maximum sum of scan lengths assigned to a TAM chain is minimized. The resulting partition is named $\mathcal{P}_{\mathcal{S}}$.

2. Assign the wrapper input cells in $\mathcal{WI}$ to TAM chains on top of $\mathcal{P}_{\mathcal{S}}$, such that the maximum scan-in time of all TAM chains is minimized.

3. Assign the wrapper output cells in $\mathcal{WO}$ to TAM chains on top of $\mathcal{P}_{\mathcal{S}}$, such that the maximum scan-out time of all TAM chains is minimized.

Note that wrapper input cells and wrapper output cells are of one-bit length, as both contain only one flip flop each. Therefore, Steps 2 and 3 of the proposed approach can yield an optimal solution in linear computation time, if Step 1 was solved to optimality.

Step 1 is the problem of partitioning of scan chains over TAM chains, and can be formalized as follows:

[PSC] **PARTITIONING OF SCAN CHAINS**

Instance: Given a set $\mathcal{S} = \{S_1, S_2, \ldots, S_{|\mathcal{S}|}\}$ of *core internal scan chains*, where scan chain S_i has *length* $l(S_i)$, and a set of $w_{\max}$ identical TAM chains. A *scan partition* is a partition $\mathcal{P} = \{P_1, P_2, \ldots, P_{w_{\max}}\}$ of $\mathcal{S}$ into $w_{\max}$ disjoint sets, one for each TAM chain. TAM chain i, $1 \leq i \leq w_{\max}$, contains all scan chains in P_i. The *scan length* for scan partition $\mathcal{P}$ is defined by $s(\mathcal{P}) = \max_{1 \leq i \leq w_{\max}} l(P_i)$, where for any $X \subseteq \mathcal{S}$, $l(X) = \sum_{S_i \in X} l(S_i)$.
Objective: Find an *optimal* scan partition $\mathcal{P}^*$, i.e. one that satisfies $s(\mathcal{P}^*) \leq s(\mathcal{P})$ for all partitions $\mathcal{P}$ of $\mathcal{S}$ into $w_{\max}$ subsets. □

The PSC problem is equivalent to the well-known problem of Multi-Processor Scheduling (MPS), sometimes referred to as Bin Design [JGJ78]. In the MPS problem, n independent tasks have to be non pre-emptively scheduled on m identical parallel processors with the objective of minimizing the 'makespan', i.e. the total time span required to process all given tasks. A formal version of the MPS problem is given below.

[MPS] **MULTI-PROCESSOR SCHEDULING**

Instance: Given a set $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ of *tasks*, where task T_i has execution *length* $l(T_i)$, and a set of m identical *processors*. A *schedule* is a partition $\mathcal{P} = \{P_1, P_2, \ldots, P_m\}$ of $\mathcal{T}$ into m disjoint sets, one for each processor. Processor i, $1 \leq i \leq m$, executes the tasks in P_i. A *finishing time* for schedule $\mathcal{P}$ is defined by $f(\mathcal{P}) = \max_{1 \leq i \leq m} l(P_i)$, where for any $X \subseteq \mathcal{T}$, $l(X) = \sum_{T \in X} l(T)$.
Objective: Find an *optimal* processor schedule $\mathcal{P}^*$, i.e. one that satisfies $f(\mathcal{P}^*) \leq f(\mathcal{P})$ for all partitions $\mathcal{P}$ of $\mathcal{T}$ into m subsets. □

It is not surprising that there is a direct one-to-one mapping between the PSC and MPS problems. In wrapper design, tasks are formed by the scan chains. The execution length of a task is equivalent to the length of a scan chain. The set of identical processors m corresponds to the set of identical TAM chains $w_{\max}$. Note that the equivalence of the two problems is also emphasized by the way they are formalized above. The MPS problem is $\mathcal{NP}$-hard [GJ79]. Because of the one-to-one mapping between PSC and MPS, it is claimed that the PSC problem is also $\mathcal{NP}$-hard.

2.6 Proposed Algorithms

In literature, various polynomial-time algorithms have been proposed for MPS that yield near-optimal schedules. A polynomial-time algorithm is evaluated by its worst-case performance ratio, which represents the possible relative error over all possible instances of the problem. A worst-case performance ratio of δ means that for every problem instance, the algorithm delivers a solution that is at most δ times the optimum. Naturally $\delta > 1$ and the closer it is to 1, the better.

Graham [Gra69] proposed the Largest Processing Time (LPT) algorithm, that first sorts the tasks such that $l(T_1) \geq l(T_2) \geq \ldots \geq l(T_n)$ and then assigns the tasks in succession to the minimally loaded processor. LPT has a time complexity of $\mathcal{O}(n \log n + n \log m)$. Graham proved that the worst-case performance ratio for LPT is $\frac{4}{3} - \frac{1}{3m}$. Algorithm 2.1 gives the pseudo-code of LPT, expressed in the variables of PSC.

Algorithm 2.1 [LPT]

(assume $w_{\max} < |\mathcal{S}|$)
1 **sort** $\mathcal{S}$ such that $l(S_1) \geq l(S_2) \geq \ldots \geq l(S_{|\mathcal{S}|})$;
2 **for** $i := 1$ **to** $w_{\max}$ **do** $P_i := S_i$; **od;**
3 **for** $i := w_{\max} + 1$ **to** $|\mathcal{S}|$ **do**
4 select $k \in \{j \mid l(P_j) = \min_{1 \leq x \leq w_{\max}} l(P_x)\}$;
5 $P_k := P_k \cup \{S_i\}$;
6 **od**;
7 **return** $\max_{1 \leq x \leq w_{\max}} l(P_x)$;

The Bin Packing problem can be seen as the dual version of the Bin Design (= MSP) problem. In Bin Design, a fixed number of bins is given, for which the minimum capacity needed to pack a set of given items has to be determined. In Bin Packing, the capacity of the bins is fixed, and the number of bins needed to pack all items has to be determined. An alternative approach to solve MPS is to utilize a bin-packing heuristic in conjunction with a search over the bin capacity C to find the minimum capacity such that all n items (tasks) will fit into (onto) the m bins (processors). For a given bin capacity, various bin-packing heuristics have been described in literature. Two very commonly used heuristics are First Fit Decreasing (FFD) [Joh73, JDU+74] and Best Fit Decreasing (BFD) [Joh73, JDU+74].

Assume that the tasks have been sorted such that $l(T_1) \geq l(T_2) \geq \ldots \geq l(T_n)$. The FFD heuristic assigns the tasks in succession to the lowest indexed processor which can complete the task within its capacity. Algorithm 2.2 gives the pseudo-code of FFD, expressed in the variables of PSC.

Algorithm 2.2 [FFD(C)]

(assume $\mathcal{S}$ is sorted such that $l(S_1) \geq l(S_2) \geq \ldots \geq l(S_{|\mathcal{S}|})$)
(assume initially $P_j = \emptyset$ for all j)
1 **for** $i := 1$ **to** $|\mathcal{S}|$ **do**
2 $j := 1$;
3 **while** $l(P_j) + l(S_i) > C$ **do**
4 $j := j + 1$;
5 **od**;
6 $P_j := P_j \cup S_i$; $l(P_j) := l(P_j) + l(S_i)$;
7 **od**;
8 **return** $\max\{j \mid P_j \neq \emptyset\}$;

Unlike the FFD, the BFD heuristic assigns the tasks in succession to the processor which is maximally loaded and can complete the current assigned tasks along with the

task in question within its capacity. In case there are more than one such processor, the processor with the lowest index is selected. Both FFD and BFD have the same worst-case performance ratio. Johnson [JDU+74] proved that the worst-case performance of FFD is $\frac{11}{9}$.

To decide on the capacity of the processor, the algorithm should use some search method. The MULTIFIT method, proposed by Coffman et al. [JGJ78], uses a bisection search over the bin capacity. Starting with known upper and lower bounds on the capacity C, at each step FFD is carried out for a value of C midway between the current upper and lower bounds. If $\text{FFD}(C) > w_{\max}$, C becomes the new lower bound; if $\text{FFD}(C) \leq w_{\max}$, C becomes the new upper bound. Initial lower and upper bounds on C are given as $C_L = \max\left(\frac{l(\mathcal{S})}{w_{\max}}, \max_{S_i \in \mathcal{S}} l(S_i)\right)$ and $C_U = \max\left(\frac{2 \cdot l(\mathcal{S})}{w_{\max}}, \max_{S_i \in \mathcal{S}} l(S_i)\right)$ respectively [JGJ78], where $l(\mathcal{S})$ represents the summed length of all scan chains in set $\mathcal{S}$.Algorithm 3 gives the pseudo-code of MULTIFIT, expressed in the variables of PSC.

Algorithm 2.3 [MULTIFIT]

(assume $\mathcal{S}$ is sorted such that $l(S_1) \geq l(S_2) \geq \ldots \geq l(S_{|\mathcal{S}|})$)
1 $C_L := \max\left(\frac{l(\mathcal{S})}{w_{\max}}, l(S_1)\right)$; $C_U := \max\left(\frac{2 \cdot l(\mathcal{S})}{w_{\max}}, l(S_1)\right)$;
2 **for** $i := 1$ **to** k **do**
3 **if** $C_L \neq C_U$ **then**
4 $C := \lfloor \frac{C_L + C_U}{2} \rfloor$;
5 **if** $\text{FFD}(C) > w_{\max}$ **then** $C_L := C$;
6 **else** $C_U := C$; **fi**;
7 **fi**;
8 **od**;
9 **return** $\mathcal{P}$;

Compared to LPT, the worst-case performance ratio of MULTIFIT is better, at the expense of additional computation time. It is important to note that this does not mean that MULTIFIT will have a better performance in all cases. Empirical tests show that in many cases LPT performs better than MULTIFIT. The time complexity of MULTIFIT is $\mathcal{O}(n \log n + kn \log m)$, where k denotes the number of iterations in the binary search. Coffman et al. [JGJ78] showed that MULTIFIT has a worst-case performance ratio of $1.22 + (\frac{1}{2})^k$. Later, Friesen [Fri84, FL86] proved that the worst-case performance ratio is even better, viz. $1.20 + (\frac{1}{2})^k$.

A problem with the binary search of MULTIFIT is that it can have the following anomalous behavior. If the tasks do not fit onto m processors with capacity C, they may still fit onto m processors with a capacity smaller than C. Hence, whereas binary search is useful to search quickly in a large search space, at the expense of additional computation time, linear search might obtain better results. Note that in many practical cases of wrapper design, the additional computation time required for a linear search will be acceptable as the linear search may provide a better solution and hence improve the test time. Based on a proposal by Lee & Massey [LM88], who use LPT to obtain a starting schedule for MULTIFIT, here a combination of LPT and LINEARSEARCH is suggested to design the proposed wrappers, if the range of C values is of accept-

able size. Algorithm 4 gives the pseudo-code of the resulting COMBINE algorithm, expressed in the variables of PSC.

Algorithm 2.4 [COMBINE] [1]

1 $A := \sum_{i=1}^{n} \frac{l(S_i)}{w_{\max}}$;
2 $X :=$ LPT;
3 **if** $X < 1.5 \cdot A$ **then**
4 $\quad C_U := X; C_L := \lfloor max(X/(\frac{4}{3} - \frac{1}{3w_{\max}}), S_1, A) \rfloor$;
6 $\quad i := C_L$; FFD(i);
7 $\quad$ **while** $i \leq C_U \wedge$ FFD$(i) > w_{\max}$ **do**
8 $\quad\quad$ FFD(i); $i := i + 1$;
9 $\quad$ **od**;
10 **fi**;
11 **return** $\mathcal{P}$;

It is important to note that although both MULTIFIT proposed by Coffman et al. [JGJ78] and COMBINE proposed here, use FFD for bin packing. However, as BFD utilizes more sophisticated partitioning rule than FFD, the use of BFD instead of FFD might obtain better results in some cases.

2.7 Experimental Results

Let us first consider wrapper design for an example core A. The example core A has five functional inputs `a[0:4]`, six functional outputs `z[0:5]`, and six internal scan chains of lengths resp. 25, 15, 18, 10, 5, and 8 flip flops. Its wrapper needs be connected to a three-bit wide TAM. Furthermore, for all TAM wires, the wrapper should have TAM chain bypass.

As the first step in a wrapper design procedure is the selection of appropriate wrapper cells, five wrapper input cells and six wrapper output cells are required for this case. For the PSC problem, the LPT algorithm yields $P_1 = \{25, 5\}$, $P_2 = \{18, 8\}$, and $P_3 = \{15, 10\}$. Hence, the longest scan-chain concatenation has 30 bits. The COMBINE algorithm improves this to the optimal partition $P_1' = \{25\}$, $P_2' = \{18, 10\}$, and $P_3' = \{15, 8, 5\}$ with a maximum length of 28 bits. Table 2.1 shows an optimal ordering and partitioning of TAM items for core A.

Table 2.1: Optimized ordering and partitioning of TAM items for core A.

TAM Wire	**TAM Input Items**		
		TAM Output Items	
TAM[0]	$\{1,1,1,1\}$	$\{25\}$	$\{1,1,1,1,1\}$
TAM[1]	$\{1\}$	$\{18,10\}$	$\{1\}$
TAM[2]	$\{\}$	$\{15,8,5\}$	$\{\}$

[1] Lee and Massey [LM88] proved that X is optimal if $X \geq 1.5 \cdot A$. Hence, if in the COMBINE algorithm, LPT already yields this result, LINEARSEARCH does not need to be executed.

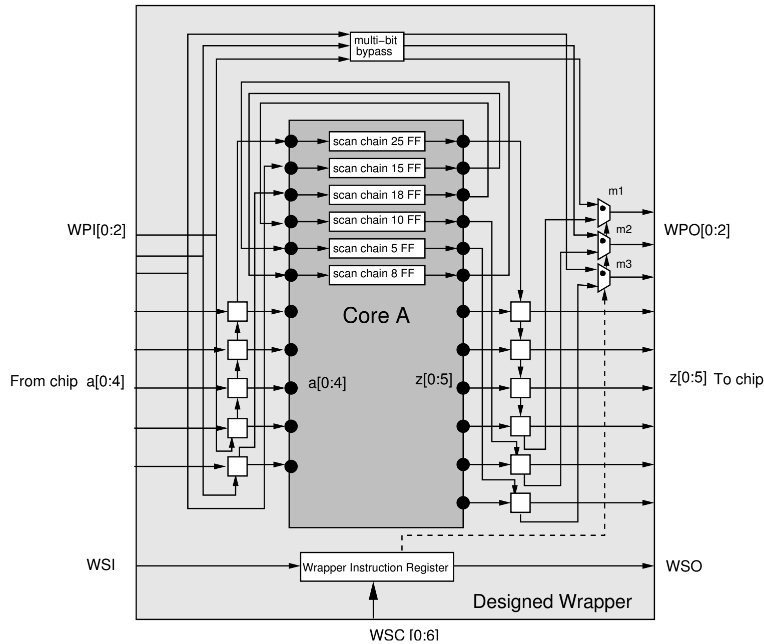


Figure 2.5: Core A with its designed wrapper.

For the optimal partition shown in the table, both the scan-in and scan-out time for core A are 29 bits. Figure 2.5 shows the corresponding wrapper for core A, including TAM chain bypass.

Next, experimental results are presented for the various wrapper design algorithms described in this chapter. As benchmarks, the set of *ITC'02 SOC Test Benchmarks* [MIC, MIC02] were used. This benchmark set contains twelve SOCs and every SOC contains a number of cores. Some of the cores inside the benchmark SOCs do not have any scan chains. Therefore the problem of PSC does not even exist for these cores. To show the impact of the wrapper design algorithm on core test time, experimental results for four large cores taken from the three out of twelve SOCs are presented. These cores are Module 26 from SOC p22810, Module 18 from SOC p34392, and Modules 1 and 13 from SOC p93791. Table 2.2 shows the test characteristics of these cores.

A test time comparison for five different wrapper design algorithms will now be presented. These algorithms are (1) LPT, (2) MULTIFIT using BFD as a subroutine,

Table 2.2: Test characteristics of four selected cores [MIC02].

Module ID	Number of					Scan Chain Length		
	Inputs	Outputs	Bi-dirs.	Scan chains	Patterns	Min.	Avg.	Max.
26	66	33	98	31	198	371	400	181
18	175	212	0	14	198	469	729	745
1	109	32	72	46	409	148	168	409
13	111	31	72	31	173	208	219	194

(3) MULTIFIT using FFD as a subroutine, (4) COMBINE using BFD, and (5) COMBINE using FFD. For all cores and all values of TAM width $w_{\max}$, the computation time for all five algorithms was less than one second. Fifty iterations were used in the binary search step of MULTIFIT $(k = 50)$. For Module 26 from SOC p22810, Figure 2.6 shows the test time results obtained from these five wrapper design algorithms for a range of TAM widths.

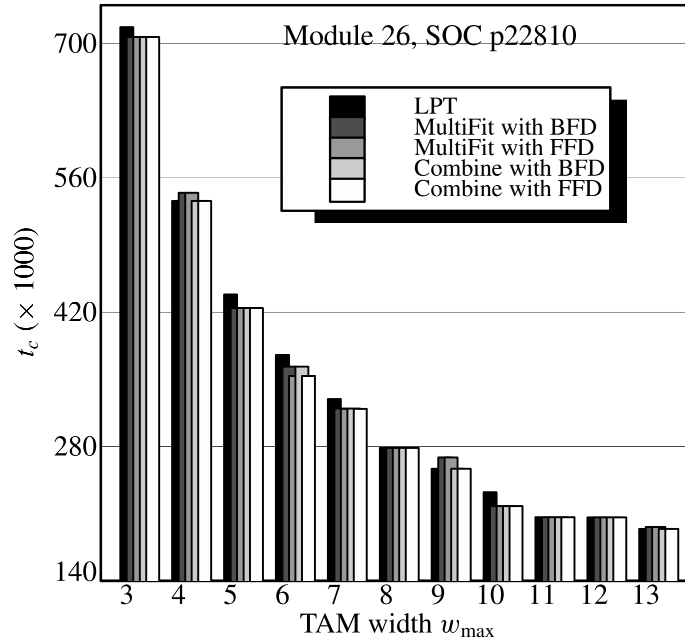


Figure 2.6: Test time comparison for Module 26, SOC p22810 [MIC02].

From Figure 2.6, one can see that the test time for a core shows a 'staircase' behavior. This is due to the fact that the wrapper design procedure involves the partitioning of the set of scan chains over the TAM wires. For increasing TAM width $w_{\max}$, the scan chains get redistributed over the TAM wires, resulting in another partitioning. However, the scan-in/out time per test pattern (and hence the overall test time for the core) only decreases if the increase in TAM width is sufficient to remove the bottleneck in scan time.

To understand this, consider a core with four internal scan chains, each having a length of 100 flip flops. If the TAM width assigned to this core is two ($w_{\max} = 2$), then the scan-in/out time per test pattern is 200 clock cycles. Now if the number of TAM wires is increased to three ($w_{\max} = 3$), the scan time does not decrease. This phenomena leads to 'staircase' behavior in case one plots the test time of a core as function of its TAM width. For Module 26 with $16 \leq w_{\max} \leq 21$ (not shown in figure), both the scan-in and scan-out time for the module are 798 clock cycles (using COMBINE with FFD). Only at $w_{\max} = 22$, both the scan-in and scan-out time reduce to 733 clock cycles and the core test time reduces from 145417 to 133405 clock cycles.

Now if one compares the results obtained from various wrapper design algorithms, one can easily see that for this core, the results obtained from the COMBINE algorithm either with FFD or BFD are slightly better or equal to the results obtained from LPT. This can be understood from the fact that the COMBINE algorithm takes the LPT solution as a starting point. One can also see that the results obtained from COMBINE are better or equal to the results obtained from corresponding MULTIFIT. This is due the fact that MULTIFIT uses a binary search which can have anomalous behavior, while COMBINE starts with LPT and uses a linear search. Here, COMBINE with FFD results in the minimum test time for all cases.

Figure 2.7 shows the test time results obtained for Module 18 from SOC p34392. Except for $w_{\max} = 3$, the results are similar to the ones obtained for previous case (Module 26, SOC p22810). For $w_{\max} = 3$, algorithms with BFD work better than the ones with FFD. Figure 2.8(a) and Figure 2.8(b) show the test time results obtained for Module 1 and Module 13 from SOC p93791 respectively. For these two cores, COMBINE with FFD always results in the minimum test time for all cases.

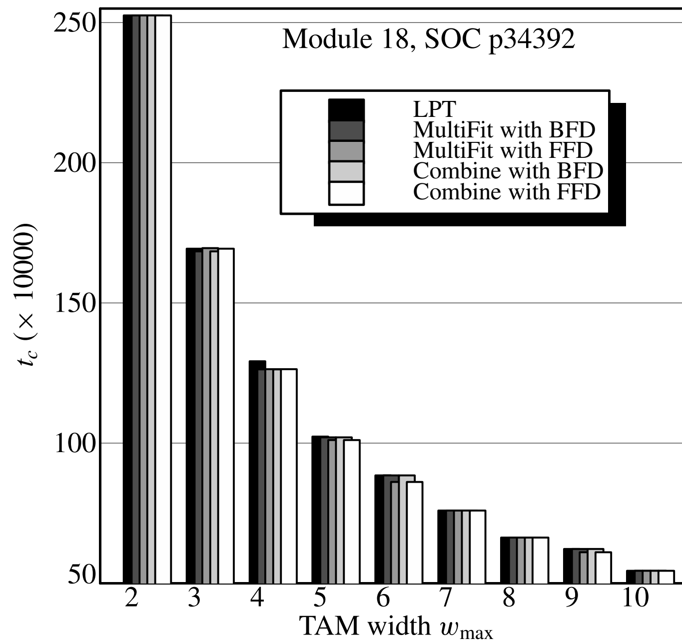
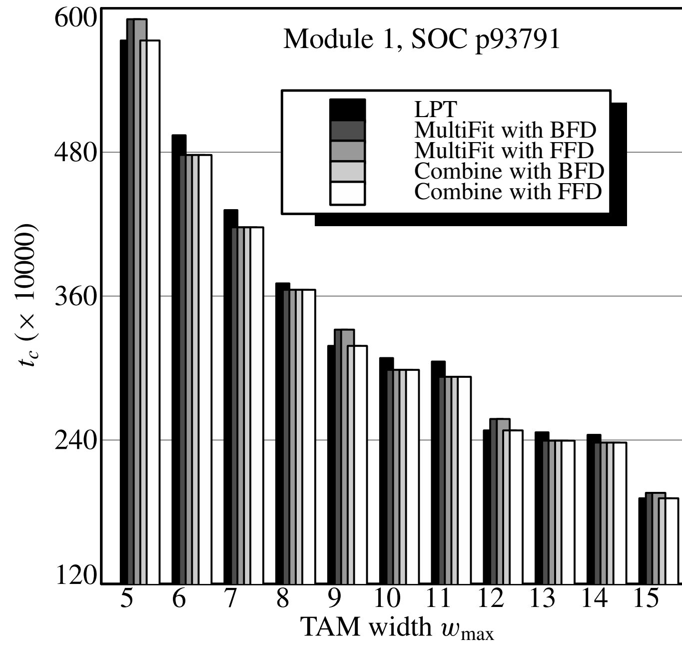
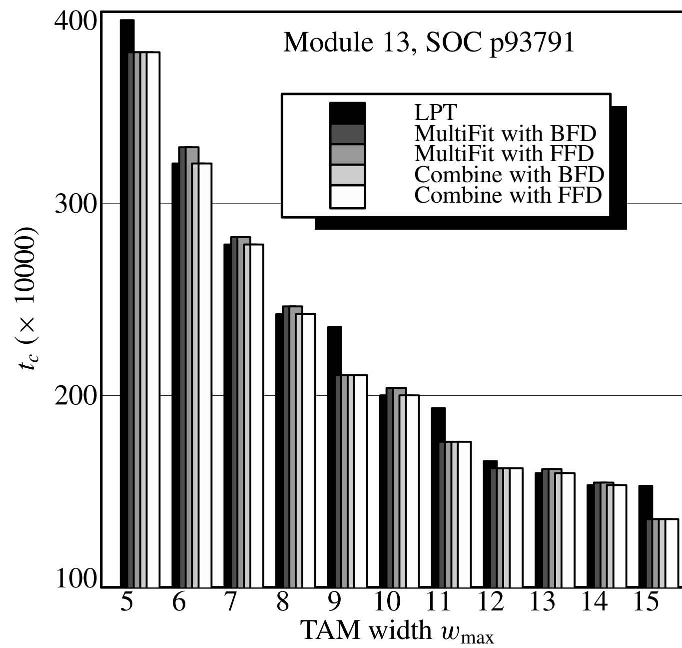


Figure 2.7: Test time comparison for Module 18, SOC p34392 [MIC02].

It is important to note here that even if there are small differences in the test times obtained from the various algorithms, selecting the algorithm that usually results the minimum test time is very important. It is due to the fact that a typical industrial SOC [MIC02] usually contains a number of cores. If for every core in the SOC, there is a small difference in the test time. In the worst-case, these small differences can add up and result in large SOC test time. As the COMBINE with FFD on an average results in the minimum test time, this algorithm is suggested for the core test-wrapper design.



(a) Module 1, SOC p93791



(b) Module 13, SOC p93791

Figure 2.8: Test time comparison for the two cores in SOC p93791 [MIC02].

2.8 Summary

Standardized, but scalable core test-wrappers play an important role in the test interoperability of embedded cores from distinct sources. In this chapter, a new wrapper architecture that unites the features of the TestShell of Philips and the IEEE P1500 wrapper was proposed. It provides facilities for functional access, as well as access for core internal and core external testing. The wrapper consists of wrapper cells and a Wrapper Instruction Register (WIR).

The interconnections of wrapper cells and core internal scan chains determine the test time of the core. Ordering these TAM chain items, together with the use of several optional bypasses, can reduce the test time. It was shown that the partitioning of TAM items over TAM chains such that test time is minimized, is equivalent to the $\mathcal{NP}$-hard problem of Multi-Processor Scheduling. Several heuristic algorithms to solve the partitioning of TAM items were described.

The proposed wrapper architecture is illustrated by means of an example. Finally, for a set of cores, a comparison of the test times obtained from the various wrapper design heuristics were presented. Experimental results show that the heuristic COMBINE in combination with FFD on an average performs better than others. Therefore in the sequel of this thesis, this algorithm will be used for core test-wrapper design.

Chapter 3

Test Architecture Design

3.1 Introduction

To design a test architecture for an SOC with a given set of cores and a given number of test pins, the SOC designer has to determine the following: (1) TAM type, (2) the number of individual TAMs, (3) the widths of these TAMs, (4) the assignment of cores to TAMs, and (5) the wrapper design for each core. These parameters need to be selected such that the total number of pins used to connect the TAMs wires does not exceed the given number of test pins, while the overall test cost is minimized. The test architecture has a large impact on two key parameters in the overall SOC test cost: the required vector-memory depth per tester channel and the test time of the SOC.

For a small SOC, having only a few cores and a few test pins, a good test architecture can be designed manually. However, the complexity of designing an architecture increases exponentially with the increase in the number of cores and test pins. Iyengar et al. [ICM01] proved that the problem of designing an optimal test architecture is $\mathcal{NP}$ hard, indicating that the required computing time increases exponentially with the problem instance size. Therefore, good heuristic algorithms which can efficiently search the solution space of feasible architectures and yield a near-optimal test architecture in a good reasonable compute time are required.

In this chapter, the issue of effective and efficient design of SOC test architectures consisting of wrappers and TAMs with respect to overall SOC test time is addressed. First, the problem of test architecture optimization is defined both for SOCs with hard and/or soft cores. Next, an architecture-independent theoretical lower bound on the test time of a given SOC is derived. Furthermore, three types of idle bits are classified and analyzed that increase the test time beyond the theoretical lower bound value. Subsequently, a novel test architecture optimization algorithm named TR-ARCHITECT is presented. Finally, experimental results are presented for the *ITC'02 SOC Test Benchmarks* [MIC, MIC02]. The test time results of TR-ARCHITECT are compared with those obtained by other methods and the theoretical lower bound. It is shown that in negligible compute time, TR-ARCHITECT drastically outperforms manual best-effort

engineering results and on average, also outperforms other-test architecture design algorithms.

3.2 Prior Work

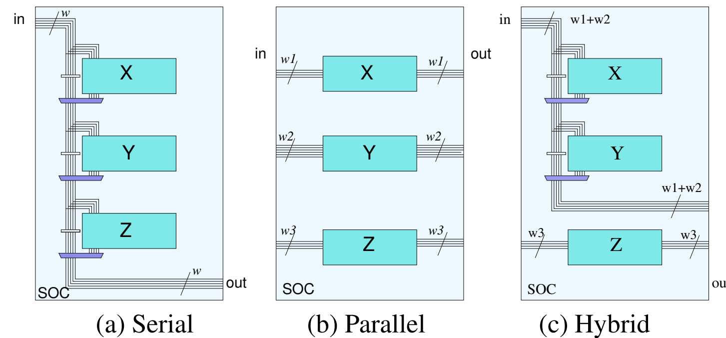
3.2.1 Test Architecture Design

In the design of test architectures, one can distinguish two issues: (1) the TAM type, and (2) the architecture type.

Two different *TAM types*, the test bus and the TestRail, have been proposed in the literature. In the *test bus* TAM as presented by Varma and Bhatia [VB98], cores connected to the same test bus can only be tested sequentially. The mutual exclusion for test bus access between multiple cores can be implemented by means of multiplexers and tri-state elements. The drawback of this TAM is that testing of logic between the cores (interconnect-logic) is difficult or impossible. This is due to the fact that in a test bus only one core wrapper can be accessed at a time, while that testing of interconnect-logic between any two cores requires access to both their wrappers simultaneously

The *TestRail* TAM as presented by Marinissen et al. [MAB+98], allows cores that are connected to the same TestRail to be tested simultaneously as well as sequentially. The TestRail can be implemented by simply concatenating the scan chains of the various cores and their wrappers. To minimize access time to cores, bypasses (optional) can be added around cores. In order to allow any number of cores to be connected to a TestRail, bypasses can be implemented as one-bit shift-registers. The advantage of a TestRail over a test bus is that it allows access to multiple or all cores wrappers simultaneously, which facilitates testing of interconnect-logic between the cores.

From literature, one can distinguish at least three different types of architectures: (1) the serial architecture, (2) the parallel architecture, and (3) the hybrid architecture. Example instances of the three architectures are shown in Figure 3.1.



(a) Serial (b) Parallel (c) Hybrid

Figure 3.1: Examples of SOC test architectures.

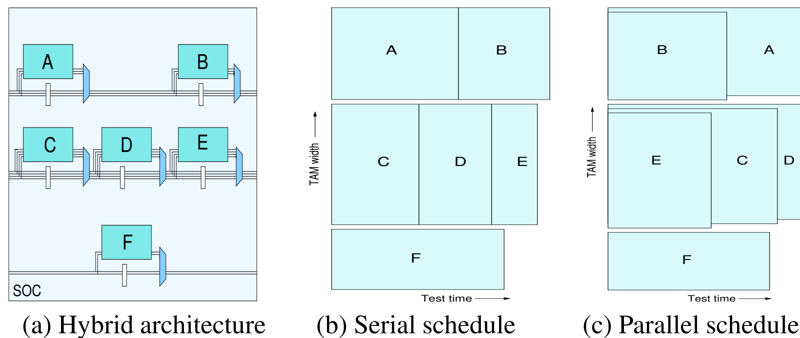
The serial architecture has only one TAM, which connects to all cores. In Figure 3.1(a), a serial architecture with TestRail TAM is shown. The shown TestRail is

connected to three cores X, Y, and Z. In this figure, optional registered-bypasses around the cores are also shown. The multiplexing and daisychain architectures in [AM98] are examples of such architectures. The multiplexing architecture has only one test bus TAM, while the daisychain architecture has only one TestRail TAM.

The parallel architecture shown in Figure 3.1(b) has a private TAM for every core. This architecture corresponds to the distribution architecture in [AM98]. As there is only one core per TAM, the TAM type is actually irrelevant in this type of test architecture. As each TAM needs to consist of at least one wire, this architecture requires that there are at least as many TAM wires as the number of cores. When designing this architecture, the partitioning of the total number of available TAM wires over the various cores has a large impact on the resulting test time.

The hybrid architecture is the hybrid combination of serial and parallel architectures. In this architecture, there are one or more TAMs, which each connects to one or more cores. This architecture is in fact a generalization, of which the serial and the parallel architectures are the two extremes. In Figure 3.1(c), a hybrid architecture with two TestRail TAMs is shown. One TestRail is connected to cores X and Y, while the other TestRail is connected to core Z.

Test architectures based on the test bus TAM only support *serial* test schedules; the cores connected to a common test bus are tested in an arbitrary, but sequential order [GM02c]. Parallelism only exists in case of multiple test buses, which operate in parallel. Test architectures based on the TestRail TAM support both *serial* and *parallel* test schedules. In a parallel test schedule, one starts to test all cores connected to a common TestRail in parallel. This continues until one of the cores runs out of test patterns. Then the bypass for this core is activated, while the testing of the remaining cores continues. This process is repeated until all cores connected to the TestRail have been completely tested. Figure 3.2(a) shows an example of a hybrid TestRail architecture with three TestRails. Figure 3.2(b) shows a possible corresponding serial test schedule, while Figure 3.2(c) shows a possible corresponding parallel test schedule. In Figure 3.2(b) and (c), the horizontal axis represents the test time, while the vertical axis represents the TestRail width. The rectangle boxes represent the tests of the cores.



(a) Hybrid architecture (b) Serial schedule (c) Parallel schedule

Figure 3.2: Example hybrid TestRail architecture with corresponding test schedules.

3.2.2 Test Architecture Optimization

Most test architecture optimization algorithms published so far, have concentrated on hybrid test bus architectures. Chakrabarty described an architecture optimization approach that minimizes test time through Integer Linear Programming (ILP) [Cha00b] and then extended the optimization criteria with place-and-route and power constraints [Cha00a]. Edabi and Ivanov replaced ILP by a genetic algorithm [EI01]. In [HCT+01], Huang et al. mapped the design of test architectures to the well-known problem of two-dimensional bin packing and used a heuristic algorithm to solve it. Iyengar et al. [ICM02a] solved the problem of integrated TAM/wrapper design by using ILP and exhaustive enumeration. In [ICM02b], the same authors presented efficient heuristics for the same problem. A heuristic optimization algorithm based on rectangle packing for a test bus architecture with one single test bus that is allowed to fork and merge was presented in [ICM02d]. In [ICM02c], the same authors extended this work by including precedence (ordering constraints between tests), pre-emption (tests can be halted and resumed later), and power constraints.

Two heuristic algorithms for co-optimization of wrappers and TAMs for hybrid TestRail architectures are described in [GM02b]. The algorithms in [GM02b] have a limitation that the total TAM width should be greater than or equal to the number of cores inside the SOC. Therefore the approaches presented in [GM02b] are not suitable for small TAM widths.

3.3 Problem Definition

One can distinguish between the problems of test architecture design for SOCs with hard cores versus soft cores. Hard cores are those, for which the *scan insertion* has been already carried out. For hard cores, the number and length of the core-internal scan chains are fixed and cannot be changed while designing the SOC-level test architecture. Soft cores are those for which the *scan insertion* is yet to be done. For these cores, the number and length of the core-internal scan chains are not decided yet, or can still be changed while creating the SOC-level design.

The problem of designing a test architecture for an SOC with hard cores can be formally defined as follows. A list of all relevant parameters used in this thesis is described in Appendix A.

[TADHC] **TEST ARCHITECTURE DESIGN WITH HARD CORES**
Instance: Given an SOC with a set of cores C. For each core $c \in C$, the number of test patterns p_c, the number of functional input terminals i_c, the number of functional output terminals o_c, the number of functional bidirectional terminals b_c, the number of scan chains s_c, and for each scan chain k, the length of the scan chain in flip flops $l_{c,k}$ are given. Furthermore, a number $w_{\max}$ is given that represents the maximum number of SOC-level TAM wires that can be used.
Objective: Determine a test architecture such that the overall SOC-level test time (in number of clock cycles) is minimized and $w_{\max}$ is not exceeded. □

The TADHC problem is $\mathcal{NP}$-hard, as was shown in [ICM02a]. A variant of the above problem is the one that assumes *soft* cores. In that case, the number of scan chains s_c and the length of these scan chains $l_{c,k}$ are not given. Instead, for each core c the number of scan flip flops f_c is given, and s_c and $l_{c,k}$ need to be determined such that $f_c = \sum_{k=1}^{s_c} l_{c,k}$.

The problem of designing a test architecture for an SOC with soft cores can be formally defined as follows:

[TADSC] **TEST ARCHITECTURE DESIGN WITH SOFT CORES**
Instance: Given all parameters as specified in the problem TADHC, and for each core $c \in C$ instead of the number of scan chains s_c and the length $l_{c,k}$ for each scan chain k, the total number of scan flip flops f_c is given.
Objective: Determine a test architecture such that the overall SOC-level test time (in number of clock cycles) is minimized and $w_{\max}$ is not exceeded. □

Many practical SOCs will actually contain a mix of hard and soft cores and to adapt for such cases, a parameter per core can be used to indicate whether a core is hard or soft. Even though both problem formulations require data on the core-internal scan flip flops, this does not mean that the problems are limited to scan-testable cores only. Both problem definitions are equally well applicable to logic cores with full scan (where f_c equals the flip flop count of the core in question), partial scan (where f_c equals the *scan* flip flop count of the core in question), and no scan (where $f_c = 0$). The latter case is also applicable to non-logic cores, such as embedded memories, which per definition have no core-internal scan chains.

3.4 Lower Bound on Test Time

As the test architecture design problem is $\mathcal{NP}$-hard, a lower bound on the SOC test time is very useful to measure the performance of any test architecture design algorithm, especially the one presented in this chapter. In this section, an architecture-independent lower bound on the SOC test time is presented. The presented lower bound LB_T consists of two components LB_1 and LB_2.

The first component LB_1 is an architecture-independent lower bound presented in [Cha01].

$$LB_1 = \max_{c \in C} \{\min_{1 \leq i \leq w_{\max}} t(c,i)\} \tag{3.1}$$

where $t(c,i)$ denotes the test time for core c with TAM width i. The idea behind this lower bound is that every core $c \in C$ requires a test time of at least $\min_{1 \leq i \leq w_{\max}} t(c,i)$, and hence the overall SOC test time cannot be smaller than the maximum of these minimum core test times.

LB_1 is a tight lower bound only in those architectures where the SOC test time is determined by one core ('bottleneck') with a large test time. A core is called a 'bottleneck' if increasing its TAM width does not further reduce its test time. LB_1 was originally defined in [Cha01] for SOCs with hard cores. However, the same equation can also be used to calculate a lower bound for an SOC with soft cores.

In many architectures, multiple cores are connected to one TAM and together determine the overall SOC test time. In such cases, LB_1 is not a tight lower bound and can be improved. The second component of the presented lower bound is referred to as LB_2 and considers the total test-data volume that needs to be shifted into and out of the SOC, given the total TAM width $w_{\max}$. Hereby, it is assumed that all cores have ideally balanced scan chains, or, in other words, all cores are considered to be 'soft' (even if, in reality, they are 'hard').

In order to derive LB_2, the lower bound on the test time of a core is tackled first. Per test pattern, test stimuli need to be loaded into the wrapper input cells as well as into the core-internal scan chains of the core-under-test. Similarly, test responses need to be unloaded from the core-internal scan chains as well as from the wrapper output cells of the core-under-test. If for a core c, terms ts_c and tr_c represent the test stimuli bits and test response bits per test pattern, then for a hard core, ts_c and tr_c can be defined as follows.

$$ts_c = i_c + b_c + \sum_{k=1}^{s_c} l_{c,k} \tag{3.2a}$$

$$tr_c = o_c + b_c + \sum_{k=1}^{s_c} l_{c,k} \tag{3.2b}$$

Similarly, for a soft core, ts_c and tr_c can be defined as follows.

$$ts_c = i_c + b_c + f_c \tag{3.2c}$$

$$tr_c = o_c + b_c + f_c \tag{3.2d}$$

In practice, the unloading of the responses of one test pattern is overlapped in time with the loading of the stimuli of the next test pattern. Therefore, a lower bound on the test time of core c can be defined as follows:

$$LB_c = \left\lceil \frac{\max(ts_c, tr_c) \cdot p_c + \min(ts_c, tr_c)}{w_{\max}} \right\rceil + p_c \tag{3.3}$$

The second term of Equation (3.3) represents the time needed to apply/capture the test patterns; this depends only on the number of test patterns and is independent from the TAM width $w_{\max}$.

An architecture-independent lower bound on the test time of the entire SOC is in principle the sum of the lower bounds for all individual cores. However, in case of multiple TAMs, it is possible that the TAM with the largest test time contains only the core with the smallest number of test patterns. Hence the second term of Equation (3.3) is taken out of the summation in Equation (3.4) and replaced by the minimum of the test pattern counts for all cores in the SOC. Also, the unloading of the responses of the last pattern for a core can be overlapped in time with the loading of stimuli of the first pattern for the another core. Therefore, a lower bound LB_2 on the test time of a SOC can be written as:

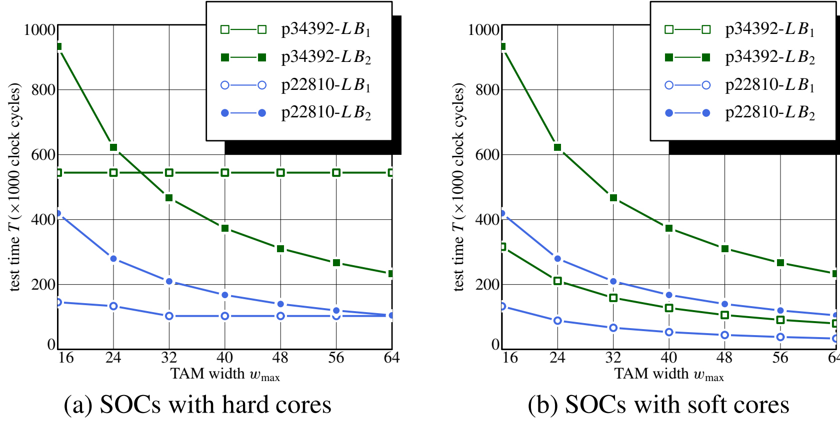
$$LB_2 = \left\lceil \sum_{c=1}^{|C|} \frac{\max(ts_c, tr_c) \cdot p_c + \min(ts_c, tr_c) - \min(tr_c, ts_{c+1})}{w_{\max}} \right\rceil + \min_{c=1}^{|C|} p_c \tag{3.4}$$

where $ts_{|C|+1} = 0$ (as there are $|C|$ cores only). The lower bound LB_2 is a tight lower bound only when the SOC test time is determined by a number of cores connected to the same TAM. On contrary, the lower bound LB_1 is a tight one in the case when the SOC test time is determined by the test time of a single core ('bottleneck'). Therefore, a general lower bound LB_T which is valid and tight in all cases, can be written as:

$$LB_T = \max(LB_1, LB_2) \tag{3.5}$$

Note that this lower bound does *not* take into account the test time required for the interconnect tests of the top-level SOC itself and assumes that all cores in the SOC are at the same level of design hierarchy.

For two Philips benchmarks SOCs [MIC], Figure 3.3(a) and (b) graphically display, for a range of values for $w_{\max}$, the values of LB_1 and LB_2 considering hard and soft cores respectively. LB_1 yields different values for the cases with hard cores and the cases with soft cores, whereas LB_2 is independent of the type of core. Exact values of LB_1 and LB_2 are listed in Table 3.1.



(a) SOCs with hard cores (b) SOCs with soft cores

Figure 3.3: Lower bounds on test time for two Philips SOCs.

Table 3.1: Architecture-independent lower bounds on test time for two Philips SOCs.

$w_{\max}$	SOC Benchmarks					
	SOC p22810			SOC p34392		
	LB_1		LB_2	LB_1		LB_2
	Hard Cores	Soft Cores	Hard/Soft Cores	Hard Cores	Soft Cores	Hard/Soft Cores
8	278641	265350	838930	663193	631857	1865451
16	145417	132856	419466	544579	316301	932790
24	133405	88631	279644	544579	211116	621903
32	102965	66609	209734	544579	158896	466459
40	102965	53324	167787	544579	127564	373193
48	102965	44406	139823	544579	105931	311016
56	102965	38218	119848	544579	91011	266603
64	102965	33486	104868	544579	79821	233294

From Figure 3.3(a), one can see that in case of hard cores, the curve for LB_1 becomes horizontal for large values of $w_{\max}$. If that happens, there is at least one core that forms the bottleneck in T and for which increasing its $w_{\max}$ does not decrease T any more. This phenomenon does not occur for LB_2, which continues to decrease for increasing $w_{\max}$. Hence, for large values of $w_{\max}$, LB_1 becomes the dominant lower bound. However, for many practical values of $w_{\max}$, the lower bound component LB_T^2 really improves the lower bound by making it tighter. In Figure 3.3(a), for SOC p34392 and $w_{\max} \geq 28$, $LB_1 > LB_2$. In all other cases, the lower bound component LB_2 provides a tighter lower bound, which underlines the value of the presented lower bound analysis.

For the cases with soft cores, values of LB_2 are usually higher than the values of LB_1 as shown in Figure 3.3(b), which means that for SOCs with soft cores also, LB_2 is a tighter lower bound than LB_1.

3.5 Test Bandwidth Utilization

The lower bound LB_T described in the previous section assumes that all available TAM width $w_{\max}$ can be used without any under-utilization for subsequent testing of all cores in the SOC. In practice, this is often not achievable. If the lower bound is not reached, this implies that one or more TAM wires are used to feed *idle bits* into and out of the SOC. In this section, three different types of idle bits are defined and analyzed.

3.5.1 Type-1 Idle Bits: Imbalanced Test Completion Times

In general, scheduling can be defined as the allocation of limited resources to tasks over time [Pin95]. In case of test scheduling, the tasks are the tests of the various cores and the limited resources are the TAMs, or defined at a finer grain, the individual TAM wires. The overall SOC test time T is defined by the completion time of the last core test on any TAM. In a concrete schedule, one TAM might complete its tasks before other TAMs do. Between the completion time of an individual TAM and the overall completion time, the TAM in question is not utilized. This type of under-utilization is referred to as *Type-1 idle bits*. In general, it is the objective of scheduling approaches to minimize this type of idle time.

Figure 3.4 shows an example of a serial test schedule. In the figure, the horizontal axis represents the test time and the vertical axis represents the TAM width. In the shown figure, there are three TAMs of widths 3, 4, and 2. The overall completion time is determined by the first TAM containing cores A and B. Due to different completion times for individual TAMs, there are some Type-1 idle bits which are indicated by means of a dark-grey shade.

3.5.2 Type-2 Idle Bits: Assigned to Non Pareto-Optimal Width

As described in Chapter 2, wrapper design for a core involves the partitioning of the set of scan elements of the core over the TAM wires assigned to the core. The set of scan

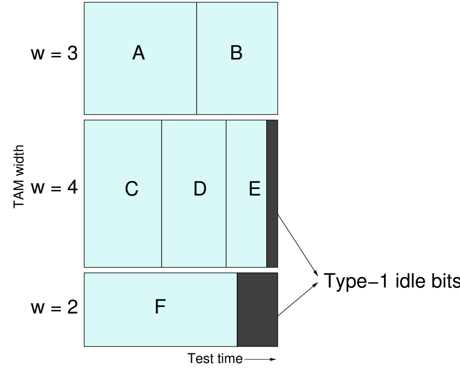


Figure 3.4: Example of a serial test schedule showing Type-1 idle bits.

elements of a core consists of the wrapper input cells, the wrapper output cells, and the core-internal scan chains. For an increasing TAM width $w_{\max}$, the scan elements get redistributed over the TAM wires, resulting in another partitioning. However, the scan time per test pattern (and hence the test time for the core) only decreases if the increase in TAM width is sufficient to remove the bottleneck in scan time.

Consider a core with four fixed-length internal scan chains of length 100 flip flops each. If assigned two TAM wires ($w = 2$), the scan time per test pattern is 200 clock cycles, as shown in Figure 3.5(a). In Figure 3.5(a), the vertical axis represent the TAM width and the horizontal axis represents the scan time (in number of clock cycles). If the number of TAM wires is increased to three ($w = 3$), the scan time does *not* decrease, as shown in Figure 3.5(b). Instead, due to differences in the total scan length for various TAMs, there are 200 idle bits per test pattern.

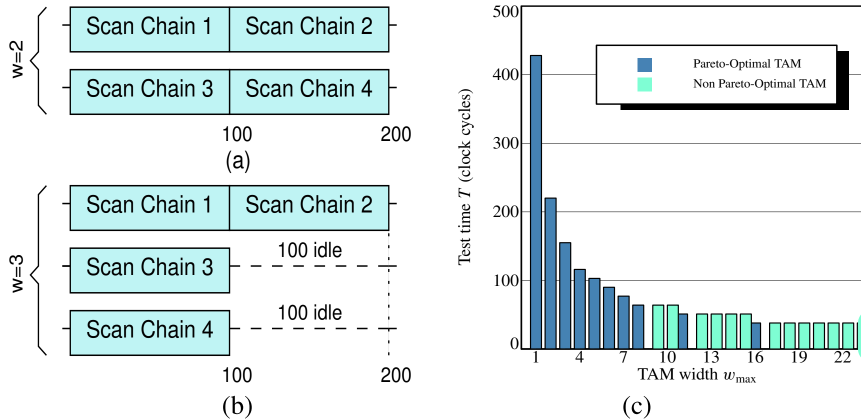


Figure 3.5: Example showing increasing TAM width $w_{\max}$ does not always lead to reduced test time.

This phenomenon leads to a 'staircase' behavior in case the test time of a core is plotted as function of its TAM width $w_{\max}$. Figure 3.5(c) shows the test time as

function of TAM width for core 1 of SOC d695 [MIC]. In this figure, the staircase behavior can be clearly recognized. For example, increasing the TAM width from 11 to 15 does not reduce the test time, as the next improvement in test time is only obtained for $w_{\max} = 16$.

For a core c, a TAM width w for which holds that $t_c(w-1) > t_c(w)$ (where t_c is the test time for core c and $t_c(0) = \infty$) is known as a *Pareto-Optimal* TAM width of core c [ICM02a]. In Figure 3.5(c), the Pareto-Optimal TAM widths are represented by the dark-colour (blue) bars and the non Pareto-Optimal TAM widths are represented by light-colour (green) bars. If a core is assigned to a TAM with a non Pareto-Optimal TAM width, the redundant bits transported along the excess TAM wires are referred to as Type-2 idle bits. Note that Type-2 idle bits are a serious problem for hard cores only, i.e. cores with fixed-length scan chains.

3.5.3 Type-3 Idle Bits: Imbalanced Scan Chains

Even if only the wrappers with the Pareto-Optimal TAM widths are considered, there might be still under-utilization of the available TAM width due to imbalanced scan chain lengths assigned to TAM wires. Idle bits due to imbalanced wrapper scan chains after wrapper design, are called Type-3 idle bits. The phenomenon is explained by means of the example depicted in Figure 3.6(a). In Figure 3.6(a), the vertical axis represent the TAM width and the horizontal axis represents the scan time (in number of clock cycles).

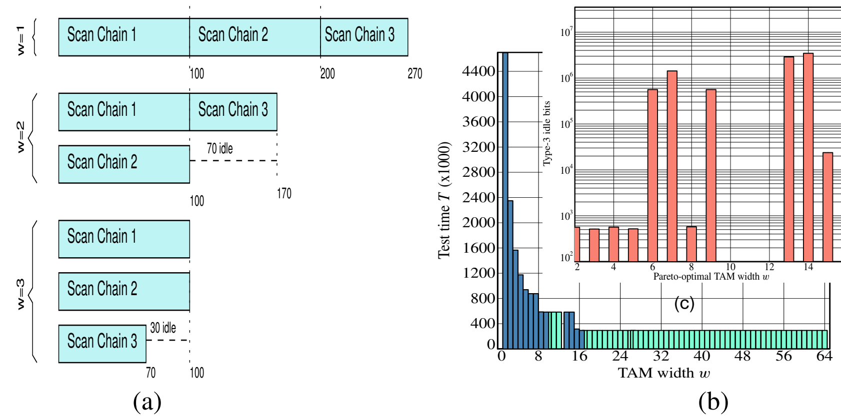


Figure 3.6: (a) Example showing the cause of Type-3 idle bits, (b) Pareto-Optimal TAM widths for core 2 in SOC p34392, and (c) number of Type-3 idle bits for core 2 in SOC p34392 at Pareto-Optimal TAM widths.

Consider a core with three internal scan chains of lengths 100, 100, and 70 respectively. In Figure 3.6(a), the cases $w = 1$, $w = 2$, and $w = 3$, are considered which are all Pareto-Optimal. For the case $w = 1$, all core-internal scan chains are assigned to this one TAM wire, and per definition, there are no imbalanced scan chains. Hence, there are no Type-3 idle bits. However, for the cases $w = 2$ and $w = 3$ there are

respectively 70 and 30 bits of Type-3 idle bits per test pattern, due to the imbalanced scan chain lengths.

Figure 3.6(b) shows that core 2 of SOC p34392 has the following set of Pareto-Optimal TAM widths: $\{1-9, 13-16\}$. In Figure 3.6(b), the Pareto-Optimal TAM widths are represented by the dark-colour (blue) bars and the non Pareto-Optimal TAM widths are represented by light-colour (green) bars. Figure 3.6(c) then shows the total amount of Type-3 idle bits for these Pareto-Optimal TAM widths, which amounts to $> 3 \cdot 10^6$ bits for some cases.

The Type-3 idle bits due to imbalanced scan chains are caused by fixed-length core-internal scan chains. For soft cores, the scan chains can be designed such that Type-3 idle bits are limited to at most one-bit per TAM wire. To understand this phenomena, consider a soft core with f_c flip flops and a TAM width of $w_{\max}$. Now if f_c is an integer multiple of $w_{\max}$, then all $w_{\max}$ TAM wires will contain the same number of flip flops and there will be no Type-3 idle bits. Otherwise, there will be a difference of at most one flip flop among the number of flip flops connected to the TAM wires. For example, if $f_c = 100$ and $w = 2$, then each of the two TAM wires will connect to 50 flip flops and there will not be any Type-3 idle bits. For case $w = 3$, one TAM wire will connect to 34 flip flops and the other two TAM wires will connect to 33 flip flops each. Therefore, one Type-3 idle bit per TAM wire for the two TAM wires. Similarly, for other values of w also, one can easily see that there will be at most one Type-3 idle bit per TAM wire.

For SOC d695 [MIC] with $w_{\max} = 16$, Figure 3.7 shows an example of a serial test schedule, corresponding to a hybrid test bus architecture. In the figure, the numbered boxes depict the tests of the cores and the number inside a box represents the core ID. At the end of each TAM, the shown number represents the test time (in number of clock cycles) for the TAM. The three types of idle bits in the schedule are indicated by means of different grey shades. This schedule contains 12,598 Type-1 idle bits which corresponds to 2% of the total bits in the schedule. Similarly, the schedule contains 25,386 Type-2 idle bits (= 4%) and 9,681 Type-3 idle bits (= 2%).

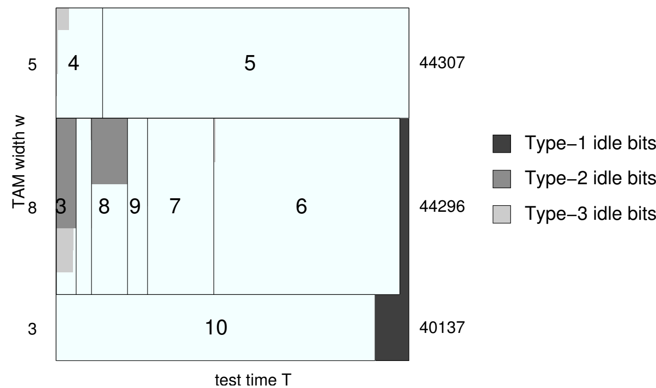


Figure 3.7: Example of a serial schedule for SOC d695 with $w_{\max} = 16$.

3.6 Test Architecture Design Algorithm

As a hybrid test architecture is a generalized form of all test architectures, in this section an effective and efficient algorithm called TR-ARCHITECT is presented for designing a hybrid test architecture for an SOC. The algorithm optimizes a test architecture for a given SOC with respect to its test time. It uses the input parameters as described in the problems TADHC and TADSC in Section 3.3. TR-ARCHITECT is based on fixed-width TAMs, i.e. it does not allow forking and merging of TAMs as in [ICM02d, ICM02c].

In TR-ARCHITECT, a TAM r is represented as a set of cores, which are connected to r. For an SOC test architecture, TR-ARCHITECT determines the following:

- the set of TAMs R, such that $C = \bigcup_{r \in R} r$ and $\forall_{r_1, r_2 \in R} \, (r_1 \cap r_2 = \emptyset)$, i.e. every core is assigned to exactly one TAM,
- the width $w(r)$ of every TAM $r \in R$, such that $\sum_{r \in R} w(r) \leq w_{\max}$, i.e. the summed width of the TAMs does not exceed $w_{\max}$,
- the wrapper design for every core in the SOC,

such that the overall SOC test time T is minimized. The overall SOC test time T is the maximum of the test times of the individual TAMs in the SOC test architecture. To determine the test time $t(r)$ for a TAM r with width $w(r)$, the existence of a procedure TESTTIME$(r, w(r))$ is assumed. For designing a wrapper around a core c, the procedure TESTTIME uses a procedure WRAPPERDESIGN$(c, w(r))$. The implementation details with regards to these procedures are given in Section 3.7.

The algorithm TR-ARCHITECT has five main steps, as shown in Algorithm 3.1. Each of these steps is explained in more detail in the sequel of this section.

Algorithm 3.1 [TR-ARCHITECT]

1 CREATESTARTSOLUTION;
2 OPTIMIZE-BOTTOMUP;
3 OPTIMIZE-TOPDOWN;
4 RESHUFFLE;
5 CHECK-EMPTYWIRE

3.6.1 Creating a Start Solution

The procedure CREATESTARTSOLUTION, as outlined in Algorithm 3.2, is meant to create an initial test architecture, which will be further optimized by the procedures to follow. It consists of a short initialization, followed by three main steps.

In Step 1 (Lines 3–8), cores are assigned to one-bit wide TAMs. If $w_{\max} \geq |C|$, each core gets assigned; if $w_{\max} < |C|$, only the largest $w_{\max}$ cores get assigned.

'Large' is here defined by the test-data volume for each core, according to which the cores have been sorted in Line 1. In case $w_{\max} = |C|$, the procedure is finished.

In case $w_{\max} < |C|$, there are still some un-assigned cores. In Step 2 (Lines 9–15), these cores are added iteratively to the one-bit wide TAM with the shortest test time. This procedure is in fact based on the Largest Processing Time (LPT) algorithm [Gra69] (see Section 2.6) for Multi-Processor Scheduling.

In case $w_{\max} > |C|$, there are still some un-used TAM wires. In Step 3 (Lines 16–22), these wires are added iteratively to the TAM with the longest test time.

Algorithm 3.2 [CREATESTARTSOLUTION]

```
 1  sort C such that TestTime({1}, 1) ≥                 /* sort cores in non-increasing order of their
      TestTime({2}, 1) ≥ ... ≥ TestTime({|C|}, 1);        test-data volume */
 2  R := ∅;                                             // initially, the set of TAMs R is empty
 3  for i := 1 to min(w_max, |C|) do                    // Step 1: iteratively, assign cores to one-bit TAMs
 4     w(r_i) := 1;                                     // create a one-bit wide TAM
 5     r_i := {i};                                      // connect a core to it
 6     t(r_i) := TestTime(r_i, w(r_i));                 // calculate the test time of the created TAM
 7     R := R ∪ {r_i};                                  // add the created TAM to the set of TAMs R
 8  od;
 9  if w_max < |C| then                                 // Step 2: if cores left, add to least-occupied TAMs
10     for i := w_max + 1 to |C| do                     // iteratively, assign remaining un-assigned cores
11        find r* for which t(r*) = min_{r∈R} t(r);     // find the TAM with the minimum test time
12        r* := r* ∪ {i};                               // add an un-assigned core to this TAM
13        t(r*) := TestTime(r*, w(r*));                 // update its test time
14     od;
15  fi;
16  if w_max > |C| then                                 // Step 3: if wires left, add to most-occupied TAMs
17     for i := |C| + 1 to w_max do                     // iteratively, assign remaining un-used wires
18        find r* for which t(r*) = max_{r∈R} t(r));    // find the TAM with the maximum test time
19        w(r*) := w(r*) + 1;                           // assign one more wire to this TAM
20        t(r*) := TestTime(r*, w(r*));                 // update its test time
21     od;
22  fi;
```

Computational complexity: The computational-time complexity of a procedure is the number of steps that it takes to solve an instance of the problem, as a function of the size of the input. For the test architecture design problem, the inputs are the TAM width $w_{\max}$ and the set of cores C. The worst-case computational-time complexity of the procedure CREATESTARTSOLUTION is $\mathcal{O}(w_{\max}|C|)$. Details about the computational-time complexity analysis for this procedure can be found in Appendix B.2.

3.6.2 Optimize Bottom Up

The procedure OPTIMIZE-BOTTOMUP tries to optimize the test time of a given test architecture. It does so by trying to merge the TAM with the shortest test time with another TAM, such that the wires that are freed up in this process can be used for an overall test time reduction. Algorithm 3.3 lists the pseudo-code for the procedure OPTIMIZE-BOTTOMUP. It is an iterative procedure, of which every iteration consists of two steps.

In Step 1 (Lines 3–12), the procedure finds a TAM $r_{\min}$ with minimum test time, i.e. $t(r_{\min}) = \min_{r \in R} t(r)$. The cores in TAM $r_{\min}$ and the cores in one of the other

TAMs, say r, are merged into a new TAM, say r^*, with width $\max(w(r_{\min}), w(r))$. TAM r is selected from $R \backslash \{r_{\min}\}$, i.e. set of TAMs R excluding $r_{\min}$, such that $t(r^*)$ is minimum and $t(r^*)$ does not exceed the current overall test time T.

In Step 2 (Lines 13–22), the merge is implemented and R is updated. As the new TAM r^* only uses $\max(w(r_{\min}), w(r))$ wires, $\min(w(r_{\min}), w(r))$ wires are now freed up. The freed-up wires are distributed over all TAMs, in order to reduce the overall test time T; here the used procedure is similar to Step 3 in CREATESTARTSOLUTION (Algorithm 3.2). The procedure ends if all TAMs have been merged into one single TAM, or if no TAM r can be found such that $t(r^*)$ does not exceed the current overall test time T.

Algorithm 3.3 [OPTIMIZE-BOTTOMUP]

1	*improve* := *true*;	// initially, improvement is possible
2	**while** $\lvert R \rvert > 1 \wedge$ *improve* **do**	// while multiple TAMs and improvement possible
3	find $r_{\min}$ for which $t(r_{\min}) = \min_{r \in R} t(r)$;	// **Step 1:** find TAM $r_{\min}$ with minimum test time
4	$T := \max_{r \in R} t(r)$; $t(r^*) := \infty$;	// calculate the current maximum test time T
5	**for all** $r \in R \backslash \{r_{\min}\}$ **do**	// iteratively, find merge candidate TAM r
6	$r_{\text{temp}} := r_{\min} \cup r$;	// create a TAM r_{temp} with cores in $r_{\min}$ and r
7	$w(r_{\text{temp}}) := \max(w(r_{\min}), w(r))$;	// assign it the maximum of the widths of $r_{\min}$ and r
8	$t(r_{\text{temp}}) :=$ TESTTIME $(r_{\text{temp}}, w(r_{\text{temp}}))$;	// calculate the test time of TAM r_{temp}
9	**if** $(t(r_{\text{temp}}) \leq T) \wedge (t(r_{\text{temp}}) < t(r^*))$ **then**	// if the test time of r_{temp} is minimum and $\leq T$
10	$r_{\text{del}} := r$; $r^* := r_{\text{temp}}$;	// accept this merge proposal
11	**fi**;	
12	**od**;	
13	**if** $t(r^*) \leq T$ **then**	// **Step 2:** if a merge proposal was found
14	$w_{\text{free}} := \min(w(r_{\min}), w(r_{\text{del}}))$;	// calculate the number of freed-up wires
15	$R := R \backslash \{r_{\text{del}}, r_{\min}\}$; $R := R \cup \{r^*\}$;	// remove merged TAMs and add proposed TAM
16	**for** $i := 1$ **to** w_{free} **do**	// iteratively, assign all freed-up wires
17	find r^* for which $t(r^*) = \max_{r \in R} t(r)$;	// find the TAM with the maximum test time
18	$w(r^*) := w(r^*) + 1$;	// assign one more wire to this TAM
19	$t(r^*) :=$ TESTTIME$(r^*, w(r^*))$;	// update its test time
20	**od**;	
21	**else** *improve* := *false*;	// if no proposal was found, no improvement possible
22	**fi**;	
23	**od**;	

The operation of one iteration of the procedure OPTIMIZE-BOTTOMUP is illustrated by means of an example depicted in Figure 3.8. Figure 3.8(a) shows a test architecture instance. TAM 3, containing cores A and D, has the shortest test time and hence $r_{\min}$ is TAM 3. Subsequently, the procedure looks for another TAM with which TAM 3 can be merged. TAM 1, containing core C, does not qualify, as it already determines the overall SOC test time T, and adding the cores of TAM 3 to it, would only increase that test time. However, TAM 2 does qualify, and hence a new TAM is created containing cores B, A, and D (Figure 3.8(b)). The $w3$ wires of TAM 3 are now freed up, and in Step 2, they are distributed over the two remaining TAMs. Figure 3.8(c) shows that this leads to a decrease in test time of both TAMs, and hence decreases the overall test time T.

Computational complexity: The computational-time complexity of this procedure heavily depends on the number of TAMs created in the CREATESTARTSOLUTION step. In the worst-case, one can assume that every core is connected to a separate TAM, i.e. $|R| = |C|$ and during optimization all TAMs are merged into a single TAM. Based on this, the worst-case computational-time complexity of the procedure OPTIMIZE-

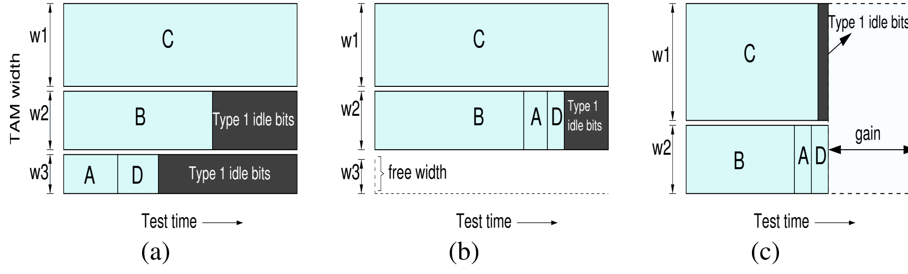


Figure 3.8: Two subsequent steps of the procedure OPTIMIZE-BOTTOMUP.

BOTTOMUP can be written as $\mathcal{O}(w_{\max}|C|^2)$. Details about the computational-time complexity analysis for this procedure can be found in Appendix B.3.

3.6.3 Optimize Top Down

The procedure OPTIMIZE-TOPDOWN tries to optimize the test time of a given test architecture in two subsequent steps. In Step 1, the algorithm iteratively tries to merge the TAM with the longest test time with another TAM, such that the overall test time is reduced. When there are no further test time improvements possible in Step 1, Step 2 is executed, In Step 2, the algorithm iteratively tries to free up wires by merging two TAMs that do not have the longest test time, under the condition that the test time of the resulting TAM does not exceed the overall test time. The wires that are freed up as a result of this merge, can be used for an overall test time reduction. Algorithm 3.4 lists the pseudo-code for the procedure OPTIMIZE-TOPDOWN.

In Step 1 (Lines 2–18), the procedure iteratively carries out the following actions. It finds a TAM $r_{\max}$ with the longest test time. Subsequently, the procedure tries to find a TAM $r \in R\backslash\{r_{\max}\}$ (a TAM other than $r_{\max}$ in R), which could be merged with TAM $r_{\max}$ into a new TAM r^* with $w(r^*) = w(r_{\max}) + w(r)$, such that $t(r^*)$ is minimum and $t(r^*)$ does not exceed the current overall test time T. If such a new TAM r^* is found, the merge is implemented and R is updated (Lines 12–13). Else, the TAM $r_{\max}$ is placed in the set R_{skip} (Lines 15–16) and Step 2 is carried out next.

Step 2 (Lines 19–47) is quite similar to Step 1, apart from the following two differences:

1. for a merge of two TAMs, it only considers the TAMs which are in R but not in R_{skip}, i.e. $r \in R\backslash R_{\text{skip}}$,
2. width $w(r^*)$ of the merged TAM r^* is determined by a linear search between the lower limit $w_L = \max(w(r_{\max}), w(r))$ and the upper limit $w_U = w(r_{\max}) + w(r)$, such that $w(r^*)$ is minimum. In this way, the freed-up TAM width, denoted as w_{fmax}, is maximized.

If search is successful , the freed-up wires w_{fmax} are distributed over the TAMs to minimize the test time of the architecture. If the search was not successful, then TAM $r_{\max}$ is added to set R_{skip}.

Algorithm 3.4 [OPTIMIZE-TOPDOWN]

```
1  improve := true;                                   // initially, improvement is possible
2  while |R| > 1 ∧ improve do                         // Step 1: while |R| > 1 and improvement possible
3    find r_max for which t(r_max) = max_{r∈R} t(r);  // find TAM r_max with maximum test time
4    T := max_{r∈R} t(r); t(r*) := ∞;                 // calculate the current maximum test time T
5    for all r ∈ R\{r_max} do                         // iteratively, find merge candidate TAM r
6      r_temp := r_max ∪ r;                           // create a TAM r_temp with cores in r_max and r
7      w(r_temp) := w(r_max) + w(r);                  // assign it the summed width of r_max and r
8      t(r_temp) := TESTTIME (r_temp, w(r_temp));     // calculate the test time of TAM r_temp
9      if (t(r_temp) < T) ∧ (t(r_temp) < t(r*)) then  // if the test time of r_temp is minimum and ≤ T
10       r_del := r; r* := r_temp; fi;                // accept this merge proposal
11   od;
12   if t(r*) ≤ T then                                // if a merge proposal was found
13     R := R\{r_del, r_max}; R := R ∪ {r*};          // remove merged TAMs and add proposed TAM
14   else                                             // no merge proposal was found
15     improve := false;                              // no improvement possible
16     R_skip := {r_max};                             // exclude r_max from further consideration
17   fi;
18 od;
19 while R_skip ≠ R do                                // Step 2: while TAMs are still under consideration
20   find r_max for which t(r_max) = max_{r∈R\R_skip} t(r); // find the TAM r_max with maximum test time
21   T := max_{r∈R} t(r);                             // calculate the current maximum test time T
22   for all r ∈ R\(R_skip ∪ {r_max}) do              // iteratively, find merge candidate TAM r
23     r_temp := r_max ∪ r;                           // create a TAM r_temp with cores in r_max and r
24     w_U := w(r_max) + w(r);                        // assign upper limit on its TAM width
25     w_L := max(w(r_max), w(r));                    // assign lower limit on its TAM width
26     found := false; w(r_temp) := w_L;              // start with TAM width equal to lower limit
27     while ¬found ∧(w(r_temp) ≤ w_U) do             // assign the actual width through linear search
28       t(r_temp) := TESTTIME (r_temp, w(r_temp));   // calculate the test time of TAM r_temp
29       if t(r_temp) ≤ T then                        // if the test time of r_temp is less than or equal to T
30         found := true; w_free := w_U − w(r_temp);  // a merge proposal is found, calculate freed-up wires
31         if w_free ≥ w_fmax                         // if the number of freed-up wires is maximum
32           w_fmax := w_free; r* := r_temp; r_del := r; // accept this merge proposal
33         fi;
34       fi;
35       w(r_temp) := w(r_temp) + 1;                  // increase the TAM width by one wire
36     od;
37   od;
38   if t(r*) ≤ T then                                // if a merge proposal was found
39     R := R\{r_del, r_max}; R := R ∪ {r*};          // remove merged TAMs and add proposed TAM
40     for i := 1 to w_fmax do                        // iteratively, assign all freed-up wires
41       find r* for which t(r*) = max_{r∈R} t(r);    // find the TAM with the maximum test time
42       w(r*) := w(r*) + 1;                          // assign one more wire to this TAM
43       t(r*) := TESTTIME(r*, w(r*));                // update its test time
44     od;
45   else R_skip := R_skip ∪ {r_max};                 /* if no merge proposal was found, exclude r_max
46   fi;                                              from further consideration */
47 od;
```

The operation of Step 1 of the procedure OPTIMIZE-TOPDOWN is illustrated in Figure 3.9. This figure shows a test architecture instance, in which TAM 1 containing only core C, has the largest test time. TAM 1 is merged with TAM 3, which contains cores A and D. The newly formed TAM contains cores C, A, and D, and gets assigned a width equal to $w1 + w3$. This leads to an overall test time reduction.

Computational complexity: The computational-time complexity of this procedure depends on the number of TAMs passed from the OPTIMIZE-BOTTOMUP step. In the worst-case, one can assume that there was no merging of TAMs possible in the OPTIMIZE-BOTTOMUP procedure, and hence the number of TAMs is equal to the number of cores, i.e. $|R| = |C|$. Based on this, the worst-case time complexity of

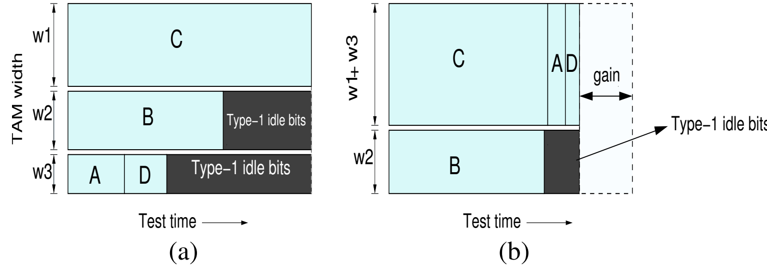


Figure 3.9: One iteration of Step 1 of the procedure OPTIMIZE-TOPDOWN.

the procedure OPTIMIZE-TOPDOWN can be written as $\mathcal{O}(w_{\max}|C|^2)$. More details about the computational-time complexity analysis for this procedure can be found in Appendix B.4.

3.6.4 Reshuffle

The procedure RESHUFFLE tries to minimize the test time of a given test architecture by moving one of the cores assigned to a TAM with the maximum test time to another TAM, provided that this decreases the overall test time. Algorithm 3.5 lists the pseudo-code for the procedure RESHUFFLE.

Algorithm 3.5 [RESHUFFLE]

```
 1  improve := true;                                  // initially, improvement is possible
 2  while improve do                                  // while improvement is possible
 3     find r_max for which t(r_max) = max_{r∈R} t(r);  // find TAM r_max with maximum test time
 4     if |r_max| = 1 then                            // if it contains only one core
 5        improve := false;                           // no improvement possible
 6     else                                           // else, TAM contains multiple cores
 7        find core j* for which t(j*) := min_{j∈r_max} t(j); // find core j* in r_max with minimum test time
 8        TAMFound := false; T := max_{r∈R} t(r);     // calculate the current maximum test time T
 9        while r ∈ R\{r_max} ∧ ¬ TAMFound do         // iteratively, find merge candidate TAM r
10           r* := r ∪ {j*}; w(r*) := w(r);           // create a TAM r* with cores in r and core j*
11           t(r*) := TESTTIME (r*, w(r*));           // calculate the test time of the created TAM r*
12           if t(r*) < T then                        // if test time of r* is less than T
13              TAMFound := true;                     // accept this merge
14              r_max := r_max\{j*}                   // remove core j* from TAM r_max
15              R := R\{r} ∪ {r*};                    // update TAM set R by removing r and adding r*
16           fi;
17        od;
18        if ¬TAMFound then                           // if no merge was found
19           improve := false; fi;                    // no improvement possible
20     fi;
21  od;
```

In Line 3, the procedure identifies a TAM $r_{\max}$ with the maximum test time. If the TAM $r_{\max}$ contains multiple cores, core j^* with the minimum test time is identified. The procedure searches through the other TAMs, to see if there is one of them to which core j^* can be added such that there is an improvement in the overall test time T. If that is the case, core j^* is indeed moved from the TAM with the maximum test time to this other TAM (Lines 12–15). This procedure is repeated until the TAM with the maximum test time contains only one core, or when no beneficial core re-assignment can be found.

Computational complexity: In the worst-case, one can assume that the TAM with the maximum test time contains $|C| - 1$ cores and during optimization all cores except one are moved from this TAM to the other TAM. Based on this, the worst-case computational-time complexity of the procedure RESHUFFLE can be written as $\mathcal{O}(|C|^2)$. Details about the computational-time complexity analysis for this procedure can be found in Appendix B.5.

3.6.5 Checking Empty Wire

Similar to the concept of Pareto-Optimal widths (See Section 3.5.2) for cores, based on the cores connected to a TAM, the TAM also has a set of Pareto-Optimal widths. The set of Pareto-Optimal widths for a TAM is determined by the union of the sets of Pareto-Optimal widths for all the cores connected to the TAM. It is possible that during the OPTIMIZE-BOTTOMUP and/or OPTIMIZE-TOPDOWN steps, the width assigned to a TAM is a non Pareto-Optimal width. Therefore, in order to use all TAM wires efficiently, this step tries to find the number of empty (redundant) wires in all TAMs. These empty wires are then re-distributed among TAMs in order to minimize the overall test time. Algorithm 3.6 lists the pseudo-code for the procedure CHECK-EMPTYWIRE. The procedure CHECK-EMPTYWIRE consists of three steps.

Algorithm 3.6 [CHECK-EMPTYWIRE]

```
 1  w_empty := 0;                                   // initially, no empty wire
 2  for all r ∈ R do                                // Step 1: iteratively, for all TAMs in set R
 3     non-pareto := true;                          // initially, TAM width is a non Pareto-Optimal width
 4     while w(r) > 1 ∧ non-pareto do               // while TAM width is more than one and non Pareto-Optimal
 5        t_old := t(r); w(r) := w(r) − 1;          // compute current time and decrease the TAM width by one
 6        t(r) := TESTTIME (r, w(r));               // calculate the new test time for the TAM
 7        if t(r) > t_old then                      // if new test time is greater than the current test time
 8           non-pareto := false;                   // previous width was Pareto-Optimal width
 9           w(r) := w(r) + 1; t(r) := t_old;       // update the TAM width and test time
10        else w_empty := w_empty + 1;              // increase the number of empty wires by one
11        fi;
12     od;
13  od;
14  for i := 1 to w_empty do                        // Step 2: iteratively, distribute all empty wires
15     find r* for which t(r*) = max_{r∈R} t(r);    // find the TAM with the maximum test time
16     w(r*) := w(r*) + 1;                          // assign one more wire to it
17     t(r*) := TESTTIME(r*, w(r*));                // update its test time
18  od;
19  non-pareto := true;                             // Step 3: TAM width is a non Pareto-Optimal width
20  while non-pareto do;                            // while non Pareto-Optimal is true
21     find r* for which t(r*) = max_{r∈R} t(r);    // find the TAM with the maximum test time
22     t_old := t(r*); w(r*) := w(r*) − 1;          // compute current time and decrease the TAM width by one
23     t(r*) := TESTTIME(r*, w(r*));                // calculate the new test time for the TAM
24     if t(r*) > t_old then                        // if new test time is greater than the current test time
25        non-pareto := false;                      // previous width was Pareto-Optimal width
26        w(r) := w(r) + 1; t(r) := t_old;          // update the TAM width and test time
27     else w_empty := w_empty + 1;                 // increase the number of empty wires by one
28     fi;
29  od;
```

In Step 1 (Lines 2–13), the procedure tries to find the number of empty wires by iteratively decreasing the width $w(r)$ of a TAM $r \in R$ and then comparing the old test time t_{old} with the new test time $t(r)$. If both the test times are the same, the number of empty wires is increased by one. If the new test time is greater than the old test time,

then it shows that the previous TAM width was a Pareto-Optimal width for the TAM. Hence, the procedure assigns this Pareto-Optimal width to TAM r.

In case, Step 1 resulted in some empty wires, these wires are added iteratively to the TAM with the longest test time in Step 2 (Lines 14–18). As the assignment of unused wires to TAMs is done iteratively, it can happen that the width of the TAM with the maximum test time is moved again from a Pareto-Optimal width to a non Pareto-Optimal width. Hence, in Step 3 (Lines 19–29), number of empty wires are again checked for the TAM with the maximum test time only.

Computational complexity: In the worst-case, one can assume that for all TAMs, the search for Pareto-Optimal widths is carried out till every TAM has only one wire left. The worst-case computational-time complexity of the procedure CHECK-EMPTYWIRE can be written as $\mathcal{O}(|C|w_{\max})$. Details about the computational-time complexity analysis for this procedure can be found in Appendix B.6.

3.6.6 TR-Architect Computational Complexity

The overall worst-case computational-time complexity of TR-ARCHITECT algorithm in principle is the sum of the worst-case computational-time complexities of the five individual steps. On contrary, a worst-case scenario for a procedure might not represent the worst-case scenario for other procedures. For example, the case with $|R| = |C|$ represents the worst-case scenario for the procedures OPTIMIZE-BOTTOMUP and OPTIMIZE-TOPDOWN, however this represent the best-case scenario for the procedure RESHUFFLE. Based on the worst-case computational-time complexities of the five steps, the overall worst-case computational-time complexity of TR-ARCHITECT can be written as $\mathcal{O}(w_{\max}|C| + w_{\max}|C|^2 + w_{\max}|C|^2 + |C|^2 + w_{\max}|C|)$. Using the rules of approximation as described in Appendix B, this can be reduced to $\mathcal{O}(w_{\max}|C|^2)$.

3.7 Test Time Calculation

TR-ARCHITECT optimizes a test architecture, independent of (1) the core type (hard or soft), (2) the TAM type (test bus or TestRail), and (3) the schedule type (serial or parallel). The user of TR-ARCHITECT selects whether to design architecture with hard cores (problem TADHC) or with soft cores (problem TADSC) or a mix of both hard and soft cores. In case of a mix of hard and soft cores, a parameter per core can be used to indicate whether a core is hard or soft. The user also needs to select the type of TAM e.g., test bus or TestRail. The selection of the test bus TAM automatically determines the schedule type, as a test bus only supports serial scheduling. In case the TestRail TAM is selected, the user of TR-ARCHITECT also needs to indicate the preference for serial or parallel test scheduling.

TR-ARCHITECT uses a procedure TESTTIME$(r, w(r))$ (see Section 3.6) to determine the test time $t(r)$ of a TAM r with width $w(r)$. The procedure TESTTIME in its turn utilizes the procedure WRAPPERDESIGN$(c, w(r))$ for designing a wrapper with TAM width $w(r)$ around core c. Based on the user-defined choices above, TR-

ARCHITECT uses different versions of the procedures TESTTIME and WRAPPERDESIGN.

The procedure WRAPPERDESIGN$(c, w(r))$ designs a wrapper for core c with TAM width $w(r)$. For *hard* cores, the procedure WRAPPERDESIGN uses the COMBINE algorithm [MGL00] presented in Chapter 2. For *soft* cores, calculating the scan-in and scan-out times for a core reduces to a simple balanced distribution of the scan flip flops and wrapper cells over $w(r)$ TAM wires.

For the procedure TESTTIME, the test time calculation for a test bus architecture is quite different from the same for a TestRail architecture. Also, in case of a TestRail architecture, the calculation for the serial and parallel schedules are different. Therefore, details about the test time calculations used by the procedure TESTTIME for test bus and TestRail architectures are described separately in the sequel of this section.

3.7.1 Test Bus Architecture Test Time

The test time of a test bus equals the sum of the test times of the individual cores connected to that test bus. If the test bus functionality is implemented by means of multiplexers or tri-state drivers, then the test access to individual cores will not require additional clock cycles. The total test time $t(r)$ for the test bus r can be written as:

$$t(r) = \sum_{c \in r} t_c \tag{3.6}$$

where t_c is the test time for core c and is defined in Eqn (2.1) in Chapter 2.

In a hybrid test bus architecture, all test buses are tested in parallel. Therefore, the total test time T for a hybrid test bus architecture is the maximum of the test time for all test buses and can be written as:

$$T = \max_{r \in R} t(r) \tag{3.7}$$

where R is set of test buses in the architecture.

3.7.2 TestRail Architecture Test Time

A TestRail TAM utilizes bypasses. If a core is not being tested, the shortest test-access path to other cores in the same TestRail is through the bypass of the core. The bypass is implemented as register, in order to provide a fully synchronous 'plug-n-play' interface, in which an infinite amount of cores can be daisychained [MAB+98]. This yields in a latency of one clock cycle per bypass register in the test-access path.

Serial Schedule: in the case of a TestRail architecture that uses a serial schedule, the procedure TESTTIME sums the test times of the individual cores in a TestRail r. For every core, it adds $|r| - 1$ to the number of clock cycles needed per test pattern, in order to account for the time latencies due to the bypass registers in rest of the cores. The total test time $t(r)$ for the TestRail r with a serial schedule can be written as:

$$t(r) = \sum_{c \in r} \{t_c + (|r| - 1) \times p_c\} \tag{3.8}$$

where t_c is the test time and p_c is the total number of test patterns for core c.

Parallel Schedule: in the case of a TestRail architecture that uses a parallel schedule, the procedure TESTTIME is a little-bit more complex. Initially, all cores $c \in r$ are tested in parallel. This continues until one of them runs out of test patterns. Then, the bypass for this core is turned on, while the testing of the remaining cores continued. After a number of test patterns, another core runs out of test patterns and its bypass is also turned on. This process is repeated until all cores have been completely tested. The time to scan-in test-stimuli for a single pattern into all cores connected to the TestRail r is $\sum_{c \in r} l_c^{\max}$ clock cycles, where $l_c^{\max}$ is the length of the longest TAM chain connecting scan chains as well as both wrapper input cells and wrapper output cells in the wrapper of core c. Similarly, it takes $\sum_{c \in r} l_c^{\max}$ clock cycles to scan-out the test-responses for a pattern.

Without loss of generality, it can be assumed that all cores in r are sorted in the non-decreasing test pattern count order, i.e. $p_1 \leq p_2 \leq \ldots \leq p_{|r|}$. If the scan-out time of a pattern is pipelined with the scan-in time for the next pattern, testing of all cores for the first p_1 number of test patterns requires the following number of clock cycles:

$$\left(1 + \sum_{c=1}^{|r|} l_c^{\max}\right) \cdot p_1 + \sum_{c=1}^{|r|} l_c^{\max} \tag{3.9a}$$

After the first p_1 test patterns, core 1 is put into bypass mode. The bypass is assumed to be implemented as a register and hence takes one clock cycle of scan time. With the assumption that the scan-in time of the first pattern in this session can be pipelined with the scan-out time of the last pattern in the previous session, the time required to execute $(p_2 - p_1)$ test patterns in this session is

$$\left\{1 + \left(1 + \sum_{c=2}^{|r|} l_c^{\max}\right)\right\} \cdot (p_2 - p_1) + \sum_{c=2}^{|r|} l_c^{\max} - \min\left(\sum_{c=1}^{|r|} l_c^{\max}, \sum_{c=2}^{|r|} l_c^{\max}\right) \tag{3.9b}$$

clock cycles. If this process is continued and all the intermediate results (Equations (3.9a), (3.9b), ...)) are summed, one can derive the equation for the test time $t(r)$ for the TestRail r and which is as follows:

$$\begin{aligned} t(r) = \sum_{j=1}^{|r|} \Bigg[& \left\{(j-1) + \sum_{c=j}^{|r|} l_c^{\max}\right\} \cdot (p_j - p_{j-1}) + \sum_{c=j}^{|r|} l_c^{\max} \\ & - \min\left(\sum_{c=j}^{|r|} l_c^{\max}, \sum_{c=j}^{|r|} l_{c+1}^{\max}\right)\Bigg] + p_{|r|} \end{aligned} \tag{3.10}$$

where $p_0 = 0$ and $l_{|r|+1}^{\max} = 0$.

To understand the test time calculations described above for the parallel schedule, consider a TestRail containing two cores A and B with $p_A = 10$ and $p_B = 20$ test patterns respectively. The maximum TAM chain lengths for cores A and B are $l_A^{\max} =$

20 and $l_B^{\max} = 30$ clock cycles respectively. Now for the first $p_A = 10$ patterns, both cores will be tested in parallel and the scan-in and scan-out time for a single pattern will be $20 + 30 = 50$ clock cycles. Therefore, testing of both cores for the first $p_A = 10$ patterns will require $(1 + 50) \times 10 + 50 = 560$ clock cycles. For the remaining 10 patterns of core B, core A is put in the bypass mode. Testing of core B for the remaining 10 patterns requires $(1 + (1 + 30)) \times 10 + 30 - \min(50, 30) = 320$ clock cycles. Therefore the total test time for this TAM is equal to $560 + 320 = 880$ clock cycles.

The total test time T for a hybrid TestRail architecture is the maximum of the test times of the individual TestRails, similar to Equation (3.7).

3.8 Experimental Results

In this section, experimental results of TR-ARCHITECT are presented. As benchmarks, the set of *ITC'02 SOC Test Benchmarks* [MIC, MIC02] have been used. In the experiments, it has been assumed that an SOC only contains one level of hierarchy, i.e. the SOC itself and all its embedded cores, even though some of these SOCs originally contain multiple levels of design hierarchy. Also, only the core-internal tests of the SOCs have been considered i.e. the interconnect tests for the top-level SOC itself have not been taken into account. As all the SOCs in the benchmark set have a fixed number of scan chains and lengths, only the test time results for the problem TADHC (SOC with all hard cores) are presented.

For SOC p22810 with $w_{\max} = 16$, Figure 3.10 illustrates how the five distinct procedures of TR-ARCHITECT contribute to the improvement in the SOC test time.

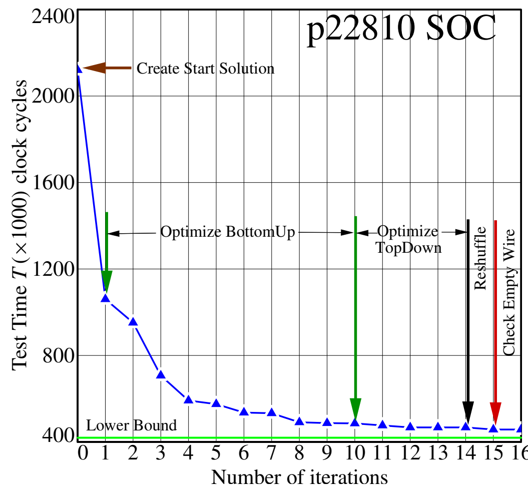
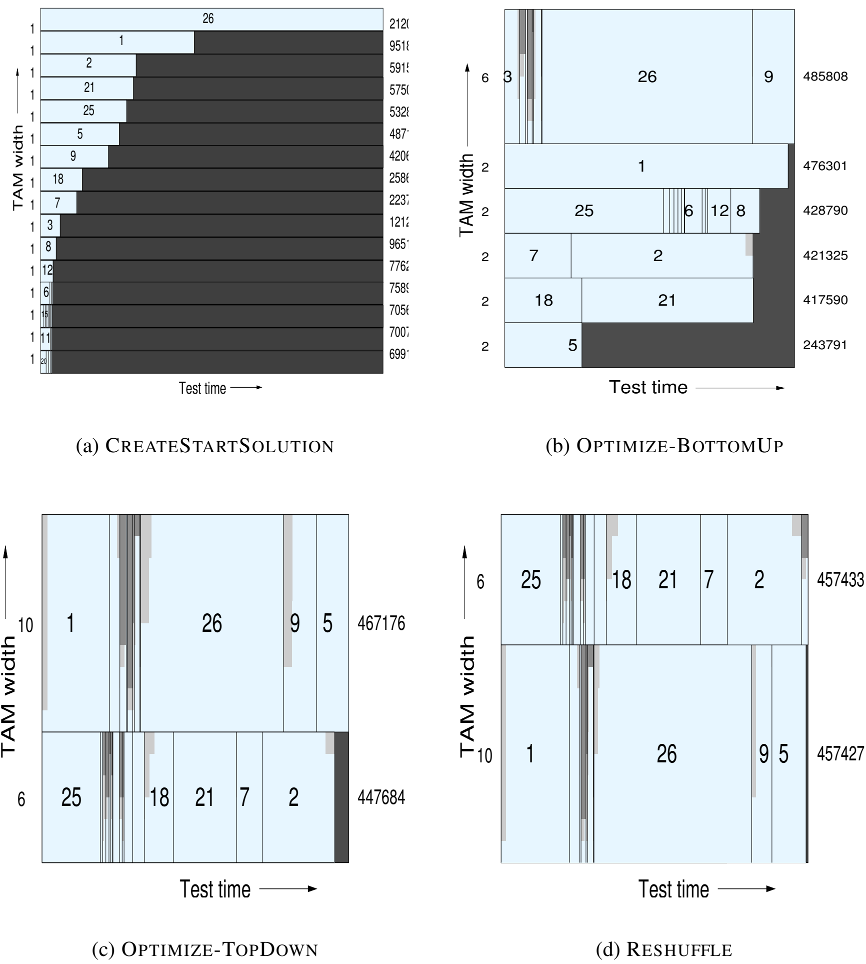


Figure 3.10: Improvement in test time for SOC p22810 using the five distinct procedures of TR-ARCHITECT.

In Figure 3.10, the horizontal axis represents the number of successful iterations (merge proposals), while the vertical axis represents the overall test time for the SOC.

The horizontal line (green colour) just above the horizontal axis represents the theoretical lower bound on the SOC test time. From Figure 3.10, one can see that initially the test time of the architecture designed by TR-ARCHITECT is quite far from the lower bound. However, as TR-ARCHITECT moves forward with the optimization steps, the test time continues to decrease and finally, a test time which is very close to the lower bound is achieved. From Figure 3.10, one can also see that for this case, most of the optimization is accomplished by the procedure OPTIMIZE-BOTTOMUP. In contrast, the procedure CHECK-EMPTYWIRE does not result in any improvement.

For the first four steps of TR-ARCHITECT, Figure 3.11 shows the corresponding test schedule of the resulting architecture after each step.



(a) CREATESTARTSOLUTION (b) OPTIMIZE-BOTTOMUP

(c) OPTIMIZE-TOPDOWN (d) RESHUFFLE

Figure 3.11: For the first four steps of TR-ARCHITECT, test schedule for SOC p22810 with $w_{\max} = 16$ after each step.

In Figure 3.11, the vertical axis represents the TAM width, while the horizontal axis shows the test time (in number of clock cycles); the later is not to scale. In the figure, the numbered boxes depict the tests of the cores and the number inside a box represents the core ID. At the end of each TAM, the shown number represents the test time (in number of clock cycles) for the TAM. The three different shades of grey in the figure, represent the three types of idle bits.

As the procedure CREATESTARTSOLUTION is used to create an initial architecture which will be further optimized by the other steps to follow, this step usually results in an architecture with a large test time and a large amount of Type-1 idle bits (see Figure 3.11(a)). The total amount of Type-1 idle bits in this schedule is $27,713,556$ (= 82% of the total bits in the schedule). From Figure 3.11(b), one can see that the procedure OPTIMIZE-BOTTOMUP improves the test time by 77%. However, it still contains a large number of Type-1 idle bits (in total $882,486$ bits, i.e. 11%). The procedures OPTIMIZE-TOPDOWN and RESHUFFLE further removed the Type-1 idle bits and shows a saving of 4% and 2% in test time respectively. From Figure 3.11(d), one can see that the resulting architecture contains only two TAMs and there is only a difference of three clock cycles in their completion time. Therefore, there are only $10 \times 3 = 60$ Type-1 idle bits in this schedule. Ideally, this means that the test time for this architecture should be very close to the lower bound of $419,466$ clock cycles. However, it is still 10% away from the lower bound. In order to understand better this difference of 10%, all three types of idle bits in the schedule are analyzed.

For the test schedule shown in Figure 3.11(d), Figure 3.12(a) shows the total amount of Type-2 and Type-3 idle bits in all cores. Figure 3.12(b) shows the percentage distribution of different types of idle bits.

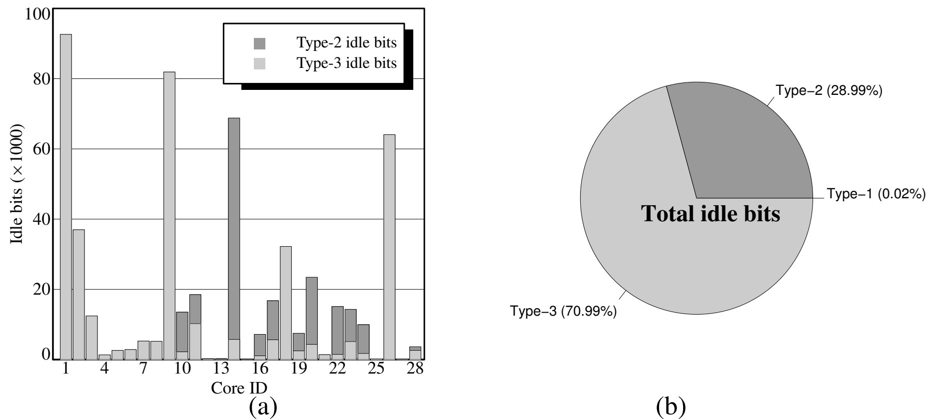


Figure 3.12: (a) Total amount of Type-2 and Type-3 idle bits in all cores, (b) percentage distribution of the three types of idle bits.

A large number of Type-3 idle bits for a core in Figure 3.12(a) shows a large imbalance in the scan chains after the wrapper design. Similarly, a large number of Type-2 idle bits shows that the TAM width assigned to the core is quite far from the nearest Pareto-Optimal width. The total number of idle bits in the schedule is $539,955$ bits.

These idle bits themselves cause the test time of architecture to be 8% away from the lower bound. The remaining 2% is contributed by the fact that the heuristic algorithm TR-ARCHITECT produces near optimal results.

Next, the test time results of TR-ARCHITECT are compared to the best test times that could be obtained by manual efforts of DfT engineers inside Philips, without access to optimization tools such as TR-ARCHITECT. Out of the twelve *ITC'02 SOC Test Benchmarks*, only the test architectures implemented on the three Philips SOCs p22810, p34392, and p93791 are known. The SOCs p22810 and p34392 originally had a distribution architecture [AM98] (i.e. all cores have their own, private TAM), while SOC p93791 was equipped with a daisychain architecture [AM98] (i.e. all cores are connected to one common single TestRail TAM).

Figure 3.13(a) shows the test schedule for an optimal distribution architecture for SOC p22810 with $w_{\max} = 64$. The horizontal axis shows the test time (in number of clock cycles), while the vertical axis represents the TAM width. The numbered boxes depict the tests of the cores and the number inside a box represents the core ID. At the end of each TAM, the shown number represents the test time (in number of clock cycles) for the TAM. The total test time for this schedule is determined by core 1 and is equal to $173,705$ clock cycles.

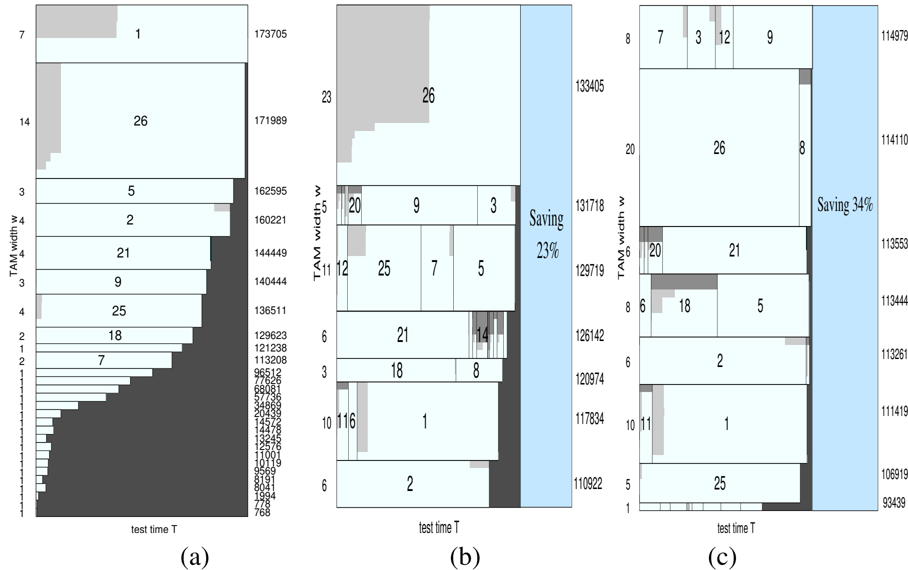


Figure 3.13: For SOC p22810 and $w_{\max} = 64$ (a) distribution architecture schedule, (b) hybrid test bus architecture schedule, and (c) hybrid test bus architecture schedule with one soft core.

From Figure 3.13(a), one can see that this schedule contains a very large amount of Type-1 idle bits. This is due to the fact that all cores in a distribution architecture have separate TAMs and irrespective of the test-data requirements of the cores, every core gets at least one (lowest integer) TAM wire. This results in less TAM wires available

for the cores with large test data. This schedule contains $3,404,593$ Type-1 idle bits ($=31\%$), 0 Type-2 idle bits ($=0\%$), and $599,951$ Type-3 idle bits ($=6\%$).

Figure 3.13(b) shows the test schedule for a hybrid test bus architecture as computed by TR-ARCHITECT for SOC p22810 with $w_{\max}=64$. The total test time for this schedule is $133,405$ clock cycles, which is a saving of 23% in test time as compared to the test time obtained from the distribution architecture. Similarly, the total amount of Type-1 idle bits reduces to $420,460$. This is due to the fact that a hybrid architecture allows more than one core to connect to the same TAM. This schedule contains $420,460$ Type-1 idle bits ($=5\%$), $94,866$ Type-2 idle bits ($=2\%$), and $1,178,495$ Type-3 idle bits ($=14\%$).

It is important to note here, that different types of idle bits shown in Figure 3.13(a) and (b) not only represent the amount of idle time in the schedules but also give an insight about the SOC design itself. For example, in Figure 3.13(b), one can see that the test schedule of core 26 contains a large amount of Type-3 idle bits and that represents $>55\%$ of the total idle bits present in the schedule. This is due to the imbalanced scan chains inside the core. If it would have been allowed to modify the internal scan chains of this core, one can reduce the overall test time further by 11% as shown in Figure 3.13(c). The test schedule shown in Figure 3.13(c) contains very few idle bits. This schedule contains $145,964$ Type-1 idle bits ($=2\%$), $141,797$ Type-2 idle bits ($=2\%$), and $207,901$ Type-3 idle bits ($=3\%$). Figure 3.14 shows the hybrid test bus architecture that corresponds to the test schedule shown in Figure 3.13(c). This architecture consists of eight TAMs with the following width distribution: $8+20+6+8+6+10+5+1=64$.

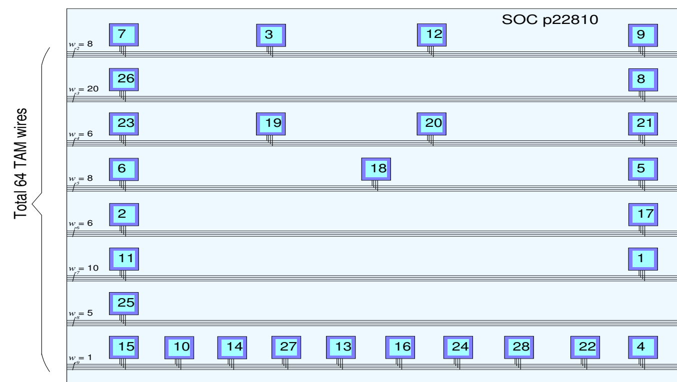


Figure 3.14: Hybrid test bus architecture with one soft core (core 26) for SOC p22810 with $w_{\max}=64$.

For a range of $w_{\max}$ values, Table 3.2 presents a comparison between the test time results obtained from TR-ARCHITECT and the manual best-effort engineering approaches i.e. the daisychain and distribution architectures. TR-ARCHITECT works for both the test bus and TestRail TAMs and for both the serial and parallel schedules. For comparison, the results for the hybrid test bus and hybrid TestRail architectures with serial schedules are presented.

Table 3.2: Test time results of TR-ARCHITECT and manual best-effort engineering approaches for SOCs with hard cores.

Input		Manual Best-Effort Eng.		TR-Architect		
SOC	w_{max}	Daisychain	Distribution	TB Hybrid (T)	TR Hybrid (T)	$\triangle T(\%)$
p22810	16	1338667	-	457433	473066	-64.7
	24	1282005	-	302737	307496	-76.4
	32	1218012	591585	222471	225743	-62.4
	40	1211583	353619	190995	190995	-46.0
	48	1196626	258602	157851	158634	-40.0
	56	1196626	206023	145417	145417	-29.4
	64	1196402	173705	133405	133405	-23.2
p34392	16	2694767	-	1010821	1019682	-62.52
	24	2636922	1693419	663193	671432	-60.84
	32	2612246	875499	584524	584524	-33.24
	40	2602957	587609	544579	544579	-7.3
	48	2592799	544579	544579	544579	0.0
	56	2592799	544579	544579	544579	0.0
	64	2568127	544579	544579	544579	0.0
p93791	16	2584315	-	1791638	1791638	-30.7
	24	1985588	-	1185434	1223881	-40.3
	32	1936078	5317007	912158	925344	-52.8
	40	1845070	1813502	718005	749228	-60.4
	48	1384343	1108358	601450	610880	-45.7
	56	1371382	918576	528925	556254	-42.4
	64	1371379	716679	455738	464351	-36.4

The header of the table indicates which architecture was used (where 'TB' stands for test-bus, and 'TR' denotes TestRail). A '–' entry denotes that the approach could not yield an architecture for the corresponding $w_{\max}$. The column ΔT shows the percentage (%) difference between the best of the TR-ARCHITECT solutions and the best of the manual best-effort engineering approaches. The test time numbers in Table 3.2 are all based on hard cores.

From the table, one can see that for all cases, TR-ARCHITECT outperforms the manual-engineering approaches and can lead to more than 75% reduction in required tester-vector memory and test time. This can be explained by recognizing the fact that daisychain and distribution architectures are the two extremes of the spectrum of architectures covered by the hybrid architecture model. In other words, the hybrid architecture model is a generalization of the first two. The presented experimental results show that typically the best test times are obtained not at the two extremes of the spectrum, but rather somewhere in between.

Next, the test time results of TR-ARCHITECT are compared to the lower bound presented in this chapter and to four others previously published approaches. These four approaches are as follows.

- ILP and exhaustive enumeration based approach in [ICM02a].
- Heuristic (*Par_eval*) based approach in [ICM02b].
- Generalized rectangle-packing (GRP) based approach in [ICM02d].
- Cluster-based optimization approach in [GM02b].

Table 3.3 presents the test time comparison for a range of $w_{\max}$ (shown as w in the table) values for four benchmark SOCs. These four SOCs are selected, as they are the

Table 3.3: Test time results for lower bound, TR-ARCHITECT and four others algorithms, for SOCs with hard cores.

SOC	w	LB_T	ILP	Par_eval	GRP	Cluster	TR-Architect		
			TB	TB	TB	TR	TB	TR	TR
			Serial	Serial	Serial	Parallel	Serial	Serial	Parallel
d695	16	40951	**42568**	42644	44545	44330	44307	44012	46035
	24	27305	**28292**	30032	31569	30021	28576	28798	31241
	32	20482	21566	22268	23306	23488	**21518**	21531	23488
	40	16388	17901	18448	18837	19034	**17617**	17832	18799
	48	13659	16975	15300	16984	16194	**14608**	14646	16194
	56	11709	13207	12941	14974	13479	**12462**	12478	13935
	64	10247	12941	12941	11984	**11033**	**11033**	**11033**	**11033**
p22810	16	419466	462210	468011	489192	-	**457433**	473066	485312
	24	279644	361571	313607	330016	-	**302737**	307496	313958
	32	209734	312659	246332	245718	259975	**222471**	225743	238289
	40	167787	278359	232049	199558	206205	**190995**	**190995**	206205
	48	139823	278359	232049	173705	173705	**157851**	158634	164580
	56	119848	268472	153990	157159	146390	**145417**	**145417**	**145417**
	64	104868	260638	153990	142342	133587	**133405**	**133405**	133587
p34392	16	932790	**998733**	1033210	1053491	-	1010821	1019682	1093066
	24	621903	720858	882182	759427	876529	**663193**	671432	743128
	32	544579	591027	663193	**544579**	585309	584524	584524	584524
	40	544579	**544579**	**544579**	**544579**	**544579**	**544579**	**544579**	**544579**
	48	544579	**544579**	**544579**	**544579**	**544579**	**544579**	**544579**	**544579**
	56	544579	**544579**	**544579**	**544579**	**544579**	**544579**	**544579**	**544579**
	64	544579	**544579**	**544579**	**544579**	**544579**	**544579**	**544579**	**544579**
p93791	16	1746657	**1771720**	1786200	1932331	-	1791638	1791638	1863436
	24	1164442	1187990	1209420	1310841	-	**1185434**	1223881	1264236
	32	873334	**887751**	894342	988039	-	912158	925344	942349
	40	698670	**698883**	741965	794027	816972	718005	749228	747165
	48	582227	**599373**	**599373**	669196	677707	601450	610880	643827
	56	499053	**514688**	**514688**	568436	542445	528925	556254	538305
	64	436673	460328	473997	517958	467680	**455738**	464351	469742

only ones for which results have been reported in [ICM02a,ICM02b,ICM02d,GM02b]. Note that SOCs p22810 and p34392 are referred to as p21241 and p33108 respectively in [ICM02b], but these constitute the same SOCs.

In the terminology used throughout this chapter, ILP/Enum [ICM02a], *Par_eval* [ICM02b], and GRP [ICM02d] use a hybrid test bus architecture in conjunction with a serial schedule, while the Cluster algorithms [GM02b] use a hybrid TestRail architecture in conjunction with a parallel schedule. TR-ARCHITECT works for the hybrid test bus and hybrid TestRail architectures and for the serial as well as parallel schedules, and hence all results are presented here. Per line of the table, the bold-font entries denote the lowest test time over all methods.

The ILP/enumeration method [ICM02a] requires computation time in the range of minutes to hours, depending on the complexity of the SOC and the total available TAM width. The *Par_eval* method [ICM02b] requires up to several minutes of computing time. For all other methods, including our TR-ARCHITECT, computational time is less than 10 seconds for all SOCs and all TAM widths, and hence negligible.

TR-ARCHITECT has proven to be very effective in optimizing the SOC test time. For hybrid test bus architectures, its main competitor is the ILP/Enumeration approach [ICM02a]. ILP/Enumeration is an exhaustive method, and hence, in principle, should produce the best results. However, as the problem is $\mathcal{NP}$-hard [ICM02a], the exhaustive ILP/Enumeration approach requires very large computation time, and hence was

restricted to cover only cases with two or three distinct TAMs. The fast heuristics of TR-ARCHITECT do not have this limitation, and show that better test times can be obtained for a larger number of TAMs. Compared to the other heuristic approaches, the heuristics of TR-ARCHITECT typically perform better. For the (preferable) hybrid TestRail architectures, TR-ARCHITECT yields comparable or better test times as compared to the Cluster algorithms [GM02b], assuming parallel schedules. In addition, TR-ARCHITECT does not have the limitation of the Cluster algorithms that the total TAM width should be greater or equal to the number of cores inside the SOC. This is the first publication on serial schedules for hybrid TestRail architectures, and hence there is no competitor yet.

For SOC p34392, the test time for this SOC does not decrease beyond $544,579$ clock cycles for large $w_{\max}$. This is an example where lower bound component LB_1 is actually achieved. This is due to core 18 of this SOC, which becomes a bottleneck core for sufficiently large enough TAM widths. Core 18 reaches its minimum test time value if assigned to a TAM of width 10, and increasing the TAM width does not further reduce its test time. All algorithms find this minimum test time. GRP [ICM02d] reaches a test time of $544,579$ clock cycles for $w_{\max} = 32$. TR-ARCHITECT reaches this lower bound for $w_{\max} = 33$ (not shown in Table 3.3) for the hybrid test bus architecture; the total TAM width is partitioned over four test buses of widths $1, 6, 16$, and 10. The overall SOC test time is determined by test bus 4, which contains core 18 only.

3.9 Summary

In this chapter, the test architecture design problem for SOCs with both hard and soft cores is presented. Based on the amount of test data to be transported into and out of the SOC and the total available TAM width, an improved architecture-independent lower bound on the overall SOC test time is derived. For SOCs with hard cores, the presented lower bound is more tight than the previously known lower bound for most practical values of $w_{\max}$; the new lower bound is virtually always an improvement for SOCs with soft cores. Next, a classification of the three types of idle bits that occur in practical schedules and might prevent from obtaining the theoretical lower bound, is presented.

A novel heuristic algorithm named TR-ARCHITECT is presented. TR-ARCHITECT optimizes test architectures with respect to required ATE vector-memory depth and test application time. TR-ARCHITECT optimizes wrapper and TAM design in conjunction. TR-ARCHITECT handles SOCs with both hard and soft cores, works for test bus as well as TestRail TAMs, and supports both serial and parallel schedules.

For the *ITC'02 SOC Test Benchmarks*, test time results of TR-ARCHITECT are compared with two manual best-effort engineering approaches that are being used in Philips, as well as to four other automated optimization approaches. Compared to the manual best-effort engineering approaches, TR-ARCHITECT can reduce the required tester-vector memory and test time up to 75%. This emphasizes the need for an automated optimization approach for designing SOC test architectures. TR-ARCHITECT

test time results are comparable or better than the four other automated optimization approaches. TR-ARCHITECT works well for TAM widths less than the number of cores in the SOC, which is an improvement over the Cluster-based approach in [GM02b]. TR-ARCHITECT requires a negligible amount of computing time and is therefore also suitable for very wide TAMs; this is especially an improvement over the CPU-intensive ILP/enumeration-based method in [ICM02a].

Chapter 4

Layout-Driven Test Architecture Design

4.1 Introduction

A large number of test architecture design procedures described in literature use SOC *test time* as minimization criterion. This is motivated by the fact that a large SOC typically has a large test-data set. This large test-data set will not only require *Automatic Test Equipment* (ATE) with deep (hence expensive) vector memory depth per channel, but will also have a long test-application time on the ATE. However, in an era of designing multi-million transistor ICs with deep-submicron technology, the wiring of TAMs is another important cost factor. Nowadays, many SOCs implement quite wide, dedicated TAMs. Especially for such SOCs, it pays off to minimize the TAM *wire length* while designing the test architecture. Short TAM wires reduce the required area cost, performance impact, power dissipation, and cross-coupling between the TAM wires.

To optimize the TAM wire length, a test architecture design procedure should take into account the layout positions of all cores in the SOC. Next to that, the procedure should try to assign the neighboring cores to the same TAM as much as possible. The reason behind this is as follows. If the layout positions of cores in an SOC are not considered while designing the test architecture for the SOC, cores positioned at different corners in the SOC-layout may get assigned to the same TAM. To connect these cores through TAM wires, TAM wires often go across the entire layout (perhaps even multiple times). For example, Figure 4.1(a) shows a nicely balanced test schedule for an example SOC with nine cores and $w_{\max}$ = 20. The test schedule contains four individual TAMs of widths 7, 5, 4, and 4. The horizontal axis in the figure represents the test time, while the vertical axis represents the TAM width.

From the SOC test time point of view, this test schedule can be considered as a very good schedule, as there is very little difference between the completion times of all four TAMs. However, if one looks at the TAM wiring for the proposed architecture in an example layout of the SOC (as shown in Figure 4.1(b)), one can clearly see that this is

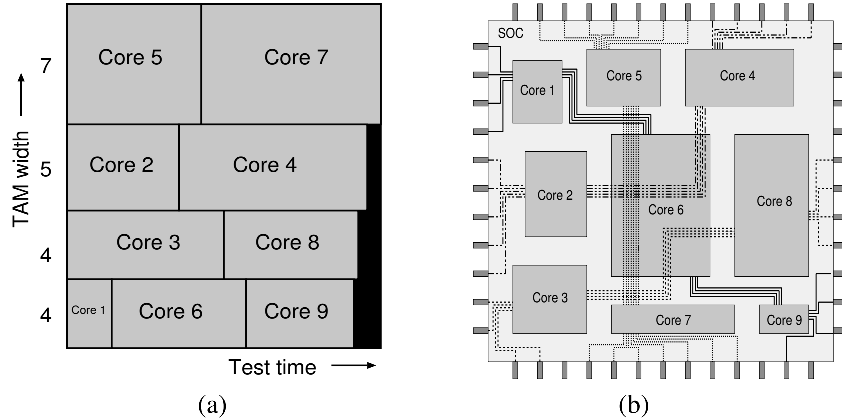


Figure 4.1: (a) An example test schedule, and (b) corresponding TAM wires connections in the SOC.

not a good test architecture. In Figure 4.1(b), different TAMs are indicated by different type of lines and they all lie across the entire layout. Therefore, it can be concluded that in order to optimize the TAM wire length, a test architecture design procedure should take into account the layout positions of all cores in the SOC.

In this chapter, a simple yet effective TAM wire length cost model is presented. The wire length of a TAM depends on its width and the ordering of cores connected to the TAM. It is shown that the problem of determining an optimal ordering of cores connected to a TAM is equivalent to the well-known *Traveling Salesman Problem* (TSP) [GJ79] and a simple heuristic algorithm is described to solve it. Subsequently, a layout-driven test architecture design problem is formulated, in which the layout positions of all cores in the SOC are assumed to be given. Next, a layout-driven version of TR-ARCHITECT is presented. The layout-driven TR-ARCHITECT combines two costs i.e. SOC test time and TAM wire length into one cost function and depending on the weight associated with each cost, computes an optimized test architecture. The presented TR-ARCHITECT minimizes the TAM wire length by assigning neighboring cores as much as possible to the same TAM and by determining a layout-driven optimal ordering of cores connected to the same TAM. Finally, experimental results for the *ITC'02 SOC TestBenchmarks* [MIC] are presented.

4.2 Prior Work

Most papers that address the issue of minimizing the wire length of the on-chip test infrastructure focus on (core-level) scan chain design. Early papers in this field [Dow96, CLH96, LCH96, NKTG97] describe the ordering of the flip flops in a single scan chain. This problem is equal to the well-known $\mathcal{NP}$-hard *Traveling Salesman Problem* (TSP) [GJ79], for which many heuristic algorithms are available. In many practical cases, multiple scan chains are designed, in order to make better use of the available bandwidth over the IC pins to bring test patterns in and out of the circuit, and hence

reducing the test-application time. This leads to a *Multi Traveling Salesmen Problem* (MTSP), in which the scan flip flops need to be both partitioned and ordered. Barbagallo et al. [BBG+98] describe a genetic algorithm to address this problem; their computational complexity seems to be an issue, as they report several hours computation time for cores with less than 2,000 flip flops.

The only researchers, that have addressed the issue of test architecture design while taking both SOC test time as well as TAM wire length into account are Chakrabarty and Iyengar [Cha00a, IC02]. [Cha00a] extends Chakrabarty's original TAM design approach that minimizes test time through ILP [Cha00b] with a rudimentary form of TAM wire length minimization. For every pair of cores, the user's preference for assigning the cores to the same TAM is expressed by a 0-1 constant. The user also specifies how many of these preferences should at least be rewarded in any solution generated by the ILP solver. [IC02] extends this approach by adding another 0-1 constant for every pair of cores, expressing the user's preference for *not* assigning the cores to the same TAM.

While it is acknowledged that these papers were the first to add wire length optimization to architecture design, their proposed solution approaches have many shortcomings. They are based on the Hybrid test bus architecture, which does not allow for core-external testing [GM02b] as discussed in the previous chapter (see Section 3.2). The approaches do not support optimization of TAMs and wrappers in conjunction. They work with a fixed user-specified number of TAMs, whereas most users only want to specify the total number of TAM wires, and leave the number of TAMs to the optimization algorithm. Both approaches lack a real wire length model. The binary 0-1 constants only provide a very coarse way to express preferences and do not allow for gradation of layout distances between cores. Despite its $\mathcal{NP}$-hard character, the problem is addressed by ILP, and hence only small problem instances can be handled within practical computation time bounds [ICM02b]. Finally, the test time penalty of taking the TAM wire length preferences into account is rather high; up to 65% for the small examples described in [Cha00a].

4.3 Wire-Length Cost Model

To implement an SOC test architecture in the corresponding SOC, the cores assigned to the same TAM have to be physically connected through one or more TAM wires, as well as the TAM wires have to be routed from the SOC pins to the respective cores and vice versa. The total wire length required to route all TAM wires in an architecture is refereed to as *TAM wire length*.

The most accurate TAM wire length can be obtained by creating an actual layout of the SOC with the proposed test architecture. However, layout generation is a computational-intensive task and usually, the SOC test architecture design is carried out by using iterative procedures. Therefore, complete layout generation cannot be afforded during the test architecture design, in which many times TAM wire length needs to be evaluated. Here, a simple TAM wire length model is presented, which, on one hand, allows for fast computation, but one the other hand, is still sufficiently accurate

to force test architecture optimization algorithm to assign neighboring cores as much as possible to the same TAM.

The proposed TAM wire length model makes the following assumptions.

1. ***Coordinate system***

    - Only the first quadrant of an orthogonal coordinate system is used. This assumption is without loss of generality; any SOC layout can be made to meet this assumption with a simple translation. The fact that the SOC is in the first quadrant means that all coordinates will be non-negative numbers, which simplifies calculations.
    - All coordinates are (non-negative) *integers*. The unit of the coordinates is not specified, but should be consistent for all coordinates belonging to the same SOC. This assumption is again without loss of generality, and meant to simplify calculations.

2. ***SOC layout position***

    - The SOC layout is assumed to be a rectangle, of which the bottom-left corner coincides with the origin $(0, 0)$ of the used coordinate system. For rectangular SOCs, this assumption is without loss of generality, as it can be met by applying a simple translation and/or rotation.
    - The coordinates of the center of the SOC layout are specified as (X, Y). From this, and from the fact that the bottom-left corner is at $(0, 0)$, one can calculate the positions of the horizontal boundaries of the SOC layout to be at $y = 0$ and $y = 2Y$, while the vertical boundaries of the SOC are at $x = 0$ and $x = 2X$.

3. ***Core layout position***

    - The position of each core c is specified by a pair of coordinates (x_c, y_c), corresponding to the center of the bounding box of the layout block of c. All TAM wires to and from c are assumed to start and end in at (x_c, y_c).

4. ***Layout distances***

    - For calculating distances between cores, the *Manhattan* distance is used instead of the *Euclidean* distance. This is motivated by the fact that SOC routing channels only allow horizontal and vertical wiring. In addition, the Manhattan distance function simplifies distance calculations. The distance between cores c_1 and c_2 is $d(c_1, c_2) = |x_{c_1} - x_{c_2}| + |y_{c_1} - y_{c_2}|$.
    - For calculating the distance between a core and the SOC boundary, the shortest distance between that core and any of the SOC boundaries is used. Hence, $d(c) = \min(x_c, y_c, 2X - x_c, 2Y - y_c)$. This assumption is based on the idea that the set of pins on which the TAM wires will be multiplexed is not pre-determined.

Note that the assumptions above are also compatible with the format of the layout information in the *ITC'02 SOC Test Benchmarks* [MIC02].

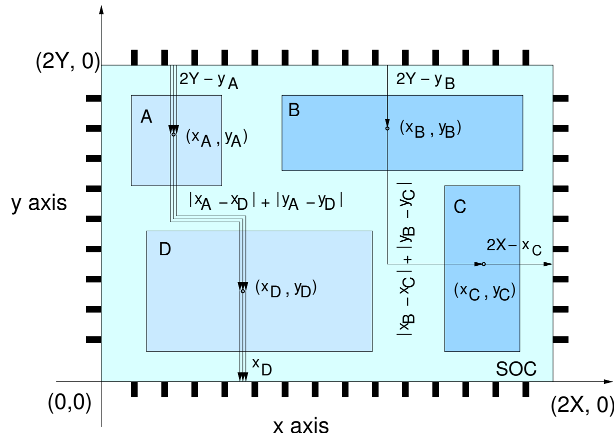
In this chapter, a TAM r is fully represented by its width $w(r)$ and an ordered list of cores $< c_1, c_2, \ldots, c_{|r|} >$. The wire length for a TAM r is the sum of the distance between the SOC boundary and core c_1, the distances between all subsequent pairs of cores in r, and the distance between core $c_{|r|}$ and the SOC boundary, multiplied by the number of wires $w(r)$. The total wire length $l(r)$ of TAM r can be written as follows:

$$l(r) = w(r) \times \left(d(c_1) + \sum_{i=1}^{|r|-1} d(c_i, c_{i+1}) + d(c_{|r|}) \right) \tag{4.1}$$

The TAM wire length L for an SOC with a set of TAMs R is the sum of the wire lengths of the individual TAMs in R and can be written as:

$$L = \sum_{r \in R} l(r) \tag{4.2}$$

A simple example SOC layout to illustrate the proposed TAM wire length model is shown in Figure 4.2. The SOC contains four cores, named A, B, C, and D. The SOC has two TAMs; TAM r_1 of width $w(r_1)$ is connected to cores A and D, while TAM r_2 of width $w(r_2)$ is connected to cores B and C.



$$l(r_1) = w(r_1) \times (2Y - y_A + |x_A - x_D| + |y_A - y_D| + x_D)$$
$$l(r_2) = w(r_2) \times (2Y - y_B + |x_B - x_C| + |y_B - y_C| + 2X - x_C)$$
$$L = l(r_1) + l(r_2).$$

Figure 4.2: Example SOC layout to illustrate the proposed TAM wire-length model.

4.4 Optimal Ordering of Cores

The wire length $l(r)$ of a TAM r depends on its width $w(r)$ and the order of cores that are connected to the TAM. As the width $w(r)$ is generally determined by the test

architecture design algorithm, minimizing the wire length of a TAM directly relates to determining an optimal order of cores with respect to wire length. The problem of determining an optimal ordering of cores that are connected to a TAM, can be formalized as follows:

[OOC] **OPTIMAL ORDERING OF CORES**

Instance: Given a TAM r with width $w(r)$ and a list of cores $< c_1, c_2, ..., c_{|r|} >$. For each core c in the list, the center coordinates (x_c, y_c) are given. Furthermore,the center coordinates (X, Y) of the SOC for which the TAM is designed are given.
Objective: Determine an optimal order of cores $c_1, c_2, ..., c_{|r|}$ such that the total wire length $l(r)$ required to connect $w(r)$ TAM wires to the cores and the SOC pins is minimized. □

Here, a two-steps approach is proposed to solve this problem. In Step 1, an optimal ordering of the cores is determined such that their TAM interconnect wire length is minimized. In Step 2, the overall TAM wire length is calculated by adding the wires from the first and the last cores in the TAM to the SOC pins.

Step 1 can be mapped to a simple graph problem. Let $G = (V, E)$ be a complete undirected weighted graph, where vertex $v_i \in V$ corresponds to core c_i and weight $W(e_{ij})$ on edge $e_{ij} \in E$ represents the wire -length cost of connecting cores c_i and c_j, i.e. $w(r) \times d(c_i, c_j)$. All cores have to appear in the connecting sequence once and at most once, and have to be connected with the minimum sum of weights on the edges. Therefore, Step 1 is equivalent to the problem of finding the shortest path through all nodes in the complete undirected weighted graph G. This problem is very similar to the well-known *Traveling Salesman Problem* (TSP) [GJ79], in which a traveling salesman has to find the shortest tour through a given set of cities with pre-defined distances.

The only difference between Step 1 and TSP is that in Step 1, one needs to find the shortest *path* through all cities (nodes), while TSP is after the shortest *tour* through all cities. Step 1 can be transformed into an instance of TSP by adding one node with equal distances to all other nodes. TSP is known to be $\mathcal{NP}$-hard [GJ79]. In order to avoid intractable computation time, an efficient, yet effective heuristic algorithm should be used. Several efficient heuristic algorithms have been proposed in literature to solve TSP. It is not the purpose here to present a heuristic which is better than the previous algorithms; hence a simple greedy heuristic [CLH96] is used for Step 1.

Algorithm 4.1 lists the pseudo-code for the proposed two-steps approach to solve OOC. Step 1 (Lines 1–13) consists of an initialization, followed by an iterative loop. In the initialization (Lines 2–5), a complete graph $G = (V, E)$ from the cores in TAM r is created. Initially, every vertex in G contains only one core. The weights on the edges are assigned as the wire length cost of connecting the cores in the corresponding nodes (Line 4). Initially, the summed wire length so far is set to zero.

In the main loop (Lines 6–13), in a greedy fashion every time two vertices which have the minimum cost on the edge are combined to form a super vertex iteratively. The merge order corresponds to the ordering of cores in the TAM. Once a super-vertex is formed, the two vertices and the edges connected to them are deleted from the set V and E respectively. The weights on the edges connecting the super-vertex to other

vertices are re-calculated and re-assigned. This step continues until a single vertex (super-vertex) is left.

Finally in Step 2 (Lines 14–15), the total wire length for the TAM r is calculated by adding the wire length costs of connecting the first and the last cores in the TAM to the current wire length.

Algorithm 4.1 [OPTIMALORDERDESIGN $(r, w(r))$]

1 // Step 1: ordering of cores
2 create a complete graph $G = (V, E)$ from cores in r;
3 **for all** edges $e_{ij} \in E$ **do**
4     $W(e_{ij}) := w(r) \times d(c_i, c_j)$ **od**;
5 *sum* := 0;
6 **while** $|V| > 1$ **do**
7   find edge e^*_{ij} for which $W(e^*_{ij}) = \min_{e \in E} W(e)$;
8   merge vertices v_i and v_j to form a super-vertex $v_{\{i,j\}}$;
9   *sum* := *sum* + $W(e^*_{ij})$;
10   delete edges incident to vertices v_i and v_j;
11     $V := V \backslash \{v_i, v_j\} \cup v_{\{i,j\}}$;
12   assign new weights on edges connecting $v_{\{i,j\}}$ to other vertices;
13 **od**;
14 // Step 2: calculation of total wire length
15 $l(r) :=$ *sum* $+ w(r) \times (d(c_{v_1}) + d(c_{v_{|r|}}))$;

A super-vertex represents a chain of vertices and only the ending vertices of the chain sequence can be connected to other vertices. For example, to connect one super-vertex to another vertex, there are two possible connections. The minimum cost of all possible connections is considered as the weight on the edge connecting the two vertices. Figure 4.3(a) shows an example calculation of weight on the edge between a vertex v_3 and a super-vertex $v_{\{1,2\}}$. The super vertex $v_{\{1,2\}}$ is formed by combining vertices v_1 and v_2.

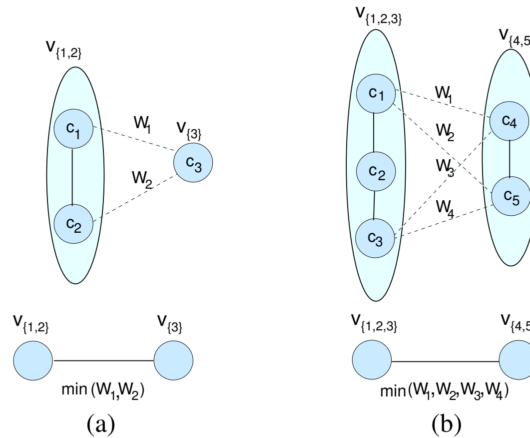


Figure 4.3: Calculation of weight on the edge between (a) a vertex and a super-vertex and (b) between two super-vertices.

For this case, there are two possible connections $(v_1 - v_3)$ and $(v_2 - v_3)$ with costs W_1 and W_2 respectively. The minimum of W_1 and W_2 is assigned as the weight on the edge connecting vertices $v_{\{1,2\}}$ and v_3. Similarly, Figure 4.3(b) shows an example calculation of weight on the edge between two super-vertices. In this case, there are four possible connections and the minimum cost of these four connections is assigned as the weight on the edge connecting the two super-vertices.

4.5 Layout-Driven Test Architecture Design

Conventional test architecture design procedures do not consider any particular ordering of cores in TAMs. Therefore, the most trivial way to minimize the TAM wire length for a given architecture is to calculate an optimal ordering of cores for every individual TAM in the architecture. For this purpose, Algorithm 4.1 can be used as a stand-alone procedure for each TAM. Figure 4.4(a) shows this flow.

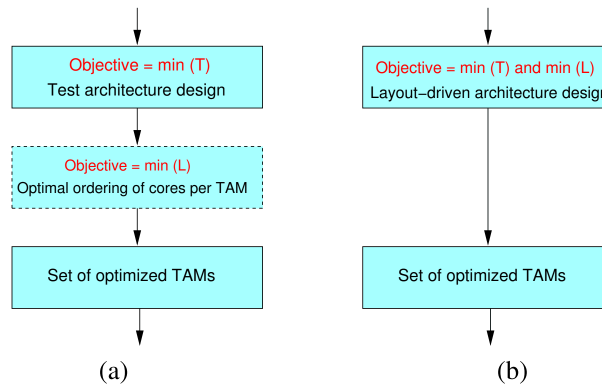


Figure 4.4: (a) Conventional test architecture design with additional cores-ordering step, and (b) the proposed layout-driven test architecture design.

Unfortunately, this scheme will not work in all cases. It is due to the fact that may be during the architecture design itself, cores placed at very far positions in the SOC layout are assigned to the same TAM in order to minimize the overall SOC test time. No matter how optimal ordering of cores is carried out for such TAMs, they will always result in large wire lengths. Therefore, to avoid such architecture designs, minimization of test time and wire length should be done in conjunction. Figure 4.4(b) shows this new flow and it is referred to as *layout-driven test architecture design*.

In the proposed layout-driven test architecture design, the objective is to minimize both the time necessary to complete all SOC tests, as well as the wire length necessary for the TAMs. These two minimization criteria can be conflicting. To provide a trade-off between the overall test time and the TAM wire length, a combined weight-based cost function Z is used, where Z can be defined as follows:

$$Z = \alpha \times T + \beta \times L \tag{4.3}$$

where, parameters α and β are user-defined inputs and represent the relative weights associated with the SOC test time (T) and the TAM wire length (L) respectively.

The problem of layout-driven test architecture design can be formally defined as follows:

[LDTAD] **LAYOUT-DRIVEN TEST ARCHITECTURE DESIGN**

Instance: Given an SOC with center layout coordinates (X, Y) and a set of cores C. Given for each core $c \in C$ its center layout coordinates (x_c, y_c), the number of test patterns p_c, the number of functional input terminals i_c, the number of functional output terminals o_c, the number of bidirectional terminals b_c, the number of scan chains s_c, and for each scan chain k, the length of the scan chain in flip flops $l_{c,k}$. Furthermore, the maximum number of SOC-level TAM wires $w_{\max}$, and the relative weights α and β are given.
Objective: Determine a set of TAMs R (i.e. the TAM widths, ordered lists of cores assigned to TAMs, and wrappers for all cores) such that number of used TAM wires does not exceed $w_{\max}$ and the overall cost Z is minimum. □

In Chapter 3, TR-ARCHITECT proved very effective in minimizing SOC test time. TR-ARCHITECT presented in Chapter 3 was unaware of the layout-positions of the SOC and its cores, and consequently neither did minimize the TAM wire length nor order the cores per TAM. To minimize the weighted sum of test time and wire length, a new, layout-driven version of TR-ARCHITECT is presented here. In the new layout-driven TR-ARCHITECT, assignment of cores which are far apart in the SOC-layout to the same TAM is avoided,

Algorithm 4.2 shows the pseudo-code for the new layout-driven TR-ARCHITECT. The layout-driven TR-ARCHITECT only contains four steps as compared to five steps in the original TR-ARCHITECT. The layout-driven TR-ARCHITECT does not require the CHECKEMPTYWIRE step and the reasons for this are explained in the sequel of this section. In each step, instead of optimizing the test time as in the original TR-ARCHITECT, the new version optimizes the total cost Z. It is important to note, that the basic concepts behind the original four steps still hold true in their new layout-driven versions. The modification details about all four steps are given below.

Algorithm 4.2 [LAYOUTDRIVEN TR-ARCHITECT]

```
1  LAYOUTDRIVEN CREATESTARTSOLUTION;
2  LAYOUTDRIVEN OPTIMIZE-BOTTOMUP;
3  LAYOUTDRIVEN OPTIMIZE-TOPDOWN;
4  LAYOUTDRIVEN RESHUFFLE
```

In the layout-driven TR-ARCHITECT, the overall cost $z(r)$ of a TAM r is calculated as follows: $z(r) = \alpha \times t(r) + \beta \times l(r)$, where $t(r)$ and $l(r)$ represent the test time and wire length for TAM r respectively. To determine the cost $z(r)$ of a TAM r with width $w(r)$, a procedure TAMCOST$(r, w(r))$ is used. The procedure TAMCOST$(r, w(r))$

uses Algorithm 4.1 to calculate an optimal ordering of cores. For test time, the procedure TAMCOST$(r, w(r))$ uses the test time calculations described in Section 3.7. To determine the overall cost Z for a given set of TAMs R, a procedure COST(R) is used.

4.5.1 Layout-Driven Creating a Start Solution

In the step LAYOUTDRIVEN CREATESTARTSOLUTION, an initial test architecture is created which is then further optimized by the steps to follow. Algorithm 4.3 outlines the pseudo-code for the procedure.

In Step 1 (Lines 3–7), cores are assigned to one-bit wide TAMs. If $w_{\max} \geq |C|$, each core gets assigned; if $w_{\max} < |C|$, only the largest $w_{\max}$ cores get assigned. 'Large' is here defined by the test-data volume for each core, according to which the cores have been sorted in Line 1. In case $w_{\max} = |C|$, the procedure is now finished.

Algorithm 4.3 [LAYOUTDRIVEN CREATESTARTSOLUTION]

```
1  sort C such that                                      /* sort cores in non-increasing order of their
     t({1}) ≥ t({2}) ≥ ... ≥ t({|C|});                     test-data volume */
2  R := ∅;                                               // initially, the set of TAMs R is empty
3  for i := 1 to min(w_max, |C|) do                      // Step 1: iteratively, assign cores to one-bit wide TAMs
4     w(r_i) := 1; r_i := {i};                           // create a one-bit wide TAM and connect a core to it
5     TAMCOST(r_i, 1);                                   // calculate the cost for the created TAM
6     R := R ∪ {r_i};                                    // add the created TAM to the set of TAMs R
7  od;
8  if w_max < |C| then                                   // Step 2: if cores left, add to TAMs that result in least cost
9     for i := w_max + 1 to |C| do                       // iteratively, assign remaining un-assigned cores
10       Z* := ∞; z(r*) := ∞
11       for all r ∈ R do                                // iteratively find the TAM that results in minimum cost
12          r_temp := r ∪ {i};                           /* create a TAM r_temp with cores in r and add an
13          R_temp := R\{r} ∪ {r_temp};                     un-assigned core it */
14          Z_temp := COST(R_temp);                      // calculate new overall cost Z_temp
15          if Z_temp < Z* then                          // if the new overall cost is less than the previous cost
16             Z* := Z_temp; r* := r_temp; r_del := r;   // accept the proposal and update the current cost
17          else if Z_temp == Z* then                    // if the new overall cost is equal to the previous cost
18             if z(r_temp) < z(r*) then                 // if cost of r_temp is less than cost of previous TAM
19                Z* := Z_temp; r* := r_temp; r_del := r; // accept the proposal and update the current cost
20             fi;
21          fi;
22       od;
23       R := R\{r_del} ∪ {r*};                          // update the TAMs in R
24    od;
25 fi;
26 if w_max > |C| then                                   // Step 3: if wires left, try to add to most-occupied TAMs
27    w_free := w_max − |C|; distribute := true;         // calculate the number of free wires
28    while w_free > 0 ∧ distribute do                   // iteratively, assign remaining un-used wires
29       Z := COST(R);                                   // calculate the current overall cost
30       find r_max for which t(r_max) = max_{r∈R} t(r); // find the TAM with the maximum test time
31       r* := r_max; w(r*) := w(r*) + 1;                // assign one more wire to this TAM
32       R* := R\{r_max} ∪ {r*};                         // create the new TAM set R*
33       Z* := COST(R*);                                 // calculate the new overall cost
34       if Z* ≤ Z then                                  // if new overall cost does not exceed the current cost
35          R := R*; w_free := w_free − 1;               // update TAM set R and number of free wires
36       else distribute = false;                        // no more free wires distribution is possible
37       fi;
38    od;
39 fi;
```

In case $w_{\max} < |C|$, there are still un-assigned cores. In Step 2 (Lines 8–25), these

cores are added iteratively to the TAMs which result in minimum overall cost Z. In case, two TAMs result in the same overall cost (Lines 17–19) then the TAM that has the smallest cost associated with it is considered.

In case $w_{\max} > |C|$, there are still un-assigned TAM wires. In Step 3 (Lines 26–39), these wires are added iteratively to the TAM with the largest test time provided there is no increase in the overall cost Z. This procedure returns a set of TAMs R and a number of unused wires w_{free}.

The next two steps, i.e. LAYOUTDRIVEN OPTIMIZE-BOTTOMUP and LAYOUT-DRIVEN OPTIMIZE-TOPDOWN, try to merge the cores of two TAMs into a new TAM, such that the new TAM requires less wires and the wires that are freed up in the process can be utilized for an overall cost reduction. The freed-up wires can reduce the overall cost in the following two ways:

1. freed-up wires can be used by the TAM with the largest test time to minimize T,
2. the TAM wire length L is reduced as less wires need to be routed.

4.5.2 Layout-Driven Optimize BottomUp

The procedure LAYOUTDRIVEN OPTIMIZE-BOTTOMUP optimizes the overall cost Z by trying to merge the TAM with the shortest test time with another TAM, such that the wires that are freed up in this process can be used for an overall cost reduction. Algorithm 4.4 lists the pseudo-code for the procedure LAYOUTDRIVEN OPTIMIZE-BOTTOMUP. It is an iterative procedure and every iteration contains two steps.

Algorithm 4.4 [LAYOUTDRIVEN OPTIMIZE-BOTTOMUP]

```
 1  improve := true;                                   // initially, improvement is possible
 2  while |R| > 1 ∧ improve do                         // while multiple TAMs and improvement possible
 3    find r_min for which t(r_min) = min_{r∈R} t(r);  // Step 1: find TAM r_min with minimum test time
 4    Z := COST(R); Z* := ∞; z(r*) := ∞                // calculate the current overall cost Z
 5    for all r ∈ R\{r_min} do                         // iteratively, find merge candidate TAM r
 6      r_temp := r_min ∪ r;                           // create a TAM r_temp with cores in r_min and r
 7      w(r_temp) := max(w(r_min), w(r));              // assign it the maximum of the widths of r_min and r
 8      w^t_free := w_free + min(w(r_min), w(r));      // calculate the possible number of free wires
 9      R_temp := R\{r_min, r} ∪ {r_temp};             // create a new TAM set R_temp considering this merge
10      DISTRIBUTEWIRES (R_temp, w^t_free);            // distribute all freed-up wires in the TAM set R_temp
11      Z_temp := COST(R_temp);                        // calculate the new overall cost Z_temp
12      if (Z_temp ≤ Z) ∧ (Z_temp < Z*) then           // if Z_temp is minimum and does not exceed Z
13        Z* := Z_temp; z(r*) := z(r_temp);            // accept this merge proposal
14        w*_free := w^t_free; R* := R_temp;
15      else if (Z_temp ≤ Z) ∧ (Z_temp == Z*) then     // if Z_temp is equal to Z* and does not exceed Z
16        if z(r_temp) < z(r*) then                    // if new proposal has less cost than the previous one
17          Z* := Z_temp; z(r*) := z(r_temp);          // accept this merge proposal
18          w*_free := w^t_free; R* := R_temp;
19        fi;
20      fi;
21    od;
22    if Z* ≤ Z then                                   // Step 2: if a merge proposal was found
23      R := R*; w_free := w*_free;                    // update the TAM set R and number of free wires
24    else improve := false; fi;                       // if no proposal was found, no improvement possible
25  od;
```

In (Lines 3–9), the procedure finds a TAM $r_{\min}$ with minimum test time, i.e. $t(r_{\min}) = \min_{r \in R} t(r)$. The cores in TAM $r_{\min}$ and the cores in one of the other TAMs, say r, are merged into a new TAM, say r_{temp}, with width $\max(w(r_{\min}), w(r))$. As the new TAM r_{temp} only uses $\max(w(r_{\min}), w(r))$ wires, $\min(w(r_{\min}), w(r))$ wires are freed up additionally to the already free wires w_{free} (if any). To reduce the overall cost Z, the total freed-up wires w^t_{free} are distributed over all TAMs in R_{temp} using a procedure DISTRIBUTEWIRES (Line 10). The procedure DISTRIBUTEWIRES is based on Step 3 in Algorithm 4.3. The TAM r is selected from $R \backslash \{r_{\min}\}$ such that Z_{temp} is minimum and Z_{temp} does not exceed the overall cost Z. In case two TAMs result in the same overall cost Z_{temp} (Lines 15–19), the TAM that has the smallest cost associated with it is selected. In Line 23, the merge is implemented and R is updated.

The procedure ends if all TAMs have been merged into one single TAM, or when no TAM r can be found such that Z_{temp} does not exceed the current overall cost Z. This procedure returns a set of TAMs R and a number of unused wires w_{free}.

4.5.3 Layout-Driven Optimize TopDown

The procedure LAYOUTDRIVEN OPTIMIZE-TOPDOWN tries to optimize the overall cost Z for a given test architecture in two subsequent steps. In Step 1, the algorithm iteratively tries to merge the TAM with the longest test time with another TAM, such that the SOC test time T is reduced. As the overall cost Z linearly depends on the SOC test time T, this step can reduce the overall cost. In case Step 1 does not yield cost improvement any more, Step 2 is executed. In this step, the algorithm iteratively tries to free up wires by merging two TAMs that do not have the longest test time, under the condition that the merge does not increase the overall cost. The wires that are freed up can be used for an overall cost reduction. Algorithm 4.5 lists the pseudo-code for procedure LAYOUTDRIVEN OPTIMIZE-TOPDOWN.

In Step 1 (Lines 1–26), the procedure iteratively finds a TAM $r_{\max}$ with the longest test time. Subsequently, the procedure tries to find a TAM $r \in R \backslash \{r_{\max}\}$, which could be merged with TAM $r_{\max}$ into a new TAM r_{temp} with $w(r_{\text{temp}}) = w(r_{\max}) + w(r)$, such that the new overall cost Z_{temp} is minimum and Z_{temp} does not exceed the current overall cost Z. If such a TAM r is found, then the merge is implemented and R is updated (Lines 21–22); else, the TAM $r_{\max}$ is put into set R_{skip} and Step 2 is executed. Step 2 (Lines 27–56) is quite similar to Step 1, apart from the following two differences:

1. as merge candidates, it only considers the TAMs in set $R \backslash R_{\text{skip}}$,
2. the width of the merged TAM $w(r_{\text{temp}})$ is determined by a linear search between the lower limit $w_L = \max(w(r_{\max}), w(r))$ and the upper limit $w_U = w(r_{\max}) + w(r)$, such that the overall cost after the merge does not exceed the overall cost Z and is minimum.

If such a search is successful, the merge is implemented and R is updated. If the search was not successful, then the TAM $r_{\max}$ is added to set R_{skip}.

Algorithm 4.5 [LAYOUTDRIVEN OPTIMIZE-TOPDOWN]

1	*improve* := *true*;	// initially, improvement is possible
2	**while** $\|R\| > 1 \wedge$ *improve* **do**	// **Step 1:** while $\|R\| > 1$ and improvement possible
3	find $r_{\max}$ for which $t(r_{\max}) = \max_{r \in R} t(r)$;	// find TAM $r_{\max}$ with maximum test time
4	$Z :=$ COST(R); $Z^* := \infty$; $z(r^*) := \infty$	// calculate the current overall cost Z
5	**for all** $r \in R \backslash \{r_{\max}\}$ **do**	// iteratively, find merge candidate TAM r
6	$r_{\text{temp}} := r_{\max} \cup r$;	// create a TAM r_{temp} with cores in $r_{\max}$ and r
7	$w(r_{\text{temp}}) := w(r_{\max}) + w(r)$; $w^t_{\text{free}} := w_{\text{free}}$;	// assign it the summed width of $r_{\max}$ and r
8	$R_{\text{temp}} := R \backslash \{r_{\max}, r\} \cup \{r_{\text{temp}}\}$;	// create a TAM set R_{temp} considering this merge
9	DISTRIBUTEWIRES $(R_{\text{temp}}, w^t_{\text{free}})$;	// distribute free wires (if any) in the set R_{temp}
10	$Z_{\text{temp}} :=$ COST(R_{temp});	// calculate the new overall cost Z_{temp}
11	**if** $(Z_{\text{temp}} \leq Z) \wedge (Z_{\text{temp}} < Z^*)$ **then**	// if Z_{temp} is minimum and does not exceed Z
12	$Z^* := Z_{\text{temp}}$; $z(r^*) := z(r_{\text{temp}})$;	// accept this merge proposal
13	$R^* := R_{\text{temp}}$; $w^*_{\text{free}} := w^t_{\text{free}}$;	
14	**else if** $(Z_{\text{temp}} \leq Z) \wedge (Z_{\text{temp}} == Z^*)$ **then**	// if Z_{temp} is equal to Z^* and does not exceed Z
15	**if** $z(r_{\text{temp}}) < z(r^*)$ **then**	// if new proposal has less cost than the previous one
16	$Z^* := Z_{\text{temp}}$; $z(r^*) := z(r_{\text{temp}})$;	// accept this merge proposal
17	$R^* := R_{\text{temp}}$; $w^*_{\text{free}} := w^t_{\text{free}}$;	
18	**fi**;	
19	**fi**;	
20	**od**;	
21	**if** $(Z^* < Z)$ **then**	// if a merge proposal was found
22	$R := R^*$; $w_{\text{free}} := w^*_{\text{free}}$;	// update the TAM set R and number of free wires
23	**else** *improve* := *false*;	// no improvement possible
24	$R_{\text{skip}} := \{r_{\max}\}$;	// exclude $r_{\max}$ from further consideration
25	**fi**;	
26	**od**;	
27	**while** $R_{\text{skip}} \neq R$ **do**	// **Step 2:** while TAMs are still under consideration
28	find r_{max} for which $t(r_{\max}) = \max_{r \in R \backslash R_{\text{skip}}} t(r)$;	// find the TAM $r_{\max}$ with maximum test time
29	$Z :=$ COST(R); $Z^* := \infty$; $z(r^*) := \infty$	// calculate the current overall cost Z
30	**for all** $r \in R \backslash (R_{\text{skip}} \cup \{r_{\max}\})$ **do**	// iteratively, find merge candidate TAM r
31	$r_{\text{temp}} := r_{\max} \cup r$;	// create a TAM r_{temp} with cores in $r_{\max}$ and r
32	$w_U := w(r_{\max}) + w(r)$;	// assign upper limit on the TAM width
33	$w_L := \max(w(r_{\max}), w(r))$;	// assign lower limit on the TAM width
34	*found* := $false$; $w(r_{\text{temp}}) := w_L$;	// start with TAM width equal to lower limit
35	**while** $\neg$*found* $\wedge (w(r_{\text{temp}}) \leq w_U)$ **do**	// assign the actual width through linear search
36	$R_{\text{temp}} := R \backslash \{r_{\max}, r\} \cup \{r_{\text{temp}}\}$;	// create a TAM set R_{temp} considering this merge
37	$w^t_{\text{free}} := w_{\text{free}} + w_U - w(r_{\text{temp}})$;	// calculate the possible number of free wires
38	DISTRIBUTEWIRES $(R_{\text{temp}}, w^t_{\text{free}})$;	// distribute free wires in the TAM set R_{temp}
39	$Z_{\text{temp}} :=$ COST(R_{temp});	// calculate the new overall cost Z_{temp}
40	**if** $(Z_{\text{temp}} \leq Z) \wedge (Z_{\text{temp}} < Z^*)$ **then**	// if Z_{temp} is minimum and does not exceed Z
41	$Z^* := Z_{\text{temp}}$; $z(r^*) := z(r_{\text{temp}})$;	// accept this merge proposal
42	$R^* := R_{\text{temp}}$; *found* := *true*; $w^*_{\text{free}} := w^t_{\text{free}}$;	
43	**else if** $(Z_{\text{temp}} \leq Z) \wedge (Z_{\text{temp}} == Z^*)$ **then**	// if Z_{temp} is equal to Z^* and does not exceed Z
44	**if** $z(r_{\text{temp}}) < z(r^*)$ **then**	// if new proposal has less cost than the previous one
45	$z(r^*) := z(r_{\text{temp}})$; $Z^* := Z_{\text{temp}}$;	// accept this merge proposal
46	$R^* := R_{\text{temp}}$; $w^*_{\text{free}} := w^t_{\text{free}}$;	
47	**fi**;	
48	**fi**;	
49	$w(r_{\text{temp}}) := w(r_{\text{temp}}) + 1$;	// increase TAM width linearly
50	**od**;	
51	**od**;	
52	**if** $(Z^* \leq Z)$ **then**	// if a merge proposal was found
53	$R := R^*$; $w_{\text{free}} := w^*_{\text{free}}$;	// update the TAM set R and number of free wires
54	**else** $R_{\text{skip}} := R_{\text{skip}} \cup \{r_{\max}\}$;	// no merge proposal was found
55	**fi**;	// exclude $r_{\max}$ from further consideration
56	**od**;	

4.5.4 Layout-Driven Reshuffle

The procedure LAYOUTDRIVEN RESHUFFLE tries to minimize the overall cost of a given test architecture by moving one of the cores assigned to the TAM with the longest

test time to another TAM, provided that this reduces the overall cost. Algorithm 4.6 lists the pseudo-code for procedure LAYOUTDRIVEN-RESHUFFLE.

In Line 3, the procedure identifies a TAM $r_{\max}$ with the longest test time. If $r_{\max}$ contains only one core, the procedure is terminated. If $r_{\max}$ contains multiple cores, core j^* with the smallest test time is identified. The procedure searches through the other TAMs, to see if there is one of them to which core j^* can be added without exceeding the overall cost Z. If that is the case, core j^* is moved from the TAM $r_{\max}$ to this other TAM (Lines 15–17). This procedure is repeated until the TAM with the longest test time contains only one core, or when no beneficial core re-assignment can be found.

Algorithm 4.6 [LAYOUTDRIVEN RESHUFFLE]

```
 1 improve := true;                                           // initially, improvement is possible
 2 while improve do                                           // while improvement is possible
 3    find r_max for which t(r_max) = max_{r∈R} t(r);         // find TAM r_max with maximum test time
 4    if |r_max| = 1 then                                     // if it contains only one core
 5       improve := false;                                    // no improvement possible
 6    else                                                    // else, TAM contains multiple cores
 7       find core j* for which t(j*) := min_{j∈r_max} t(j);  // find core j* in r_max with minimum test time
 8       TAMFound := false; Z := COST(R);                     // calculate the current overall cost Z
 9       R_temp := R\{r_max}; r_temp: = r_max\{j*};           // create a TAM set R_temp considering this move
10       while r ∈ R\{r_max} ∧ ¬ TAMFound do                  // iteratively, find merge candidate TAM r
11          r* := r ∪ {j*}; w(r*) := w(r);                    // create a TAM r* with cores in r and core j*
12          R_temp := R\{r} ∪ {r*, r_temp}; w^t_free := w_free; // update the TAM set R_temp
13          DISTRIBUTEWIRES (R_temp, w^t_free);               // distribute free wires (if any) in the set R_temp
14          Z_temp := COST(R_temp);                           // calculate the new overall cost Z_temp
15          if (Z_temp < Z) then                              // if Z_temp is less than the current overall cost Z
16             TAMFound := true;                              // accept this merge
17             R := R_temp; w_free := w^t_free;               // update TAM set R and number of free wires
18          fi;
19       od;
20       if ¬TAMFound then                                    // no merge was found
21          improve := false;                                 // no improvement possible
22       fi;
23    fi;
24 od;
```

It is important to note here that, due to weight dependencies on the TAM wire length and the SOC test time in the overall cost function Z, it is possible that for the cases with a small value of α and a large value of β, a large number of TAM wires remains un-assigned. This is due to the fact that adding one more wire to a TAM directly increases the wire length but does not necessarily decreases its test time. However, it is also possible that if all the wires are added iteratively to the TAM with the longest test time, the decrease in the SOC test time might be larger than the increase in the TAM wire length. To account for this phenomenon, in case TAM wires are left after the LAYOUTDRIVEN RESHUFFLE step, all unused wires are added iteratively to the TAM with the longest test time. The new architecture is accepted only if the new overall cost is less than the overall cost without the TAM wires distribution.

As all four steps described above always keep track of the number of unused wires, unlike the original TR-ARCHITECT, the CHECKEMPTYWIRE step is not required in the layout-driven TR-ARCHITECT to calculate the total number of unused wires.

4.6 Experimental Results

In this section, experimental results for the proposed layout-driven version of TR-ARCHITECT are presented. For experiments, five SOCs were selected from the *ITC'02 SOC Test Benchmarks* [MIC02]. These five SOCs were selected because they are the only ones with a large number of cores, and for which test time continues to decrease for increasing values of TAM width $w_{\max}$ [GM02c]. In the experiments, it has been assumed that an SOC only contains one level of hierarchy, i.e. the SOC itself and all its embedded cores, even though some of these SOCs originally contain multiple levels of design hierarchy. Also, only the core-internal tests of the SOCs have been considered i.e. the interconnect tests for the top-level SOC itself have not been taken into account. As all cores in the SOC benchmark set have a fixed number of scan chains and lengths, all cores are considered as hard cores. The proposed layout-driven TR-ARCHITECT is able to generate both Hybrid test bus and TestRail architectures like the original TR-ARCHITECT in Chapter 3. Here, only Hybrid test bus architecture results are presented, but similar results could be obtained for TestRail architectures.

As the original benchmark SOCs do not have any data related to the positions of cores in the SOC layout, randomly-generated, but feasible floor plans are used for the benchmark SOCs. For the randomly generated floor-plans for all SOCs (except SOC a586710), the TAM wire length happened to be the same order of magnitude as the SOC test time. Therefore, the same order values were used for both α and β in the experiments. The values of α and β were selected, such that $\alpha + \beta = 1$. In the experiments, for all SOCs and all TAM widths, first a minimum weight on the test time ($\alpha = 0.1$) and a maximum weight on TAM wire length ($\beta = 0.9$) were selected. Iteratively, the value of α was increased and the value of β was decreased by 0.1.

In this section, test time and TAM wire length results for three approaches are compared. The baseline approach is obtained by the original version of TR-ARCHITECT, followed by a lexicographical ordering of the cores per TAM. The lexicographical ordering represents a random ordering with respect to the layout positions of the cores. The second approach uses the test architecture obtained by the original version of TR-ARCHITECT, followed by a layout-driven ordering of cores per TAM (as described in Section 4.4). This approach is also depicted in Figure 4.4(a). The third, and best approach, is based on the new layout-driven version of TR-ARCHITECT; it includes both a layout-driven test architecture design and a layout-driven ordering of cores per TAM.

Table 4.1 presents results for a range of $w_{\max}$ values for five benchmark SOCs. Columns 1 and 2 show the SOC name and the value of $w_{\max}$ respectively. Columns 3 and 4, present the SOC test time T (in number of clock cycles) and the TAM wire length L (in units) for the original TR-ARCHITECT with random ordering of cores per TAM respectively. Column 5 presents the TAM wire length results for the original TR-ARCHITECT with a layout-driven ordering of cores per TAM. Test times are not listed for this approach, as they are the same as the test times in Column 3. Columns 5 also show percentage (%) savings in TAM wire length ($\triangle L$) compared to the baseline approach.

Columns 6 and 7 present the results for the new layout-driven TR-ARCHITECT.

Table 4.1: Experimental results for SOC test time T and TAM wire length L of original and layout-driven TR-ARCHITECT.

SOC	w_{max}	Original TR-Architect		Layout-Driven Cores Ordering		Layout-Driven TR-Architect			
		T	L	L	$\triangle L$	T	$\triangle T$	L	$\triangle L$
d695	16	44307	86451	63339	26.7	42548	4.0	49404	42.9
	24	28576	78908	69344	12.1	30132	-5.4	32892	58.3
	32	21518	87860	80430	8.5	22438	-4.3	39241	55.3
	40	17677	146120	137044	6.2	17677	0.0	57132	60.9
	48	14608	163444	134697	17.6	16194	-10.9	56492	65.4
	56	12462	136828	136828	0.0	13354	-7.2	66024	51.7
	64	11033	87837	87837	0.0	11274	-2.2	73736	16.1
p22810	16	457433	541750	338608	37.5	462030	-1.0	104189	80.8
	24	302737	388607	331032	14.8	298054	1.5	147340	62.1
	32	222471	288573	234779	18.6	243791	-9.6	120342	58.3
	40	190995	415412	355063	14.5	194193	-1.7	94460	77.3
	48	157851	358205	310436	13.3	156472	0.9	143297	60.0
	56	145417	427962	316938	25.9	145417	0.0	105952	75.2
	64	133405	444676	338647	23.8	135571	-1.6	126948	71.5
p34392	16	1010821	118596	100316	15.4	1010821	0.0	59007	50.2
	24	663193	263968	211415	19.9	698844	-5.4	137900	47.8
	32	584524	259450	219562	15.4	584524	0.0	78771	69.6
	40	544579	474171	362181	23.6	544579	0.0	93499	80.3
	48	544579	474171	362181	23.6	544579	0.0	93499	80.3
	56	544579	474171	362181	23.6	544579	0.0	93499	80.3
	64	544579	474171	362181	23.6	544579	0.0	93499	80.3
p93791	16	1791638	649338	406713	37.4	1791638	0.0	289826	55.4
	24	1185434	513619	376939	26.6	1211149	-2.2	263828	48.6
	32	912158	1054240	759708	27.9	946879	-3.8	151676	85.6
	40	718005	704481	543507	22.9	745824	-3.9	313788	55.5
	48	601450	690570	538308	22.0	617782	-2.7	286215	58.6
	56	528925	1125430	904718	19.6	538346	-1.8	291288	74.1
	64	455738	764913	662107	13.4	480003	-5.3	181099	76.3
a586710	16	41523868	132700	132700	0.0	42117536	-1.4	52765	60.2
	24	28716501	112157	98547	12.1	28716501	0.0	82064	26.8
	32	22475033	228561	228561	0.0	22475033	0.0	88961	61.1
	40	19048835	303370	303370	0.0	19048835	0.0	118018	61.1
	48	15212440	411104	377216	8.2	15212440	0.0	323372	21.3
	56	13401034	206520	206520	0.0	13401034	0.0	137550	33.4
	64	12510356	232168	232168	0.0	12700205	-1.5	182267	21.5

The test times listed in Column 6 are the minimum SOC test times obtained from the layout-driven TR-ARCHITECT. For most cases, the minimum test time was achieved for $\alpha = 0.9$ and $\beta = 0.1$. This was to be expected, as this value of α most strongly emphasizes the test time. Columns 6 and 7 also show percentage (%) savings in SOC test time ($\triangle T$) and TAM wire length ($\triangle L$) respectively compared to the baseline approach.

From the table, it can be seen that for the original TR-ARCHITECT, a layout-driven ordering of cores alone itself shows the savings in TAM wire length ranging from 0 to 37% when compared to the random ordering of cores. However, the new layout-driven TR-ARCHITECT outperforms and shows savings up to 85%. For SOC d695, the savings in L vary from 16 to 65%. It is interesting to note here, that for $w_{\max} = 16$, the new layout-driven approach also improves the test time with 4%. Similarly, for SOC p22810, the test time is improved by 1.5% and 0.9% for $w_{\max} = 24$ and $w_{\max} = 48$ respectively. For SOC p22810, the savings in TAM wire length are 58 to 81%, at the cost of an increase of between 1.0 and 10% in the overall test time. Similarly for SOCs

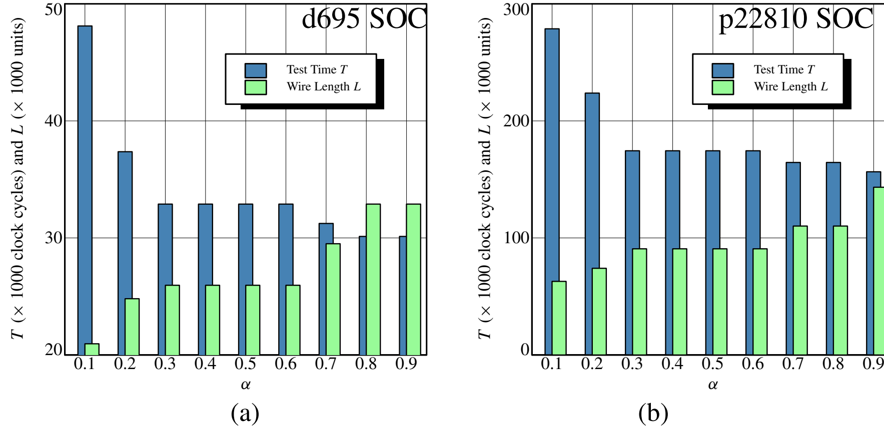


Figure 4.5: Variation in T and L with α for (a) SOC d695, $w_{\max} = 24$, and (b) SOC p22810, $w_{\max} = 48$.

p34392 and p93791, the savings in the TAM wire length vary from 47 to 80% and 48 to 85% respectively. The penalties in test time for these SOCs are in range of 2 to 6%. For SOC a586710, the savings in wire length are also comparable to the other SOCs, and are in the range of 21 to 61%.

For SOCs d695 with TAM width $w_{\max} = 24$, Figure 4.5(a) shows the variation in SOC test time T and TAM wire length L with parameters α and β. In the figure, the horizontal axis shows the value of α and the corresponding value of β can be calculated as $\beta = 1 - \alpha$. The vertical axis shows the value of test time and TAM wire length. Figure 4.5(b) shows the same but for SOC p22810 and with TAM width $w_{\max} = 48$. Both the figures confirm that both α and β indeed work as weighting factors between the two, often conflicting, cost factors.

For SOC p93791 with $w_{\max} = 32$, Figure 4.6(a) shows the used layout together with the cores assignments to TAMs as obtained by the original TR-ARCHITECT. For the same case, Figure 4.6(b) shows the cores assignments obtained by our new layout-driven TR-ARCHITECT with $\alpha = 0.7$ and $\beta = 0.3$. Cores with the same color and pattern are connected to the same TAM. The original version of TR-ARCHITECT divides the total TAM width over three TAMs as shown in Figure 4.6(a).

In this case, cores placed at different corners in the layout are assigned to the same TAM; for example cores 6, 20, 14, 29 32, 5, 10 and 22 are connected to the same TAM. The same is true for other TAMs also. This results in a long TAM wire length. The layout-driven version of TR-ARCHITECT generates a test architecture as shown in Figure 4.6(b). The total TAM width is partitioned over nine TAMs. It is clear from Figure 4.6(b), that usually only the cores that are physically close to each other are connected to the same TAM, which minimizes the TAM wire length. For this case, the new layout-driven TR-ARCHITECT obtained a saving of 85.6% in TAM wire length at an expense of 3.8% in the SOC test time.

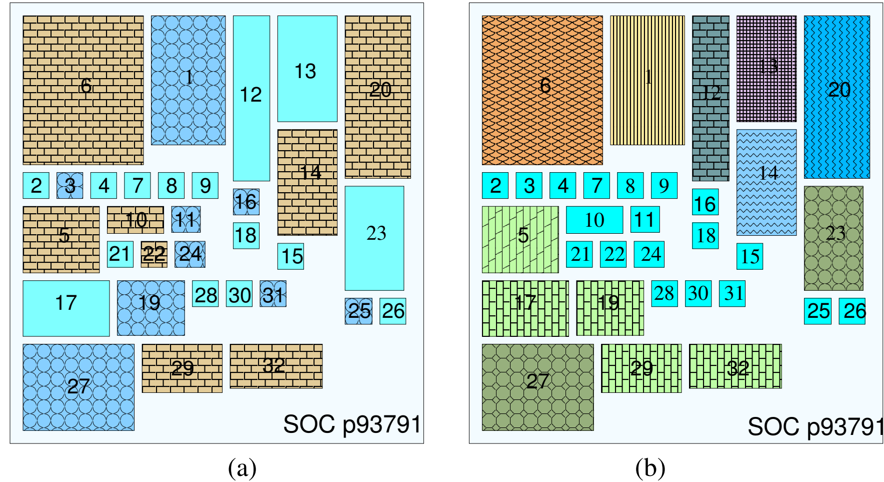


Figure 4.6: Cores assignments to TAMs for SOC p93791 with $w_{\max} = 32$, for (a) the original TR-ARCHITECT, and (b) the new layout-driven TR-ARCHITECT.

4.7 Summary

In this Chapter, an existing SOC test architecture design approach that minimizes SOC test time is extended with the capability to minimize the TAM wire length as well. To calculate the TAM wire length for a given architecture, a simple, yet effective TAM wire-length model was presented. In the presented wire-length model, the layout positions of all cores in the SOC layout were assumed to be given. The wire length of a TAM depends on the ordering of cores connected to the TAM. Therefore, to minimize the wire length for a TAM, an optimal ordering of cores had to be found. The problem of determining an optimal ordering of cores with respect to the wire length of the TAM was shown to be equivalent to the well-known Traveling Salesman Problem (TSP) and a heuristic algorithm was described to solve it.

Subsequently, it was shown that the minimization of test time and TAM wire length should be carried out in conjunction and a layout-driven test architecture design problem was formulated. For layout-driven test architecture design, a layout-driven version of TR-ARCHITECT was presented. The layout-driven TR-ARCHITECT uses a combined weight-based cost function that provides a trade-off between the two, often competing, cost factors SOC test time and TAM wire length. The layout-driven TR-ARCHITECT minimizes the total wire length by assigning neighboring cores as much as possible to the same TAM.

For five benchmark SOCs, results obtained from the original TR-ARCHITECT with both random and layout-driven ordering of cores per TAM and layout-driven TR-ARCHITECT were compared. For all SOCs and a range of $w_{\max}$ values, the layout-driven TR-ARCHITECT resulted in savings up to 85% in TAM wire length at the expense of less than 5% in test time.

Chapter 5

Control-Aware Test Architecture Design

5.1 Introduction

In a hybrid test architecture, distinct TAMs can operate independently and hence the cores connected to different TAMs can be tested in parallel. Depending on the TAM-type, the cores connected to a single TAM can be tested either in series or in parallel. Before starting a test of a core, the wrapper of the core needs to be configured in the corresponding test mode. Setting a particular test mode in the wrapper of a core requires a few pseudo-static test-control signals, while parallel test execution of cores connected to different TAMs requires at-least one *scan-enable* signal per TAM. Based on the nature of test-control signals, test control can be classified into two categories:

1. Pseudo-static test control.
2. Dynamic test control

Test-control signals can be generated on-chip, off-chip (by means of dedicated chip pins), or a combination of both (by means of shift-registers). Each of these generation types provides a trade-off between test time and area overhead. Pseudo-static test-control signals are often provided by means of a shift-register, where as dynamic test-control signals often require dedicated test pins. As the total number of chip pins available for test is limited, a large number of test-control pins results in less TAM bandwidth available for test-data transportation. Therefore, test architecture design should take the test control into account.

This chapter deals with *control-aware* test architecture design for SOCs. To deal with pseudo-static test control, two test strategies are presented and their impact on the SOC test schedule are discussed. For dynamic test-control, a pin-constrained design of test architectures is presented. Finally, experimental results for *ITC'02 SOC Test Benchmarks* are presented.

5.2 Prior Work

None of the test architecture design algorithms [Cha00a, EI01, SW02, ICM01, ICM02c, HRC+02, KI02, LAFP02, GM02b, GM02a, GM02c, GM03c] available in literature accounts for the signals and the time required to set up an appropriate test mode at the start of each core test. Also, all these methods seem to assume that a sufficiently large number of dedicated test-control pins required for scan-enable signals are available, on top of the constrained TAM width $w_{\max}$. As the total number of chip pins that can be used for test purposes is limited, this assumption is often not realistic.

The only paper, that has addressed the issue of test control for SOC test architectures is [AM98]. The paper considers that *not* the maximum number of TAM wires $w_{\max}$ is specified, but rather the total number of test pins K that can be used during test. The required number of test-control pins K_c are taken out from the total number of available test pins K. Hence, the maximum number of TAM wires $w_{\max}$ depends on both K and K_c following: $w_{\max} = \lfloor (K - K_c)/2 \rfloor$. The test time T can be reduced if the TAM width $w_{\max}$ is increased, and hence it is important to minimize the number of pins K_c spent on test control.

In [AM98], for test-mode settings, only one pin is used and the test modes are applied by means of a shift register. Furthermore, for each scan-enable signal, a dedicated pin is used. Cores that work with a common scan-enable pin, need to base their scan-in and scan-out times on the longest of all of their scan chains. This leads to a trade-off between the number of scan-enable pins and the number of TAM wires. Less scan-enable pins means more TAM wires; per two freed-up scan-enable pins, one can afford one additional TAM wire and hence increase $w_{\max}$ by one. In [AM98], the trade-off problem between test time and scan-enable pins was mapped to the *Interval Hitting Problem* (IHP) and a polynomial-time algorithm was presented to solve it. In the proposed algorithm, cores were allowed to share scan-enable pins based on the overlap between the feasible scan intervals for the cores.

While it is acknowledged that this paper was the first one to consider test control for an SOC test architecture, there are several limitations of the approach described in [AM98]. The paper only considers SOCs with all soft cores, while in practice SOCs usually contain all hard cores or a mix of hard and soft cores. The paper does not account for the time required for the test-mode settings in cores wrappers. The approach does not support optimization of TAMs and wrappers in conjunction. Only the basic architectures, i.e. the serial and parallel type test architectures are addressed in the paper, while in practice, a hybrid test architecture always performs better than corresponding serial and parallel test architectures. Nevertheless, the algorithm [AM98] based on *Interval Hitting Problem* (IHP) cannot be extended for hybrid test architectures.

5.3 Test-Control Classification

In an SOC test architecture, all TAMs are tested in parallel and as soon as a core completes its test in a TAM, it is put into the 'bypass' mode and another core is put into

the 'core internal test' mode. Therefore, the complete test schedule for an architecture consists of several test sessions and in each test session one or more cores are tested in different TAMs. Setting a particular test mode in the wrapper of a core and the execution of its test require a few test-control signals. Based on the nature of test-control signals, they can be classified into the following two categories:

- **Pseudo-static test control**
  Pseudo-static test-control signals are those signals which remain constant during one or more test sessions. Examples are test-mode setting signals in the wrapper of a core.

- **Dynamic test control**
  Dynamic test-control signals change very often during a test session and their change-over time, i.e. the time in which a signal should change its value, is generally very small. An example is a scan-enable signal which needs to be changed after every scan-in cycles. The change-over time for a scan-enable signal is equal to the number of apply/capture cycles and depends on the type of test. For example, the change-over time for a scan-enable signal is one clock cycle for a stuck-at-fault test.

Test control signals can be generated on-chip, off-chip (by means of dedicated chip pins), or a combination of both (by means of shift-registers). Each of these generation types provides a trade-off between test time and area overhead. In the next sections, the generation of both pseudo-static and dynamic test-control signals and their impact of test time and area overhead are discussed.

5.4 Pseudo-Static Test Control

To set a core into a particular test mode, the wrapper of the core needs to be configured according to the test mode. As a wrapper provides various test modes, setting a particular test mode requires a few pseudo-static test-control signals. Pseudo-static test-control signals cannot be generated on-chip as it is hard to determine the actual test-mode data beforehand. As the total number of pseudo-static signals for a test architecture is usually large, these should not be provided by means of dedicated chip pins and instead a shift register should be used. In the Philips TestShell [MAB+98], this shift register is called *Test Control Block* (TCB), while the proposed IEEE P1500 standard [HM, DZW+03] has a *Wrapper Instruction Register* (WIR) for this purpose. The wrapper architecture proposed in Chapter 2 also has a WIR for this purpose (see Section 2.3). The WIR consists of a shift and an update register.

At SOC level, these shift registers (WIRs) can be controlled either by the IEEE 1149.1 TAP [Soc00] controller or by simple pin connections. In this thesis, the use of the IEEE 1149.1 TAP controller is considered to control all WIRs. Therefore, a WIR can be considered as a data register connected between the TDI and TDO pins in the corresponding boundary-scan architecture [Soc00] as shown in Figure 5.1. To set a

particular test mode in the wrapper of a core, the corresponding WIR has to be selected between the TDI and the TDO pins in the TAP controller. This requires that one user instruction (e.g. PROGRAM_WIR) per WIR should be defined.

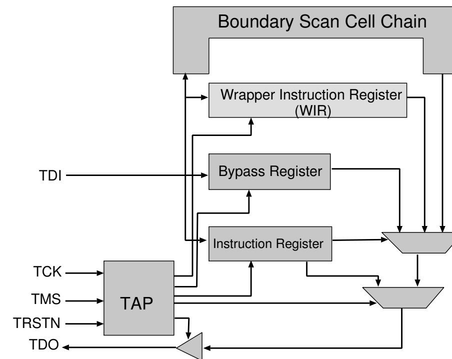
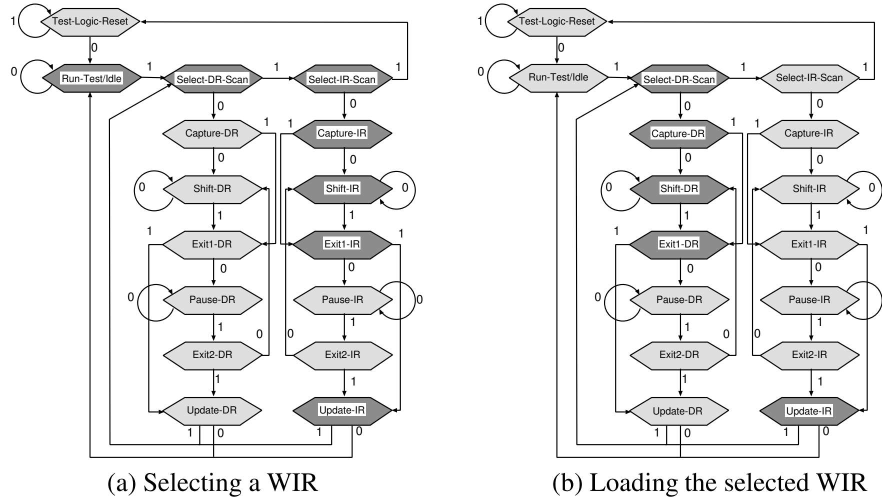


Figure 5.1: Boundary-scan architecture with a WIR as data register.

The time required to select a WIR and setting the corresponding core into its respective test mode is called *test-mode re-load time*. The test-mode re-load time depends on the WIR length L_{WIR} and the length L_{IR} of the instruction register. If the 'Run-Test/Idle' state is the default state in the TAP state diagram as shown in Figure 5.2(a), then it requires $6 + L_{\mathrm{IR}}$ clock cycles to select a WIR as a data register. The number of transitions between the dark-shaded states in Figure 5.2(a) is a representation of this. It is important to note that state 'Shift-IR' requires L_{IR} clock cycles to shift the corresponding instruction in the instruction register.



(a) Selecting a WIR (b) Loading the selected WIR

Figure 5.2: The IEEE 1149.1 TAP state diagram.

Once a data register (WIR) has been selected then loading of the required test-mode data into the selected data register (WIR) requires $(4 + L_{\mathrm{WIR}})$ clock cycles,

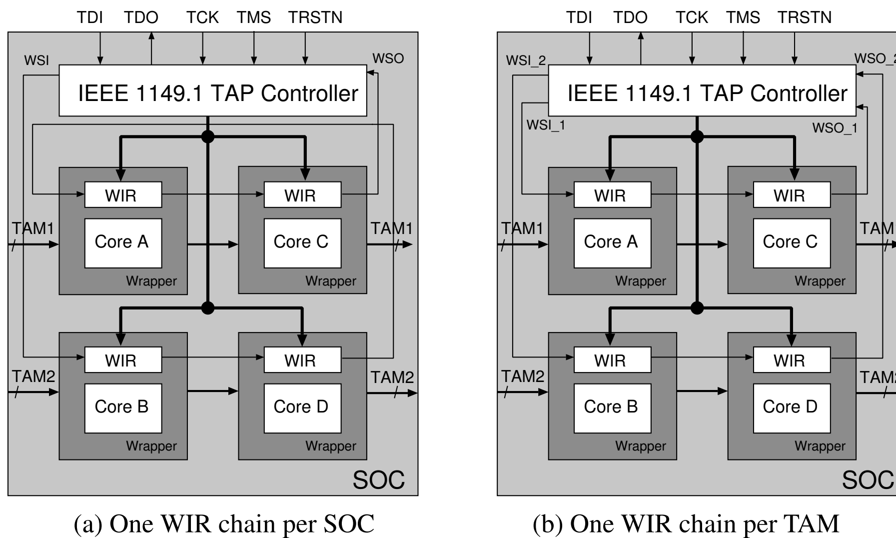
which is again shown by the number of transitions between the dark-shaded states in Figure 5.2(b).

Depending on the SOC test architecture, it is possible to make one or multiple WIR chains as separate boundary-scan data registers. However, it is not advisable to control each core WIR separately at the SOC level. This is due to the fact, that this will not only require a large instruction set, but also the time to select a single WIR will be substantial as compared to the time required to program the WIR. Therefore, there exists a trade-off between the number of WIR chains and the time required to select and program the wrappers of the cores in an architecture.

Here, the following two configurations are proposed for the WIRs connections at the SOC level:

1. One WIR chain per SOC.
2. One WIR chain per TAM.

In the first configuration, all the WIRs in an SOC are concatenated to form one WIR chain for the SOC. In the second configuration, for each TAM, only the WIRs for the cores connected to the same TAM are concatenated to form a WIR chain per TAM. Figure 5.3 shows the above-mentioned two configurations for an example SOC test architecture with two TAMs, each containing two cores.



(a) One WIR chain per SOC (b) One WIR chain per TAM

Figure 5.3: SOC-level WIRs connections for the two configurations.

For the configuration of one WIR chain per SOC as shown in Figure 5.3(a), all four WIRs in the wrappers of four cores are concatenated to form a WIR chain with WSI as input terminal and WSO as output terminal. In this case, the WIR chain needs to be selected only *once*, but for every change in the wrapper of a core, all WIRs in the chain need to be programmed. In contrast, for the configuration of one WIR chain per

TAM as shown in Figure 5.3(b), there are two WIR chains. In this case, every time one or more wrappers in a TAM need to be programmed, and only the corresponding WIR chain can be selected between the terminals TDI and TDO and programmed.

In the sequel of this section, the impact of the test-mode re-load time on the test schedule of a given test architecture is analyzed for both the configurations.

5.4.1 One WIR Chain per SOC

As described earlier, the complete test schedule for an architecture consists of several test sessions and in each test session one or more cores are tested in different TAMs.

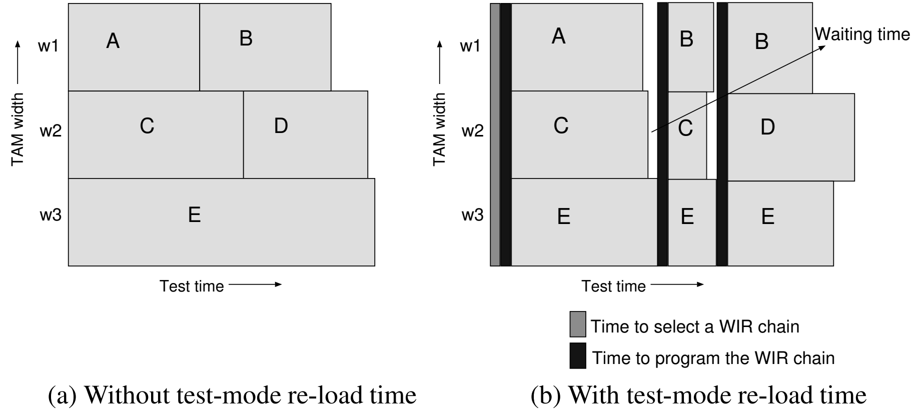
In this case as there is only one WIR chain, only one user instruction has to be defined and the WIR chain has to be selected only once for the complete schedule. However, the WIR chain needs to be programmed for every test session in the schedule. If $l_{\mathrm{WIR}}(c)$ represents the length of the WIR for core c, then the total length of the WIR chain L_{WIR} for an SOC test architecture with a set of cores C can be written as follows:

$$L_{\mathrm{WIR}} = \sum_{c \in C} l_{\mathrm{WIR}}(c) \tag{5.1}$$

therefore, the test-mode re-load time for the first test session will be $(6 + L_{\mathrm{IR}} + 4 + L_{\mathrm{WIR}})$ clock cycles and for every other test session, it will be $(4 + L_{\mathrm{WIR}})$ clock cycles. L_{IR} represents the length of the instruction register in the boundary-scan architecture of the SOC.

In this case, the test time of a test session is determined by the minimum of the test times of all cores that are being tested in that session. Due to the concatenation of all WIRs, programming of a WIR in a TAM cannot be carried out without disturbing the WIRs in other TAMs. Therefore, as soon as the core with the minimum test time finishes its test, the tests of other cores have to be pre-empted (halted) in order to program the WIR chain again. Whether the test of a core can be pre-empted or not is determined by the requirements for the integrity of the test. Here, it is assumed that the tests of all cores are pre-emptable. The test of a core should not be pre-empted in the middle of the test pattern, i.e. the test should be halted after completing both scan-in and scan-out cycles. As the cores in different TAMs can have different scan-in and scan-out times, tests of various cores will be halted at different points in time and this introduces a *waiting time* at different TAMs. To minimize this waiting time, cores in each TAM can be sorted on the basis of the maximum of scan-in and scan-out times.

Figure 5.4(a) shows an example test schedule without any test-mode re-load time. In Figure 5.4(a), there are three TAMs. TAM 1 with width w_1 contains cores A and B, TAM 2 with width w_1 contains cores C and D, while TAM 3 with width w_1 contains core E. In the figure, the horizontal axis represents the test time, while the vertical axis represents the TAM width. The overall test time of the test schedule shown in Figure 5.4(a) is determined by the test time of TAM 3. Figure 5.4(b) shows the same example test schedule but with test-mode re-load times. In Figure 5.4(b), first the WIR chain is selected and programmed, then the testing of cores A, C and E is carried out.



(a) Without test-mode re-load time (b) With test-mode re-load time

Figure 5.4: An example test schedule with one WIR chain per SOC configuration.

Core A has the minimum test time so in the first test session it finishes its test. After completion of the test of core A, testing of cores C and E is halted (pre-empted) in order to re-program the WIR chain. Due to the difference in the scan lengths, the tests of cores C and E are halted at different points in time. As a result of this, there is some waiting time at TAMs 1 and 2. Once all tests have been halted, the WIR chain is programmed again and the testing of cores is resumed. From Figure 5.4(b), one can see that the overall test time is not any longer determined by TAM 3 but is determined by the test time of TAM 2.

5.4.2 One WIR Chain per TAM

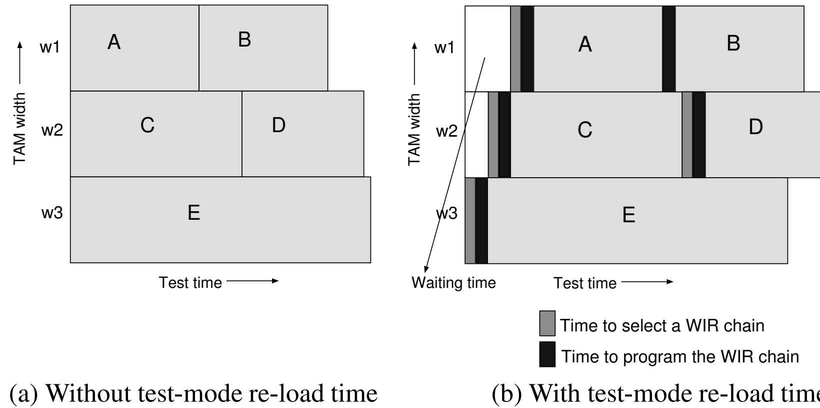
In this case, one user instruction per WIR chain (or per TAM) has to be defined. Test-mode re-load times for all WIR chains will primarily depend on their individual lengths and might be different. For a TAM r, the length of the corresponding WIR chain $L_{\mathrm{WIR}}(r)$ can be written as:

$$L_{\mathrm{WIR}}(r) = \sum_{c \in r} l_{\mathrm{WIR}}(c) \tag{5.2}$$

therefore, the test-mode re-load time for every test session in a TAM r is equal to $(10 + L_{\mathrm{IR}} + L_{\mathrm{WIR}}(r))$ clock cycles.

Unlike the previous case, here the cores tests do not require pre-emptions as the cores in different TAMs (WIR chains) can be programmed independently. However, if more than one WIR chain need to be programmed at the same time, the WIR chains are programmed sequentially. This is due to the fact that all WIR chains share the same TDI and TDO pins in the boundary-scan architecture. The sequential programming of WIR chains introduces a *waiting time* at the TAMs. In order to minimize the waiting time at TAMs, cores in TAMs can be ordered in such a way that a minimum number of cores are tested out at a time. Furthermore, the programming of WIR chains in the decreasing order of cores test times can also reduce the waiting time at TAMs.

Figure 5.5(a) shows the same example test schedule as shown in Figure 5.4(a). This schedule does not consider any test-mode re-load time. In Figure 5.5(a), there are three TAMs. TAM 1 with width w_1 contains cores A and B, TAM 2 with width w_1 contains cores C and D, while TAM 3 with width w_1 contains core E. In the figure, the horizontal axis represents the test time, while the vertical axis represents the TAM width.



(a) Without test-mode re-load time (b) With test-mode re-load time

Figure 5.5: An example test schedule with one WIR chain per TAM configuration.

Figure 5.5(b) shows the same example test schedule but with test-mode re-load times for one WIR chain per TAM configuration. In the first session, cores A, C and E need to be tested. As core E has the maximum test time, the WIR chain corresponding to its TAM is selected and programmed first. While the testing of core E is carried out, the WIR chains corresponding to the TAMs containing cores C and A are selected and programmed respectively. Core A has the minimum test time, so as soon as it has been tested, the WIR chain corresponding to its TAM is programmed (it was already selected during the last session). In a similar way, as soon as core C is tested out in TAM 2, its WIR chain is selected and programmed.

From Figure 5.5(b), one can see that if cores connected to different TAMs can be programmed independently, cores tests do not require pre-emptions. However, as only one WIR chain can be programmed at a time, a small amount of waiting time is introduced at TAMs 1 and 2. The overall test time is determined by the test time of TAM 2.

In the next section, the generation of dynamic test-control signals is discussed. As dynamic test-control signals, only scan-enable signals required for cores tests execution are being considered.

5.5 Dynamic Test Control

In an SOC test architecture, cores connected to a TAM can be tested either in series or in parallel. In case of serial testing of cores per TAM, only one scan-enable signal

per TAM is required as only one core is tested at a time. However, for parallel testing of cores per TAM also, only one scan-enable signal per TAM is required. This is due to the fact that in parallel testing of cores in a TAM, the test data is shifted-in and out from all cores serially and only the apply/capture cycle is applied to cores simultaneously [GM02c]. Hence, a common scan-enable signal can be used for all cores connected to the same TAM. Therefore, independent of the scheduling type per TAM, one needs to consider only one scan-enable signal per TAM. Based on this, the total number of scan-enable signals for an SOC test architecture is equal to the number of TAMs in the architecture.

The generation of scan-enable signals can be accomplished in three ways:

1. on-chip generation,
2. by means of a shift-register,
3. dedicated test pins for all signals.

Each of these types provides a trade-off in terms of test time and area overhead; it is discussed below.

5.5.1 On-Chip Generation

Figure 5.6(a) shows an example of a test protocol for a scan-testable core. In the figure, the horizontal axis represents the time (in clock cycles), while the vertical axis shows different types of signals. Here, it is assumed that the scan-out time for a test pattern can be pipelined with the scan-in time for the next pattern. The signal se in Figure 5.6(a) represents the scan-enable signal, while the signals TAM in and TAM out represent the TAM input and output. From this figure, one can see that the signal se has to change its value after every $\max(si, so)$ clock cycles. Here, si and so, represent the scan-in and the scan-out time (in clock cycles) for the core respectively.

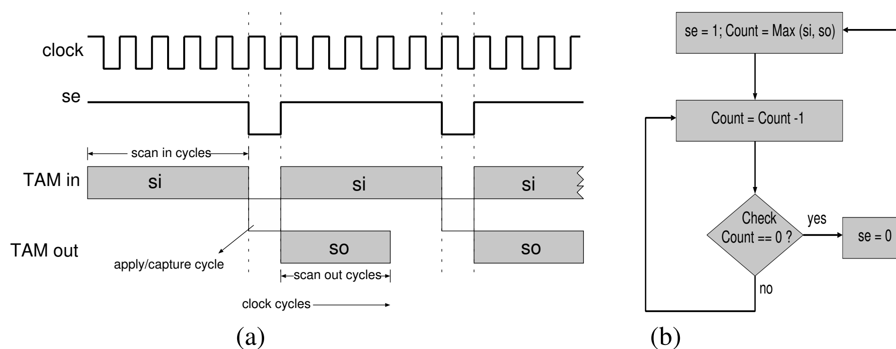


Figure 5.6: (a) An example of a scan-test protocol, and (b) flow-chart for generating a scan-enable signal.

Figure 5.6(b) shows a flow-chart for generating a scan-enable signal. From Figure 5.6(b), one can see that the generation of such a signal requires at least one counter and one comparator and may also require some additional glue logic. The length of the counter depends on the maximum of the scan-in and the scan-out times of the core for which the scan-enable signal is being generated.

The major advantage of on-chip generation of scan-enable signals is that no extra chip pins are required for them and hence, all available test pins can be used for SOC test time minimization. However, on-chip generation requires some extra silicon area. To minimize the silicon area overhead due to on-chip generation of test-control signals, hardware resources such as counters and comparators can be shared among various cores. Unfortunately, this can have a negative impact on the overall SOC test time. This is due to the fact that if the cores sharing the hardware resources are tested in parallel (on different TAMs), they will be tested at a common scan frequency; the frequency of the scan-enable signal will be determined by the maximum of the individual scan frequencies of the cores.

In practice, the on-chip generation of scan-enable signals is not preferred. This is due to the fact that it requires extra silicon area. Next, any design modification may require re-generation of the scan-enable signal and the corresponding hardware. Nevertheless, it provides less flexibility and freedom to test engineers.

5.5.2 Shift-Register Implementation

A scan-enable signal should not be provided by means of a shift register, otherwise for every change in the signal, a new value has to be loaded into the shift register. This may require a large amount of time. If the load time for the shift register is larger than the change-over time for the scan-enable signal, then during the load of new data into the shift register, the scan chains should keep their contents. This can be accomplished by providing a hold functionality into all flip-flops in the scan chains or by stopping all clocks. The idea of using hold functionality is expensive in terms of silicon area and may also create some problems in timing critical paths. Also, stopping all clocks may not be safe from the data invalidation point of view [GV02]. Therefore, it is not advisable to have scan-enable signals coming out from a shift register.

5.5.3 Dedicated Chip Pins

This is the most preferred way of applying scan-enable signals. In this case, a dedicated chip pin is used to provide a scan-enable signal. For a given SOC test architecture that requires K_c pins for scan-enable signals, there are two possibilities. In the first case, one can assume that the SOC designer can free up K_c chip pins as required for the scan-enable signals. In the second case, the required K_c pins are taken out from the existing $2 \times w_{\max}$ test pins used in the architecture.

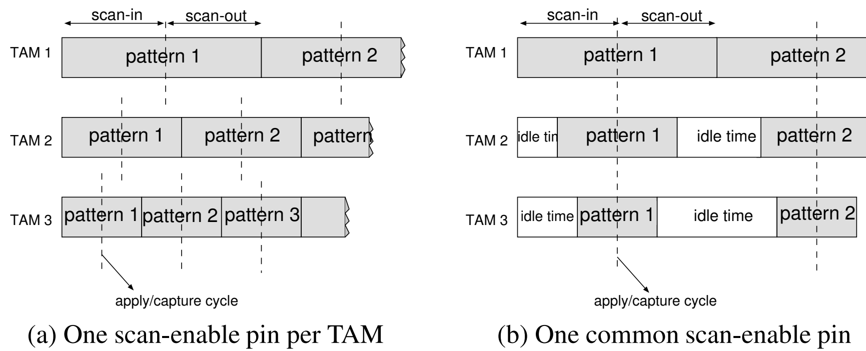
The first case is often not realistic due to the fact that it is very hard to find a large number of chip pins on the top of the given $2 \times w_{\max}$ test pins, where $w_{\max}$ is the

total TAM width used in the architecture. Therefore, in this chapter the second case is preferred and it is considered that for an SOC test architecture, the required number of scan-enable pins K_c is taken out from the total number of available test pins K. Therefore, the total number of TAM wires $w_{\max}$ available for test time optimization can be written as:

$$w_{\max} = \left\lfloor \frac{K - K_c}{2} \right\rfloor \tag{5.3}$$

To minimize the number of scan-enable pins K_c and hence minimize the overall test time by means of a large TAM bandwidth $w_{\max}$, scan-enable pins can be shared among various TAMs. However, sharing of scan-enable pins can also lead to an increase in the overall test time. This is due to the fact that the scan-in time for the cores connected to the TAMs sharing a scan-enable pin will be determined by the maximum of the scan-in times for the cores that are being tested in those TAMs. Similarly, the scan-out time for the cores will be determined by the maximum of the scan-out times for the cores that are being tested in those TAMs. This results in an increase in the test time of the TAMs that share the same scan-enable pin and might increase the overall test time T.

Consider an example test architecture with three TAMs, where each TAM contains one core only. For the case with one scan-enable pin per TAM, i.e. $K_c = 3$, Figure 5.7(a) shows the test-execution time for the cores in the three TAMs. In Figure 5.7(a), the horizontal axis represents the test time in clock cycles, while the vertical axis represents the TAM. For clarity of the figure, the pipelining between the scan-out time of a pattern and the scan-in time for the next pattern is not considered. However, the presented analysis is also valid for the case with pipelining.



(a) One scan-enable pin per TAM (b) One common scan-enable pin

Figure 5.7: Example test schedule showing relationship between test time and number of scan-enable pins.

Every scan-test pattern consists of three phases, being scan-in, apply/capture, and scan-out. In Figure 5.7(a), a rectangular box represents a test pattern and shows these three phases. The dotted vertical line in the middle of the box represents the execution of the apply/capture cycle. From Figure 5.7(a), one can see that due to a dedicated scan-enable signal for each TAM, the apply/capture cycle for a test pattern of the core in a TAM can be executed independently of the apply/capture cycles in other TAMs.

Figure 5.7(b) shows the same test execution but in the case, where all three TAMs are sharing the same scan-enable signal. In this case, the apply/capture cycles on the three TAMs should be aligned and all cores have to base their scan lengths on the maximum of the scan lengths of the cores that are being tested. This results in idle time [MG02] at various TAMs. From Figure 5.7(b), one can see that sharing a scan-enable pin results in a large amount of idle time per test pattern at TAM 2 and TAM 3. Based on the number of test patterns for the cores connected to TAMs 2 and 3, this can result in a large increase in the overall test time T.

In this chapter, an individual scan-enable pin for each TAM is considered. To minimize the number of scan-enable pins, a test architecture design procedure should take into account the pins required for scan-enable signals. The problem of test architecture design that takes into account the number of scan-enable pins can be formally defined as follows:

[PCTAD] **PIN CONSTRAINED TEST ARCHITECTURE DESIGN**

Instance: Given an SOC with a set of cores C. For each core $c \in C$, the number of test patterns p_c, the number of functional input terminals i_c, the number of functional output terminals o_c, the number of functional bidirectional terminals b_c, the number of scan chains s_c, and for each scan chain k, the length of the scan chain in flip flops $l_{c,k}$ are given. Furthermore, a number K is given that represents the maximum number of chip pins available for test architecture design.
Objective: Determine a test architecture with a set of TAMs R such that the overall test time T is minimized and the maximum number of TAM wires $w_{\max}$ used by the TAMs in the architecture does not exceed $\lfloor (K - |R|)/2 \rfloor$. □

To solve the PCTAD problem, a pin-constrained version of the test architecture design algorithm TR-ARCHITECT described in Chapter 3 is presented. The new pin-constrained version of TR-ARCHITECT uses the same five steps as in the original version: (1) CREATESTARTSOLUTION, (2) OPTIMIZE-BOTTOMUP, (3) OPTIMIZE-TOPDOWN, (4) RESHUFFLE, and (5) CHECK-EMPTYWIRE. Instead of repeating the pseudo-codes for the five steps here, only the modification details together with a brief summary about each individual step are given below.

In the step CREATESTARTSOLUTION, an initial test architecture is created which is then further optimized by the steps to follow. In this step, cores are assigned to one-bit wide TAMs based on their test-data volumes. Assignment of a core to a one-bit wide TAM requires three chip pins, one for scan-enable and two to form a one-bit wide TAM. Therefore if $|C| < \lfloor \frac{K}{3} \rfloor$, each core gets assigned, else only the first $\lfloor \frac{K}{3} \rfloor$ cores get assigned. The total number of unused pins K_{un} is equal to $(K - 3 \times |R|)$, where $|R|$ is the number of TAMs. If there are some unassigned cores left, cores are added iteratively to the TAM which results in a minimum overall test time. Similarly, if there are some unused pins K_{un}, TAM wires $w_{\text{free}} = \lfloor \frac{K_{\text{un}}}{2} \rfloor$ formed by these pins are added iteratively to the TAM with the maximum test time. This step returns the number of unused pins K_{un} and a set of TAMs R.

In the next two steps, being OPTIMIZEBOTTOMUP and OPTIMIZETOPDOWN, it is tried to merge the cores of two TAMs into one new TAM, such that the TAM wires that

are freed-up in the process can be utilized for an overall test time reduction. A merge of two TAMs can result in free TAM wires because of the following two reasons:

1. a merged TAM might require less wires than the sum of the wires of the merging candidate TAMs,

2. as a TAM requires only one scan-enable pin, a pin is also freed-up as a result of a merge. If there is already an unused pin then by using the freed-up pin, a TAM wire can be formed.

In the new pin-constrained version, it is important to note that if a merge of two TAMs does not show any improvement in the overall test time T, the merge is still accepted. This is due to the fact that a merge will always result in a free test pin that can be used in the subsequent steps to form a TAM wire. These two steps also return the number of unused pins K_{un} and a set of TAMs R.

In the step RESHUFFLE, an individual core is moved from the TAM with the largest test time to another TAM, if and only if this decreases the overall test time. As compared to the basic RESHUFFLE step described in Section 3.6.4 (Chapter 3), there are no major changes in the new version.

In the last step CHECK-EMPTYWIRE, the number of redundant wires (empty wires) are searched for all TAMs. If any empty wire is found, then it is tried to assign it to the TAM with the maximum test time in order to minimize the overall test time. Similar to the RESHUFFLE step, there are no major changes in the new version of this step as compared to the basic CHECK-EMPTYWIRE step described in Section 3.6.5 (Chapter 3).

5.6 Experimental Results

In this section, experimental results for four of the twelve *ITC'02 benchmarks SOCs* [MIC] are presented. These SOCs were selected, as they are the only ones with a large number of cores and for which the test time continues to decrease for increasing values of TAM width up to $w_{\mathrm{max}} = 64$. In the experiments, it has been assumed that an SOC only contains one level of hierarchy, viz. (1) the SOC itself and (2) all its embedded cores, even though some of these SOCs originally contain multiple levels of design hierarchy. Also, only the core-internal tests of the SOCs have been considered i.e. the interconnect tests for the top-level SOC itself have not been taken into account. As all the SOCs in the benchmark set have a fixed number of scan chains and lengths, only the test time results considering all hard cores are presented. Here, only the test time results for hybrid test-bus architectures are presented; however similar results could be obtained for hybrid TestRail architectures also.

For the dynamic test control, a comparison of the test time results for the following three cases is presented.

Table 5.1: For dynamic test control, test time results for the considered cases.

SOC	Test Pins K	Dynamic Test Control					
		Original TR-Architect				Pin-Constrained TR-Architect (Case 3)	
		No Test Control (Case 1)	Only One SE (Case 2a)	One SE Per TAM (Case 2b)			
		T	T	K_c	T	K_c	T
d695	32	44307	86859	3	60128	2	48216
	48	28576	48275	3	53381	4	32218
	64	21518	37571	3	25578	4	23275
	80	17617	22227	3	22512	4	19094
	96	14608	27562	5	27913	4	15761
	112	12462	20209	5	20778	5	13652
	128	11033	24692	6	12199	5	11420
p22810	32	457433	653559	2	576979	2	506147
	48	302737	645901	4	398403	4	351559
	64	222471	626917	6	292232	4	250862
	80	190995	507242	6	228117	6	208667
	96	157851	510854	7	217241	5	170893
	112	145417	423974	7	177868	6	145417
	128	133405	363376	7	189287	6	138927
p34392	32	1010821	1921513	4	1442956	4	1174702
	48	663193	1191443	4	967132	3	832498
	64	584524	1276404	4	622902	4	594231
	80	544579	908899	4	544579	4	544579
	96	544579	908899	4	544579	4	544579
	112	544579	908899	4	544579	4	544579
	128	544579	908899	4	544579	4	544579
p93791	32	1791638	3899852	3	2304109	2	1949321
	48	1185434	2146012	4	1480950	2	1288461
	64	912158	1527645	3	1083384	3	990373
	80	718005	1792717	6	954863	3	798940
	96	601450	1280548	6	737132	3	690709
	112	528925	958681	4	605243	3	567009
	128	455738	1076121	7	584806	4	516237

Case 1 is the original TR-ARCHITECT as described in Chapter 3, which does not take any test control into account. In this case, the complete TAM bandwidth is used to minimize the test time and therefore, this case has the lowest test time.

Case 2 is again the original TR-ARCHITECT but followed by the dynamic test control strategies presented in this chapter. The original TR-ARCHITECT does not consider the test pins required for scan-enable signals and works with a given TAM width $w_{\max}$. Therefore to account for scan-enable pins, either the TAM wires have to be stripped-off from the TAMs in order to generate the required number of scan-enable pins or one can assume that the SOC designer can still free-up some chip pins. As for each $w_{\max}$, TR-ARCHITECT results in a different number of TAMs, it is not practical to assume that the designer can free up as many chip pins as the number of TAMs. Therefore, for this case, two possibilities are considered. In Case 2a, it is considered that at most one pin can be freed-up on the top of the given $(2 \times w_{\max})$ TAM pins, and hence one common scan-enable is used for all cores in the architecture. In Case 2b, it is considered that wires are stripped-off from the TAMs in order to generate one scan-enable pin per TAM.

Case 3 is the new pin-constrained version of TR-ARCHITECT. In this case, one scan-enable signal per TAM is already taken into account while designing the test architecture.

Table 5.1 presents the test time (in number of clock cycles) results for the above-mentioned cases. In the table, Column 2 lists the total number of test pins K. Column 3 presents the test time T for Case 1, i.e. the original TR-ARCHITECT without any test control. Column 4 presents the test time for the original TR-ARCHITECT with only one scan-enable (SE) pin for all TAMs (Case 2a). Columns 5 and 6 present the number of scan-enable pins K_c and the test time results respectively for Case 2b, which is the original TR-ARCHITECT with one scan-enable pin per TAM. Column 7 and 8 present the same results for the new pin-constrained TR-ARCHITECT. It is important to note that the test time results presented in Table 5.1 are with dynamic test control only and do not include test mode re-load times (pseudo-static test control).

From Table 5.1, one can see that for the original TR-ARCHITECT, as compared to the test schedules without considering the scan-enable signals for TAMs (Case 1), there is a large increase in the test time if one or more scan-enable pins are considered (Cases 2a and 2b). For Case 2a in which only one scan-enable pin is used, test time explodes due to a common scan frequency. The test times shown in Column 5 for Case 2b in which TAM wires are stripped-off to generate one scan control pin per TAM are slightly better. However, the new pin-constrained TR-ARCHITECT performs very good and gives test times closer to the original TR-ARCHITECT test times as shown in Column 3.

From the test time results shown in the table, one can conclude that a scan-enable signal should not be shared among all TAMs in an architecture and instead one scan-enable signal per TAM should be used. Furthermore, while designing an architecture, taking the chip pins required for the scan-enable signals into account helps in getting good test times. By comparing the test time results shown in Columns 3 and 8, one can also conclude that if the dynamic test control is taken into account, on an average an 8.5% penalty is paid in the overall test time T.

Next, the impact of pseudo-static test control is analyzed for the cases with one scan-enable signal per TAM, i.e. Cases 2b and 3. For pseudo-static test control, both the strategies presented in Section 5.4, i.e. one WIR chain per SOC and one WIR chain per TAM, are considered. Here, a seven-bit long ($l_{\text{WIR}} = 7$) WIR for every core in all SOCs and a six-bit long ($L_{\text{IR}} = 6$) instruction register are considered. The seven-bit long WIR represents a typical WIR as described by the IEEE P1500 standard [DZW+03], while the six-bit long instruction register ensures that sufficient instructions ($2^6 = 64$) can be used. Table 5.2 presents the modified test time (in number of clock cycles) results considering the pseudo-static test control.

Column 3 in Table 5.2 shows the test time results for the original TR-ARCHITECT (Case 2b) with one WIR chain per SOC, while Column 6 shows the same but now with one WIR chain per TAM. Column 4 shows the test time results for the new pin-constrained version of TR-ARCHITECT (Case 3) with one WIR chain per SOC, while Column 7 shows the same but with one WIR chain per TAM. Columns 5 and 8 show the savings (%) in test time obtained from the new pin-constrained TR-ARCHITECT as compared to the original TR-ARCHITECT under the same test control settings.

From Table 5.2, one can see that if the pseudo-static test control is considered, the overall test time is increased. This was to be expected, as a few clock cycles are

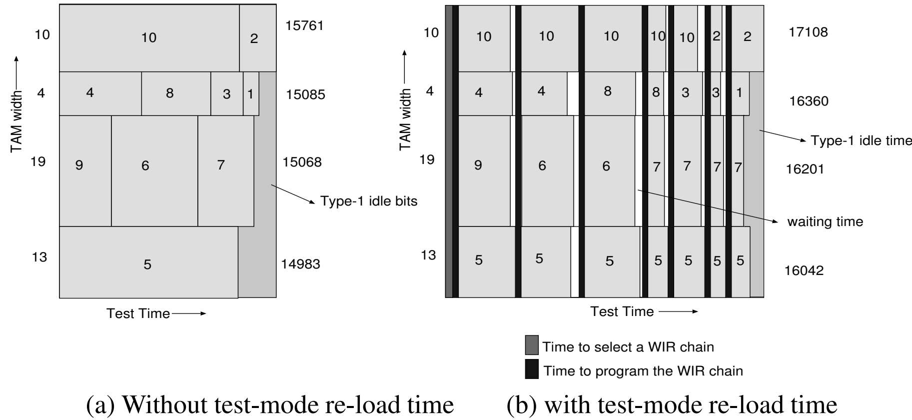
Table 5.2: For pseudo-static test control, the modified test time results considering one scan-enable per TAM.

SOC	Test Pins K	Pseudo-Static Test Control					
		One WIR Chain Per SOC			One WIR Chain Per TAM		
		Original TR-Architect (Case 2b)	Pin-Constrained TR-Architect (Case 3)		Original TR-Architect (Case 2b)	Pin-Constrained TR-Architect (Case 3)	
		T	T	$\triangle T(\%)$	T	T	$\triangle T(\%)$
d695	32	67701	49259	27.2	60150	48266	19.8
	48	62528	33221	46.9	53417	32247	39.6
	64	26560	24170	9	25635	23304	9.1
	80	23345	20229	13.3	22555	19123	15.2
	96	29603	17108	42.2	27949	15790	43.5
	112	21730	14230	34.5	20814	13681	34.3
	128	12795	12280	4.0	12286	11442	6.9
p22810	32	593278	510693	13.9	577071	507238	12.1
	48	407029	384493	5.5	398467	351644	11.8
	64	316604	275037	13.1	292282	250891	14.2
	80	245494	237232	3.4	228146	208773	8.5
	96	236894	181799	23.3	217284	170936	21.3
	112	194957	167584	14.0	177897	145439	18.3
	128	200667	155345	22.6	189323	139019	26.6
p34392	32	1453953	1216373	16.3	1443069	1174724	18.6
	48	993060	843673	15.0	967182	832527	13.9
	64	628853	607466	3.4	623015	594260	4.6
	80	558341	558341	0	544601	544601	0
	96	558341	558341	0	544601	544601	0
	112	558341	558341	0	544601	544601	0
	128	558341	558341	0	544601	544601	0
p93791	32	2352311	1957867	16.8	2304152	1949462	15.4
	48	1546032	1288926	16.6	1480986	1289318	12.9
	64	1102145	1018563	7.6	1083455	990409	8.6
	80	1036648	858032	17.2	954885	798962	16.3
	96	781491	707276	9.5	737161	690766	6.3
	112	631883	576136	8.8	605272	567052	6.3
	128	635183	539643	15.0	584835	516266	11.7

required for test-mode re-load time and pre-emptions of cores tests or serial programming of WIR chains introduce some waiting time at TAMs. The test time results with one WIR chain per TAM are better than those with one WIR chain per SOC. From Table 5.2, one can observe that the new pin-constrained TR-ARCHITECT performs well and results in savings up-to 46% in test time if compared to the original TR-ARCHITECT with the same test-control settings.

For SOC d695 and K = 96, Figure 5.8(a) shows the test schedule with one scan-enable per TAM and without test-mode re-load time, as obtained by the new pin-constrained TR-ARCHITECT. In the figure, the horizontal axis represents the test time, while the vertical axis represents the TAM width. The numbered boxes depict tests of various cores and the number inside a box represents the core ID. The number shown at the end of each TAM represents its test time. In the shown schedule, there are four TAMs and hence four pins are required for scan-enable signals. The remaining 92 pins result in a total TAM width $w_{\max} = 46$, which is distributed among four TAMs in the following fashion: $10 + 4 + 19 + 13 = 46$. The total test time for the schedule is determined by TAM 1 and is equal to 15761 clock cycles.

Figure 5.8(b) shows the same schedule but with test-mode re-load times using one



(a) Without test-mode re-load time (b) with test-mode re-load time

Figure 5.8: Test schedules obtained from the pin-constrained TR-ARCHITECT for SOC d695 with K = 96.

WIR chain per SOC strategy. From this figure, one can see that every time a core finishes its test, tests of all cores are halted and the complete WIR chain is re-programmed. This leads to some waiting time at TAMs (indicated by white space) and an increase in the overall test time. For the shown case, there is an 8.5% increase in the overall test time T as compared to the test schedule shown in Figure 5.8(a).

5.7 Summary

In this chapter, the test control for SOC test architectures is described. The term test control refers to controlling the mode of operation of all cores and the execution of their tests. Based on the nature of test-control signals, it is shown that test control can be divided into two categories: (1) pseudo-static test control, and (2) dynamic test control. Test-control signals can be generated in three ways: (1) by on-chip hardware, (2) by means of a shift-register, and (3) by dedicated chip pins. Each of these types provides a trade-off between the test time and the area-overhead.

Pseudo-static test control refers to setting test modes in the cores wrappers between various test sessions. As pseudo-static signals do not change very often, shift-register based generation of test-control signals is preferred for them. To account for the time required for test mode re-loads, two test strategies are presented: (1) one WIR chain per SOC, and (2) one WIR chain per TAM. Each of these strategies provides a trade-off between the time to select a WIR chain and programming the WIR chain.

Dynamic test control refers to controlling the execution of cores tests, i.e. scan-enable control. As scan-enable signals change very often and the change-over time for them is very small, they cannot be provided by means of a shift-register and instead dedicated chip pins are preferred. As there is a limited number of chip pins available for test, a large number of scan-enable pins will result in less TAM bandwidth available for test time minimization. Therefore, SOC test architecture design should take into

account the pins required for scan-enable signals. For this purpose, a pin-constrained version of TR-ARCHITECT is presented.

Experimental results for the ITC02 benchmark SOCs show that as compared to the test times obtained from the original TR-ARCHITECT, if the dynamic test control (scan-enable signals) is taken into account, there is a large increase in test time. It is shown that sharing a scan-enable signal among TAMs is not a good idea and instead, one scan-enable signal per TAM should be used. Similarly if the pseudo-static test control (test-mode re-load time) is considered during test scheduling, it also increases the overall test time. For pseudo-static test control, one WIR chain per TAM strategy works better than one WIR chain per SOC strategy. The presented new pin-constrained version of TR-ARCHITECT performs very good and can save up to 46% in test time if compared to the test times obtained from the original TR-ARCHITECT with the same test control settings.

Chapter 6

User-Constrained Test Architecture Design

6.1 Introduction

All test architecture design algorithms published so far, compute an optimized SOC test architecture fully autonomously. This can lead to situations where test architectures proposed by the optimization procedure might not be acceptable to SOC designers due to some design constraints which were either not modeled or very hard to model directly in the optimization procedure. For example, a wrapped third-party IP core requires a specific TAM width; analog and digital cores or cores running at different clock frequencies need to be assigned to separate TAMs; some test architecture design constraints are inherited from a previous SOC design. Ideally, there exists a whole spectrum of user constraints, in which at one extreme is the fully specified architecture by the user and at the other extreme the fully designed architecture by the optimization procedure. User constraints might be application- or design-specific and hence hard to generalize into general cost functions or constraints.

Therefore, SOC designers and test engineers do not always want to leave their entire SOC-level test architecture design to an optimization tool. They often want to influence the number of TAMs, the width of the TAMs, the assignment of cores to TAMs, and/or the cores ordering within a TAM. As the SOC designers have the ultimate decision power over *their* SOC, a test architecture optimization tool can only become successful in the SOC designer community if it is able to satisfy their constraints. Currently, there are no test architecture design procedures available in literature which can take user constraints into account and design the test architecture accordingly.

This chapter presents a novel Test Architecture Specification (TAS) language that can be used to specify a full or partial test architecture in a concise way. The usage of the TAS language is illustrated by means of an example. In addition, it is described how the original version of TR-ARCHITECT has been modified and extended in order to accommodate a wide range of user constraints. All architecture parameters specified

by the user in an input TAS file to TR-ARCHITECT are considered as constraints, while everything which is not specified, is left for the tool to optimize for minimal test time, TAM wire length, etc. This extension yields a whole spectrum of use scenarios for TR-ARCHITECT. The spectrum ranges from an empty user specification (in which the tool fully optimizes the resulting test architecture) to a full-user specification (in which the tool only calculates the corresponding test schedule and associated costs), and everything in between. Finally, the operation of the modified TR-ARCHITECT is illustrated by means of an example and experimental results for the *ITC'02 SOC TestBenchmarks* are presented.

6.2 Prior Work

Various test architecture design algorithms have been described in literature [Cha00b, ICM01, HCT+01, ICM02b, ICM02c, ICM02d, GM02b, GM02c, KI02, LF03, GM03d]. For a given SOC with a given TAM width $w_{\max}$, these algorithms determine the number of TAMs, the widths of the TAMs, and the assignment of cores to TAMs such that the total SOC test time is minimized. Some algorithms, in addition to test time minimization, also allow for minimization of the number of TAM wires or their lengths [GM03c], or take additional constraints into account, such as the maximum power dissipation during testing [HRC+02], or test control [GM03b]. It important to note that in hindsight, some of the these papers [Cha00b, ICM01, EI01] also provide a partial solution to the issue addressed in this chapter, as they correspond to designing a test-access architecture in the case of a user-specified number of TAMs.

However, the methods described in these papers have the following shortcomings. These methods only allow for one fixed number of TAMs, and do not allow the user to specify a minimum and/or maximum number of TAMs. Next, these methods only work if all TAM widths are fully specified, and do not allow for only some specified TAM widths, or minimum and/or maximum TAM widths. In addition, these methods only work for TAMs of the type *test-bus* [VB98] (which only allows mutually exclusive access to cores in the same TAM) and *hard* cores (i.e. cores for which the number and length of internal scan chains are fixed). In practice, one also need to work with TAMs of the type *TestRail* [MAB+98], which do not constrain access to multiple cores in the same TAM to be mutually exclusive, and a mix of hard and soft cores [GM03d]. Furthermore, these methods lack the ability to accommodate user-specified core-to-TAM assignments and ordering constraints.

6.3 Test Architecture Specification

To model user constraints in a concise and un-ambiguous way, a Test Architecture Specification (TAS) language is defined in this section. With such a language, a user should be able to fully control all parameters that constitute a test architecture. Hence such a specification should be able to describe the following elements:

- A fixed number of TAMs, or a minimum and/or maximum number of TAMs
- For each user-specified TAM in the architecture:
  - A fixed TAM width, or a minimum and/or maximum TAM width
  - A maximum number of cores to be assigned to this TAM
  - A list of user-assigned cores
  - A full or partial order of the user-assigned cores
- For remaining (i.e. unassigned) cores:
  - A list of TAMs to which this core is allowed to be assigned.

The proposed Test Architecture Specification language can be used both to specify user constraints for test architecture design (i.e. as input to a test architecture design tool such as TR-ARCHITECT), as well as to describe an optimized architecture (i.e. as output from the tool). This section describes the TAS language in detail. First, all keywords of the TAS language are described. Next, the usage of the TAS language is illustrated by means of an example.

6.3.1 Keywords

The TAS language has the following keywords:

- `SocName`: Unique identifier of the SOC
- `TotalTAMs`: The total number of TAMs in the architecture. The user can either specify a fixed number, or one can specify a minimum and/or a maximum bound on the number of TAMs. The two bounds are separated by a hyphen (" – "). If a user specifies a fixed number of TAMs or a minimum bound on the number of TAMs, then one has to specify those TAMs subsequently. However, if a user only specifies an maximum bound on the number of TAMs, then one does not need to specify any TAM. Please note that these assumptions are just to avoid any ambiguity in the specification.

Per TAM, the following information is specified:

- `TAM`: Uniquely identifying TAM name. This can help a test architecture design algorithm to keep this TAM separate from other TAMs
- `Width`: A user can either specify a fixed TAM width, or a minimum and/or a maximum bound on the TAM width. The two bounds are separated by a hyphen (" – ")
- `MaxCores`: The maximum number of cores that are allowed to be connected to this TAM

- `FixCores`: A list (c_1, c_2, ..., c_n) of user-assigned cores to the TAM. Core names in the list are separated by commas

- `Order`: Order (c_1 - c_2 - ...- c_k) of the cores connected to the TAM. Core names in the list are separated by hyphens. A user can specify a partial or full ordering of cores. The Kleene [Koz90] star operator ($*$) is used to denote zero or more unspecified cores in the order list; similarly, the Kleene [Koz90] plus operator ($+$) is used to denote one or more unspecified cores.

For each of the remaining (i.e. unassigned) cores in the SOC, the following information can optionally be specified:

- `Core`: Uniquely identifying core name

- `FlexTAMs`: ru_1, ru_2, ..., ru_k: A list of TAMs to which this core is allowed to be assigned. The TAMs in the list are separated by commas. If a user does not want to assign a core to the user-specified TAMs, but instead to a TAM created by the optimization algorithm, then one can use a reserved TAM name `rx`. Hence, this name should *not* be used to identify a user-specified TAM.

The TAS language allows for white spaces and blank lines. Furthermore, text following '`//`' on a line is considered as comment.

6.3.2 Example

Figure 6.1 illustrates the usage of the TAS language by means of an architecture example for SOC p22810, taken from the set of *ITC'02 SOC Test Benchmarks* [MIC02]. Please note that the line numbers in the example are not part of the TAS language, but are added here for explanation purposes only.

Example 1 [p22810.tas]

```
1  SocName p22810

2  // TAM Specification
3  TotalTAMs 3-5
4  TAM ru1 Width 4-  MaxCores 3 FixCores :  18    Order :  18-+
5  TAM ru2 Width 1-9 MaxCores 4 FixCores :  4,5   Order :  *-5-*-4-*
6  TAM ru3 Width 5-  MaxCores 5 FixCores :  6,7,9 Order :  *-9-7-6-*

7  // Core Specification
8  Core 2 FlexTAMs :  ru2,ru3,rx
```

Figure 6.1: Example TAS file for SOC p22810.

The first line in the TAS example identifies the SOC, viz. SOC p22810. Line 3 specifies the total number of TAMs in the architecture. In this example, the minimum number of TAMs is specified as 3, while the maximum number of TAMs is specified as 5.

The next three lines (Lines 4–6) specify three TAMs, corresponding to the minimum number of three TAMs specified in Line 3. Line 4 specifies TAM $ru1$. Its minimum specified width is 4; no maximum width is specified. The maximum number of cores that are allowed to connect to this TAM is set to 3. Core 18 is specified to be the first core in this TAM. During test architecture design, the used architecture design algorithm has to add one or more cores (as specified by the Kleene [Koz90] plus (+) operator) after core 18, as long as TAM $ru1$ in total does not have more than three cores (as specified by the `MaxCores` constraint).

TAM $ru2$ is specified in Line 5. Its width is specified to be between 1 and 9. TAM $ru2$ can contain at most four cores, and cores 4 and 5 are amongst those. The specified order of the TAM states that core 5 comes in front of core 4, while additional cores (if any) can be placed anywhere in front, in between, and after those two cores. Line 6 gives a similar specification for TAM $ru3$.

In total, only six cores are pre-assigned to the three user-defined TAMs $ru1$, $ru2$, and $ru3$. From [MIC02], it is known that SOC p22810 contains 28 cores (in the benchmark set, cores are referred to as modules). Therefore, the assignment of the remaining 22 cores is left to the test architecture design algorithm, within the constraints specified in Lines 8–9. Line 8 states that core 2 is only allowed to be assigned to either TAM $ru2$ or TAM $ru3$ or TAM rx, which represents a TAM created by the test architecture design algorithm itself.

6.4 Test Architecture Design

The problem of test architecture design that takes the user's constraints (specified by means of a TAS file) into account can be defined as follows:

[UCTAD] **USER-CONSTRAINED TEST ARCHITECTURE DESIGN**

Instance: Given is an SOC with a set of cores C. For each core $c \in C$, the number of test patterns p_c, the number of functional input terminals i_c, the number of functional output terminals o_c, the number of functional bidirectional terminals b_c, the number of scan chains s_c, and for each scan chain k, the length of the scan chain in flip flops $l_{c,k}$ are given. Furthermore, a TAS file that contains user constraints, and a number $w_{\max}$ that represents the maximum number of SOC-level TAM wires, are given.
Objective: Determine a test architecture with a set of TAMs R such that (i) the sum of individual TAM widths does not exceed $w_{\max}$, (ii) all user constraints are satisfied, and (iii) the overall SOC test time T (in clock cycles) is minimized. □

TR-ARCHITECT proved effective in minimizing the overall test time of SOCs. The original version of TR-ARCHITECT (presented in Chapter 3) was unaware of user constraints. To solve the UCTAD problem, a new user-constrained version of TR-ARCHITECT is presented here. This version addresses all user constraints that can be expressed in the TAS language.

Figure 6.2 shows the input and output files used in the user-constrained version of TR-ARCHITECT. There are three inputs to the tool. The user options contain informa-

tion such as top-level TAM width $w_{\max}$, TAM type, architecture type, etc. The SOC data file contains the information about the SOC, coded in ".soc" format introduced for the *ITC'02 SOC Test Benchmarks* [MIC02]. New is the (optional) input TAS file, which contains the user constraints in the TAS language.

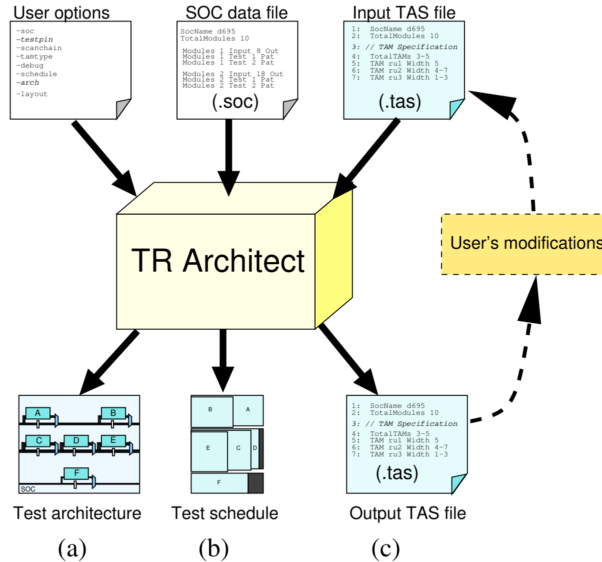


Figure 6.2: User-constrained TR-ARCHITECT version.

TR-ARCHITECT yields (a) a test architecture which is optimized within the bounds of the user constraints, (b) the optimal test schedule for this architecture, and (c) the corresponding new TAS file. If for some reason the user does not like the proposed architecture, the user can take the output TAS file, modify it, and feed it back again for a new TR-ARCHITECT run.

Apart from the ordering constraints, all constraints specified in an input TAS file can be divided into three distinct domains. This division is specially useful for including user constraints in the various optimization steps of TR-ARCHITECT. These three domains along with the user constraints they contain, are described below.

6.4.1 User Constraints

In the user-constrained version of TR-ARCHITECT, for a TAM r, $w(r)$ denotes its width of r, $C(r)$ denotes the set of cores connected to it, and $t(r)$ denotes its test time. For the architecture parameters specified in the TAS file, the following symbols are used.

- R_{user} denotes the set of user-specified TAMs
- R_L and R_U denote the minimum and maximum bounds on the number of TAMs in the architecture

- For each TAM $ru \in R_{\text{user}}$,
  - $w_L(ru)$ and $w_U(ru)$ denote the minimum and maximum bounds on its width,
  - $c_{\max}(ru)$ denotes the maximum number of cores that can be assigned to it,
  - $C_{\text{user}}(ru)$ denotes the set of cores specified by the user to be assigned to it.

Apart from the ordering constraints, the user constraints in the TAS input file can be categorized into 'minimum' and 'maximum' constraints in the following three domains.

1. **Number of TAMs**

$$\text{Constraint } 1_{\min}: \quad R_L \leq |R| \text{ and } R_{\text{user}} \subseteq R \tag{6.1a}$$

$$\text{Constraint } 1_{\max}: \quad |R| \leq R_U \tag{6.1b}$$

Constraint $1_{\min}$ means that the total number of TAMs $|R|$ in the architecture should be more than or equal to the minimum bound R_L specified with regard to the number of TAMs. Furthermore, the TAMs in the architecture (set R) should include the TAMs specified by the user (set R_{user}). Constraint $1_{\max}$ means that the total number of TAMs $|R|$ in the architecture should be less than or equal to the maximum bound R_U specified with regard to the number of TAMs the number of TAMs.

2. **TAM widths**

$$\text{Constraint } 2_{\min}: \quad w_L(ru) \leq w(ru), \;\; \forall\, ru \in R_{\text{user}} \tag{6.2a}$$

$$\text{Constraint } 2_{\max}: \quad w(ru) \leq w_U(ru), \;\; \forall\, ru \in R_{\text{user}} \tag{6.2b}$$

Constraint $2_{\min}$ means that for all user-specified TAMs, width $w(ru)$ of a TAM ru should be more than or equal to the minimum bound w_L specified on its width. Similarly, Constraint $2_{\max}$ means that for all user-specified TAMs, width $w(ru)$ of a TAM ru should be less than or equal to the maximum bound w_U specified on its width.

3. **Cores in TAM**

$$\text{Constraint } 3_{\min}: \quad C_{\text{user}}(ru) \subseteq C(ru), \;\; \forall\, ru \in R_{\text{user}} \tag{6.3a}$$

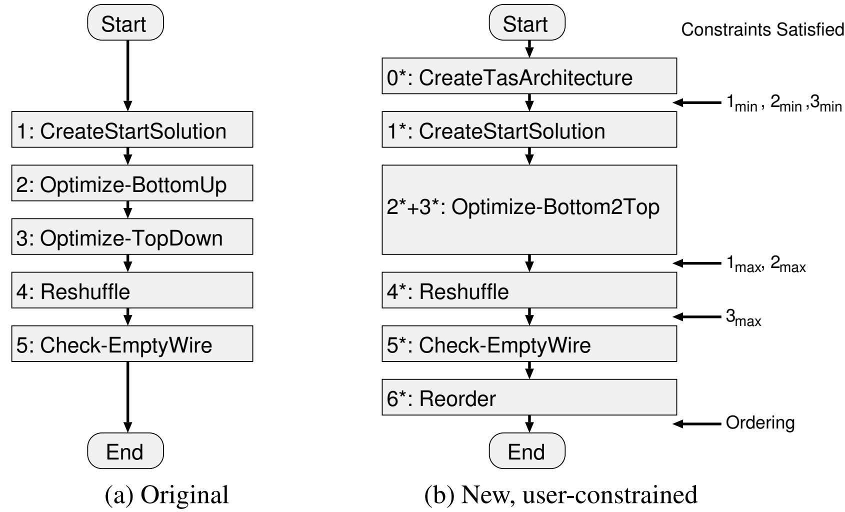
$$\text{Constraint } 3_{\max}: \quad |C(ru)| \leq c_{\max}(ru), \;\; \forall\, ru \in R_{\text{user}} \tag{6.3b}$$

Constraint $3_{\min}$ means that for all user-specified TAMs, cores connected to a TAM ru (set $C(ru)$) should at least contain the cores specified by the user (set $C_{\text{user}}(ru)$) for this TAM. Similarly, Constraint $3_{\max}$ means that for all user-specified TAMs, the total number of cores $|C(ru)|$ connected to a TAM ru should not exceed the maximum number of cores that are allowed $c_{\max}(ru)$ to be connected to this TAM.

Next, a user-constrained version of TR-ARCHITECT is presented. The new user-constrained TR-ARCHITECT takes into account the above-described user constraints while designing a test architecture.

6.4.2 User-Constrained Test Architecture Design

Figure 6.3 depicts the steps of the heuristic algorithm of TR-ARCHITECT. Figure 6.3(a) shows the five steps of the original version of TR-ARCHITECT as described in Chapter 3.



(a) Original (b) New, user-constrained

Figure 6.3: Algorithm steps in (a) the original version and (b) the new, user-constrained version of TR-ARCHITECT.

The algorithm steps of the new, user-constrained version of TR-ARCHITECT are depicted in Figure 6.3(b). It contains a modified version of the five steps of the original algorithm. A new, pre-initialization step has been added: Step 0*, CREATETASARCHITECTURE. This step ensures that all 'minimum' user constraints (i.e. Constraints 1_{min}, 2_{min}, and 3_{min}) are met. In Steps 0* and 1*, the algorithm temporarily ignores Constraints 1_{max}, 2_{max}, and 3_{max}. The reasoning behind this strategy is that some 'maximum' constraints might be met automatically by TR-ARCHITECT, i.e. without additional actions. In order to ensure that these 'maximum' constraints are met in the final architecture, the subsequent steps of the algorithm have been modified.

In the original algorithm, steps 2 and 3 are the last ones in the sequence to influence the number of TAMs and the TAM widths. In the new algorithm, they have been merged into one combined Step 2*+3*: OPTIMIZE-BOTTOM2TOP that guarantees that Constraints 1_{max} (the maximum number of TAMs) and 2_{max} (maximum TAM widths) are met. Step 4 is the last step to influence the number of cores per TAM. Its new version, step 4*, has been modified such that it now enforces the corresponding Constraint 3_{max}. Step 5* checks for the empty wires at all TAMs. Finally, a new step: step 6* handles the order constraints. In the remainder of this section, all algorithm steps are described in more detail.

Figure 6.4 shows the sequence of actions performed in CREATETASARCHITECTURE. First, the TAS file is parsed and checked for syntax errors. Subsequently, the

user specification is checked for inconsistencies, such as a sum of minimum widths of the user-specified TAMs exceeding $w_{\max}$ or if a core is assigned to multiple user-specified TAMs. In case of errors or inconsistencies, the tool sends appropriate error or warning messages and then exits. Otherwise, an initial test architecture consisting only of user-specified TAMs is created. All user-specified TAMs get assigned their minimum number of wires, and all their user-assigned cores. The resulting architecture meets all 'minimum' user constraints ($1_{\min}$, $2_{\min}$, $3_{\min}$); subsequent steps maintain this as invariant. Furthermore, all 'maximum' user constraints ($1_{\max}$, $2_{\max}$, $3_{\max}$) will be satisfied during next five steps.

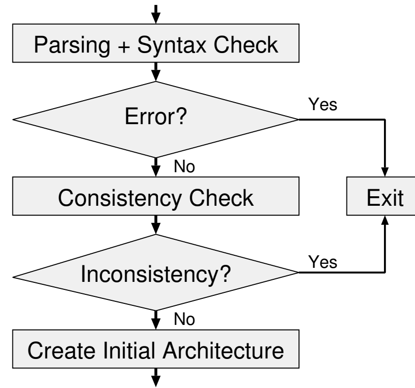


Figure 6.4: Sequence of actions performed in CREATETASARCHITECTURE.

In the step CREATESTARTSOLUTION, the remaining cores and wires (if any) are added to the architecture. First, the cores which are completely free, i.e. the cores which do not have a `FlexTAMs` constraint, are assigned to one-bit wide TAMs. These cores are assigned in the decreasing order of their test-data volume. If there are more wires than such cores, each core gets assigned; else some of them remain unassigned. These remaining cores, if any, are iteratively assigned to TAMs, such that the assignment yields the smallest increase in overall test time. Subsequently, cores with `FlexTAMs` constraints are assigned to TAMs according to the same algorithm, while satisfying those constraints. Finally, if there are some unused wires left, these wires are added iteratively to the TAM with the maximum test time, while satisfying Constraint $2_{\max}$ regarding their maximum TAM width.

Steps OPTIMIZE-BOTTOMUP and OPTIMIZE-TOPDOWN try to merge a pair of TAMs into a new TAM, such that the new TAM requires less wires. The freed-up wires can be used for an overall test time reduction. In order to maintain the user-specified TAMs in the architecture, a merge of two user-specified TAMs is *not* allowed. In addition, it is taken into account that a user-specified TAM width (Constraint $2_{\max}$), if any, is not violated while assigning the TAM width.

Steps 2^* and 3^* might initially finish with a number of TAMs which is too large, and hence violate Constraint $1_{\max}$, if existing. If that is the case, more TAMs need to be merged in order to get the number of TAMs down. Therefore, an iterative loop around steps 2^* and 3^* has been added, as shown in Figure 6.5. The acceptance criterion for

TAM merging is relaxed from "no increase in overall test time" (i.e. $T_{\text{new}} \leq T_{\text{old}}$; $\alpha = 0$) to "at most $\alpha\%$ increase in overall test time" (i.e. $T_{\text{new}} \leq (1 + \alpha/100) \cdot T_{\text{old}}$; $\alpha > 0$). The relaxation percentage α is iteratively increased, until the number of TAMs meets the user constraint. Forcing the number of TAMs down to the user-specified maximum in this way does increase the compute time of TR-ARCHITECT, and might also negatively influence the overall test time.

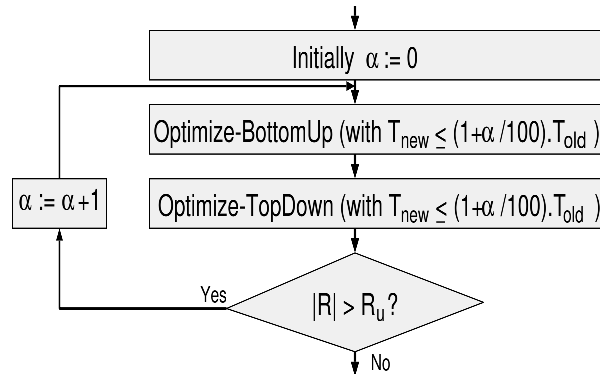


Figure 6.5: Sequence of actions performed in OPTIMIZE-BOTTOM2TOP.

In step 4* RESHUFFLE, individual cores are moved from one TAM to another. This step was originally meant only for fine-tuning of the test architecture with respect to test time. Now, it is also used to resolve violations of Constraint $3_{\max}$, i.e. the user-specified maximum number of cores per TAM. First, the TAMs are identified for which Constraint $3_{\max}$ is violated. Iteratively, excess cores from these TAMs are removed and re-assigned to other TAMs, such that Constraint $3_{\max}$ is not violated and the corresponding increase in overall test time is minimal. If the TAM from which cores are removed is the TAM with maximum test time, the overall test time is also minimized. Subsequently, the procedure tries to improve the overall SOC testing time in the same way as the original version of RESHUFFLE, that is iteratively moving the core with minimum test time from the bottleneck TAM to another TAM, while keeping Constraint $3_{\max}$ satisfied.

In step 5* CHECK-EMPTYWIRES, empty (redundant) wires are searched at all TAMs under the condition that Constraint $2_{\min}$, being the minimum width of the TAM, is not violated. If empty wires are found, they are assigned iteratively to the TAM with the maximum test time such that the overall test time is minimized and Constraint $2_{\max}$ is not violated.

Finally, in step 6*, REORDER, the ordering of cores within a TAM is handled. First, the user-specified cores ordering constraints are satisfied. In a second step, other cores in the TAMs are sorted accordingly to minimize TAM wire length [GM03c]. The test architecture obtained after this step, satisfies all user-constraints specified in the input TAS file.

6.5 Experimental Results

This section presents experimental results obtained with the new user-constrained version of TR-ARCHITECT. For the experiments, SOCs from the *ITC'02 SOC Test Benchmarks* [MIC] were used. First, the operation of the new user-constrained version of TR-ARCHITECT for SOC p22810 with $w_{\max} = 64$ is illustrated by means of an example case. For the user constraints, the TAS file as shown in Figure 6.1 is used. The example TAS file contains three user-specified TAMs with minimum width equal to 4, 1, and 5 respectively. The total number of TAMs in the architecture is limited to five.

For the first four steps, Figure 6.6 shows the resulting test schedules after every step of TR-ARCHITECT. In these schedules, the horizontal axis displays test time (in number of clock cycles), while the vertical axis displays the TAM width. The numbered boxes depict the various core tests. The number in the box is the core identification number; for very small boxes, this number is omitted. The cores which were specified by the user to be assigned to certain TAMs are depicted as boxes of a slightly darker color than the other cores. The three darkest shades of gray in the schedules represent three types of idle bits 3.5.

Figure 6.6(a) shows the initial test schedule after Step 0^*, CREATETASARCHITECTURE. This schedule contains only the three user-specified TAMs and six cores. TAMs $ru1$, $ru2$, and $ru3$ get assigned their minimum TAM widths, respectively 4, 1, and 5. In the shown schedule, out of 64 given TAM wires, only 10 TAM wires are used. Similarly, out of 28 cores in the SOC, only six cores are assigned. Therefore, the assignment of the remaining 54 TAM wires and 22 cores have to be carried out in the remaining five steps.

Figure 6.6(b) shows the test schedule after Step 1^*, CREATESTARTSOLUTION. In this step, the remaining cores are assigned to one- or multiple-wire TAMs, depending on their test-data volume and the number of available TAM wires. For this example, TR-ARCHITECT created 24 distinct TAMs, including the three user-specified TAMs. Hence, the architecture violates Constraint $1_{\max}$, which allows for at most five TAMs. The total test time of this schedule is $173,705$ clock cycles, and is dominated by core 1. There is a large imbalance in the completion time of the various TAMs.

The test schedule obtained after Step 2^*+3^*, OPTIMIZE-BOTTOM2TOP, is shown in Figure 6.6(c). In order to satisfy Constraint $1_{\max}$, TR-ARCHITECT has reduced the number of TAMs from 24 down to five; three user-specified TAMs and two TAMs created by the tool. The reduction of the number of TAMs was achieved for $\alpha = 9$, meaning an individual merge was accepted in OPTIMIZE-BOTTOM2TOP even if that merge resulted in a 9% increase in the SOC test time. However, despite the user constraints, this optimization step yielded in an overall test time of $144,981$ clock cycles, which is a 16% reduction with respect to the previous step. After completing Step 2^*+3^*, the only user constraint which can still be violated is Constraint $3_{\max}$, being the maximum number of cores per TAM. In Figure 6.6(c), TAM $ru2$ of width five is a user-specified TAM which contains 15 cores, while at most three cores are allowed to be assigned to this TAM.

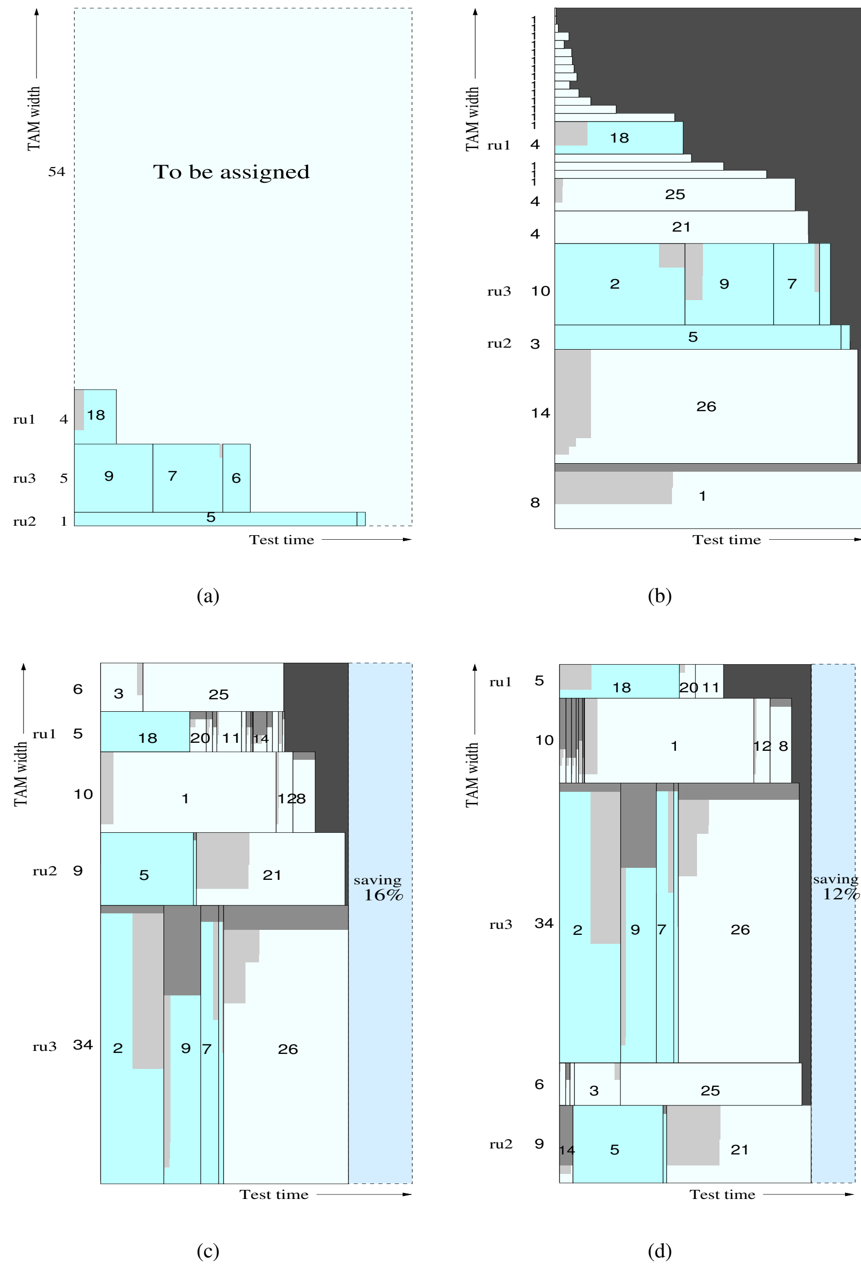


Figure 6.6: Test schedule for SOC p22810 with $w_{\max} = 16$ after (a) CREATETASARCHITECTURE, (b) CREATESTARTSOLUTION, (c) OPTIMIZE-BOTTOM2TOP, and (d) RESHUFFLE steps of the user-constrained TR-ARCHITECT.

Step 4*, RESHUFFLE, resolves this violation and results in the test schedule shown in Figure 6.6(d). The overall test time is $151,611$ clock cycles, which is an increase of 4.5% compared to the previous step. This is due to the fact that TR-ARCHITECT was forced to move cores from TAM $ru1$ to other TAMs to satisfy Constraint $3_{\max}$, despite the associated test time increase. Step 5*, CHECK-EMPTYWIRES, did not result in any test time improvement. Finally, Step 6* (REORDER) satisfies the cores-ordering constraints in TAMs. The resulting test architecture and corresponding test schedule now satisfies all user constraints. For this case, the original version of TR-ARCHITECT, without user constraints, achieves a test time of $133,405$ clock cycles (see Section 3.8 in Chapter 3). The test time cost of implementing this set of user constraints was 13.6%, when compared to the original, unconstrained test time of $133,405$ clock cycles.

Next, the relation between a user-specified maximum number of TAMs and the SOC test time is investigated. In this experiment, for a given SOC and $w_{\max}$, first TR-ARCHITECT is executed unconstrained, in order to get the 'optimal' number of TAMs. Subsequently, TR-ARCHITECT is executed with user constraints, in which iteratively only the user-specified maximum number of TAMs is specified and decreased by one. Figure 6.7 shows these experimental results.

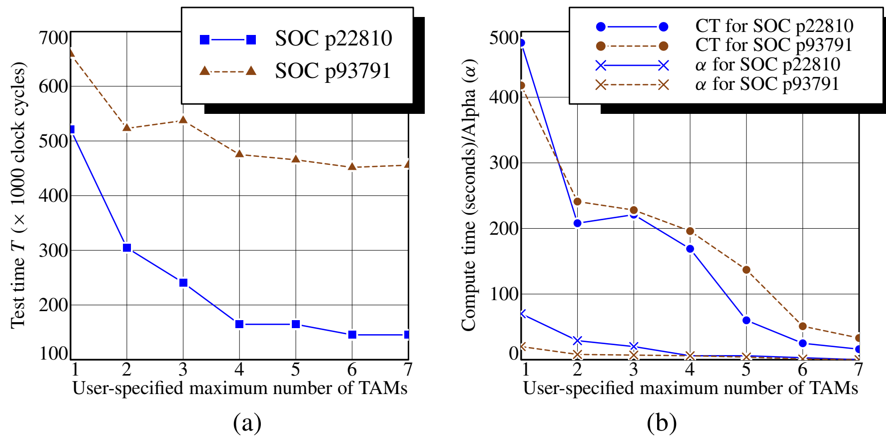


Figure 6.7: Variation in SOC test time, computational time (CT), and α with the user-specified maximum number of TAMs.

Figure 6.7(a) shows the SOC test time as a function of the user-specified maximum number of TAMs for (1) SOC p22810 and $w_{\max} = 56$, and (2) SOC p93791 and $w_{\max} = 64$. In both cases, the unconstrained run of TR-ARCHITECT yields seven TAMs. Figure 6.7(a) shows that restricting the number of TAMs usually leads to an increase in test time. However, due to the heuristic nature of the optimization algorithm, in some cases decreasing the number of TAMs also decreases the overall test time. For example, for SOC p93791, going from seven to six TAMs and three to two TAMs, decrease the overall test time.

Figure 6.7(b) shows the variation in the value of α and computational time as function of the user-specified maximum number of TAMs. The value of α increases for a decreasing maximum number of allowed TAMs. This was to be expected as a small

number of TAMs means more iteration loops in OPTIMIZE-BOTTOM2TOP, in order to merge more TAMs and to reduce the total number of TAMs further. The increase of α leads to an increase of computational time. In general, one can state that more user constraints lead to longer compute times of TR-ARCHITECT.

6.6 Summary

In this chapter, an approach to extend basic SOC test architecture design approaches that minimize the overall SOC test time with the capabilities to include user constraints is presented. To model user constraints in a concise and un-ambiguous way, a Test Architecture Specification (TAS) language is defined. In a TAS file, a user can express constraints with respect to (1) the number of TAMs, (2) the TAM widths, (3) the assignment of cores to TAMs, and (4) the core ordering with a TAM. A TAS file can serve as an input to the test architecture design tool. With the help of this TAS format, a user can now specify a partial or full test architecture specification to the test architecture optimization algorithm.

A user-constrained version of TR-ARCHITECT that takes into account user constraints specified in the TAS language is presented. The new proposed TR-ARCHITECT uses an approach of, six steps instead of the five-step approach used in the original TR-ARCHITECT. The operation of the proposed TR-ARCHITECT is illustrated for SOC p22810 taken from the *ITC'02 SOC Test Benchmarks*. Finally, for two benchmark SOCs, it was shown that increasingly strict user constraints typically lead to an increase in both test time and computational time.

Chapter 7

Test Architecture Design for SOCs with Hierarchical Cores

7.1 Introduction

Modern SOC designs are not limited to only one level of hierarchy (SOC and cores), instead they consist of multiple levels of design hierarchy. For example, [DJR01, GCM+04] describe SOCs for digital video, for which the design is partitioned into design units called *chiplets*, which in turn consist of cores. In general, cores or design units that contain other cores within them are referred to as *hierarchical cores*. Contrary to this, cores that do not contain other cores are referred to as *flat cores*. Based on the hierarchical relation, a hierarchical core is also called a *parent core*, while the cores which are at one-level below and embedded in this core, are referred to as *child cores*. By definition, a child core itself can be a parent core for the cores at deeper levels of hierarchy. An example of a hierarchical core is an older generation SOC embedded in a current generation SOC. In the *ITC'02 SOC TestBenchmarks* suite [MIC], a large number of industrial SOCs have hierarchical cores with multiple levels of hierarchy. For example, SOC p93791 [MIC] contains 32 cores, out of which *eight* are hierarchical cores and contain embedded memory cores.

While designing test architectures for SOCs with hierarchical cores, wrappers for hierarchical cores have to be designed such that the test-access is available to both parent and child cores. Testing of a parent core requires access to its own elements such as scan chains and wrapper cells as well as to the elements of its child cores. For simplicity, most prior work on wrapper design as well as on test architecture design have assumed only one level of hierarchy (SOC and cores), i.e. even if there is a hierarchy among the cores, all cores in an SOC are treated at the same level of hierarchy. This includes the wrapper and test architecture design algorithms presented in previous chapters. Due to this assumption, all these methods design wrappers for both hierarchical and flat cores in the same fashion. For a parent core testing, all these methods do not consider the time required to access to elements in its child

cores. Furthermore, optimal test schedules proposed by these methods allow parallel testing of parent and child cores, which is *not* supported by the existing wrapper architectures [MAB+98, VB98, DZW+03]. Therefore, test solutions proposed by these methods are not directly applicable to real-life SOCs and support for multiple levels of hierarchy requires substantial modifications to these approaches.

In this chapter, the problems of wrapper design for hierarchical cores and test architecture design for SOCs with hierarchical cores are addressed. First, a generic hierarchical core model is presented, and four different practical design scenarios that occur between two adjacent hierarchy levels are identified. Next, the testing requirements for a hierarchical core are discussed and an improved wrapper architecture that allows for parallel testing of parent and child cores is presented. The proposed wrapper architecture is compatible with the proposed IEEE P1500 wrapper architecture [DZW+03] and the wrapper architecture presented in Chapter 2, and presents an extension to that. The proposed wrapper architecture optimizes both core internal as well as external test time. By using the proposed wrapper architecture, optimal test schedules obtained from the existing test architecture design methods that consider SOCs with flat cores can be implemented directly for the same SOCs with hierarchical cores. Hence, optimal test times can be obtained for SOCs with hierarchical cores. Finally, experimental results for the *ITC'02 SOC Test Benchmarks* [MIC02] are presented.

7.2 Prior Work

All wrapper architectures and wrapper design methods available in literature [DZW+03, MAB+98, VB98, MGL00, Kor02], including the one which is presented in Chapter 2, assume flattened hierarchy, i.e. flat cores. All wrapper architectures support three mandatory modes. These modes are the normal functional mode, the Inward-facing mode, and the Outward-facing mode. In the normal functional mode, the wrapper is transparent and the core is in the functional operation mode. The Inward-facing mode is used to test the circuitry inside the core. In the Outward-facing mode, the circuitry outside the core is tested. Some wrapper architectures [DZW+03, MAB+98] also support Bypass mode, which is used to bypass the entire core. Due to conflicting requirements of wrapper cells in various modes, a wrapper can only be configured in one of the modes at a time.

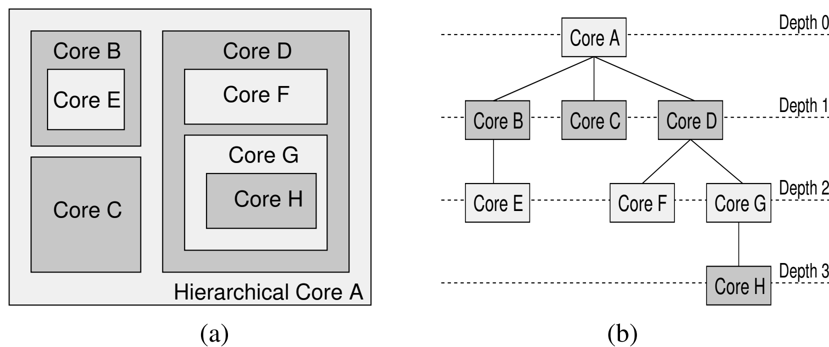
Most test architecture design algorithms described in literature [Cha00b, ICM01, EI01, HCT+01, GM02c, HRC+02, KI02, GM03a, LF03, ZRPH03, GM04a, KGM+04] assume *flat* cores, i.e. even if there is a hierarchy among the cores, all cores in an SOC are treated at the same level of hierarchy. Therefore, to minimize SOC test time, all the methods propose test schedules which allow for parallel testing of parent and child cores which is not possible with the used wrapper architecture. Only limited work has been carried out on test architecture design for SOCs with hierarchical cores [BCC+00, CBC00, BMM02]. While these paper have addressed the design of hierarchical test architectures, they have not focused on optimizing wrappers and TAM design for test time minimization and presenting results for industrial benchmarks. Recently [ICKK03], test architecture optimization techniques for non-hierarchical SOCs

have been used to optimize test architectures for SOCs with multiple levels of hierarchy. However, access to child cores elements for parent cores tests and the constraints on parallel testing of parent and child cores have been neglected.

7.3 Hierarchical Core Model

Before going into the details with regard to testing of hierarchical cores, first a generic hierarchical core model is presented in this section. Hierarchical cores can have multiple levels of design hierarchy. They contain embedded cores, which in turn can contain other embedded cores at deeper levels of hierarchy. Therefore, by definition, a hierarchical core model is a recursive model.

The design hierarchy present in a hierarchical core can be easily represented by its design hierarchy tree. In a design hierarchy tree, nodes represent cores and an edge between two nodes represents the hierarchical relation between the corresponding cores. All leaf nodes in a design hierarchy tree represent non-hierarchical cores, while the root node represents the top-level design entity (core or SOC). The depth of a node represents the level of the corresponding core in the design hierarchy. A node (core) at depth n in the design tree is called a *parent* node (core) with respect to the nodes (cores) that are connected to it and are at depth $(n+1)$. Conversely, nodes at depth $(n+1)$ are called *child* nodes with respect to the node which is at depth n and connected to these nodes. Parent nodes may have multiple-child nodes, which in turn can be parent nodes for some-other nodes.



(a) (b)

Figure 7.1: Example of a generic (a) hierarchical core, and its (b) design hierarchy tree.

Figure 7.1(a) shows an example of a generic hierarchical core A. The hierarchical core shown in the figure contains three child cores B, C and D, out of which cores B and D are again hierarchical cores. Core B contains only one child core E, while core D contains two child cores F and G. Core G itself contains a child core H. Therefore, core G is a child core of core D but the parent core for core H. Figure 7.1(b) shows the design hierarchy tree for core A. In Figure 7.1(b), at the top-level (depth 0) is core A itself, while core H is at the lowest-level (depth 3) in the design hierarchy tree.

In case of hierarchical cores, there are three parameters that influence the problems

of wrapper and test architecture design. These three parameters are scan chains, wrapper and TAM architecture. These parameters might be *hard* or *soft* in-terms of their implementation. A hard implementation of a parameter is the one in which the parameter is fixed and cannot be changed. On the contrary, a soft implementation allows for changes according to the requirements. In a test architecture, a TAM can be assigned to a core, only after the core has been wrapped and a core can be wrapped only after the scan chains have been designed. Therefore, a soft implementation of scan chains cannot follow a hard implementation of a wrapper. Similarly, a soft implementation of wrapper cannot follow a hard implementation of TAM architecture. Based on the *hard* and *soft* implementation of scan chains, wrapper and TAM architecture, there are four feasible design scenarios that can be identified for a parent or a child core [SGMC04]. These four design scenarios are as follows:

1. hard scan chains, hard wrapper, hard TAM architecture,
2. hard scan chains, hard wrapper, soft TAM architecture,
3. hard scan chains, soft wrapper, soft TAM architecture,
4. soft scan chains, soft wrapper, soft TAM architecture.

For example, consider a third-party hierarchical IP core which contains several other embedded cores. To protect IP rights, all child cores in the hierarchical IP core might already be wrapped and TAMed by the core provider. This corresponds to design scenario 1 for all child cores. In addition, the hierarchical IP core might be wrapped itself which corresponds to design scenario 2 for the parent core. Examples of design scenarios 3 and 4 are in-house designed hierarchical cores, for which all three parameters can be soft. In general, a hard implementation of the scan chains of a core, wrapper and TAM architecture restricts the flexibility in test time minimization. On the other hand, soft implementation of scan chains, wrappers and TAM architectures are more suitable for optimization, since the design of the test architecture can be optimized globally for test time. Therefore, design scenarios 3 and 4 are more likely to occur in large design houses such as Philips. These design scenarios also correspond to the SOC models considered in the test architecture problems in the previous chapters.

Based on the reasoning mentioned above, only design scenarios 3 and 4 are considered in this chapter and design scenarios having fixed wrappers and TAM architectures are excluded from the discussion [SGMC04]. In this chapter, it is assumed that irrespective of the level in the design hierarchy, all cores in an SOC are not equipped with wrappers and TAMs. Furthermore, all cores can have a hard or soft implementation of the scan chains inside the cores, i.e. fixed-length scan chains or flexible-length scan chains are allowed.

7.4 Testing of Hierarchical Cores

To understand the problem why traditional wrapper architectures and test architecture design algorithms available in literature are not sufficient for SOCs with hierarchical

cores, one needs to look at the testing requirements for hierarchical cores. Let us consider the testing of a hierarchical core consisting of one parent and one child core only. The parent core can be considered at any level n, while the child core can be considered at level $(n + 1)$. This allows the possibility for the parent core to be a child core itself for some other core at level $(n - 1)$. Since the hierarchical core model is a recursive model, the presented analysis for two adjacent levels of hierarchy can be easily extended to any number of levels and with any number of child cores.

Figure 7.2(a) shows an example of a hierarchical core consisting of one parent and one child core. The parent core has two scan chains, two functional input terminals `Pa[0:1]`, and two functional output terminals `Pz[0:1]`. The child core also has two scan chains, but three functional input terminals `Ca[0:2]` and two functional output terminals `Cz[0:2]`.

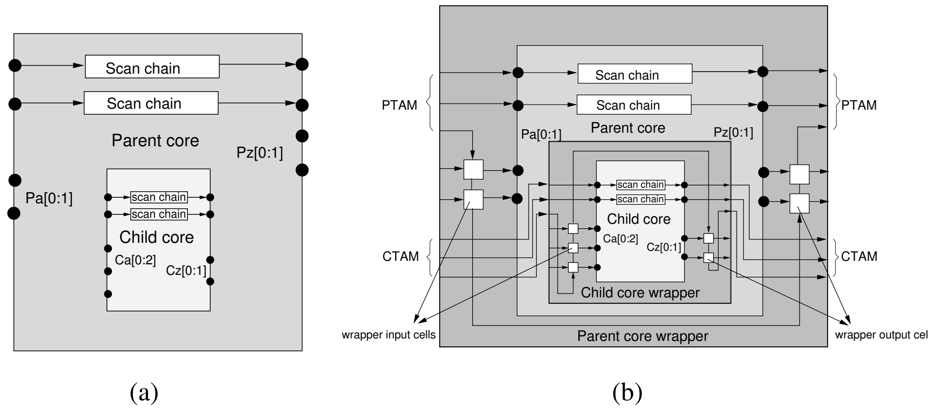


Figure 7.2: (a) Example of a hierarchical core, and (b) its wrapper.

Figure 7.2(b) shows the IEEE P1500 compliant wrapper architecture [DZW+03] for this core. For clarity, the control circuitry, the bypass circuitry, and the connection through the one-bit serial TAM are not shown. In the shown wrapper, the test-access requirement for the parent core is served by a three-bit wide TAM called `PTAM`, while the test-access requirement for the child core is served by a three-bit wide TAM called `CTAM`. All functional terminals are connected to TAM wires via wrapper cells, while the scan chains are directly connected to TAM wires. It is important to note that in practice, the parent and child cores may share the same TAM. Furthermore, if connected to different TAMs as shown in Figure 7.2(b), the width of `PTAM` and `CTAM` can also be different. Now the testing of the child and parent cores are discussed below.

Child Core Test

To test a core, one needs to apply test stimuli and subsequently observe test responses. To apply test data to the child core, test stimuli have to be loaded into its scan chains, and the wrapper input cells that are connected to its functional input terminals marked as `Ca[0:2]`. Similarly, to observe test data from the child core, test responses have

to be unloaded from its scan chains, and the wrapper output cells that are connected to its functional output terminals marked as `Cz[0:1]`. This requires the wrapper of the child core to be configured in the Inward-facing mode. It is important to note here that during the testing of a core, wrapper input cells are used to apply test data, while the wrapper output cells are used to capture test responses.

Parent Core Test

Testing of the parent core is somewhat more complex. To apply test data to the parent core, test stimuli have to be loaded into its scan chains, and the wrapper input cells that are connected to its functional input terminals marked as `Pa[0:1]`. As the primary output terminals (`Cz[0:1]`) of the child core acts as input terminals to the parent core, test stimuli have to be loaded into the wrapper output cells that are connected to the functional output terminals (`Cz[0:1]`) of its child core. Figure 7.3 shows this case. In Figure 7.3, all elements, like scan chains and wrapper cells that take part in this phase are enclosed in a dotted dark grey box.

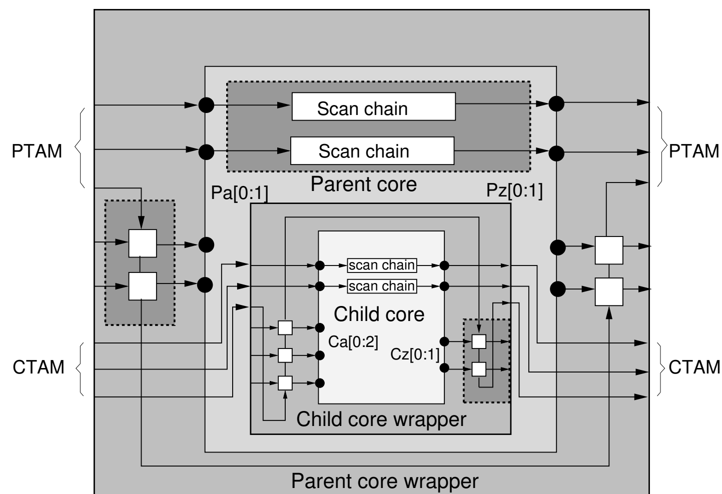


Figure 7.3: Applying test data to the parent core.

To observe test data from the parent core, test responses have to be unloaded from the parent core scan chains and the wrapper output cells connected to its functional output terminals marked as `Pz[0:1]`. As the primary input terminals (`Ca[0:1]`) of the child core acts as output terminals from the parent core, test responses have to be unloaded from the wrapper input cells connected to the functional input terminals (`Ca[0:2]`) of its child core. Figure 7.4 shows this case. In Figure 7.4, all elements, like scan chains and wrapper cells that take part in this phase are enclosed in a dotted dark grey box.

It is clear from Figures 7.3 and 7.4 that testing of a parent core requires not only access to its own elements such as scan chains and wrapper cells, but also to the el-

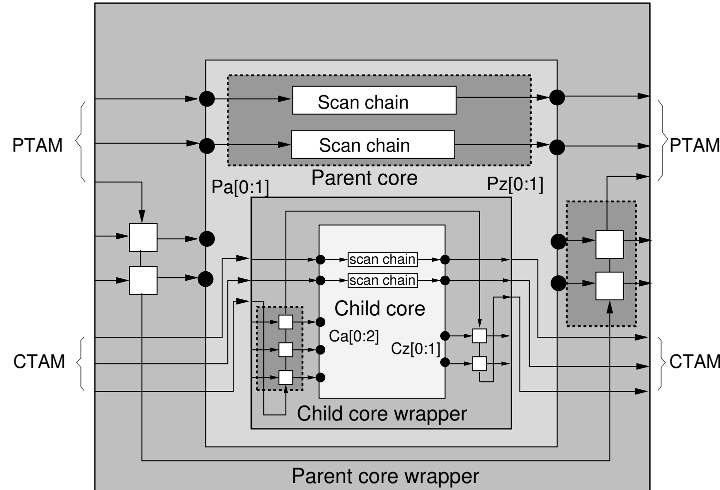


Figure 7.4: Observing test data from the parent core.

ements of its child core(s). Furthermore, the wrapper input cells in the child-core wrapper are used to capture test responses, while the wrapper output cells are used to apply test data. Therefore, during the parent-core testing, the role of wrapper cells in the child-core wrapper is reversed compared to their role during the child core testing itself. Due to this, testing of a parent core requires its wrapper to be configured in the Inward-facing mode and the wrapper(s) of its child core(s) to be configured in the Outward-facing mode. As the TAM (`PTAM`) connected to the parent core is not connected to the wrapper cells in the child core wrapper, while testing the parent core, both TAMs `PTAM` and `CTAM` should be used to transport test data for the parent-core test.

Current wrapper architectures [MAB+98, VB98, MGL00, DZW+03] only allow a wrapper to be configured in one mode at a time. Therefore, due to conflicting mode requirements for the child core wrapper, testing of a child core is not possible while the parent core is being tested. Based on this analysis, one can conclude that for hierarchical cores with multiple-levels of design hierarchy, parallel testing cannot be carried out for the cores that are hierarchically related and are at two adjacent levels in the design hierarchy. Furthermore, there is no restriction on parallel testing of a core and its grand-child cores. For example, for the hierarchical core example shown in Figure 7.1, testing of core D cannot be done in parallel with the testing of cores F and G, however it can be done in parallel with the testing of core H.

All test architecture design methods available in literature assumes flat cores and hence use the same wrapper architecture for both hierarchical and flat cores. To optimize SOC test time, these methods propose test schedules that allow for parallel testing of parent and child cores, which is *not* possible (as shown above) with the current wrapper architectures. Furthermore, for parent cores testing, these methods do not consider the time required to access the elements in the child cores wrappers. Therefore, test architectures and schedules proposed by these methods are not directly applicable to

real-life SOCs. This justifies the statement that existing test architecture design methods are not optimally suited for SOCs with hierarchical cores.

To gain a better understanding, for example consider the test architecture of a hierarchical SOC and the corresponding test schedule as shown in Figure 7.5. The SOC shown in Figure7.5(a) contains four cores, out of which core A is a hierarchical core and contains core B. The shown test architecture contains two TAMs of widths $w1$ and $w2$. The TAM with width $w1$ connects to cores C and B, while the other TAM connects to cores A and D.

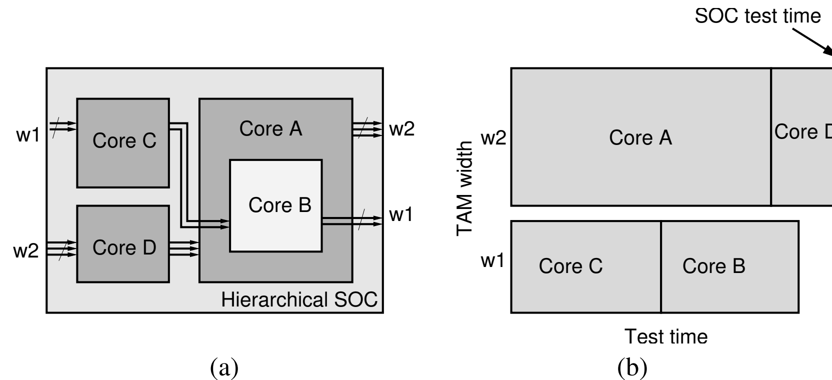


Figure 7.5: (a) Example of a test architecture for a hierarchical SOC, and (b) optimal test schedule considering no hierarchy.

The corresponding test schedule shown in Figure 7.5(b) does not consider any hierarchy among the cores, i.e. all cores are considered at the same hierarchical level. In Figure 7.5(b), the horizontal axis represents the test time, while the vertical axis represents the TAM width. In the case shown, the TAM containing cores A and D determines the SOC test time. Please note that for each core, the test time shown in the figure is the time required to test the circuitry inside the core, i.e. the wrapper of the core is configured in the Inward-facing mode only.

It is clear from Figure 7.5(b) that if all cores are considered at the same level, quite a good test completion time can be obtained for this SOC. Unfortunately, in reality there is a hierarchy between core A and core B, as core A is a parent core for core B. Due to this, cores A and B *cannot* be tested in parallel. Therefore, the test schedule shown in Figure 7.5(b) is not valid anymore and changes are required in order to respect the hierarchy present in the SOC.

A trivial solution to this problem is to modify the given test schedule in such a way that only one of the two (parent and child) cores is tested at a time. Unfortunately, this can lead to serialization of various tests and hence can severely affect the SOC test time. For example, Figure 7.6(b) shows the modified test schedule considering the hierarchy present in the SOC. In Figure 7.6(b), while testing core A, its child core is put into Outward-facing mode. It can be seen from Figure 7.6 that the modified test schedule results in a large penalty (65%) in test time.

In terms of minimum SOC test time, the best solution to this problem would be to

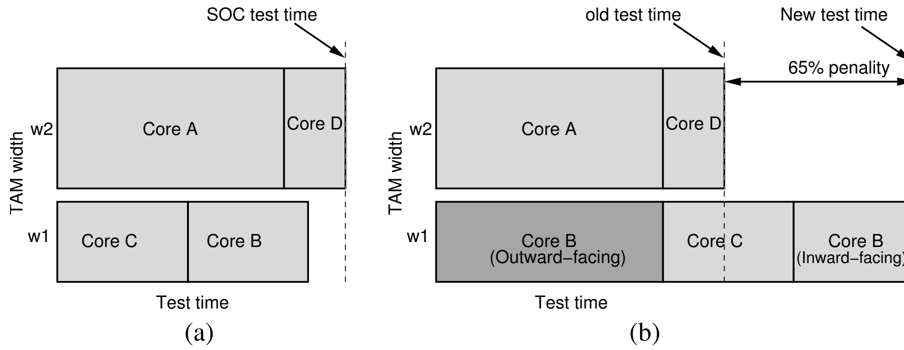


Figure 7.6: (a) Original test schedule considering no hierarchy, and (b) modified test schedule considering hierarchy among cores A and B.

modify the wrapper architecture such that parallel testing of parent and child cores is possible. By doing so, a test architecture design algorithm will have the full flexibility in terms of arranging tests of various cores, and optimal test times could be obtained for hierarchical SOCs as well. In the next section, an improved wrapper architecture is presented that allows for parallel testing of the parent and child cores.

7.5 Improved Wrapper Architecture

To allow the testing of the parent and child cores in parallel, it is proposed to change the wrapper cells in the child core wrapper [Goe04]. Unlike the conventional wrapper cell which is connected to only one TAM, the proposed wrapper cell is connected to the following two TAMs:

1. child-core TAM, to serve the test-data requirements for the child core,
2. parent-core TAM, to serve the test-data requirements for the parent core.

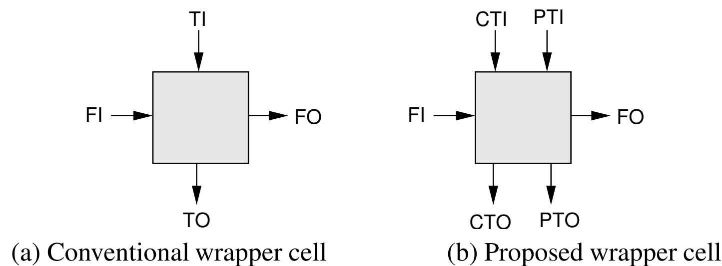
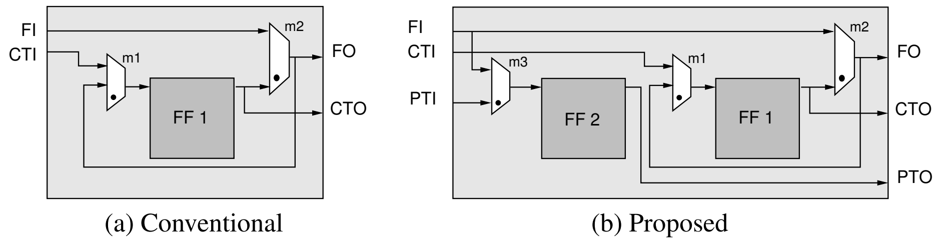


Figure 7.7: Conceptual views of wrapper cells.

Figure 7.7(a) shows a conceptual view of the conventional wrapper cell, while Figure 7.7(b) shows a conceptual view of the newly proposed wrapper cell. In Figure 7.7(a), signals `FI` and `TI` represent the primary (functional) and test (TAM) inputs

to the wrapper cell respectively. Similarly, signals FO and TO represent the primary (functional) and test (TAM) outputs from the wrapper cell. In Figure 7.7(b), signals CTI and CTO represent the test input and output corresponding to the child-core TAM, while signals PTI and PTO represents the same for the parent-core TAM.

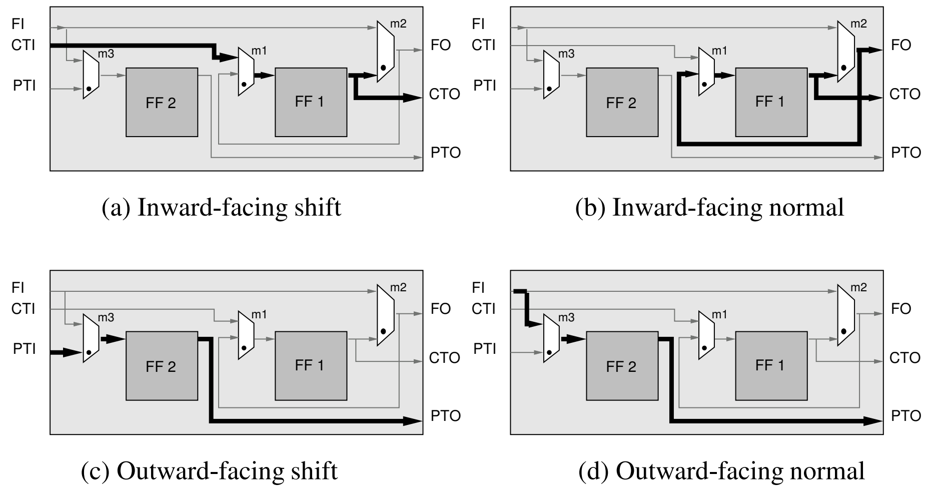
Figure 7.8(a) shows an example implementation for the conventional wrapper input cell. In this cell, flip-flop FF 1 is used to store test date for the core-under-test (in this case the child core). Figure 7.8(b) shows an example implementation of the proposed wrapper input cell. In this cell, there are two flip-flops; flip-flop FF 1 is again used to store test data for the child core test, while the newly added flip-flop FF 2 is used to store test data for the parent-core test. In both the cells, terminal FI is connected to the primary signal coming from the parent core. Similarly, terminal FO is connected to the primary signal going to the child core.



(a) Conventional (b) Proposed

Figure 7.8: Example implementations for the wrapper input cell.

Figure 7.9 shows the proposed wrapper input cell configured in various modes. The thick black line in the figure shows the active path in the corresponding mode. The Inward-facing mode is used to test the child core itself, while the Outward-facing mode is used during the parent-core test.



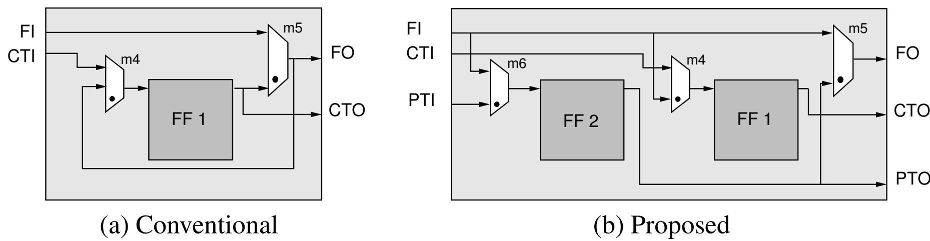
(a) Inward-facing shift (b) Inward-facing normal

(c) Outward-facing shift (d) Outward-facing normal

Figure 7.9: Configuration of the proposed wrapper input cell in various modes situations.

Figure 7.9(a) shows the Inward-facing mode during shift operation. In this mode, the wrapper cell is configured such that test data can be shifted in/out from flip-flop FF1 via the terminals CTI and CTO. Figure 7.9(b) shows the Inward-facing mode during the normal step. In this step, the data stored in flip-flop FF1 is applied to terminal FO. Similarly, Figures 7.9(c) and (d) show the same sequence but now for the Outward-facing mode. During the shift operation in this mode, the wrapper cell is configured such that test data can be shifted in/out from flip-flop FF2 via the terminals PTI and PTO. During the normal step, test response available at terminal FI is captured in flip-flop FF2.

Figures 7.10(a) and (b) show example implementations for the conventional and the proposed wrapper output cells respectively. In practice, the conventional wrapper input cell can also be used as the wrapper output cell, except that in the wrapper output cell, terminal FI is connected to the primary signal coming from the child core. Similarly, terminal FO is connected to the primary signal going to the parent core.



(a) Conventional (b) Proposed

Figure 7.10: Example implementations for the wrapper output cell.

Figure 7.11 shows the proposed wrapper output cell configured in various modes. Figure 7.11(a) shows the Inward-facing mode during shift operation. In this mode, the wrapper cell is configured such that test data can be shifted in/out from flip-flop FF1 via the terminals CTI and CTO.

Figure 7.11(b) shows the Inward-facing mode during the normal step. In this step, test response available at terminal FI is captured in flip-flop FF1. Similarly, Figures 7.11(c) and (d) show the same sequence but now for the Outward-facing mode. During the shift operation in this mode, the wrapper cell is configured such that test data can be shifted in/out from flip-flop FF2 via the terminals PTI and PTO. During the normal step, test stimuli stored in flip-flop FF2 is applied at the terminal FO.

Table 7.1 shows the required multiplexers settings for both the proposed wrapper input and output cells in the various supported modes. In the table, the symbol 'X' represents the *don't care* term. From the table, one can see that the settings for the Inward-facing mode and the Outward-facing mode are compatible with each other. Hence, with this type of wrapper cell, a core can be configured in both the Inward-facing and Outward-facing modes at the same time. Therefore, testing of the parent and child cores can be done in parallel, if they are connected to different TAMs.

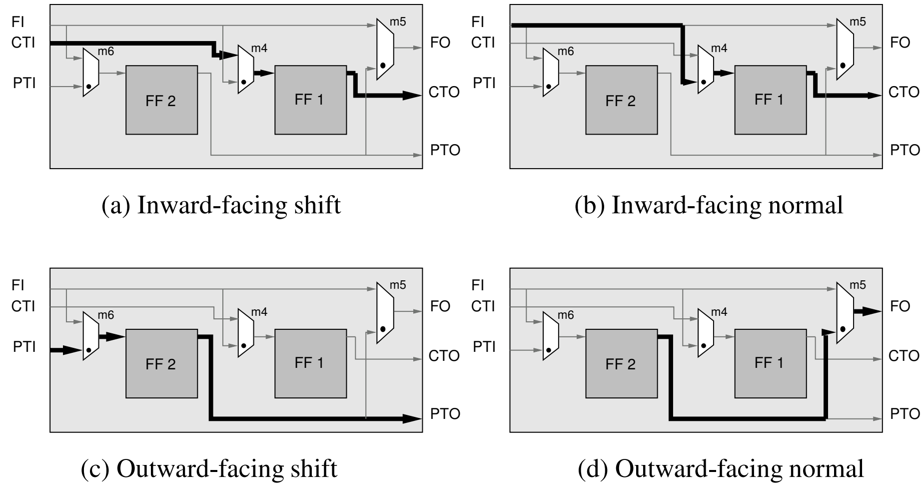


Figure 7.11: Configuration of the proposed wrapper output cell in various modes situations.

Table 7.1: Multiplexer settings in the proposed wrapper cells for various modes.

Wrapper mode	Required multiplexer settings					
	Wrapper input cell			Wrapper output cell		
	m1	**m2**	**m3**	**m4**	**m5**	**m6**
Functional	X	1	X	X	1	X
Inward-facing shift	1	X	X	1	X	X
Inward-facing normal	0	0	X	0	X	X
Outward-facing shift	X	X	0	X	X	0
Outward-facing normal	X	X	1	X	0	X

7.5.1 Testability of the Proposed Wrapper Cells

The proposed wrapper input cell is fully testable as all nodes in the cell are fully controllable and observable. In contrast, the wrapper output cell is not fully testable. This is due to the fact that unless connected directly to the chip pin, the value at the output signal of the multiplexer `m5` cannot be observed during any test mode. In order to make the proposed wrapper output cell fully testable, an additional multiplexer has be be added.

Figure 7.12 shows an example of the fully testable wrapper output cell. To observe the value at the output signal of the multiplexer `m5`, the bottom input signal of multiplexer `mx` need to be selected. Therefore, by setting the control signal of the multiplexer `mx` to logic value '`0`' in the normal operation during the Outward-facing mode, the value at the output signal of multiplexer `m5` can be captured in flip-flop FF 2 and subsequently shifted-out during the shift operation. For the rest of the modes, setting of multiplexer `mx` is not important and can hence be considered as *don't care* ('`X`').

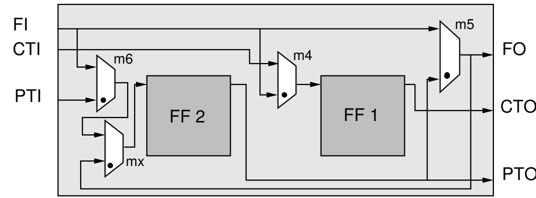


Figure 7.12: Example implementation for the fully testable wrapper output cell.

7.5.2 Ordering of Elements in a TAM

In the proposed wrapper cells for the child core, the parent TAM also connects to wrapper cells in the child core wrapper. Therefore, in the improved wrapper architecture that uses the proposed wrapper cells in the child core wrapper, the parent TAM is connected to the following elements:

- scan chains in the parent core
- wrapper input cells connected to the parent core functional input terminals
- wrapper output cells connected to the parent core functional output terminals
- wrapper input cells connected to the child core functional input terminals
- wrapper output cells connected to the child core functional output terminals.

To minimize the test time for the parent core, an optimal ordering for the above-mentioned elements in a TAM connected to the parent core is presented here. As described earlier, in order to test the parent core one needs to shift test stimuli into its scan chains, its wrapper input cells and also to the wrapper output cells of its child core. Similarly, one needs to shift-out test responses from its scan chains, its wrapper output cells and also from the wrapper input cells of its child core. Figure 7.13 shows the proposed optimal ordering of the various elements in a single TAM wire that is connected to the parent core.

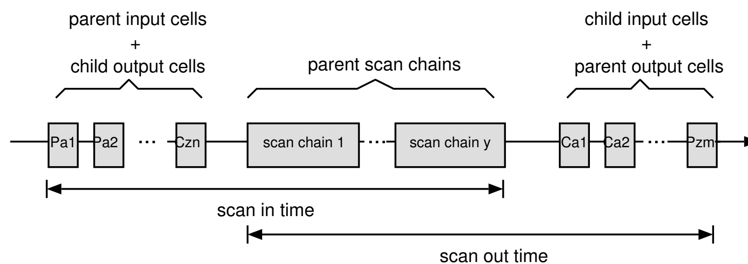
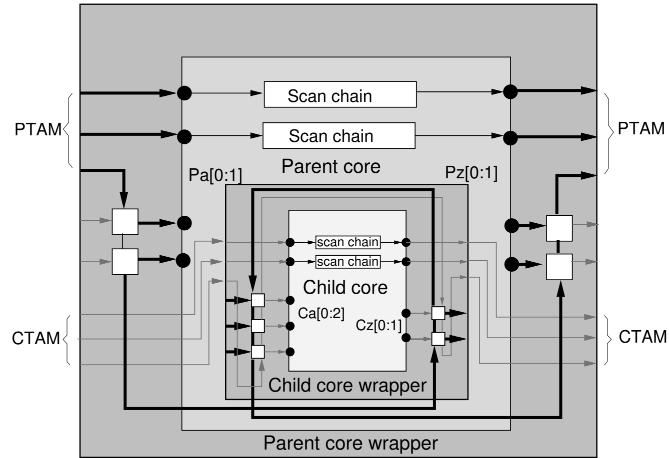


Figure 7.13: Ordering of various elements in a TAM connected to the parent core.

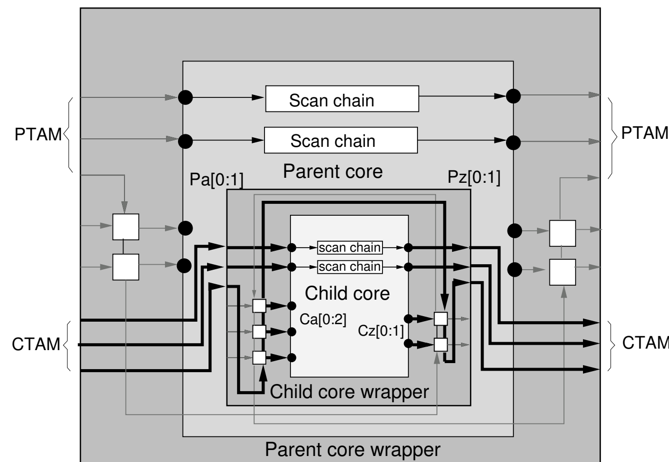
In Figure 7.13, boxes containing ids Pa_i and Pz_j represent parent wrapper input and output cells respectively. Similarly, boxes containing ids Ca_i and Cz_j represent

child wrapper input and output cells respectively. As the scan chains take part both in applying and observing test data, they are in the middle of the wrapper cells. The wrapper input cells for the parent core together with the wrapper output cells for the child core are connected in front of the scan chains. Likewise, the wrapper input cells for the child core and the wrapper output cells for the parent core are connected after the scan chains.

Based on the above described ordering, Figure 7.14 shows the improved wrapper architecture for the hierarchical core using the proposed wrapper cells.



(a) Parent core test



(b) Child core test

Figure 7.14: Improved wrapper architecture configurations for parent and child core tests

Figure 7.14(a) shows the improved wrapper configured in the parent Inward-facing and child Outward-facing modes. Figure 7.14(b) shows the improved wrapper architecture configured in the child Inward-facing mode. In both the figures, active connections are shown by thick black lines, while the inactive connections are shown by grey lines.

As far as the impact of new wrapper cell on the functional performance of the core is concerned, there will not be any difference as compared to the conventional wrapper cell. This is due to the fact that in the new wrapper cell also, there is only one multiplexer in the functional path from `FI` to `FO`. The only drawback of the proposed wrapper cell is the area overhead. Compared to the conventional wrapper cell, which only requires one flip-flop, the new wrapper cell requires two flip-flops.

7.6 Experimental Results

In this section, experimental results are presented for four SOCs taken from the *ITC'02 SOC Test Benchmarks* [MIC02]. These four SOCs are p22810, p34392, p93791, and a586710. These four SOCs were selected, as they are the only ones which contain multiple levels of design hierarchy. As the proposed wrapper architecture allows parallel testing of hierarchical cores, the cores inside an SOC and the SOC itself (top-level test) can also be tested in parallel. Unfortunately, not all SOCs have their top-level test listed in the *ITC'02 SOC Test Benchmarks* [MIC02]. Therefore, only the core-internal tests for all cores in the SOCs are considered. As all the SOCs in the benchmark set have a fixed number of scan chains and lengths, only the test time results considering hard scan chains, soft wrapper and soft TAM architecture (design scenario 3) are presented. Furthermore, only the test time results for hybrid test-bus architectures are presented; however similar results could be obtained for hybrid TestRail architectures also.

In the experiments, basic version of TR-ARCHITECT (described in Chapter 3) was used for the test architecture design. However, the proposed wrapper architecture is not limited to this design method only and other test architecture design algorithms available in literature can be used instead. Here, the test time results for three cases will be compared. In the first two cases, the conventional wrapper cells [MKL+02, DZW+03] are used in the wrappers of all cores. In the third case, the proposed wrapper cells are used in the wrappers of all child cores.

Case 1 is the original test architecture design as presented in Chapter 3. In this case, all cores in an SOC are considered at the same level of design hierarchy. Therefore, in all SOCs, all cores are considered to be *flat*. It is important to note that in Case 1, since no hierarchy is assumed, testing of a core requires access to its own scan chains and wrapper cells only.

In Case 2, it is assumed that the test architecture for the SOC has been already designed as in Case 1 and it is only allowed to modify the test schedules in order to respect the hierarchy present in the SOC. In this case, the wrapper of a child core is configured in the Outward-facing mode during the test of its parent core. Testing of a hierarchical core not only requires access to its own scan chains and wrapper cells, but also to the wrapper cells of its child cores.

In Case 3, the design hierarchy present in the SOC is considered from the very beginning. In this case, the proposed wrapper cells are used in the wrappers of all child cores. As the proposed wrapper cells allow for the testing of hierarchical cores in parallel, neither the test architecture design algorithm nor the test schedule obtained from the algorithm needs to be modified. However, for test time calculation of a parent core, one needs to consider the access to elements of both the parent core as well as of its child cores.

Table 7.2 shows the test time results (in terms of the number of clock cycles) for the three above mentioned cases. The first two columns in Table 7.2 show the SOC name and the number of TAM wires $w_{\max}$ available for the SOC test architecture design. Column 3 shows the test time results for Case 1, i.e. considering flat cores. Column 4 shows the test time results for Case 2, in which the test schedules obtained from Case 1 are modified to respect the design hierarchy. Column 5 ($\triangle T$) shows the percentage (%) difference between the test times for Case 1 and Case 2.

Table 7.2: Experimental results for the test times of the three mentioned cases.

SOC	$w_{\max}$	**Using Conventional Wrapper Cells**			**Using Proposed Wrapper cells**		
		Flat cores (Case 1)	**Hierarchical Cores (Case 2)**		**Hierarchical Cores (Case 3)**		
		T [GM02c]	T	$\triangle T(\%)$	T	$\triangle T(\%)$	**#NWC**
p22810	16	457433	621895	36	466667	2	600
	24	302737	547039	81	309641	2	600
	32	222471	493290	122	229899	3	600
	40	190995	460885	141	191978	1	600
	48	157851	279949	77	157226	0	600
	56	145417	315469	117	145417	0	548
	64	133405	296445	122	133405	0	600
p34392	16	1010821	2327127	130	1019766	1	624
	24	663193	1977631	198	702852	6	336
	32	584524	1449581	148	584524	0	624
	40	544579	643939	18	544579	0	301
	48	544579	855644	57	544579	0	388
	56	544579	855644	57	544579	0	388
	64	544579	855644	57	544579	0	388
p93791	16	1791638	3363819	88	1792354	0	886
	24	1185434	3438009	190	1211510	2	2157
	32	912158	1504806	65	917246	1	2007
	40	718005	4021279	460	730713	2	1875
	48	601450	1965642	227	610037	1	2127
	56	528925	1059869	100	528407	0	2270
	64	455738	1908099	319	458600	1	2127
a586710	16	41523868	60294453	45	41523868	0	1678
	24	28716501	53295859	86	28716501	0	1678
	32	22475033	40729137	81	22475033	0	1678
	40	19048835	36478145	91	19048835	0	1678
	48	15315467	21723090	42	15212440	-1	1678
	56	13401034	13401034	0	13401034	0	1148
	64	12700205	12769440	1	12510356	-1	1148

Columns 6, 7, and 8 show the results for Case 3. Column 6 shows the test time obtained using the new wrapper cells and considering the hierarchy present in the SOCs.

Column 7 shows the percentage (%) difference between the test times for Case 1 and this case (Case 3). Column 8 shows the required number of new wrapper cells (NWC). As a new wrapper cell requires two flip-flops instead of the one used in the conventional wrapper cell, this number can also be seen as the amount of area-overhead (excluding area for multiplexers and wires) resulting from using the new proposed wrapper cells.

From the table, one can see that compared to the test times obtained considering flat cores (Case 1), if the design hierarchy present in the SOCs is considered and only the conventional wrapper cells are used (Case 2), an average increase of 113% (based on all cases shown in the table) in test time is obtained. For SOC p93791 with $w_{\max} = 40$, the penalty in test time is even more than 400%, which is not acceptable. As this increase is due to the design hierarchy, a design house will not appreciate a hierarchical-test development, no matter what benefits it may provide. Because sometimes, the design hierarchy cannot be overruled, one needs to find a good test solution.

Test time results for Case 3 show that with the use of the proposed wrapper cells, test times considering the design hierarchy can be comparable or better than the same design without hierarchy. From Column 7, one can see that for most cases, Case 3 obtained the same test times as for Case 1 (SOCs with flat cores). Surprisingly, for some cases the test times for Case 3 are even lower than that of Case 1. This is due to heuristic nature of TR-ARCHITECT. For SOC p93791 with $w_{\max} = 40$, by using 1875 new wrapper cells, one can decrease the penalty in test time from 460% (Case 2) to 2%. Figure 7.15 shows the design hierarchy present in SOC p93791. This SOC contains 32 cores, out of which *eight* are hierarchical cores. All hierarchical cores in this SOC contain embedded memory cores.

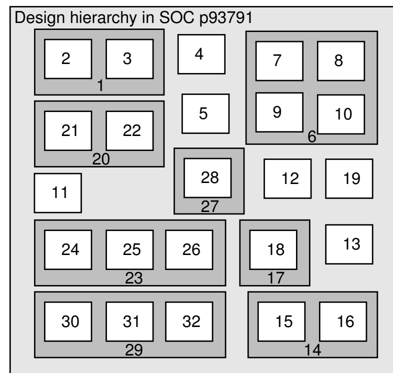
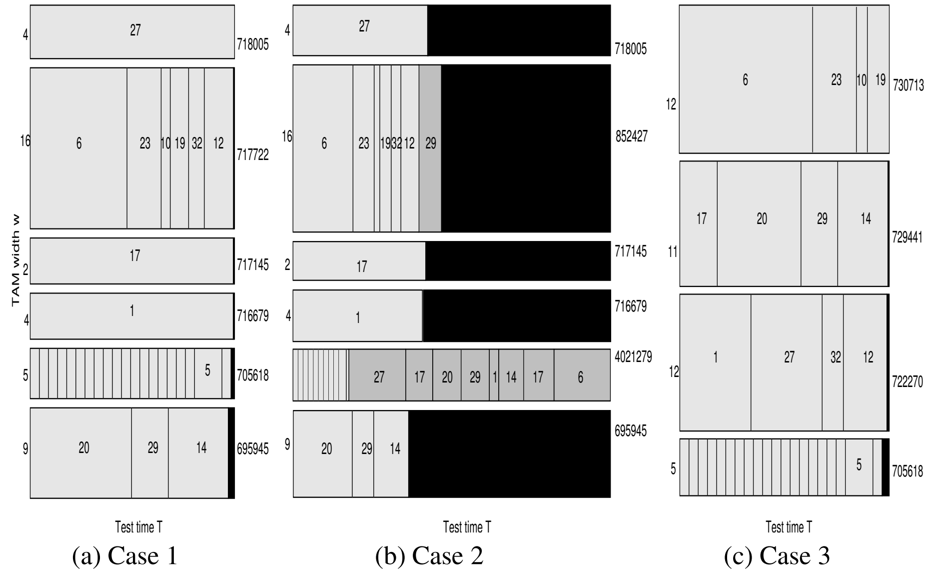


Figure 7.15: Conceptual view of the design hierarchy in SOC p93791.

For SOC p93791 with $w_{\max} = 40$, Figure 7.16 shows the test schedules obtained for all three cases. In Figure 7.16, the horizontal axis represents the test time (in the number of clock cycles), while the vertical axis represents the TAM width. The rectangle boxes represent the tests of various cores and the number inside a box represents the core *ID*. For very small boxes, this number is omitted. At the end of each TAM, the shown number represents the test time for the TAM. Please note that the schedules shown are not drawn to scale and idle time is shown at the end of a TAM only.



(a) Case 1 (b) Case 2 (c) Case 3

Figure 7.16: Test schedules for the three cases for SOC p93791 with $w_{\max} = 40$.

Figure 7.16(a) shows quite a good schedule if hierarchy is not considered. The reality comes into the picture with the test schedule shown in Figure 7.16(b). The dark grey boxes show the tests for the child cores with their wrappers in the Outward-facing mode for the parent core ID shown in the box. From this figure, one can see that TAM 5 contains a large number of child cores. Therefore, during testing of a parent core in other TAMs, corresponding child cores in this TAM are configured in the Outward-facing mode. Due to this, this TAM requires a large amount of time and determines the overall test time.

With the help of the new wrapper cells, the test schedule as shown in Figure 7.16(c) obtains the test time which is very close to the test time of the schedule shown in Figure 7.16(a). Please note that in Case 3, TR-ARCHITECT results in a new TAM assignment. This is due to the fact that during optimization itself, TR-ARCHITECT considers access to elements of both the parent and child cores for the test time calculation for parent cores. Therefore, due to its heuristic nature, TR-ARCHITECT provides a new TAM assignment. However, for the same TAM assignment as in Case 1, the use of new wrapper cells results in an overall test time of 738175 clock cycles, which shows an increase of 2.4% in test time.

7.7 Summary

Modern SOCs are designed by using IP cores which are usually delivered by several companies. Often, in-house designed cores contain one or more IP cores. Therefore, modern SOC designs are not limited to only one level of hierarchy (SOC and cores), instead they consist of multiple levels of hierarchy.

In this chapter, a generic hierarchical core model has been presented and various design scenarios that may exist in practice have been identified. Next, an improved wrapper architecture for hierarchical cores was presented. The improved wrapper architecture uses new types of wrapper cells in the wrapper of a child core. The use of new wrapper cells allows for parallel testing of parent and child cores provided they are connected to different TAMs. Therefore, all optimal test architecture design methods available in literature which assume only one level of hierarchy (SOC and cores) and proposes parallel testing of parent and child cores, can also be used without any modification for SOCs with multiple levels of hierarchy. The proposed wrapper architecture is compatible with the IEEE P1500 wrapper architecture and is actually an extension to it.

Experimental results for four SOCs taken from the *ITC'02 SOC Test Benchmarks* have been presented. It is shown that due to the multiple levels of design hierarchy present in the SOCs, optimal test schedules obtained using the conventional wrapper cells result in a test time penalty up to 400% when compared to the same for SOCs with only one level of hierarchy. It has been shown that by using the proposed wrapper architecture for SOCs with multiple levels of hierarchy, test schedules comparable or better than the test schedules for the same SOCs with one level of hierarchy can be obtained. The new wrapper cells do not have any impact on the functional performance of the core; however, they require extra silicon area. There is always a trade-off between test time and area-overhead. In this chapter, we have given higher preference to test time. The savings obtained in test times using the proposed wrapper cells justifies the small increment in the silicon area.

Chapter 8

Conclusions

Modern semiconductor design methods and manufacturing technologies enable the creation of a complete system on a one single die, the so called *System-On-Chip* or SOC. To reduce time-to-market for large SOCs, reuse of pre-designed and pre-verified blocks called *cores* is employed. This leads to a new core-based design paradigm. Like the design style, testing of SOCs can be best approached in a core-based fashion. In order to enable core-based test development, an embedded core should be isolated from its surrounding circuitry and electrical test access from chip pins to the core should be provided. Isolation of a core is done by designing a wrapper around the core, while the test access to the core is provided by means of a dedicated Test Access Mechanism (TAM).

One of the challenges while developing a core-based test solution for an SOC is to design the on-chip test architecture consisting of wrappers and TAMs in such a way, that it enables effective scheduling of the various core tests, and fits the total amount of test data onto the given ATE vector memory. The computational-time complexity of designing a test architecture increases exponentially with the number of cores and test pins. Therefore, for large real-life SOCs, there is a need for a tool which can efficiently search the solution space of feasible architectures and yield a (near-)optimal test architecture.

The objective of the work described in this thesis was to develop an automated tool that can assist SOC designers in selecting cost-effective test architectures and test schedules for their embedded-core based system chips. Therefore, in this thesis, the problem of effective and efficient test architecture design for core-based SOCs has been discussed. To design a test architecture for a SOC with a given number of cores and a given number of test pins, the SOC designer has to determine the following items:

1. the number of individual TAMs and their widths,
2. the assignment of cores to TAMs, and
3. a wrapper design for each core.

These parameters need to be selected in such a way that the total number of pins used for the architecture design does not exceed the given number of test pins, while the overall test cost is minimized.

In this thesis, the problem of wrapper design for a core with a given TAM width is tackled first. A wrapper is a thin shell around a core and provides both functional and test access to the core. A new wrapper architecture that unites the feature of the TestShell of Philips and the IEEE P1500 wrapper has been proposed in Chapter 2. In the proposed wrapper architecture, functional terminals of the core are connected to the TAM wires via wrapper cells, while the core-internal scan chains are directly connected to the TAM wires. The interconnection of wrapper cells and core-internal scan chains determine the test time of the core. Therefore, to design a wrapper around a core, one needs to partition the total number of scan chains and the wrapper cells over the given TAM wires in a balanced way.

In case of soft cores for which the *scan insertion* is yet to be carried out, the problem of wrapper design is very trivial and can be solved optimally. It is a result of the fact that for soft cores, the number and length of the core-internal scan chains can still be changed while creating the wrapper design. Hence, the total number of scan flip-flops and wrapper cells can be distributed optimally over the TAM wires. For hard cores, it is shown that the partitioning of scan chains and wrapper cells over TAM wires such that the test time is minimized, is equivalent to the $\mathcal{NP}$-hard problem of Multi-Processor Scheduling (MPS). In Section 2.5, various heuristic algorithms are described to solve the problem of wrapper design for hard cores. Experimental results for a set of cores with a range of TAM widths show that the heuristic COMBINE in combination with FFD on an average performs better than others. Therefore, for core test-wrapper design this heuristic is recommended and used in the later parts of the thesis.

Once a wrapper architecture and its design procedure is selected, the next step in designing a test architecture is to calculate the total number of TAMs, their widths and the assignment of cores to TAMs. This step is addressed in Chapter 3. As the test architecture design problem is $\mathcal{NP}$-hard, a lower bound on the SOC test time is very useful to measure the performance of any test architecture design algorithm. Based on the amount of test data that needs to be transported into and out of the SOC and the total available TAM width, an improved architecture-independent lower bound on the overall SOC test time is derived in Section 3.4. For SOCs with hard cores, the presented lower bound is more tight than the previously known lower bound; the new lower bound is almost always an improvement for SOCs with soft cores. In order to analyze why a lower bound cannot be achieved in practice, a classification of three types of idle bits that occur in practical schedules, is presented.

For test architecture design, a novel five-step heuristic algorithm TR-ARCHITECT is presented in Section 3.6. TR-ARCHITECT designs and optimizes test architectures with respect to the required ATE vector-memory depth and test-application time. TR-ARCHITECT optimizes wrapper and TAM design in conjunction. The first step of TR-ARCHITECT is CREATESTARTSOLUTION and its main purpose is to create an initial test architecture which will be further optimized by the steps to follow. In the next two steps, i.e. OPTIMIZE-BOTTOMUP and OPTIMIZE-TOPDOWN, it is tried to merge the

cores of two TAMs into a new TAM, such that the new TAM requires less wires; the wires that are freed-up in the process can be utilized for an overall test time reduction. In the step OPTIMIZE-BOTTOMUP, the TAM with the shortest test time is merged with another TAM, while in the step OPTIMIZE-BOTTOMUP, the TAM with the largest test time is merged with another TAM. The fourth step is RESHUFFLE. In this step, fine tuning of the test architecture is carried out by moving individual cores from the TAM with the maximum test time to other TAMs, provided that this reduces the overall test time. The last and fifth step is CHECK-EMPTYWIRES, in which the number of redundant wires (empty wires) are searched for all TAMs. If any empty wire is found, then it is tried to assign it to the TAM with the maximum test time in order to minimize the overall test time.

It has been shown that TR-ARCHITECT handles SOCs with hard and soft cores, works for test bus as well as TestRail TAMs, and supports both serial and parallel scheduling. Experimental results for the *ITC'02 SOC Test Benchmarks* shows that compared to the manual best-effort engineering approaches used within Philips, TR-ARCHITECT can save upto 75% in test time at negligible computational time. For all SOCs and for all TAM widths, the run time for TR-ARCHITECT was around one second. These large savings in test time at negligible computational time also emphasize the need for an automated optimization approach for designing SOC test architectures. Experimental results also show that TR-ARCHITECT test time results are comparable or better than the four other previously published approaches.

The original version of TR-ARCHITECT as presented in Chapter 3, has been stepwise extended to include the following practical constraints related to real-life SOC designs.

- To minimize the total wire length required for routing the TAM wires in the designed test architecture, layout-constraints have been included in TR-ARCHITECT in Chapter 4. To calculate the total wire length for a test architecture, a simple yet effective wire-length cost model has been presented. The wire-length cost model assumes that the layout positions of all cores in the SOC are given. The wire length of a TAM depends on the ordering of the cores connected to the TAM. Therefore, to minimize the wire length of a TAM, an optimal ordering of cores connected to the TAM has to be found. The problem of determining an optimal ordering of cores with respect to the wire length of the TAM has been shown equivalent to the well-known Traveling Salesman Problem (TSP) and a simple heuristic algorithm has been described to solve it.

  It has been shown that for a test architecture, minimization of test time and wire length should be done in conjunction. As minimization of these two costs can be conflicting, a weight-based cost function has been used in the layout-driven TR-ARCHITECT. The new layout-driven TR-ARCHITECT minimizes the total wire length by assigning neighboring cores as much as possible to the same TAM. Experimental results for several benchmark SOCs show that the layout-driven TR-ARCHITECT can save upto 85% in TAM wire length at an expense of less than 5% in test time.

- In Chapter 5, TR-ARCHITECT is extended to take the test control required for the test architecture into account. The term test control refers to setting appropriate test modes in the wrappers of all cores and the execution of cores tests. It has been shown that based on the nature of test-control signals, the test control can be divided into two categories: (1) pseudo-static test control, and (2) dynamic test control. Test-control signals can be generated in different ways and each of them provides a trade-off between the test time and area-overhead. Pseudo-static test control refers to setting test modes in cores wrappers between various test sessions. As these signals do not change very often, a shift-register based generation of signals is preferred for them. To account for the time required for test mode re-load, two test strategies have been presented.

  Dynamic test control refers to controlling the execution of cores tests, i.e. scan-enable control. As scan-enable signals change very often and the change-over time for them is very small, they should be provided by means of dedicated chip pins. In order to minimize the impact of scan-enable signals on the overall test time, a SOC test architecture design procedure should take into account the pins required for scan-enable signals. TR-ARCHITECT has been extended to take into account both test mode re-load time and the pins required for scan-enable signals. Experimental results for several benchmark SOCs show that the extended TR-ARCHITECT can save upto 40% in test time if compared to the results obtained from the original TR-ARCHITECT followed by the same test control settings.

- To include a wide range of user constraints which might be application- or design-specific and hence hard to generalize into general cost functions, a novel Test Architecture Specification (TAS) language has been described in Chapter 6. In a TAS file, a user can fully or partially specify test architecture parameters. TR-ARCHITECT has been extended to take into account the user constraints specified in an input TAS file. All parameters specified by the user are considered as constraints, while everything which is not specified, is left for the tool to optimize for minimal test time, TAM wire length, etc. This extension yields a whole spectrum of use scenarios for TR-ARCHITECT. The spectrum ranges from an empty user specification (in which the tool fully optimizes the resulting test architecture) to a full user specification (in which the tool only computes the corresponding test schedule and associated costs) and everything in between.

- Support for designing optimal test architectures for SOCs with hierarchical cores has been presented in Chapter 7. Most of the test architecture design algorithms (including TR-ARCHITECT) assume no hierarchy inside cores, i.e. even if there is a hierarchy among cores, all cores in a SOC are considered at the same level. To minimize SOC test time, all these methods allow parallel testing of cores which are hierarchically related (parent and child relation). However, this is not supported by the existing wrapper architectures. Therefore, test schedules obtained by these methods are not directly applicable to SOCs with hierarchical cores and therefore require modifications. To allow parallel testing and hence reuse of existing design algorithms, a new wrapper architecture for hierarchical cores has been presented. The presented wrapper architecture allows parallel

testing of cores which are hierarchically related. Therefore, optimal test schedules can be obtained for SOCs with hierarchical cores also. Experimental results for several benchmark SOCs show that if the design hierarchy present in the SOC is considered, the use of new wrapper architecture can save upto 400% in test time if compared to modified schedules with conventional wrapper architectures.

The work described in this thesis has lead to the creation of a Philips internal tool called TR-ARCHITECT. The first practical use of TR-ARCHITECT has been reported for the Philips PNX8550 SOC. The SOC PNX8550 contains more than 60 cores and has 280 test pins. TR-ARCHITECT was only available halfway the design trajectory of the PNX8550 SOC. Despite of that it helped to optimize the test architecture further within the given constraints and designers successfully managed to fit the test data onto the target ATE. Experimental results show that, if TR-ARCHITECT would have been available from the project start onwards, further test time reductions up to 50% would have been possible.

For future work, constraints like total power consumption during test, multiple clock domains, precedence between various cores tests (For example, tests of core A should be carried out before tests of core B), can be included in TR-ARCHITECT.

Bibliography

[ABF94] Miron Abramovici, Melvin A. Breuer, and Arthur D. Friedman. *Digital System Testing and Testable Design*. Wiley-IEEE Press, 1994.

[AM98] Joep Aerts and Erik Jan Marinissen. Scan Chain Design for Test Time Reduction in Core-Based ICs. In *Proceedings IEEE International Test Conference (ITC)*, pages 448–457, Washington, DC, October 1998.

[BA00] Michael L. Bushnell and Vishwani D. Agrawal. *Essentials of Electronic Testing of Digital, Memory and Mixed-Signal VLSI Circuits*. Kluwer Academic Publishers, Boston, 2000.

[BBG+98] Stefano Barbagallo, G. Borgonovo, D. Grassi, Davide Medina, Fulvio Corno, Paolo Prinetto, and Matteo Sonza Reorda. Scan Chain Partitioning and Re-ordering Based on Layout Information: An Industrial Experience. In *Proceedings Design, Automation, and Test in Europe (DATE) – Designers Track*, pages 123–127, Paris, France, March 1998.

[BBT95] Frans Beenker, Ben Bennetts, and Loek Thijssen. *Testability Concepts for Digital ICs - The Macro Test Approach*, volume 3 of *Frontiers in Electronics Testing*. Kluwer Academic Publishers, Boston, USA, 1995.

[BCC+00] Alfredo Benso, Silvia Chiusano, Stefano Di Carlo, Paolo Prinetto, Fabio Ricciato, Maurizio Spadari, and Yervant Zorian. HD^{2}BIST: A Hierarchical Framework for BIST Scheduling, Data Patterns Delivering and Diagnosis in SoCs. In *Proceedings IEEE International Test Conference (ITC)*, pages 892–901, Atlantic City, NJ, October 2000.

[BMM02] Mounir Benabdenbi, Walid Maroufi, and Meryem Marzouki. CAS-BUS: A Test Access Mechanism and a Toolbox Environment for Core-Based System Chip Testing. *Journal of Electronic Testing: Theory and Applications*, 18(4/5):455–473, August 2002.

[BR83] Zeev Brazilai and Barry K. Rosen. Comparison of AC Self-Testing Procedures. In *Proceedings IEEE International Test Conference (ITC)*, pages 89–94, October 1983.

[CBC00] Tapan J. Chakraborty, Sudipta Bhawmik, and Chen-Huan Chiang. Test Access Methodology for System-On-Chip Testing. In *Digest of Papers of IEEE International Workshop on Testing Embedded Core-Based Systems (TECS)*, pages 1.1–1–7, Montreal, Canada, May 2000.

[Cha00a] Krishnendu Chakrabarty. Design of System-on-a-Chip Test Access Architectures Under Place-and-Route and Power Constraints. In *Proceedings ACM/IEEE Design Automation Conference (DAC)*, pages 432–437, Los Angeles, CA, June 2000.

[Cha00b] Krishnendu Chakrabarty. Design of System-on-a-Chip Test Access Architectures Using Integer Linear Programming. In *Proceedings IEEE VLSI Test Symposium (VTS)*, pages 127–134, Montreal, Canada, April 2000.

[Cha01] Krishnendu Chakrabarty. Optimal Test Access Architectures for System-on-a-Chip. *ACM Transactions on Design Automation of Electronic Systems*, 6(1):26–49, January 2001.

[CLH96] Chau-Shen Chen, Kuang-Hui Lin, and TingTing Hwang. Layout Driven Selecting and Chaining of Partial Scan Flip-Flops. In *Proceedings ACM/IEEE Design Automation Conference (DAC)*, pages 262–267, Las Vegas, NV, June 1996.

[CT97] Sreejit Chakravarty and Paul J. Thadikaran. *Introduction to IDDQ Testing*. Kluwer Academic Publishers, Boston, 1997.

[DJR01] Santanu Dutta, Rune Jensen, and Alf Rieckmann. VIPER: A Multiprocessor SOC for Advanced Set-Top Box and Digital TV Systems. In *IEEE Design & Test of Computers*, pages 21–31, 2001.

[Dow96] Shane Dowd. Post Placement Scan Chain Re-Ordering. *Electronic Product Design*, 17(10):33–38, October 1996.

[DZW+03] Francisco DaSilva, Yervant Zorian, Lee Whetsel, Karim Arabi, and Rohit Kapur. Overview of the IEEE P1500 Standard. In *Proceedings IEEE International Test Conference (ITC)*, pages 988–997, Charlotte, NC, September 2003.

[EI01] Zahra Sadat Ebadi and Andre Ivanov. Design of an Optimal Test Access Architecture Using a Genetic Algorithm. In *Proceedings IEEE Asian Test Symposium (ATS)*, pages 205–210, Kyoto, Japan, November 2001.

[Eld59] R.D. Eldred. Test Routines Based On Symbolic Logical Statements. *Journal of the ACM*, 6(1):33–36, January 1959.

[FL86] Donald K. Friesen and M.A. Langston. Evaluation of a MULTIFIT-Based Scheduling Algorithm. *Journal of Algorithms*, 7:35–59, 1986.

[Fri84] Donald K. Friesen. Tighter Bounds for the Multifit Processor Scheduling Algorithm. *SIAM Journal of Computing*, 13(1):170–181, February 1984.

[GCM+04] Sandeep Kumar Goel, Kuoshu Chiu, Erik Jan Marinissen, Toan Nguyen, and Steven Oostdijk. Test Infrastructure Design for the Nexperia™ Home Platform PNX8550 System Chip. In *Proceedings Design, Automation, and Test in Europe (DATE)*, pages 108–113, Paris, France, February 2004.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability - A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.

[GM01] Sandeep Kumar Goel and Erik Jan Marinissen. TAM Architectures and Their Implication on Test Application Time. In *Digest of Papers of IEEE International Workshop on Testing Embedded Core-Based Systems (TECS)*, pages 3.3–1–10, Marina del Rey, CA, May 2001.

[GM02a] Sandeep Kumar Goel and Erik Jan Marinissen. A Novel Test Time Reduction Algorithm for Test Architecture Design for Core-Based System Chips. In *Proceedings IEEE European Test Workshop (ETW)*, pages 7–12, Corfu, Greece, May 2002.

[GM02b] Sandeep Kumar Goel and Erik Jan Marinissen. Cluster-Based Test Architecture Design for System-on-Chip. In *Proceedings IEEE VLSI Test Symposium (VTS)*, pages 259–264, Monterey, CA, April 2002.

[GM02c] Sandeep Kumar Goel and Erik Jan Marinissen. Effective and Efficient Test Architecture Design for SOCs. In *Proceedings IEEE International Test Conference (ITC)*, pages 529–538, Baltimore, MD, October 2002.

[GM03a] Sandeep Kumar Goel and Erik Jan Marinissen. A Test Time Reduction Algorithm for Test Architecture Design for Core-Based System Chips. *Journal of Electronic Testing: Theory and Applications*, 19(4):425–435, August 2003.

[GM03b] Sandeep Kumar Goel and Erik Jan Marinissen. Control-Aware Test Architecture Design for Modular SOC Testing. In *Proceedings IEEE European Test Workshop (ETW)*, pages 57–62, Maastricht, The Netherlands, May 2003.

[GM03c] Sandeep Kumar Goel and Erik Jan Marinissen. Layout-Driven SOC Test Architecture Design for Test Time and Wire Length Minimization. In *Proceedings Design, Automation, and Test in Europe (DATE)*, pages 738–743, Munich, Germany, March 2003.

[GM03d] Sandeep Kumar Goel and Erik Jan Marinissen. SOC Test Architecture Design for Efficient Utilization of Test Bandwidth. *ACM Transactions on Design Automation of Electronic Systems*, 8(4):399–429, October 2003.

[GM04a] Sandeep Kumar Goel and Erik Jan Marinissen. *Test Resource Management and Scheduling for Modular Manufacturing Test of SOCs*. In Wim

Verhaegh, Emile Aarts, Jan Korst (eds.), ‘Algorithms in Ambient Intelligence’, Philips Research Book Series, Volume 2, Kluwer Academic Publishers, The Netherlands, 2004.

[GM04b] Sandeep Kumar Goel and Erik Jan Marinissen. TR-Architect: DfT and Test Support for SOC Designers. In *Proceedings of the IEEE/ProRISC Workshop on Circuits, Systems and Signal Processing*, pages 30–38, Veldhoven, The Netherlands, November 2004.

[Goe04] Sandeep Kumar Goel. A Novel Wrapper Cell Design for Efficient Testing of Hierarchical Cores in System Chips. In *Proceedings IEEE European Test Symposium (ETS)*, pages 147–152, Ajaccio, Corsica, May 2004.

[Gra69] R.L. Graham. Bounds on Multiprocessing Anomalies. *SIAM Journal of Applied Mathematics*, 17:416–429, 1969.

[GV02] Sandeep Kumar Goel and Bart Vermeulen. Hierarchical Data Invalidation Analysis for Scan-Based Debug on Multiple Clock System Chips. In *Proceedings IEEE International Test Conference (ITC)*, pages 1103–1110, Baltimore, MD, October 2002.

[GZ97] Rajesh K. Gupta and Yervant Zorian. Introducing Core-Based System Design. *IEEE Design & Test of Computers*, 14(4):15–25, December 1997.

[HCT+01] Yu Huang, Wu-Tung Cheng, Chien-Chung Tsai, Nilanjan Mukherjee, Omer Samman, Yahya Zaidan, and Sudhakar M. Reddy. Resource Allocation and Test Scheduling for Concurrent Test of Core-Based SOC Design. In *Proceedings IEEE Asian Test Symposium (ATS)*, pages 265–270, Kyoto, Japan, November 2001.

[HM] Alan Hales and Erik Jan Marinissen. IEEE P1500 Web Site. http://grouper.ieee.org/groups/1500/.

[HRC+02] Yu Huang, Sudhakar M. Reddy, Wu-Tung Cheng, Paul Reuter, Nilanjan Mukherjee, Chien-Chung Tsai, Omer Samman, and Yahya Zaidan. Optimal Core Wrapper Width Selection and SOC Test Scheduling Based on 3-D Bin Packing Algorithm. In *Proceedings IEEE International Test Conference (ITC)*, pages 74–82, Baltimore, MD, October 2002.

[IC02] Vikram Iyengar and Krishnendu Chakrabarty. Test Bus Sizing for System-on-a-Chip. *IEEE Transactions on Computers*, 51:449–459, May 2002.

[ICKK03] Vikram Iyengar, Krishnendu Chakrabarty, Mark D. Krasniewski, and Gopind N. Kumar. Design and Optimization of Multi-level TAM Architectures for Hierarchical SOCs. In *Proceedings IEEE VLSI Test Symposium (VTS)*, pages 299–304, Napa, CA, April 2003.

[ICM01] Vikram Iyengar, Krishnendu Chakrabarty, and Erik Jan Marinissen. Test Wrapper and Test Access Mechanism Co-Optimization for System-on-Chip. In *Proceedings IEEE International Test Conference (ITC)*, pages 1023–1032, Baltimore, MD, October 2001.

[ICM02a] Vikram Iyengar, Krishnendu Chakrabarty, and Erik Jan Marinissen. Co-Optimization of Test Wrapper and Test Access Architecture for Embedded Cores. *Journal of Electronic Testing: Theory and Applications*, 18(2):213–230, April 2002.

[ICM02b] Vikram Iyengar, Krishnendu Chakrabarty, and Erik Jan Marinissen. Efficient Wrapper/TAM Co-Optimization for Large SOCs. In *Proceedings Design, Automation, and Test in Europe (DATE)*, pages 491–498, Paris, France, March 2002.

[ICM02c] Vikram Iyengar, Krishnendu Chakrabarty, and Erik Jan Marinissen. Integrated Wrapper/TAM Co-Optimization, Constraint-Driven Test Scheduling, and Tester Data Volume Reduction for SOCs. In *Proceedings ACM/IEEE Design Automation Conference (DAC)*, pages 685–690, New Orleans, LO, June 2002.

[ICM02d] Vikram Iyengar, Krishnendu Chakrabarty, and Erik Jan Marinissen. On Using Rectangle Packing for SOC Wrapper/TAM Co-Optimization. In *Proceedings IEEE VLSI Test Symposium (VTS)*, pages 253–258, Monterey, CA, April 2002.

[JDU+74] D.S. Johnson, A. Demers, J.D. Ullman, M.R. Garey, and R.L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal of Computing*, 3(1):299–325, 1974.

[JGJ78] E.G. Coffman Jr., M.R. Garey, and D.S. Johnson. An Application of Bin-Packing to Multiprocessor Scheduling. *SIAM Journal of Computing*, 7(1):1–17, February 1978.

[Joh73] D.S. Johnson. *Near-Optimal Bin Packing Algorithms*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1973.

[KB99] M. Keating and P. Bricaud. *Reuse Methodology Manual for System-on-a-chip Designs*. Kluwer Academic Publishers, Norwell, Massachusetts, 1999.

[KGM+04] Ludovic Krundel, Sandeep Kumar Goel, Erik Jan Marinissen, Marie-Lise Flottes, and Bruno Rouzeyre. User-Constrained Test Architecture Design for Modular SOC Testing. In *Proceedings IEEE European Test Symposium (ETS)*, pages 153–158, Ajaccio, Corsica, May 2004.

[KI02] Sandeep Koranne and Vikram Iyengar. On the Use of k-tuples for SoC Test Schedule Representation. In *Proceedings IEEE International Test Conference (ITC)*, pages 539–548, Baltimore, MD, October 2002.

[KKK+99] Rohit Kapur, Brion Keller, Bernd Koenemann, Maurice Lousberg, Paul Reuter, Tony Taylor, and Prab Varma. P1500-CTL: Towards a Standard Core Test Language. In *Proceedings IEEE VLSI Test Symposium (VTS)*, pages 489–490, Dana Point, CA, April 1999.

[KLT+01] Rohit Kapur, Maurice Lousberg, Tony Taylor, Brion Keller, Paul Reuter, and Douglas Kay. CTL – The Language for Describing Core-Based Test. In *Proceedings IEEE International Test Conference (ITC)*, pages 131–139, Baltimore, MD, October 2001.

[Kor02] Sandeep Koranne. A Novel Reconfigurable Wrapper for Testing of Embedded Core-Based SOCs and its Associated Scheduling Algorithm. *Journal of Electronic Testing: Theory and Applications*, 18(4/5):415–434, August 2002.

[Koz90] Dextor Kozen. On Kleene algebras and closed semirings. In *Rovan, Editor, Proceeding Mathematics Foundation of Computer Science, volume 452 of Lecture Notes in Computer Science*, pages 26–47. Springer (Online at http://www.cs.cornell.edu/kozen/papers/kacs.ps), 1990.

[LAFP02] Erik Larsson, Klas Arvidsson, Hideo Fujiwara, and Zebo Peng. Integrated Test Scheduling, Test Parallelization and TAM Design. In *Proceedings IEEE Asian Test Symposium (ATS)*, pages 397–404, Tamuning, Guam, USA, November 2002.

[LCH96] K.-H. Lin, C.-S. Chen, and T.-T. Hwang. Layout-Driven Chaining of Scan Flip-Flops. *IEE Proceedings - Computers and Digital Techniques*, 143(6):421–425, November 1996.

[LF03] Erik Larsson and Hideo Fujiwara. Test Resource Partitioning and Optimization for SOC Designs. In *Proceedings IEEE VLSI Test Symposium (VTS)*, pages 319–324, Napa, CA, April 2003.

[LM88] Chung-Yee Lee and J. David Massey. Multiprocessor Scheduling: Combining LPT and MULTIFIT. *Discrete Applied Mathematics*, 20:233–242, 1988.

[MAB+98] Erik Jan Marinissen, Robert Arendsen, Gerard Bos, Hans Dingemanse, Maurice Lousberg, and Clemens Wouters. A Structured And Scalable Mechanism for Test Access to Embedded Reusable Cores. In *Proceedings IEEE International Test Conference (ITC)*, pages 284–293, Washington, DC, October 1998.

[MG02] Erik Jan Marinissen and Sandeep Kumar Goel. Analysis of Test Bandwidth Utilization in Test Bus and TestRail Architectures for SOCs. In *Proceedings IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop (DDECS)*, pages 52–60, Brno, Czech Republic, April 2002.

[MGL00] Erik Jan Marinissen, Sandeep Kumar Goel, and Maurice Lousberg. Wrapper Design for Embedded Core Test. In *Proceedings IEEE International Test Conference (ITC)*, pages 911–920, Atlantic City, NJ, October 2000.

[MIC] Erik Jan Marinissen, Vikram Iyengar, and Krishnendu Chakrabarty. ITC'02 SOC Test Benchmarks Web Site. http://www.extra.research.philips.com/itc02socbenchm/.

[MIC02] Erik Jan Marinissen, Vikram Iyengar, and Krishnendu Chakrabarty. A Set of Benchmarks for Modular Testing of SOCs. In *Proceedings IEEE International Test Conference (ITC)*, pages 519–528, Baltimore, MD, October 2002.

[MKL+02] Erik Jan Marinissen, Rohit Kapur, Maurice Lousberg, Teresa L. McLaurin, Mike Ricchetti, and Yervant Zorian. On IEEE P1500's Standard for Embedded Core Test. *Journal of Electronic Testing: Theory and Applications*, 18(4/5):365–383, August 2002.

[ML99] Erik Jan Marinissen and Maurice Lousberg. The Role of Test Protocols in Testing Embedded-Core-Based System ICs. In *Proceedings IEEE European Test Workshop (ETW)*, pages 70–75, Konstanz, Germany, May 1999.

[NKTG97] Kazushi Nakamura, Susumu Kobayashi, Shigeyoshi Tawada, and Takeshi Goto. Scanpath's Wire Length Minimization and Its Short Path Error Correction. *NEC Research & Development Journal*, 38(1):22–27, January 1997.

[Pap94] C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley, New Jersey, 1994.

[Pin95] M. Pinedo. *Scheduling - Theory, Algorithms, and Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[SGMC04] Anuja Sehgal, Sandeep Kumar Goel, Erik Jan Marinissen, and Krishnendu Chakrabarty. P1500-Compliant Test Wrapper Design for Hierarchical Cores. In *Proceedings IEEE International Test Conference (ITC)*, page (To appear), Charlotte, NC, 2004.

[Smi85] G. L. Smith. Model for Delay Faults Based upon Paths. In *Proceedings IEEE International Test Conference (ITC)*, pages 342–349, November 1985.

[Smi97] Gary Smith. Test and System Level Integration. *IEEE Design & Test of Computers*, 14(4):19, December 1997.

[Soc99] IEEE Computer Society. IEEE Standard Test Interface Language (STIL) for Digital Test Vector Data - Language Manual - IEEE Std. 1450.0-1999. IEEE, New York, September 1999.

[Soc00] IEEE Computer Society. IEEE Standard Test Access Port and Boundary-Scan Architecture - IEEE Std. 1149.1-2000. IEEE, New York, 2000.

[SW02] Chih-Ping Su and Cheng-Wen Wu. Graph-Based Power-Constrained Test Scheduling for SOC. In *Proceedings IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop (DDECS)*, pages 61–68, Brno, Czech Republic, April 2002.

[VB98] Prab Varma and Sandeep Bhatia. A Structured Test Re-Use Methodology for Core-Based System Chips. In *Proceedings IEEE International Test Conference (ITC)*, pages 294–302, Washington, DC, October 1998.

[VSI96] Virtual Socket Interface Architectural Document. VSIA Alliance (http://www.vsi.org), November 1996.

[Wag85] K. D. Wagner. The Error Latency of Delay Faults in Combinational and Sequential Circuits. In *Proceedings IEEE International Test Conference (ITC)*, pages 334–341, November 1985.

[ZMD98] Yervant Zorian, Erik Jan Marinissen, and Sujit Dey. Testing Embedded-Core Based System Chips. In *Proceedings IEEE International Test Conference (ITC)*, pages 130–143, Washington, DC, October 1998.

[Zor97] Yervant Zorian. Test Requirements for Embedded Core-Based Systems and IEEE P1500. In *Proceedings IEEE International Test Conference (ITC)*, pages 191–199, Washington, DC, November 1997.

[ZRPH03] Wei Zou, Sudhakar M. Reddy, Irith Pomeranz, and Yu Huang. SOC Test Scheduling Using Simulated Annealing. In *Proceedings IEEE VLSI Test Symposium (VTS)*, pages 325–330, Napa, CA, April 2003.

Appendix A

List of Parameters

Here a list of all relevant parameters used in this thesis is presented.

- $\boldsymbol{w_{\max}}$: maximum number of TAM wires available for test architecture design
- $\boldsymbol{C}$: set of cores in an SOC
- For every core $\boldsymbol{c} \in \boldsymbol{C}$
  - $\boldsymbol{i_c}$: total number of functional input terminals
  - $\boldsymbol{o_c}$: total number of functional output terminals
  - $\boldsymbol{b_c}$: total number of functional bi-directional terminals
  - $\boldsymbol{p_c}$: total number of test patterns
  - $\boldsymbol{\mathcal{S}}$: set of scan chains
  - $\boldsymbol{l(S_i)}$: length of scan chain $S_i \in \mathcal{S}$
  - $\boldsymbol{s_c}$: total number of scan chains
  - $\boldsymbol{l_{c,k}}$: length of the kth scan chain
  - $\boldsymbol{f_c}$: total number of scan flip flops
  - $\boldsymbol{l_c^{\max}}$: length of the longest TAM chain in the core wrapper
  - $\boldsymbol{si_c}$: maximum scan-in time
  - $\boldsymbol{so_c}$: maximum scan-out time
  - $\boldsymbol{ts_c}$: total number of test stimuli bits per test pattern
  - $\boldsymbol{tr_c}$: total number of test response bits per test pattern
  - $\boldsymbol{t_c}$: total test time
  - $\boldsymbol{l_{\mathrm{WIR}}}$(c): length of the WIR in the wrapper of the core
- $\boldsymbol{T}$: total test time for the SOC

- $\boldsymbol{R}$: set of TAMs in a test architecture
- For every TAM $\boldsymbol{r} \in \boldsymbol{R}$
  - $\boldsymbol{r}$: set of cores that are connected to the TAM
  - $\boldsymbol{w(r)}$: width of the TAM
  - $\boldsymbol{t(r)}$: total test time for the TAM
  - $\boldsymbol{l_{\mathrm{WIR}}}$(r): added length of all WIRs in the wrappers of the cores connected to the TAM
- K: total number of chip pins available for test
- K_c: total number of chip pins required for test control
- L_{IR}: length of the instruction register in the boundary-scan architecture.

Appendix B

Computational Complexity

B.1 Introduction

Computational complexity analysis [Pap94] deals with the calculation of the resources required during computation to solve a given problem. The most common resources are *time* (how many steps does it take to solve a problem) and *space* (how much memory does it take to solve a problem). Complexity analysis differs from *computability* analysis [GJ79], which deals with whether a problem can be solved at all, regardless of the resources required.

The *computational-time complexity* of an algorithm is the number of steps that it takes to solve an instance of the problem, as a function of the *size of the input*. To avoid any implementation dependency, *Big O notation* is used. If for a problem of size n, an algorithm has a complexity of $\mathcal{O}(g(n))$ on one machine, then it will also have complexity of $\mathcal{O}(g(n))$ on most other machines.

Suppose $f(n)$ and $g(n)$ are two functions defined on some subset of real numbers, then the function $f(n)$ is $\mathcal{O}(g(n))$ if and only if

- increase of $f(n)$ is not faster than that of $g(n)$ or
- $\lim_{n\to\infty} f(n)/g(n)$ exits and finite or
- there exists a number n_0 and a non-negative M such that for all $n > n_0$, $0 \leq f(n) \leq Mg(n)$.

For example, lets consider that to solve a problem of size n, an algorithm $f(n)$ requires $5n^2 - 3n + 2$ steps. As n grows large, the term n^2 will start to dominate and all other terms can be neglected. Furthermore, the constants will depend on the precise details of the implementation and the hardware it runs on, so they should also be neglected as well. The big O notation captures what remains, i.e. $f(n) = \mathcal{O}(n^2)$ and it is concluded that the algorithm $f(n)$ has *order of* n^2 time complexity.

Based on the above mentioned theory, analysis of the computational-time complexity for all five steps of TR-ARCHITECT is described below. For the test architecture design problem, the inputs are the TAM width $w_{\max}$ and the set of cores C.

B.2 Creating a Start Solution

The small initialization step (Line 1) in the beginning of Algorithm 3.2 requires sorting of cores and can be implemented such that it requires $\mathcal{O}(|C| \log |C|)$ compute time. The other three steps are simple *for* loops and their actual computational complexities depend on the TAM width $w_{\max}$ and the number of cores $|C|$. It is clear that Step 1 will always be executed and only if $w_{\max} \neq |C|$, then either Step 2 or 3 will be executed, and hence $w_{\max} \neq |C|$ represents the worst-case scenario.

Now lets first consider the case where $w_{\max} > |C|$, then Step 1 will require $|C|$ iterations and Step 3 will require $(w_{\max} - |C|) \times |C|$ iterations. Therefore, the worst-case time complexity for this case can be written as $\mathcal{O}(|C| \log |C| + |C| + (w_{\max} - |C|) \times |C|)$. This can be approximated to $\mathcal{O}(w_{\max}|C|)$. Similarly, for the case where $w_{\max} < |C|$, the worst-case time complexity can be written as $\mathcal{O}(|C| \log |C| + w_{\max} + (|C| - w_{\max}) \times w_{\max})$, which again can be approximated to $\mathcal{O}(w_{\max}|C|)$. Therefore the overall worst-case computational-time complexity of the procedure CREATESTARTSOLUTION is $\mathcal{O}(w_{\max}|C|)$.

B.3 Optimize Bottom Up

The computational-time complexity of this procedure heavily depends on the number of TAMs created in the CREATESTARTSOLUTION step. In the worst-case, one can assume that every core is connected to a separate TAM, i.e. $|R| = |C|$. Finding a TAM with the minimum test time (Line 3) in Step 1 requires a linear search and can be done in $|C|$ iterations. Finding a merge candidate TAM (Lines 5–12) also requires $|C|$ iterations. Therefore, the worst-case time complexity of Step 1 can be written as $\mathcal{O}(|C| + |C|)$.

In Step 2, free wires obtained in Step 1 are distributed among the TAMs. This step is similar to Step 3 in CREATESTARTSOLUTION (Algorithm 3.2) and hence the worst-case time complexity of this step can be written as $\mathcal{O}(w_{\max}|C|)$. The case in which all the TAMs are merged into a single TAM represents the worst-case for the procedure OPTIMIZE-BOTTOMUP. Based on this, the overall worst-case computational-time complexity of OPTIMIZE-BOTTOMUP step can be written as $\mathcal{O}(|C| - 1 \times (|C| + |C| + w_{\max}|C|))$, which can be approximated to $\mathcal{O}(w_{\max}|C|^2)$.

B.4 Optimize Top Down

The computational-time complexity of this procedure depends on the number of TAMs passed from the OPTIMIZE-BOTTOMUP step. In the worst-case, one can assume that

there were no merging of TAMs possible in the OPTIMIZE-BOTTOMUP step, and hence the number of TAMs is equal to the number of cores, i.e. $|R| = |C|$. Step 1 of the OPTIMIZE-TOPDOWN procedure is similar to the procedure OPTIMIZE-BOTTOMUP, except that there are no free wires here. Therefore, the worst-case time complexity of Step 1 can be written as $\mathcal{O}(|C|^2)$.

In Step 2, finding a merged TAM and assigning its width using a linear search requires $\mathcal{O}(|C|w_{\max})$ compute time. Furthermore, the distribution of free wires among TAMs requires $\mathcal{O}(w_{\max}|C|)$ compute time. Therefore, the worst-case time complexity of Step 2 can be written as $\mathcal{O}(w_{\max}|C|^2)$. Based on the worst-case time complexities of Step 1 and Step 2, the worst-case computational-time complexity of the procedure OPTIMIZE-TOPDOWN can written as $\mathcal{O}(|C|^2 + w_{\max}|C|^2)$, which can be approximated to $\mathcal{O}(w_{\max}|C|^2)$.

B.5 Reshuffle

The worst-case computation scenario for this procedure will be the case in which all cores except one are moved from the TAM with the maximum test time to other TAMs. In the worst-case, the TAM with the maximum test time will contain $|C| - 1$ cores. Finding the TAM with the maximum test time requires $\mathcal{O}(|R|)$ compute time. Next, finding the core with the minimum test time in this TAM requires $\mathcal{O}(|C| - 1)$ compute time. Now moving this core from this TAM to another TAM requires $\mathcal{O}(|R|)$ compute time. Therefore, a single iteration (moving only one core) of this procedure require$\mathcal{O}(|R| + |C| - 1 + |R|)$ compute time. Therefore, moving $|C| - 2$ cores will require $\mathcal{O}((|C|-2) \times (|R|+|C|+|R|))$ compute time. Since, the number of TAMs $|R|$ can never be greater than the number of cores $|C|$, the worst-case computational-time complexity of the procedure RESHUFFLE can be approximated to $\mathcal{O}(|C|^2)$.

B.6 Checking Empty Wire

For Step 1, the case in which the search for Pareto-Optimal widths is carried out till every TAM has only one wire left represents the worst-case. Therefore, in the worst-case, searching for a Pareto-Optimal width for a single TAM requires $\mathcal{O}(w_{\max})$ compute time. Since the search is carried out for all TAMs, the worst-case time complexity of Step 1 can be written as $\mathcal{O}(|C|w_{\max})$.

In Step 2, the emptied wires from Step 1 are re-distributed among the TAMs to minimize the overall test time. This step is similar to Step 3 in CREATESTARTSOLUTION (Algorithm 3.2) and hence the worst-case time complexity of this step can be written as $\mathcal{O}(w_{\max}|C|)$. In Step 3, the search for the Pareto-Optimal width is carried out only for the TAM with the maximum test time and hence its worst-case time complexity is $\mathcal{O}(w_{\max})$. Therefore, the worst-case computational-time complexity of the procedure CHECK-EMPTYWIRE can be written as $\mathcal{O}(|C|w_{\max} + |C|w_{\max} + w_{\max})$, which can be approximated to $\mathcal{O}(|C|w_{\max})$.

Summary

Advances in the semiconductor process technology enable the creation of a complete system on one single die, the so-called *system chip* or SOC. To reduce time-to-market for large SOCs, reuse of pre-designed and pre-verified blocks called *cores* is employed. Like the design style, testing of SOCs can be best approached in a core-based fashion. In order to enable core-based test development, an embedded core should be isolated from its surrounding circuitry and electrical test access from chip pins to the core should be provided. Isolation of a core is done by designing a wrapper around the core, while the test access to the core is provided by means of a dedicated Test Access Mechanism (TAM).

To design a test architecture for a SOC with a given number of cores and a given number of test pins, the SOC designer has to determine the following: (1) the number of individual TAMs, (2) TAMs widths, (3) the assignment of cores to TAMs, and (4) wrapper design for each core. These parameters need to be chosen such that the total number of pins used for the TAM wires does not exceed the given number of test pins, while the total amount of test data fits onto the given Automatic Test Equipment (ATE) and the SOC test time is minimized. The complexity of designing an architecture increases exponentially with the number of cores and test pins. For a small SOC, having only a few cores and a few test pins, a good test architecture can be designed manually. However, for large real-life SOCs such as the Philips PNX8550 that contains more than 60 logic cores and 280 test pins, there is a need for a tool which can efficiently search the solution space of feasible architectures and yield a (near-)optimal test architecture.

To assist SOC designers in selecting cost-effective test architectures and test schedules for their embedded-core based system chips, such a tool, named TR-ARCHITECT has been developed inside Philips. This thesis describes parts of the research work that has been carried out at Philips Research Laboratories, Eindhoven, related to the development of TR-ARCHITECT. TR-ARCHITECT uses a five-step heuristic algorithm to design a test architecture. TR-ARCHITECT designs and optimizes test architectures with respect to the required ATE vector-memory depth and test-application time. TR-ARCHITECT optimizes wrapper and TAM design in conjunction. Experimental results for the *ITC'02 SOC Test Benchmarks* show that, compared to manual best-effort engineering approaches that are being used within Philips, TR-ARCHITECT can save up to 75% in test time at negligible computation time. The original version of TR-ARCHITECT has been step-wise extended to include various practical constraints related to real-life SOC designs. The included constraints are layout, test control, user-

specific, and design hierarchy.

First practical use of TR-ARCHITECT was reported for the Philips PNX8550 SOC. TR-ARCHITECT was only available halfway the design trajectory of PNX8550 SOC. Despite of that it helped to optimize the test architecture further within the given constraints and we successfully managed to fit the test date onto the target ATE. Experimental results show that, if TR-ARCHITECT would have been available from the project start onwards, further test time reductions up to 50% would have been possible.

Samenvatting

De vooruitgang in de halfgeleider technologie heeft geleid tot de realisatie van een compleet systeem op een plakje silicium, de zogenoemde systeem chip of kortweg SOC. Grote SOCs maken gebruik van vooraf ontworpen en geverifieerde blokken genaamd cores om de “time-to-market” te verkorten. Zoals de ontwerpstijl kan ook het testen van SOCs het best worden benaderd op een core-gebaseerde manier. Om core-gebaseerd testen mogelijk te maken dient een core geisoleerd te worden van zijn omgeving en elektrisch toegankelijk te zijn vanaf de pinnen van de chip. Een core wordt geisoleerd door het ontwerpen van een omhulling, terwijl de test toegang mogelijk wordt gemaakt doormiddel van een Test Toegang Mechanisme (TAM).

Voor een SOC met een gegeven aantal cores en test pinnen dient de SOC ontwerper bij het ontwerp van een test architectuur de volgende parameters te bepalen: (1) het aantal afzonderlijke TAMs, (2) de breedte van een TAM, (3) de toewijzing van cores aan TAMs, en (4) het ontwerp van de omhulling voor elke core. Deze parameters dienen zodanig gekozen te worden dat het aantal pinnen die nodig zijn voor de TAM verbindingen niet het gegeven aantal test pinnen overschrijdt, terwijl de totale hoeveelheid test data past op het gegeven Automatische Test Apparatuur (ATE) en de SOC test tijd is geminimaliseerd. De complexiteit van het test architectuur ontwerp neemt exponentieel toe met een toenemend aantal cores en test pinnen. Voor een kleine SOC, dat slechts enkele cores en test pinnen bevat, kan een goede test architectuur handmatig worden ontworpen. Echter voor grote SOCs bestaat de behoefte voor automatisering om efficint te kunnen zoeken binnen de oplossingsruimte van bruikbare architecturen voor het verkrijgen van een (sub-)optimale test architectuur. Een voorbeeld van een grote SOC is de Philips PNX8550 dat meer dan 60 logica cores en 280 test pinnen bevat.

Binnen Philips Research Eindhoven is het tool TR-ARCHITECT ontwikkeld om SOC ontwerpers te assisteren in hun keuze voor een kosteneffectieve test architectuur en test schema’s voor hun core-gebaseerde systeem chips. Dit proefschrift beschrijft een gedeelte van het onderzoek gerelateerd aan de ontwikkeling van TR-ARCHITECT. TR-ARCHITECT maakt gebruik van een vijf-staps heuristiek algoritme om een test architectuur te ontwerpen. TR-ARCHITECT ontwerpt en optimaliseert test architecturen voor de diepte van het benodigde ATE vectorgeheugen en de testapplicatie tijd. Het ontwerp van omhulling en TAM wordt door TR-ARCHITECT gelijktijdig geoptimaliseerd. Experimentele resultaten voor de ITC’02 SOC Test Benchmarks laten zien dat TR-ARCHITECT tot 75tijd kan besparen bij een verwaarloosbare rekentijd

in vergelijking met de handmatige best-effort technieken die worden gebruikt binnen Philips. De oorspronkelijke versie van TR-ARCHITECT is stapsgewijs uitgebreid om een aantal praktische limitaties van realiseerbare SOC ontwerpen mee te nemen. De opgenomen beperkingen zijn layout, test controle, gebruikerspecifieke zaken en de hirarchie van het ontwerp.

De eerste toepassing van TR-ARCHITECT was gerapporteerd voor de PNX8550 SOC. TR-ARCHITECT was slechts beschikbaar vanaf de helft van het PNX8550 SOC ontwerp traject. Desondanks is de bestaande test architectuur verder geoptimaliseerd binnen de gestelde beperkingen en hebben we de test data passend kunnen maken voor de specifieke ATE. Volgens experimentele resultaten was er een verdere besparing tot 50% in test tijd mogelijk, indien TR-ARCHITECT vanaf het begin van het project beschikbaar was geweest.

About the Author

Sandeep Kumar Goel was born on 31st of August 1976 in a small city called Meerut in Uttar Pradesh, India. In 1998, he obtained his bachelor's degree (B.Tech.) in Electronics and Tele-communication from Institute of Engineering & Technology (IET), Lucknow, India. In 1998, he joined master's program (M.Tech.) in VLSI Design Tools and Technology (VDTT) at Indian Institute of Technology (IIT), Delhi, India. During the master's program, he was sponsored by Philips Semiconductors, The Netherlands. His master's project was also partially carried out at Philips Research Laboratories, Eindhoven, The Netherlands. In 1999, he received his master's degree.

Since 2000, he is a member of scientific staff in the group Digital Design & Test at Philips Research Laboratories, Eindhoven, The Netherlands. His research interests include all topics in the domains of test and debug of digital VLSI circuits. He has published several papers in the field of core-based testing and debugging of large SOCs. He is a member of the IEEE and the IEEE computer society.

Current address

Sandeep Kumar Goel
WAY 41, Philips Research Laboratories
Prof. Holstlaan 4, 5656AA
Eindhoven, The Netherlands

E-mail: sandeepkumar.goelphilips.com