Reconfigurable On-Chip Instrument Access Networks

Analysis, Design, Operation, and Application

Ghani Zadegan, Farrokh

2017

Document Version:
Publisher's PDF, also known as Version of record

Reconfigurable On-Chip Instrument Access Networks

Analysis, Design, Operation, and Application

Farrokh Ghani Zadegan

LUND UNIVERSITY

Doctoral Thesis
Electrical Engineering
Lund, March 2017

Farrokh Ghani Zadegan
Department of Electrical and Information Technology
Electrical Engineering
Lund University
P.O. Box 118, 221 00 Lund, Sweden

Series of licentiate and doctoral theses
ISSN 1654-790X; No. 94
ISBN 978-91-7753-026-8 (print)
ISBN 978-91-7753-027-5 (pdf)

Printed in Sweden by Tryckeriet i E-huset, Lund University, Lund.

To my family

Popular Science Summary

Electronic systems have become inseparable parts of our lives. They are used in home appliances, cars, medical equipments, credit cards, etc. An electronic system is typically composed of a circuit board (or a number of circuit boards) where each board hosts some components such as integrated circuits (also known as ICs or chips). An IC contains a number of *transistors*: components that are basic building blocks of electronic circuits. Developments in the manufacturing technology has enabled dramatic reduction in the size of transistors and the interconnects between them, thereby making it possible to produce complex ICs hosting up to a few billion transistors. Such complexity has brought about many challenges for the designers of electronic systems:

- In the design phase, it is likely that errors, commonly known as *bugs*, are introduced into the systems. The more the complexity becomes, the more advanced the *debugging* techniques should be.
- The manufacturing process is not perfect, and therefore, each electronic product should be tested for defects. To keep the test cost for complex systems low, test techniques should be constantly enhanced.
- As transistors and interconnects have become much smaller, the correct operation of modern electronic systems has become more sensitive to environmental conditions (such as cosmic rays) and *aging* (which is the unwanted gradual change in system behavior as time goes by). Such increased sensitivity causes malfunctions and necessitates constant monitoring of systems during their lifetime.

Designers of electronic systems found the solution to many of these issues to be the embedding of the debugging, testing, and monitoring instruments

into the IC itself: the so called *on-chip instruments*. As benefits of on-chip instruments become increasingly evident, more advanced instruments and higher number of them are embedded into the new generations of ICs. Accessing such abundance of instruments efficiently (with respect to time) requires proper on-chip infrastructure, referred to as *network*.

In this thesis, we address the design of reconfigurable on-chip instrument access networks that allow for efficient access to on-chip instruments. In particular, to speed up detection of malfunctions in systems at run-time, we propose special networks that are able to reconfigure themselves automatically such that the monitoring instruments that have detected errors are quickly accessed. Moreover, operation of on-chip networks requires automation tools that translate human-readable commands to streams of data applicable to on-chip networks. We, therefore, propose methods to help in performing the translation such that transfer time of the generated data stream is minimized.

Abstract

The constant need for higher performance and more advanced functionality has made the design and manufacturing of modern electronic chips highly demanding. Moreover, the use of smaller transistors in modern chips has increased their sensitivity to aging and faults, hence the need to constantly monitor the correct operation of these chips. To address the challenges and requirements, it has become common to embed extra hardware modules in the chips to assist in the design and manufacturing processes, as well as in monitoring the correct operation of the chips. Such modules, commonly referred to as on-chip instruments, are used through the entire life cycle of the chip, from the early prototyping phase to when the system incorporating that chip becomes operational at the customer's site.

The increasing trend in the number and complexity of the on-chip instruments called for methodologies that allow for scalable, fast, and easy access to these instruments. As an alternative to in-house methods, which although effective might be expensive to maintain, two IEEE standards, namely, IEEE Std 1687 and IEEE Std 1149.1-2013, have recently come into existence. These standards provide a common base for describing reconfigurable on-chip instrument access networks, as well as for describing the operation of each embedded instrument by using high-level description languages. Such common base motivates the development of relevant design automation tools, and facilitates the integration of instruments developed by multiple vendors. These standards, however, have left the arising optimization problems in the design and operation of such networks to be addressed by the electronic design automation community.

In this thesis, we address some of these optimization problems whose objective is to minimize the instrument access time, i.e., the time it takes to transport data to/from the instruments over reconfigurable on-chip instru-

ment access networks. In particular, we present access time analysis that helps to determine the contributing factors to the access time overhead. Using the analysis, we present methods for design of reconfigurable networks that are optimized with respect to instrument access time. Moreover, to operate such on-chip networks, there is a need for automation tools that translate (*retarget*) high-level descriptions of instrument access procedures specified at instruments' boundaries, into low-level description languages or bit vectors applicable from the chip's boundary. The reconfigurability of these networks, makes it challenging to perform the retargeting such that the generated vectors are optimized with respect to the time it takes to apply them to the chip. In this thesis, we explore opportunities for optimization in retargeting. In particular, we present a method to assist in optimal bit vector generation, by reducing the solution space without removing the optimal vector from it. Finally, considering the application of on-chip networks in in-field monitoring of the correct operation of chips, we propose a self-reconfiguring network that upon detection of errors, automatically reconfigures itself to reduce the time it takes to identify the faulty resources.

Preface

This thesis sums up my research work between the years 2010 and 2017. Part of this work was carried out when I worked at the Embedded Systems Laboratory (ESLAB), Department of Computer and Information Science (IDA), Linköping University, from 2010 to 2013, and the rest was done from 2014 to 2017 at the Digital ASIC Group, Department of Electrical and Information Technology (EIT), Lund University.

The material presented in this thesis is based on the following publications.

JOURNAL ARTICLES

- F. Ghani Zadegan, U. Ingelsson, G. Carlsson, and E. Larsson, "Access time analysis for IEEE P1687", IEEE Transactions on Computers, Vol. 61, No. 10, pp. 1459–1472, 2012.

  ► *I had the main responsibility for the analysis, the proposed algorithms, the corresponding implementations required for the experiments, and the writing.*

- F. Ghani Zadegan, U. Ingelsson, G. Carlsson, and E. Larsson, "Reusing and retargeting on-chip instrument access procedures in IEEE P1687", IEEE Design and Test of Computers, Vol. 29, No. 2, pp. 79–88, 2012.

  ► *I provided the code listings for the examples presented in the article, and did the writing.*

- F. Ghani Zadegan, D. Nikolov, and E. Larsson, "On-chip fault monitoring, using self-reconfiguring IEEE 1687 networks", submitted to IEEE Transactions on Computers.

► *I assisted in the development of the analysis and the design method. I carried out the hardware implementations and was responsible for the writing.*

- F. Ghani Zadegan, R. Krenz-Baath, and E. Larsson, "An upper-bound computation method for optimal retargeting in IEEE 1687 networks", submitted to IEEE Transactions on Computers.

  ► *I had the main responsibility for the implementation of the proposed techniques, development of the benchmarks, performing the experiments, and the writing.*

- F. Ghani Zadegan, G. Carlsson, and E. Larsson, "Analysis and design of reconfigurable on-chip networks for multiple access schedules", to be submitted to IEEE Transactions on Computers.

  ► *I had the main responsibility for the presented analysis and design methods, for performing the experiments, and for the writing.*

PEER-REVIEWED CONFERENCE PAPERS

- F. Ghani Zadegan, U. Ingelsson, G. Carlsson, and E. Larsson, "Test time analysis for IEEE P1687", Asian Test Symposium (ATS), 2010.

  ► *I had the main responsibility for the analysis, the proposed algorithms, the corresponding implementations required for the experiments, and the writing.*

- F. Ghani Zadegan, U. Ingelsson, G. Carlsson, and E. Larsson, "Design automation for IEEE P1687", Design, Automation, & Test in Europe (DATE), 2011.

  ► *I devised the heuristics for optimized network design, performed the experiments, wrote the paper, and did the presentation at the conference.*

- F. Ghani Zadegan, U. Ingelsson, G. Asani, G. Carlsson, and E. Larsson, "Test scheduling in an IEEE P1687 environment with resource and power constraints", Asian Test Symposium (ATS), 2011.

► *I had the main responsibility for the writing, as well as for the access time analysis and the corresponding implementation required for the experiments.*

- K. Petersen, D. Nikolov, U. Ingelsson, G. Carlsson, F. Ghani Zadegan, and E. Larsson, "Fault injection and fault handling: an MPSoC demonstrator using IEEE P1687", International On-Line Testing Symposium (IOLTS), 2014.

  ► *I helped with the integration, as well as hardware-software co-simulation of the proposed demonstrator.*

- F. Ghani Zadegan, G. Carlsson, and E. Larsson, "Robustness of TAP-based scan networks", International Test Conference (ITC), 2014.

  ► *I had the main responsibility for the presented analysis, performing the experiments, and the writing. I also presented the paper at the conference.*

- R. Krenz-Baath, F. Ghani Zadegan, and E. Larsson, "Access time minimization in IEEE 1687 networks", International Test Conference (ITC), 2015.

  ► *I contributed to the writing, to the computation of upper-bound, and to the development of benchmarks for the experiments.*

- F. Ghani Zadegan, D. Nikolov, and E. Larsson, "A self-reconfiguring IEEE 1687 network for fault monitoring", European Test Symposium (ETS), 2016.

  ► *I assisted in the development of the analysis and the design method, carried out the hardware implementations, was responsible for the writing, and did the presentation at the conference.*

- F. Ghani Zadegan, R. Krenz-Baath, and E. Larsson, "Upper-bound computation for optimal retargeting in IEEE 1687 networks", International Test Conference (ITC), 2016.

  ► *I had the main responsibility for the implementation of the proposed techniques, development of benchmarks, performing the experiments, and the writing. I presented the paper at the conference.*

INVITED PAPERS

The material in the above publications, which are included in this thesis, have also been presented in following invited papers:

- E. Larsson and F. Ghani Zadegan, "Accessing embedded DfT instruments with IEEE P1687", Asian Test Symposium (ATS), 2012.

- F. Ghani Zadegan, E. Larsson, A. Jutman, S. Devadze, and R. Krenz-Baath, "Design, verification, and application of IEEE 1687", Asian Test Symposium (ATS), 2014.

- E. Larsson and F. Ghani Zadegan, "Accessing on-chip instruments through the life-time of systems", Latin-American Test Symposium (LATS), 2016.

- F. Ghani Zadegan, D. Nikolov, and E. Larsson, "In-field system-health monitoring based on IEEE 1687", System-on-Chip Conference (SoCC), 2016.

PEER-REVIEWED PAPERS NOT INCLUDED IN THE THESIS

- R. Cantoro, M. Montazeri, M. Sonza Reorda, F. Ghani Zadegan, and E. Larsson, "On the testability of IEEE 1687 networks", Asian Test Symposium (ATS), 2015.

- R. Cantoro, M. Montazeri, M. Sonza Reorda, F. Ghani Zadegan, and E. Larsson, "Automatic generation of stimuli for fault diagnosis in IEEE 1687 networks", International Symposium on On-Line Testing and Robust System Design (IOLTS), 2016.

- A. Tšertov, A. Jutman, S. Devadze, M. Sonza Reorda, E. Larsson, F. Ghani Zadegan, R. Cantoro, M. Montazeri, and R. Krenz-Baath, "A suite of IEEE 1687 benchmark networks", International Test Conference (ITC), 2016.

The research work has been supported by the following:

- European Union's 7th Framework Programme's collaborative research project FP7-ICT-2013-11-619871 BASTION - Board and SoC Test Instrumentation for Ageing and No Failure Found

- European Union's 7th Framework Programme's collaborative research project FP7-2009-IST-4-248613 DIAMOND - Diagnosis, Error Modeling and Correction for Reliable Systems Design

- The Swedish National Graduate School in Computer Science (CUGS)

- Travel grants from the Royal Physiographic Society of Lund, and the Faculty of Engineering LTH

Acknowledgments

I would like to express my heartfelt gratitude to my main advisor, Prof. Erik Larsson, for encouraging me to start the PhD studies, and for his wholehearted support throughout these years. This thesis would not have been possible without his knowledge, guidance, and dedication.

It was a great opportunity to work with Dr. Urban Ingelsson during the early years of my research work. I acknowledge all the fruitful discussions we had and what we accomplished together.

During my PhD studies, I had the privilege of experiencing two research environments: the Embedded Systems Laboratory (ESLAB) in Linköping University and the Digital ASIC group in Lund University. I'm grateful to all friends and colleagues in these two places for creating such friendly working environments, and for providing valuable feedback on my work.

Special thanks to Prof. Zebo Peng, Prof. Mariam Kamkar, Prof. Nahid Shahmehri, and Anne Moe in Linköping University for their kind support, especially during the transition period to Lund University.

This thesis revolves entirely around IEEE Std 1687, which is an industrial standard. Without receiving advice from industrial experts, it would have been very difficult to carry out relevant research in this area. I am, therefore, deeply grateful to my co-advisor Gunnar Carlsson, from Ericsson AB, and Alfred L. Crouch, one of the main driving forces behind the IEEE 1687 standardization effort, for their valuable advice and feedback on my ideas.

I was fortunate to be part of two European projects, namely, DIAMOND and BASTION. I gained invaluable experience and met wonderful people from both academia and industry across Europe. In particular, I'd like to thank Prof. René Krenz-Baath from Hochschule Hamm-Lippstadt, for sharing his ideas on retargeting for IEEE 1687 networks and helping me painstakingly in their implementation. Also many thanks to Artur Jutman and his

colleagues in Testonica Lab for all the effort they put into the progress of these projects, and especially into the preparation of IEEE 1687 benchmarks. Further, I'm grateful to Prof. Matteo Sonza Reorda and Riccardo Cantoro from Politecnico di Torino for the rewarding collaboration on the test and diagnosis for IEEE 1687 networks.

Research work is not possible without the right tools and a well-functioning working environment. A big thank you to all the technical support staff and administrators at EIT, especially to Anne Andersson, Pia Bruhn, Erik Jonsson, Bertil Lindvall, Stefan Molund, Elisabeth Nordström, and Josef Wajnblom.

It was fun to share office and work with Dimitar Nikolov. Thank you Dimitar for helping me with my research problems, giving me feedback on my work, and all the good discussions we had.

I would like to thank Oskar Andersson, Rakesh Gangarajaiah, and Hemanth Prabhu, for helping me get the hang of the synthesis and place & route processes.

Many thanks to Mojtaba Mahdavi for his help and constructive feedback on the early draft of this thesis.

I would also like to thank all my friends who supported me kindly throughout these years. Special thanks to Breeta Sengupta, Dimitar Nikolov, Mehdi Fander, Mehrnaz Safaee, Amir Aminifar, Adrian Lifa, Arian Maghazeh, Nima Aghaee, and Ke Jiang, for the great time we shared.

I could have barely accomplished anything in my life without the love and support of my wonderful family, to whom I have dedicated this thesis: to the memory of my father, to my mother whom I cannot thank enough for her endless love and selfless dedication to our success, and to my brother and my sister, for being the amazing people that they are.

Farrokh Ghani Zadegan
Lund, March 2017

Contents

Acronyms and Abbreviations

ASIC	Application Specific Integrated Circuit
ATE	Automatic Test Equipment
BIST	Built-In Self-Test
CSU	Capture-Shift-Update
DFT	Design for Testability
EDA	Electronic Design Automation
EMR	Error code/Mask Register
FIFO	First In First Out
FSM	Finite State Machine
I2C	Inter-Integrated Circuit
IC	Integrated Circuit
ICL	Instrument Connectivity Language
IEEE	Institue of Electrical and Electronics Engineers
IJTAG	Internal JTAG
IR	Instruction Register
JTAG	Joint Test Action Group
LBIST	Logic Built-In Self-Test
MBIST	Memory Built-In Self-Test
OAT	Overall Access Time
PCB	Printed Circuit Board
PDL	Procedural Description Language
RTL	Register Transfer Level
SAT	Satisfiability (Boolean satisfiability problem)
SerDes	Serializer-Deserializer
SI	Scan Input
SIB	Segment Insertion Bit
SO	Scan Output

SoC	System-on-Chip
SRAM	Static Random-Access Memory
SVF	Serial Vector Format
TAP	Test Access Port
TCK	Test Clock
TCL	Tool Command Language
TDI	Test Data Input
TDO	Test Data Output
TDR	Test Data Register
TMS	Test Mode Select

List of Figures

List of Tables

1

Introduction

The advances in semiconductor technology have enabled manufacturing of increasingly complex integrated circuits (also known as ICs or chips) composed of up to a few billion transistors. Some of these ICs, referred to as system-on-chips (SoCs), host a complete system consisting of a number of general-purpose processors, digital signal processors, memories, accelerators, high-speed communications links, etc. The complexity of such SoCs has brought about tougher challenges for their designers in almost every stage in the life cycle of these SoCs: from verification efforts happening early on in the design flow, to the measures taken to ensure the correct in-field operation.

Firstly, no matter how carefully an SoC is designed, errors, commonly known as *bugs*, might find their ways into its design. Therefore, it is imperative that validation (and when necessary) *debugging* are performed on prototypes.

Moreover, the semiconductor manufacturing process is not perfect and some of the produced devices might be defective, making it mandatory to screen all products for such defects via *manufacturing tests*. Traditionally, automatic test equipments (ATEs) have been used to perform the manufacturing tests. These machines are now having a hard time providing the means for testing complex SoCs at reasonable prices [2, 3].

Additionally, as modern SoCs are manufactured using very small transistors, the electronic systems incorporating these SoCs are more prone to in-field malfunctioning, due to phenomena such as *soft errors*, *intermittent faults*, and *aging*. Soft errors are unwanted changes in the computation results or the state of a system as a result of external energy sources such as alpha particles or cosmic rays [4]. Intermittent faults result from defects that manifest themselves only in specific operational conditions such as an increase or a decrease in temperature and voltage [5]. Aging in electronic systems refers to gradual

changes in system properties that can result in malfunctioning. For example, a mechanism called *electromigration* can lead to open-circuit or short-circuit failures in metal interconnects in ICs [6]. Addressing the above-mentioned reliability issues has become crucial [7].

1.1. ON-CHIP INSTRUMENTS

The need for advanced debugging techniques has been addressed by embedding dedicated hardware modules, such as shadow registers, trace buffers, and hardware breakpoints, in the SoCs. These embedded debugging modules facilitate the process of finding hardware (as well as software) design errors. Similarly, over the years, engineers found the key to increasing testability (i.e., the quality of tests in detecting more defects) to be the use of embedded design-for-test (DFT) features. For example, built-in self-test (BIST) engines are now widely used for at-speed test of different blocks inside SoCs, such as memory blocks, logic blocks, and high-speed interconnects. These BIST engines are now equipped with configuration registers for choosing test algorithms that fit the current design best. Likewise, to enhance the in-field operation of electronic systems, they are equipped with many built-in monitors such as sensors for temperature and voltage drop, as well as error detection mechanisms.

The number, diversity, and complexity of these non-mission-mode modules have been gradually increasing ever since they came into existence. Nowadays, such embedded modules—referred to as on-chip *instruments*—are used to assist in tasks such as test, debug, configuration, and on-line monitoring. Here, we present examples of such instruments:

- Test: Memory BIST (MBIST) and logic BIST (LBIST) are widely used throughout prototype debugging and validation, manufacturing test, printed circuit board (PCB) assembly test [8], and power-on self-test. Core wrappers (e.g., those introduced by IEEE Std 1500 [9]) are used for core isolation and for hierarchical tests.

- Debug and validation: as examples, trace buffers, memory access, access to processor debug features, hardware breakpoints, shadow capturing of registers, overwriting registers [10, 11], and programmable random stimuli generators used to exercise the design-under-validation [12], can be mentioned.

- Configuration and calibration: programming fuse bits for disabling functional units [10], characterization of high-speed I/O (e.g., SerDes characterization [13]), and calibration of analog circuits [14] can be mentioned.

- Monitoring: Memory error detection [15], in-field checkers for assertions [16], temperature sensors, and reliability (aging and electromigration) monitors [17, 18] can be mentioned as examples.

1.2. MOTIVATION OF THE THESIS

As the number of on-chip instruments grew, it became clear that the traditional methods of accessing such instruments did not scale well with number of instruments [19]. Neither did those methods lend themselves well to electronic design automation (EDA). To address these issues, two standards came into existence: IEEE Std 1687 [20] and IEEE Std 1149.1-2013 [21]. These two standards laid the basis for scalable and automatable on-chip instruments access methods. More specifically, they propose

1. to use dynamically reconfigurable instrument access infrastructures—hereinafter *networks*—so that at any point in time, only those instruments that are needed become accessible, and

2. to describe instruments' operational procedures at their terminals by using a human readable language and let automation tools translate those procedures into bit vectors that are applicable at the IC's terminals. This translation is referred to as *retargeting*.

Both standards leave out the following challenges to be taken up by the EDA community:

- optimized design of instrument access network with respect to, e.g., area or access time (which is the time it takes to transport data to/from the instruments over on-chip reconfigurable access networks), and

- optimized retargeting algorithms such that translation of instrument operational procedures is done reasonably fast, and results in bit vectors with low application time, i.e., the time it takes to perform the procedures.

Regarding optimization of access time, it should be noted that in some scenarios such as test, configuration, and calibration, which happen as part of the manufacturing process, lowering the access time results in reduced costs, and is therefore very important. In the following section, the contributions of this thesis regarding the optimization problems in the area of reconfigurable on-chip instrument access networks are presented.

1.3. THESIS CONTRIBUTIONS

The contributions of this thesis can be summarized as follows:

- **Time analysis:** For solving an optimization problem, it is necessary to determine which variables (and each to what degree) affect the objective function. Optimizing the objective is then a matter of adjusting those variables. In this thesis, our main objective in all optimization problems is to minimize the instrument access time. Therefore, in order to determine the relevant variables, we provide time analyses and access time calculation methods for a number of reconfigurable network types.
- **Network design:** The use of reconfigurable instrument access networks opens up the opportunity for access time optimization. In this thesis, we present optimization methods for a number of reconfigurable network types.
- **Operation:** Optimal operation of reconfigurable access networks requires advanced retargeting tools. In retargeting for reconfigurable instrument access networks, finding bit vectors that are optimal with respect to application time is a hard problem, as too many candidate vectors exist. In this thesis, we improve the search by proposing (1) a method to explore the instrument access time reduction by exploiting the opportunities for concurrent access, and (2) a method to reduce the number of candidate vectors without eliminating the optimal vector from the solution space.
- **Application:** The use of reconfigurable networks to access on-chip instruments during in-field operation of electronic systems, for the purpose of fault monitoring, has been proposed [1, 22, 23]. In this thesis, we improve prior work w.r.t. fault localization time (i.e., the time it takes to identify the faulty resource and extract error information).

1.4. THESIS ORGANIZATION

The rest of this thesis is organized as follows. First, in Chapter 2, we review the basic concepts used throughout this thesis. In particular, reconfigurable on-chip access networks and the common off-chip to on-chip interfaces will be introduced.

In Part I, we detail three reconfigurable instrument access network types, namely, *SIB-based* networks, *Daisy-chained* networks, and *Remote* networks, and present access time calculation methods for each network type. We conclude Part I by presenting a parametric analysis, which helps us identify possibilities for access time reduction.

In Part II, based on the observations from the analysis in Part I, we present methods for construction of reconfigurable instrument access networks optimized with respect to access time.

In Part III, we focus on how reconfigurable instrument access networks can be efficiently operated. Part III consists of Chapter 5 and Chapter 6. In Chapter 5, we describe a basic retargeting flow and explore opportunities for optimization in it. In Chapter 6 we focus on a specific step in the retargeting flow and present a method to support the vector search for optimal retargeting.

In Part IV, we consider the application of reconfigurable networks in the area of in-field monitoring and fault management, and present our proposed self-reconfiguring networks that reduce fault localization time significantly over prior work.

Chapter 8 presents concluding remarks, as well as directions for future research in the area of this thesis work.

2

Background

In this chapter, the technical background needed to follow the discussion throughout this thesis is laid out. In particular, interfaces to on-chip instruments, reconfigurable networks for accessing on-chip instruments, and the related standards are introduced. Moreover, some key concepts, such as *retargeting*, which will be discussed in more details in later chapters, are introduced here.

2.1. INTERFACES TO ON-CHIP INSTRUMENTS

Given the widespread availability of IEEE Std 1149.1 (JTAG[1]) test access port (TAP) on majority of devices for the purpose of boundary-scan testing [24], this port has been traditionally used to access on-chip instruments, as well [10]. Other standard interfaces are also being used especially under stringent pin-count limitations, most notably the inter-integrated circuit (I2C) bus [25]. For example, I2C has been used to access [26], test [27], and calibrate [14] on-chip sensors. There are also devices that provide the access to on-chip instruments via multiple interfaces [28].

The TAP is ubiquitous and is the recommended (or the only) access port in the standards [20, 21, 9] related to on-chip instrument access networks. Therefore, in this thesis, it will be assumed that the TAP is the interface to the on-chip instrument access networks. In the rest of this section, we explain the TAP circuitry and operation as much as needed for the discussion in this thesis.

Figure 2.1 shows a conceptual view of the IEEE 1149.1 (hereinafter 1149.1) circuitry in a chip [29]. Two Test Data Registers (TDRs) are mandatory, namely,

[1]Joint Test Action Group

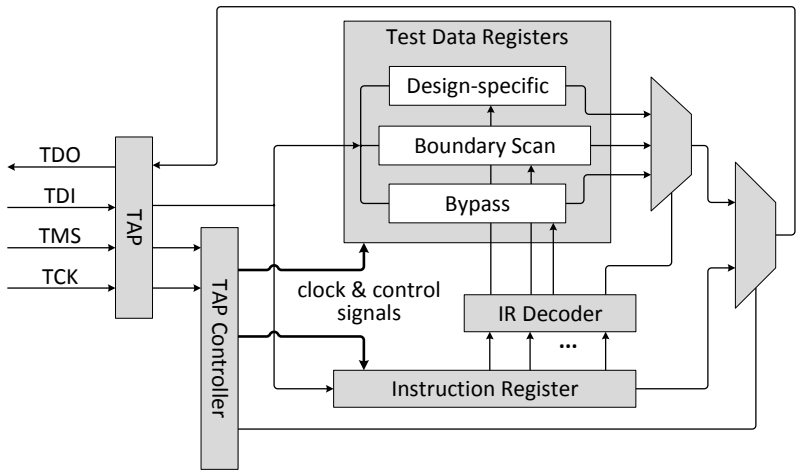


Figure 2.1. A conceptual view of IEEE 1149.1 circuitry

Boundary Scan Register and the Bypass Register. It is also possible to include a number of *design-specific* TDRs in the 1149.1 circuitry. 1149.1 uses a serial protocol, and at any time either the Instruction Register (IR) or one of the TDRs is accessible serially. Accessing the on-chip 1149.1 circuitry is done through the TAP, which includes four mandatory signals, namely test data input (TDI), test data output (TDO), test mode select (TMS), and test clock (TCK). The TMS signal is decoded by a state diagram (see Figure 2.2) to generate the control signals required for the *capture*, *shift* and *update* operations on IR and TDRs. The capture operation is defined as parallel loading a value into the IR (or any of the TDRs), the update operation is defined as transferring logic values from the shift-register stage of the IR (or any of the TDRs) to their latched parallel outputs, and the *shift* operation is defined as shifting the data serially into and out of the IR (or any of the TDRs) one bit per TCK.

In the following, it is explained how the above-mentioned control signals are generated and used to transfer data to and from the IR and TDRs. The state diagram in Figure 2.2 is implemented as a finite state machine in the TAP controller (Figure 2.1). The TAP controller state machine has two similar branches: the IR branch used for performing operations on the IR, and the DR branch used for performing operations on the currently selected TDR. In this thesis, the currently selected TDR will be referred to as the active TDR. The select signal of each TDR comes from the IR decoder, which activates the TDR corresponding to the instruction currently loaded into the IR. Since 1149.1 uses a serial protocol, input data for a TDR are transformed into a number of scan vectors (hereinafter vectors). Input vectors are shifted into

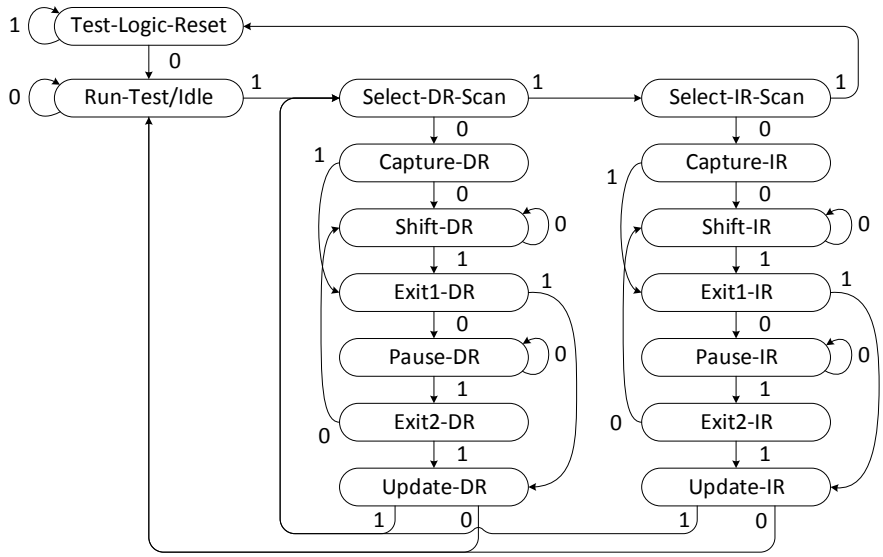


Figure 2.2. IEEE 1149.1 TAP Controller state diagram

the active TDR by shifting the data when the TAP controller is in the Shift-DR state. By keeping TMS at logic '0' it is possible to shift in as many bits as required for a vector. Moving to the Update-DR state makes the shifted vector appear at the parallel outputs of the TDR. Appendix B presents RTL circuitry for a 1149.1-style TDR. The data that should be parallel loaded into the TDR, i.e., the output vectors, are captured at the Capture-DR state and are shifted out by moving to the Shift-DR state. It is possible to shift in the next input vector while shifting out the output vector corresponding to the previous input vector. For applying inputs and capturing outputs between two shift operations, the Exit1-DR, Update-DR, Select-DR, and Capture-DR states are traversed in the state machine, which takes four TCKs. In this thesis, we refer to the process of shifting input vectors and going through update and capture operations as *vector application*.

2.2. NON-RECONFIGURABLE INSTRUMENT ACCESS NETWORKS

Instrument access networks connect chip interfaces (such as 1149.1 and I2C) to on-chip instruments. Interfaces such as 1149.1 and I2C use serial protocols, whereas it might be needed to access the terminals of on-chip instruments in parallel. It is, therefore, common to use shift-registers with parallel I/O (similar to 1149.1-style TDRs) in order to access those terminals over serial

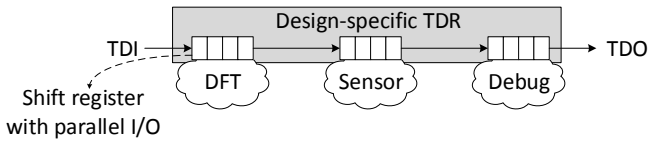


Figure 2.3. Non-reconfigurable network

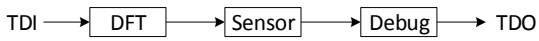


Figure 2.4. Non-reconfigurable network (simpler illustration)

interfaces. A straightforward way to design an on-chip access network is to serially connect instrument shift-registers between TDI and TDO terminals as a design-specific TDR. In this thesis, we refer to such a network as a *non-reconfigurable network*. As an example, Figure 2.3 shows such a network connecting three instruments (namely, a DFT instrument, a sensor, and a debug feature) to the TAP. In this figure, only the TDI–TDO scan path[2] is shown. There are, however, other signals involved for operation of such a network, such as clock, reset, and control (namely, capture, shift, and update) signals. To keep the drawings simple, in this thesis, only the scan path will be shown (except for when the focus is on the underlying circuitry). Further, an instrument along with its interface shift-register will be represented by a box, as shown in Figure 2.4.

In the network shown in Figure 2.4, the length of the TDI to TDO scan path is fixed, as all instruments are always on the scan path. Therefore, when only a subset of them is being accessed, dummy bits (i.e., meaningless data or repetition of previous data to those instruments) should be shifted in for those instruments that are not being accessed. Shifting dummy bits might increase the access time significantly especially when only a small subset of instruments is being frequently accessed. It will then be helpful to use reconfigurable (a.k.a. variable-length or flexible) scan path, which allows bypassing instruments that are not needed for the current access.

According to [30], the vector application time for a non-reconfigurable network is calculated as

$$t = (p + 1) \cdot l + p \cdot T_a \tag{2.1}$$

where p is the number of vectors, 1 denotes shifting out the output bits for the last vector, $+l$ is the sum of the length of shift-registers (in number of flip-flops), and T_a is the time it takes to go through the update and capture operations. In [30], T_a is equal to one.

[2]Scan path refers generally to the path over which non-functional serial data is transmitted.

2.3. RECONFIGURABLE INSTRUMENT ACCESS NETWORKS

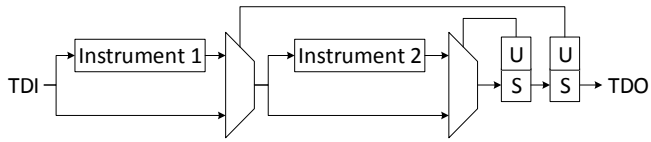
The idea of reconfigurable scan path has been around for quite a while. The authors in [31] proposed the use of reconfigurable scan-chains to reduce test time for designs with partial scan (by grouping frequently used scan cells in one chain and less frequently used scan cells into another one). Optimal design for a single reconfigurable scan-chain (based on the frequency of access) was presented in [32]. The work in [30] presented analysis and optimized design of scan-chains for core-based ICs, where two of the considered architectures were reconfigurable.

Adding reconfigurability to the scan path is done by placing multiplexers (hereinafter muxes) on the scan path. The control signals for the muxes can be provided from multiple sources, such as chip pins, 1149.1 IR decoder, other TDRs, and controllers on the same scan path on which the mux is placed (the so called *in-line control*). The choice of source of mux control signals directly impacts the instrument access time, as well as the achievable flexibility. Assume that at design time we precisely know how many configurations of the instrument access network are needed throughout the chip's life-time, though this might not be a quite realistic assumption, as we will discuss in Chapter 4. For example, for a total of 256 configurations, we will need eight pins if we choose to control the muxes directly from the chip boundary, which might be simply too many for pin-constrained designs. Therefore, one should trade number of configurations for number of control pins. If for this example, we choose to control the muxes directly via 1149.1 circuity, we will need 256 additional 1149.1 instructions, which also means more complexity in the IR decoder (Figure 2.1). In practice, 256 different configurations might be what is needed for a very small network of only eight instruments. Clearly, the use of chip pins or 1149.1 instructions does not scale well for large networks including hundreds or even thousands of instruments. In the rest of this work, we focus on large networks for which we assume full flexibility in the network in order to achieve low instrument access time.

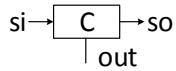
In the following, we will review three IEEE standards that facilitate automation for reconfigurable on-chip instrument access networks.

2.3.1. IEEE STD 1500

IEEE Std 1500 [9] targets testing of the embedded cores. Each core is equipped with a wrapper circuitry, which allows testing that core individually or in combination with other cores, as well as testing the user-defined logic between the cores. The wrapper circuitry and test patterns for each core are described in Core Test Language (CTL). EDA tools can automatically port the patterns described in CTL to the boundary of the parent core (in case of hierarchical designs that have cores nested inside other cores) or to the chip boundary.



(a) A network of two instruments, configured by two ScanMux control bits



(b) Symbol

Figure 2.5. ScanMux control bit

CTL, however, cannot be used to fully describe the behavior of instruments with complicated access procedures. For example, CTL lacks the flexibility of a programming language, which is required to describe the dynamic access procedures (that make decisions based on the status of the system). Furthermore, CTL cannot be used to describe the on-chip instrument access infrastructure.

2.3.2. IEEE STD 1687 (IJTAG)

IEEE Std 1687 [20], also known as internal JTAG (IJTAG), describes a methodology for accessing on-chip instruments via the TAP over dynamically reconfigurable networks. The dynamic reconfiguration in IEEE 1687 networks (hereinafter 1687 networks) allows for reduction of instrument access time by including only those instruments in the TDI to TDO scan path that are needed for the current operation. In this thesis, we refer to the part of scan path that is currently accessible from TDI-TDO terminals as the active scan path.

To enable dynamic reconfiguration in 1687 networks, multiplexers are used on the scan path, which are referred to as scan multiplexers or *ScanMuxes*. A two-input ScanMux is configured via a *control bit*, which is a shift-update register that can be placed anywhere on the scan path to configure one or more ScanMuxes. Larger ScanMuxes are configured by using multiple control bits (i.e., a *control register*). As an example, Figure 2.5(a) shows two ScanMux control bits used to configure a network of two instruments. To program the control bits to any desired configuration, the right values should be placed in their shift cells (denoted by S) during the *Shift* phase, and copied to their parallel latch (denoted by U) during the *Update* phase. We will use the symbol in Figure 2.5(b) to represent a ScanMux control bit in the rest of this thesis, where *si* is the scan input terminal, *so* is the scan output terminal, and *out* is the ScanMux control signal.

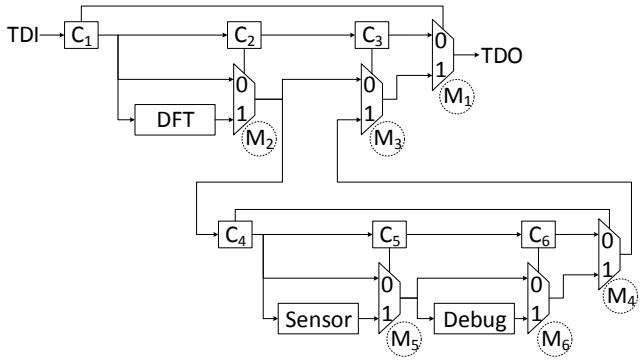


Figure 2.6. An example IEEE 1687 network connecting three instruments to the TAP

IEEE 1687 specifies the 1149.1 TAP as the primary interface between the chip boundary and the on-chip network of instruments. Interfacing is performed by connecting the IEEE 1687 network as a design-specific TDR to the 1149.1 circuitry. Since the TAP controller state machine is primarily used to operate 1687 networks, configuring the control bits or applying input vectors to instruments involves cycling through the capture, shift, and update states in the TAP controller state machine, each cycle referred to as a CSU operation [20] (hereinafter CSU). Performing each CSU consists of applying a number of clock cycles (TCKs) for shifting instrument data, as well as applying some clock cycles to take the TAP controller state machine through the update and capture phases back to the shift state.

In the following, with the help of Figure 2.6, it will be explained how a 1687 network is operated. Figure 2.6 illustrates a small 1687 network consisting of three instruments, namely, a DFT instrument, a sensor, and a debugging feature, as well as six ScanMux control bits. The instruments are interfaced to the scan path through shift-registers with parallel I/O (similar to 1149.1-style TDRs). To access the instruments, ScanMux control bits should be programmed to include the required shift-registers into the scan path. For example, to access only the DFT feature, C_1 and C_2 should be set to logic value '1', and C_3 should be set to '0' to bypass (via input 0 of mux M_3) the network segment containing the Sensor and Debug instruments, as well as C_4, C_5, and C_6 components.

Reconfiguring the network to the desired configuration might need several CSUs. For example, assuming an initial configuration of $C_1 = \ldots = C_6 = 0$ in Figure 2.6, accessing the Debug instrument needs two cycles of shift and update. In the first cycle, only C_1, C_2, and C_3 are accessible and by setting $C_2 = 0$ and $C_1 = C_3 = 1$, C_4, C_5, and C_6 become accessible. It is in the second

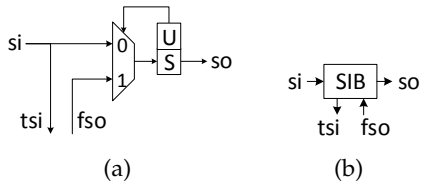


Figure 2.7. SIB: (a) simplified schematic, and (b) symbol

cycle when C_4, C_5, and C_6 can be configured to the right values, i.e., $C_5 = 0$ and $C_4 = C_6 = 1$, so that the Debug instrument becomes accessible.

In Figure 2.6, the select signals used to gate the capture, shift, and update control signals are not shown. In this thesis, it is assumed that only the components on the selected input of a mux get their select signal asserted.

The select signal for C_1 is asserted from the 1149.1 circuitry when the design-specific TDR corresponding to the 1687 network is active, meaning that C_1 is always accessible when working with this 1687 network.

2.3.2.1. SEGMENT INSERTION BIT (SIB)

By using ScanMuxes, it is possible to create reconfigurable networks with arbitrary architectures. A particular architecture, however, stands out throughout the examples provided by IEEE Std 1687, in which ScanMuxes bypass instrument shift-registers. To implement such architecture, a special combination of a two-input ScanMux and a control bit can be used, which is referred to as Segment Insertion Bit (SIB)[3].

The use of SIBs makes it possible to construct reconfigurable networks with low access time overhead (as is discussed in Chapter 3) for which retargeting can be done efficiently. A similar concept is added to the latest revision of 1149.1, as well (Section 2.3.3).

Figure 2.7(a) shows a simplified schematic of a possible SIB implementation. The *select* and control signals (namely, *capture*, *shift*, and *update*) are not shown for simplicity. The SIB has a shift (S) flip-flop, an update (U) flip-flop, and a two-input ScanMux. SIBs in the network are programmed by shifting a bit into their S flip-flop and latching that bit into the parallel U flip-flop. If the latched bit is a '0', the SIB is *closed* and the scan path is from the **si** (scan-in) terminal, to the **so** (scan-out) terminal via the S flip-flop, bypassing the segment between the **tsi** (to scan-in) and **fso** (from scan-out) terminals. If the latched bit is a '1', the SIB is *opened* and the scan path includes the segment connected between **tsi** and **fso** terminals of the SIB—referred to as the *host port* of a SIB in this thesis. In Appendix B, a detailed schematic of the SIB

[3]The idea for such a component first appeared in [33].

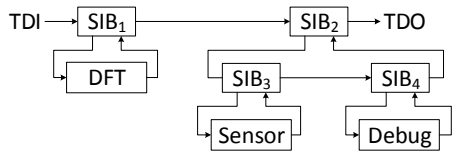


Figure 2.8. A SIB-based IEEE 1687 network with three instruments

component is presented. The symbol shown in Figure 2.7(b) will be used in the rest of this thesis to represent a SIB.

Figure 2.8 illustrates an example of a SIB-based IEEE 1687 network that connects three instruments to the TAP. Similar to Figure 2.6, the instruments are interfaced to the scan path through shift-registers with parallel I/O (similar to 1149.1-style TDRs). Initially, the SIBs are closed and the scan path consists of SIB_1 and SIB_2. To access the instruments, SIBs must be programmed to include corresponding shift-registers into the scan path. For example, to access the Sensor instrument, in one CSU, SIB_2 is opened and in the second CSU, SIB_3 is opened.

2.3.2.2. DESCRIPTION LANGUAGES AND RETARGETING

IEEE 1687 introduces two description languages, namely, Instrument Connectivity Language (ICL) and Procedural Description Language (PDL). ICL is used to describe the network, that is, how the instruments are connected to the TAP. PDL is used to describe the operation of instruments at their terminals. PDL commands allow to perform read/write operations on the instrument shift-registers and configurable components, as well as to wait for an instrument (such as a BIST engine) to finish its operation.

Given the PDL of each instrument, a retargeting tool generates scan vectors to configure the network and transport the required data bits from the TAP to/from the instruments' shift-registers. A retargeting tool relieves the designer from dealing with network configuration (i.e., writing the PDL to configure ScanMux Control bits directly). For example, assuming that the goal is to read the value from the sensor instrument in Figure 2.6, the PDL developer might simply use a write command to activate the sensor, a wait command to wait for the sensor to capture the value, and a read command to read the captured value out. It is then the task of the retargeting tool to generate one scan vector to configure C_1, C_2, and C_3, one vector to configure C_4, C_5, and C_6, one vector to write to the enable bit in the sensor's shift-register, a wait cycle of enough length, and finally one vector to scan the captured value out.

PDL has a basic set of commands to specify how to operate an instrument

by, for example, reading from/writing to its terminals/registers. These basic commands are referred to as Level-0 PDL commands. To make it possible to describe the operation of complex instruments (which might require the flexibility of a programming language, such as looping, branching, and so), PDL is designed to be used as an extension to TCL [34], which is a language well-known to the users of EDA tools. PDL, when used as an extension to TCL, is referred to Level-1 PDL. In this thesis, the focus will be on Level-0 PDL, which is a flat and sequential language consisting of two types of commands: *setup* commands and *action* commands. Setup commands (e.g., iRead, iWrite, and iScan) are queued, as if acting on a model of the instrument in the retargeter's memory, and take effect upon running the first subsequent action command (e.g., iApply). Another action command is iRunLoop which specifies a number of clock cycles to wait for an instrument to perform its operation (useful for describing the operation of BIST instruments).

To clarify the setup and action commands, we pick iRead and iWrite (used to read from and write to instrument terminals/registers) as setup commands, and iApply as an action command. A typical scenario is that multiple iRead and iWrite commands are queued and applied upon the first subsequent iApply command. A group of setup commands followed by an iApply command is referred to as one *iApply group*. Assuming that the TAP is used to access the 1687 network, the process of applying one iApply group is as follows:

1. The data to be written to instruments, specified by the queued iWrite commands, and the network configuration bits are formed into a *scan time frame* to be shifted in from TDI as input vector

2. The data to be read from the instruments, specified by the queued iRead commands, is captured into the shift registers (see Figure 2.6) when the TAP controller state machine is in the *Capture-DR* state

3. When the TAP controller state machine is in the *Shift-DR* state, the prepared scan frame is shifted in from TDI while the captured data is shifted out from TDO and stored into the ATE's memory[4]

4. When the TAP controller state machine is in the *Update-DR* state, the shifted-in vector is copied to the (parallel latches of the shift registers for the) instruments

Therefore, each iApply group is translated into a series of TAP operations to capture the instrument data, shift out the captured data while shifting in the

[4]In general, to the memory of the system or component operating the IEEE 1687 network (see Chapter 7).

prepared scan time frame, and apply the shifted-in scan time frame. It might be that an instrument is not currently on the active scan path. To perform the read/write operations on that instrument, the translation of an iApply group should be such that the required TAP operations for putting that instrument on the active scan path are also generated. A *retargeting step* will then be to generate a number of scan vectors to (1) change the configuration of the network (from its current state) to a configuration in which the specified registers are accessible, and (2) to perform the read/write operation. Each of these vectors is then applied to the network through a number of CSU operations. A complete retargeting flow might involve many retargeting steps.

The retargeting tool can perform a retargeting step in many ways. It suffices that the instruments to be accessed in the given iApply group become part of the active scan path. However, it is also possible to have additional instruments on the scan path, though not needed for the given iApply group. Presence of such additional instruments on the scan path results in higher access time, as the data should anyway be shifted through them. On the other hand, removal of other instruments from the scan path might also contribute to access time overhead (as it might require extra CSUs). Therefore, activating the required instruments such that the access time is minimized is an optimization problem. As a PDL script is a sequence of iApply groups, the optimization process should consider the complete PDL script when minimizing the access time, as reductions in access time in some step might counter reductions in another step.

Finally, we should note that PDL allows for concurrent application of multiple action commands by the use of *merge blocks* [20]. A merge block gives the retargeting tool the freedom to execute the stated actions in any arbitrary order, which is an opportunity to decrease the access time. Merging will be discussed in Chapter 5.

2.3.3. IEEE STD 1149.1-2013

The new revision of IEEE Std 1149.1 [21] has added support for reconfigurable on-chip instruments access networks. The reconfigurability is added to an IEEE 1149.1-2013 TDR by defining segments of that TDR as *selectable*. A selectable segment mux with a one-bit wide control, is similar to the SIB component specified by IEEE 1687. Moreover, IEEE 1149.1-2013 also allows for controlling a selectable segment mux from another part of the scan path or from other TDRs. The selectable segments can be nested to create a hierarchical network for accessing instruments, similar to what is achievable by a hierarchical 1687 network. Finally, IEEE 1149.1-2013 and IEEE 1687 use a similar Procedural Description Language (PDL) for describing the operation of embedded instruments.

2.4. CHAPTER CONCLUSIONS

The similarities between IEEE 1149.1-2013 and IEEE 1687 make much of the discussions and conclusions in this thesis work applicable to both of them. It should be noted that IEEE 1687 allows more flexibility when it comes to designing the on-chip instrument access networks, and therefore, brings up more opportunities and challenges in the design and optimization areas. Therefore, in this work, the focus will be on 1687 networks.

Part I

Analysis

3

Access Time Analysis

Reconfigurability allows for construction of many different networks for the same set of instruments. Some of these networks might be preferable to the others with respect to the ease of design, ease of operation, having lower hardware overhead, and allowing for faster access to instruments. To know if and how much each network is better than the others, there is a need for comparison metrics. In this chapter, we provide one such metric for comparison with respect to access time: the *overall access time (OAT)*.

Loosely defined, OAT is the time in terms of of clock cycles it takes to transport data to/from all embedded instruments over an on-chip instrument access network. The low OAT becomes particularly important when instruments are being accessed in the course of production, as the access time might affect the final product cost, or during in-field monitoring of a chip's operation, as the access time can affect the reliability of the chip. In this chapter, we present OAT analysis to identify contributing factors to OAT. Based on the analysis, we present methods for calculation of OAT for 1687 networks. The methods are used to assess how much each contributing factor affects OAT, which helps us develop methods for designing 1687 networks optimized for OAT (Chapter 4). There are, however, many ways to design a 1687 network. In order to be able to draw conclusions, in this chapter, we focus on only three network types which we refer to as *SIB-based*, *Daisy-chained*, and *Remote* networks.

In this chapter, after covering the basic definitions and assumptions in Section 3.1, we present OAT calculation methods for the aforementioned three 1687 networks types in Section 3.2, Section 3.3, and Section 3.4. Also in Section 3.4, we will review prior work on time analysis for reconfigurable scan path [35, 30]. The OAT calculation methods are used in a parametric analysis, presented in Section 3.5, to identify opportunities for OAT reduction.

3.1. PRELIMINARIES

In this section, we give definitions for the terms used in this chapter, such as *access*, *instrument data*, OAT, and access time *overhead* (Section 3.1.1). In Section 3.1.2, we define the *access schedule*, and describe the schedules considered in this thesis.

3.1.1. ACCESS

In this thesis, access to an instrument is defined as:

1. shifting input bits into the instrument's shift-register,
2. latching the contents of the shift-register to be applied to the internal circuitry of the instrument,
3. capturing the output of the instrument into the shift-register, and
4. shifting the captured values out.

When performing multiple accesses, shifting out the instrument outputs can overlap in time with shifting in the input bits for the next access. This, however, requires that the outputs to the previously applied inputs are ready to be captured and shifted out by the time the next inputs are being shifted in. In this regards, considering the relatively slow clock applied to a 1687 network (i.e., TCK applied to the TAP) [13, 36], we assume the time it takes an instrument to process the applied inputs and make the outputs ready to be captured, is less than the time it takes to move from Update-DR to Capture-DR in the TAP controller state machine.

It is important to note that not all instrument types are accessed as described above. For example, a BIST engine might be selected (by opening its corresponding SIB) and activated (by launching the BIST) and then be deselected (by closing its SIB) while still active and running. Later in the access schedule, the BIST can be selected again and its Done and Fail signals be polled. Nevertheless, from the access time analysis point of view, the above-mentioned BIST engine is accessed two times—once when launching the BIST and once when checking the results—and the number of clock cycles it takes to run the test is not part of the access time. Since the aim of our access time analysis is to study the overhead incurred by the network when data is transmitted through it, the amount of time an instrument spends running on its own (without any new inputs) can be disregarded.

considering the above-mentioned assumption on time overlap in performing multiple accesses, it takes

$$L_i \times (A_i + 1) \tag{3.1}$$

clock cycles to perform A_i accesses to instrument i with the shift-register length L_i, where $+1$ denotes shifting out the outputs for the last access. When there are N instruments in a network, the total number of clock cycles needed to perform all accesses to all instruments is referred to as *instrument data* in this thesis, and is calculated as:

$$\text{Instrument data} = \sum_{i=1}^{N} L_i \times (A_i + 1) \tag{3.2}$$

where N is the number of instruments, L_i is the length of shift-register for instrument i, and A_i is the number of accesses for instrument i.

Ideally, when performing all accesses to each of the N instruments, the OAT should be the same as the instrument data. In practice, however, more clock cycles should be spent, to operate the TAP controller state machine, program reconfigurable components such as SIBs, and shift data through bypass flip-flops. In this thesis, we refer to any clock cycle spent on an operation other than shifting instrument data, as access time *overhead*. When the overhead clock cycles are spent on shifting, we refer to them as *shift overhead*, and when they are spent are any TAP operation other than shifting we refer to them as *TAP overhead*.

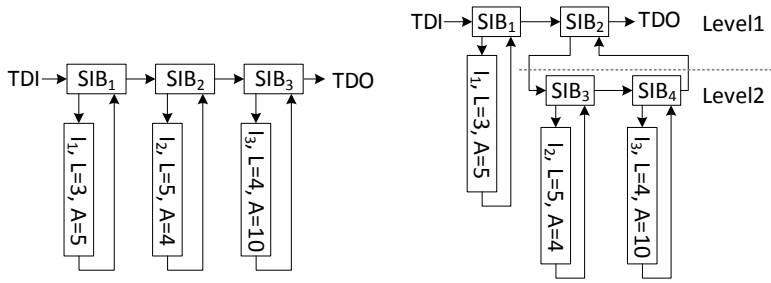
3.1.2. ACCESS SCHEDULES

An access schedule is an abstract model detailing how many times, in which order, and in what combinations the instruments are to be accessed. We refer to performing accesses according to a given schedule as *application* of that schedule. For accessing instruments according to any given schedule, it is crucial that the network has the flexibility to allow switching instruments on and off the scan path individually. The three reconfigurable network types that we consider in this thesis are designed to have such flexibility.

In this thesis we assume that in a given schedule accesses to an instrument can start at any point relative to other instruments, and that once started, all accesses to each instrument are performed consecutively without interruption. We refer to this schedule type as *generic*[1] throughout this thesis. Two extreme cases of generic schedules are of special interest, namely, the (fully) concurrent schedule and the sequential schedule, as they accentuate components of access time that are important in access time optimization.

In the concurrent schedule, as we define and use in this thesis, accesses for all instruments start as soon as possible. In this regard, recall from the discussion on Figure 2.6 in Section 2.3.2 that depending on the network design,

[1] Also referred to as non-preemptive session-less [37] or partitioned testing with run to completion schedules [38] in SoC test scheduling terminology.



(a) Flat architecture (b) Hierarchical architecture

Figure 3.1. Flat and hierarchical SIB-based 1687 networks

some instruments might becomes accessible earlier than the others. When an instrument is no more active (i.e., there are no more inputs to be applied to it), it is excluded from the scan path, by closing its corresponding SIB. This makes the scan path shorter for accessing the rest of the instruments.

In the sequential schedule, the instruments are accessed one at a time, and the assumed order of access is the order that the instruments appear on the scan path when all SIBs are open. The order of access can affect OAT in hierarchical networks, if it causes closing the already opened SIBs and reopening them again to access instruments in segments connected to the host ports of those SIBs. It is also assumed that the access for each instrument is completed before accessing any other instrument.

In the following sections, we present OAT calculation methods for the three network types mentioned earlier, namely, SIB-based, Daisy-chained, and Remote networks. For each network type, we consider concurrent, sequential, and generic schedules. We start our OAT analysis by considering the SIB-based networks.

3.2. SIB-BASED NETWORKS

Figure 3.1 shows two SIB-based networks for the same three instruments I_1, I_2, and I_3. For each instrument, length of shift-register (denoted by L) and number of accesses to perform (denoted by A) are shown. The type of architecture in Figure 3.1(a) is called a flat architecture in the remainder of this thesis. In the flat architecture, no SIB is connected to the host port of another SIB. Figure 3.1(b) shows another network for the same three instruments. Here, there are four SIBs and two of these SIBs are connected to the TAP through the host port of SIB_2. This type of architecture is called hierarchical architecture in the

Table 3.1. OAT calculation steps for the concurrent schedule

Row	Scan path	Shifted bits				Capture & update	Number of CSUs	Sum for scan path
		SIBs	I_1	I_2	I_3			
1	Figure 3.2(a)	2	0	0	0	4	1	$(2+4)\times 1$
2	Figure 3.2(b)	4	3	0	0	4	1	$(7+4)\times 1$
3	Figure 3.2(c)	4	3	5	4	4	5	$(16+4)\times 5$
4	Figure 3.2(d)	4	0	0	4	4	6	$(8+4)\times 6$
OAT								$\sum$=189

remainder of this thesis. In this thesis, a SIB having only an instrument on its host port is referred to as an *instrument SIB* and a SIB having one or more SIBs on its host port is called a *doorway SIB*.

3.2.1. ANALYSIS FOR A SMALL EXAMPLE

As the flat architecture is a one-level hierarchical architecture, it suffices to only discuss the hierarchical example. Therefore, we begin our analysis by showing how OAT can be calculated for the hierarchical network in Figure 3.1(b) according to concurrent, sequential, and an example of generic schedules. The analysis for this small network, explains the main idea behind the proposed OAT calculation algorithms.

3.2.1.1. CONCURRENT SCHEDULE

In the following, we describe how to calculate the OAT for the hierarchical architecture shown in Figure 3.1(b) according to the concurrent schedule. Figure 3.2 presents different scan path configurations for the network in Figure 3.1(b). In Figure 3.2, the gray boxes represent the S flip-flops inside the correspondingly numbered SIBs (see Figure 2.7). Note that the mux is placed after the host port, therefore, when a SIB is open, the segment connected to its host port appears before the S register on the scan path.

We use Table 3.1 to describe the OAT calculation. As the scan path initially consists only of SIB_1 and SIB_2 (Figure 3.2(a)), these SIBs should be opened before accessing the instruments. To open the SIBs, two bits with logic value of '1' are shifted in (one bit for each SIB) and subsequently applied. The two bits each corresponds to the S cell of a closed SIB, and they are accounted for on the row marked 1 in Table 3.1, column "SIBs". Applying shifted bits requires going through update and capture states in the TAP controller state machine, which takes four clock cycles (TCKs) as indicated in the column "Capture & update". After applying the two shifted bits, instrument I_1, as well as SIB_3 and SIB_4 are included in the scan path, as shown in Figure 3.2(b). At this point, input data can be applied to instrument I_1, SIB_1 and SIB_2 should

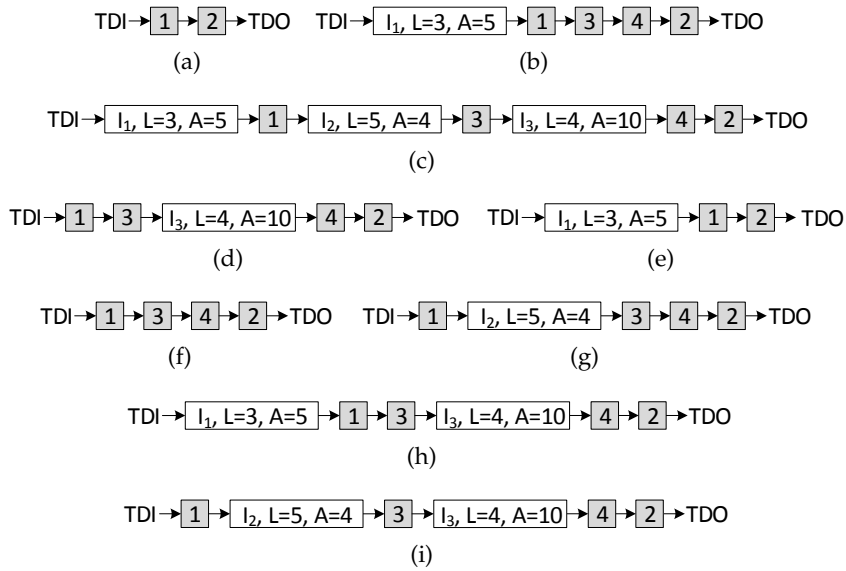


Figure 3.2. Different scan path configurations of the network shown in Figure 3.1(b)

be programmed to stay opened, and SIB_3 and SIB_4 should be programmed to be opened. The length of scan path in this configuration is 3 (for I_1) + 4 (for SIB_1 to SIB_4) = 7 bits. After applying the required vector to perform one access to I_1 and program SIB_1 to SIB_4, the scan path will be as shown in Figure 3.2(c). At this point, the second access for I_1 can be performed while performing the first access for instruments I_2 and I_3 (while programming the SIBs to stay open). In fact, in the current configuration, all remaining accesses for I_1 and I_2 can be performed. Note that after finishing the accesses for I_1 and I_2, one final access should be performed to shift the final set of outputs out, during which, one more access is performed to I_3. Therefore, in this configuration, a total of five CSUs are performed. The last of the five should also close SIB_1 and SIB_3 to make the scan path shorter for performing the remaining accesses to I_3, which results in the scan path shown in Figure 3.2(d). At this point, the remaining five accesses can be performed on I_3, plus one last CSU to shift out the final responses from I_3.

Table 3.1 shows the number of bits of different types that are shifted in for each CSU and the number of CSUs performed on each scan path configuration. The scan path configuration corresponding to each row is specified under the column "Scan path". The column "Sum for scan path" shows the total number of bits that are shifted in for each scan path. The OAT is the

Table 3.2. OAT calculation steps for the sequential schedule

Row	Scan path	Shifted bits				Capture & update	Number of CSUs	Sum for scan path
		SIBs	I_1	I_2	I_3			
1	Figure 3.2(a)	2	0	0	0	4	1	$(2+4)\times 1$
2	Figure 3.2(e)	2	3	0	0	4	6	$(5+4)\times 6$
3	Figure 3.2(f)	4	0	0	0	4	1	$(4+4)\times 1$
4	Figure 3.2(g)	4	0	5	0	4	5	$(9+4)\times 5$
5	Figure 3.2(d)	4	0	0	4	4	11	$(8+4)\times 11$
OAT								$\sum$=265

sum of the values in this last column, as shown on the last row, which for this example is 189 clock cycles.

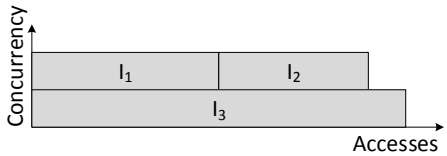
In Section 3.1.1, we mentioned that any clock cycle spent on an operation other than shifting instrument data is considered overhead. In this example, shifting the SIB control bits contributes to OAT by $2\times 1+4\times 1+4\times 5+4\times 6=50$ clock cycles, which we regard as SIB overhead. Furthermore, the number of clock cycles spent on update and capture operations is $4\times(1+1+5+6)=52$, which we consider as TAP overhead. The rest of clock cycles, i.e., $189-(50+52)=87$, are spent on shifting instrument data, which can also be calculated directly by using Eq. (3.2) as $3\times(5+1)+5\times(4+1)+4\times(10+1)=87$. It can be confirmed that OAT consists of three components, namely, instrument data, shift overhead, and TAP overhead. In this case, the total overhead is about 54 percent $((50+52)/189\times 100)$ of the OAT.

3.2.1.2. SEQUENTIAL SCHEDULE

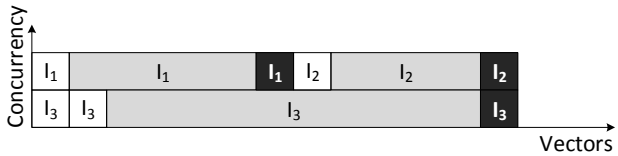
Similar to how Table 3.1 described the access for the concurrent schedule, Table 3.2 details the steps of sequential access to the instruments in the network shown in Figure 3.1(b).

For the sequential schedule, it is assumed that only those doorway SIBs are open that are on the shortest scan path to the instrument being accessed. Table 3.2 shows that for the sequential schedule, OAT is 265 clock cycles. The reason for OAT increase in case of the sequential schedule is that more scan vectors are applied each incurring shift overhead and TAP overhead. Seen from another perspective, in case of the concurrent schedule, overhead is shared by multiple concurrent accesses.

From Table 3.2, the TAP overhead as calculated as $4\times(1+6+1+5+11)=96$, and the shift overhead is calculated as $1\times 2+6\times 2+1\times 4+5\times 4+11\times 4=82$. In this case, the total overhead is about 67 percent $((82+96)/265\times 100)$ of the OAT.



(a) Given schedule



(b) Applied schedule

Figure 3.3. Example generic schedule

3.2.1.3. A GENERIC SCHEDULE

In practice, the instrument access schedule will be the output of the retargeting process, and can have partial concurrency (as opposed to the strictly concurrent and sequential schedules discussed above). The aim of the study in this section is not, however, to discuss the exact ordering a retargeting tool considers for the access in a given PDL script, but merely to study OAT for different networks under partially concurrent schedules. We take the example of a generic schedule representation shown in Figure 3.3(a) to explain our OAT calculation approach. In Figure 3.3(a), the horizontal axis represents the number of accesses, and the vertical axis shows how many instruments are accessed concurrently. Each rectangle represents one of the three instruments in the network shown in Figure 3.1(b), where the width denotes the number of accesses and the height is one unit. The given schedule can be interpreted as accesses to instruments I_1 and I_3 start at the same time (i.e., concurrency of two), and after accesses to instrument I_1 are complete, I_2 should be accessed concurrently with I_3.

The representation in Figure 3.3(a) is abstracted from the network configuration steps that might be needed to apply that given schedule to a 1687 network. The representation in Figure 3.3(b) shows how the given schedule will look like when we take into account the reconfigurations required for application of this given schedule to the network in Figure 3.1(b). In Figure 3.3(b), the white boxes represent the reconfiguration(s) needed to place the correspondingly numbered instrument on the scan path, the gray boxes represent the accesses to perform on each instrument, and the blackened boxes represent shifting out the results of the final access. The configuration for including

Table 3.3. OAT calculation steps for the generic schedule given in Figure 3.3(a)

Row	Scan path	Shifted bits				Capture & update	Number of CSUs	Sum for scan path
		SIBs	I_1	I_2	I_3			
1	Figure 3.2(a)	2	0	0	0	4	1	$(2+4)\times 1$
2	Figure 3.2(b)	4	3	0	0	4	1	$(7+4)\times 1$
3	Figure 3.2(h)	4	3	0	4	4	5	$(11+4)\times 5$
4	Figure 3.2(d)	4	0	0	4	4	1	$(8+4)\times 1$
5	Figure 3.2(i)	4	0	5	4	4	5	$(13+4)\times 5$
OAT								$\sum$=189

instrument I_2 could be done at the same time with closing SIB_1 to exclude instrument I_1. However, for simplicity, we assume that network reconfiguration is not done prospectively with respect to the next set of instruments in the given schedule.

Table 3.3 details the steps needed to apply the given schedule in Figure 3.3 to the network in Figure 3.1(b). The OAT for the given generic schedule is the same as the OAT for the concurrent schedule. This can be explained by noting that in both schedules instrument I_3 becomes active from the third vector (corresponding to Row 3 in Table 3.1 and Table 3.3) and remains active until the end of both schedules due to its high number of accesses, thus both schedules incur the same TAP overhead. Moreover, no matter what other instruments are being accessed concurrently with I_3, all the SIBs will be on the scan path for both schedules, thus shift overhead will be the same.

From the table, the TAP overhead as calculated as $4\times(1+1+5+1+5)=52$, and the shift overhead is calculated as $2\times 1+4\times 1+4\times 5+4\times 1+4\times 5=50$. In this case, the total overhead is about 54 percent $((50+52)/189\times 100)$ of the OAT.

3.2.2. ACCESS TIME CALCULATION ALGORITHMS

In this section, we will present OAT calculation algorithms for the SIB-based networks, when concurrent, sequential, and generic schedules are applied. The algorithms calculate OAT in the same way as presented in the analysis in Section 3.2.1, that is, by performing accesses and summing up the number of clock cycles.

For the algorithms, we model a 1687 network as a tree in which each internal node corresponds to a doorway SIB and each leaf node corresponds to an instrument SIB. The root of the tree is the TAP. Each node in the tree is associated with the following attributes:

- an *accesses* attribute that for a leaf node signifies how many accesses are to be performed on the node's corresponding instrument. For the

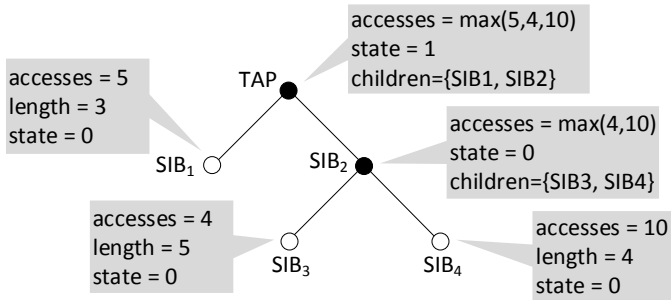


Figure 3.4. Tree representation for the network in Figure 3.1(b)

concurrent schedule, this attribute is associated with internal nodes as well, to signify how many accesses are to be performed in the subtree of that node,

- a *length* attribute that signifies length of the instrument shift-register associated with a leaf node,
- a *state* attribute that signifies whether a SIB is opened (state = 1) or closed (state = 0), and
- a *children* attribute that is a set containing node's child nodes. For a leaf node, this set is empty.

As an example, the tree in Figure 3.4 represents the SIB-based network in Figure 3.1(b). The values assigned to the attributes show their initial values.

3.2.2.1. CONCURRENT SCHEDULE

For the concurrent schedule, the OAT calculation steps are captured by Algorithm 3.1. As input, the algorithm receives a tree that models a SIB-based network (similar to Figure 3.4) and whose root node is denoted by *root*. In this algorithm, each access to the instruments (Lines 2–4) comprises of

1. resetting the variable SL, which stores the number of clock cycles needed to shift data through the scan path,
2. a call to TraverseConcSIB() (Line 3), which updates SL and returns the number of remaining accesses, and
3. adding the counted number of cycles to OAT (Line 4), which involves shifting SL bits followed by performing update and capture operations (represented by T_{FSM}[2]).

[2]To signify that this overhead is caused by the TAP controller finite state machine.

Algorithm 3.1: SIB-based network, concurrent schedule

Input: A tree T modeling the SIB-based network, whose root node is *root*
Output: OAT

```
1 while root.accesses > −1 do
2 |   SL := 0                                  // Scan path length for the current access
3 |   root.accesses := TraverseConcSIB(root)
4 |   OAT := OAT + SL + T_FSM
5 end
```

The algorithm terminates when there are no more accesses to be performed and the last responses are also shifted out (i.e., $root.accesses \leq -1$).

Function TraverseConcSIB() receives a tree *node* as input, corresponding to a doorway SIB, and by recursively calling itself

1. calculates the number of clock cycles needed to shift data for the current access (stored in SL), and

2. calculates and updates the remaining number of accesses for each instrument/segment in the *node*'s subtree (stored in variable *remaining*, which is initialized to -1 in Line 1).

The function iterates over *child* nodes in the subtree of *node* (Lines 2–22) and updates the *remaining* number of accesses while calculating the length of scan path (SL). Each child SIB increases the scan path length by one (Line 3), no matter if it is opened or closed. If a child node has accesses to be performed (Line 4) and it is closed (Line 5), it is opened (Line 6). If the SIB is opened and there are accesses to perform, one of the following is applicable:

- If the child node corresponds to an instrument SIB, the number of accesses to its corresponding instrument is decremented and the number of required clocks for shifting the data through that instrument's shift-register is added to SL (Lines 10–11).

- If the child node is a doorway SIB, the function calls itself recursively with that child node as input parameter (Line 14).

When there are no more accesses for a child node, that node is closed (or kept closed if it has already been closed) (Line 19). That is, if the child node is an instrument SIB and there are no more accesses to perform to its corresponding instrument, or if the child node is a doorway SIB and there are no more accesses to perform to any instrument connected directly or indirectly to its host port, that child node is closed. After analysis of each child node, the

```
Function TraverseConcSIB(node)
1  remaining := −1                          // number of remaining accesses in node's subtree
2  foreach child ∈ node.children do
3      SL := SL + 1                                                  // +1 for the SIB's S cell
4      if child.accesses > −1 then
5          if node.state = 0 then
6              node.state := 1
7          end
8          else
9              if child.children = ∅ then
10                 child.accesses = child.accesses − 1
11                 SL = SL + child.length
12             end
13             else
14                 child.accesses := TraverseConcSIB (child)
15             end
16         end
17     end
18     else
19         node.state := 0
20     end
21     remaining := max{remaining, child.accesses}
22 end
23 return remaining
```

remaining variable is set to the maximum number of accesses among the child nodes analyzed so far (Line 21), and is returned after all child nodes are considered (Line 23).

3.2.2.2. SEQUENTIAL SCHEDULE

This section describes function TraverseSeqSIB() for OAT calculation for the sequential schedule. The basic idea behind the OAT calculation for the sequential schedule is that there are $A_i + 1$ accesses for each instrument i, for which the number of shifted bits per access is constant. This can be seen in the examples of Table 3.2. The number of shifted bits during the instrument access, depends on the length of that instrument's shift-register and the number of SIBs on the scan path to that instrument.

To calculate OAT, TraverseSeqSIB() should be called with the TAP as parameter (TAP being the root node of the tree). Before the call to TraverseSeqSIB(), the global variables $SIBs$ and OAT should be set to 0. Here, $SIBs$ is a variable that counts the number of SIBs on the scan path, and OAT is the vari-

Function TraverseSeqSIB(*node*)

```
1  if node.children ≠ ∅ then
2  |   SIBs := SIBs + |node.Children|
3  |   OAT := OAT + SIBs + T_FSM
4  |   foreach child ∈ node.children do
5  |   |   TraverseSeqSIB (child)
6  |   end
7  |   SIBs := SIBs − |node.Children|
8  end
9  else
10 |   OAT := OAT + (node.length + SIBs + T_FSM) × (node.accesses + 1)
11 end
```

able that will contain the OAT when TraverseSeqSIB() terminates. The number of SIBs on the scan path will vary according to the location of the instrument that is being accessed within the network. Therefore, TraverseSeqSIB() keeps track of the SIBs that must be traversed to reach the level of hierarchy on which the accessed instrument is located. Each level of hierarchy is marked by a recursive call (line 5).

When TraverseSeqSIB() is called, it checks whether the current *node* (which is a SIB) has any child SIBs (Line 1). If *node* has children, the $SIBs$ variable should be increased by the number of children (Line 2), and OAT should be increased to represent the initial SIB programming required for the newly opened level of hierarchy (Line 3). Similarly when the function is leaving this level, $SIBs$ is reduced to the previous value, corresponding to the previous level of hierarchy (Line 7). TraverseSeqSIB() should be called recursively for the children of the current *node* (Lines 4–6). If the current *node* has no child (Line 9), OAT will be increased by the access time required for applying all the instrument's input vectors and the shift out of the last output vector (Line 10).

3.2.2.3. GENERIC SCHEDULES

We assume that a generic schedule is given as Figure 3.3(a), which does not capture the application details such as network configuration. To perform accesses according to such a generic schedule, we need to perform the network configuration, which makes the actual applied schedule similar to Figure 3.3(b).

In this section, we present a strategy for OAT calculation according to a given generic schedule. A given generic schedule is first broken into a number of sessions. A session starts when accesses to an instrument (or a number of instruments) begin, and finishes when a new session starts. We can think of

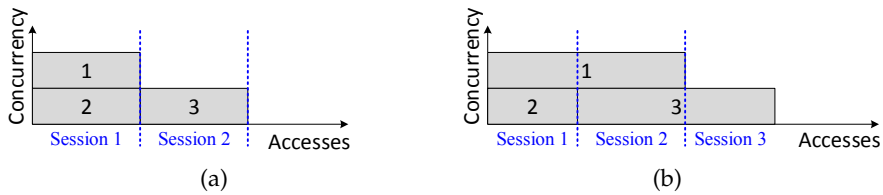


Figure 3.5. Example given generic schedules

a session as a list of instruments and their associated number of accesses. As we need to deviate from the given schedule to add the network configuration steps, we consider that the main rule to follow from the given schedule is the concurrency constraints. That is, if two instruments are not accessed at the same time in the given schedule, they are considered to be in *conflict*, and should not be accessed at the same time in the applied schedule.

Let us clarify the above with the help of an example. Assume the given schedule is the one shown in Figure 3.5(a). Further, assume that accesses to instrument 2 can start after one CSU spent on configuration while accesses to instrument 1 start after more CSUs (because, e.g., this instrument is in a deeper hierarchical level). In this situation, we consider that although the second session can start immediately after accesses to instrument 2 are finished, it should wait until accesses to instrument 1 are finished, as well. In contrast, if the given schedule is similar to the one shown in Figure 3.5(b), which allows for concurrent access between instrument 1 and instrument 3, the second session can start immediately after accesses to instrument 2 are complete. We emphasize again that the aim of this OAT calculation is not to represent the behavior of a retargeting tool, but to lay a common basis for comparison between different network types and architectures.

The flowchart in Figure 3.6 shows how a generic schedule is interpreted and applied. Initially, the *accesses* attribute (see Section 3.2.2) for all nodes is set to -1. The application stops when there are no more accesses left in the current session and no more sessions left in the schedule. After reading each new session, it is checked if any instrument in the newly read session is in conflict with any instrument in the session currently being applied:

- In case of no conflicts, the new session is *activated*. Activation of a new session is performed by setting the *accesses* attribute of the instruments listed in the session to the number of accesses specified in that session for that instrument.

- In case of conflicts, application of currently active session continues until accesses to an instrument are complete. Performing each access

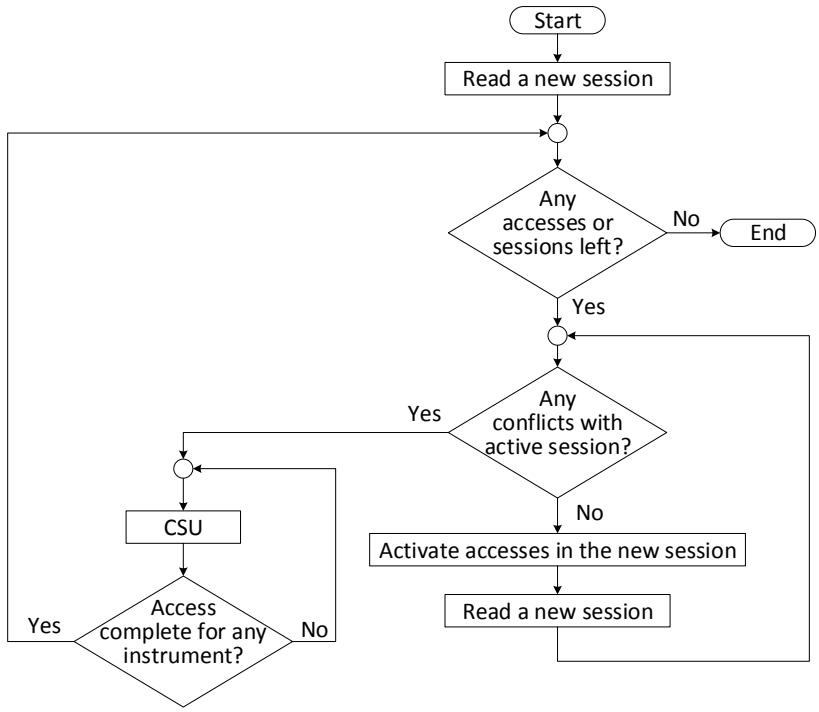
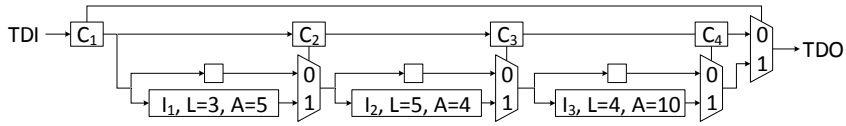


Figure 3.6. How a generic schedule is interpreted and applied.

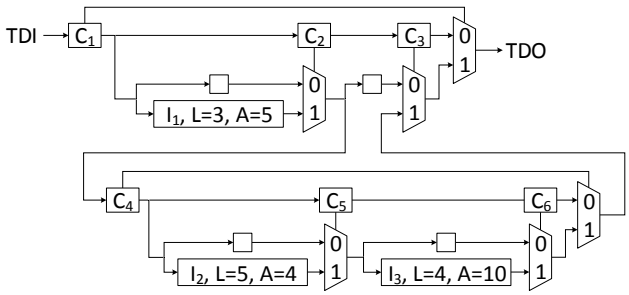
can entail applying a number of CSUs for network configuration before a final CSU for the actual read/write operation. In our implementation of the presented flowchart we used the Function TraverseConcSIB() to perform the CSU operation.

3.3. DAISY-CHAINED NETWORKS

What in this thesis is referred to as Daisy-chained was first presented in [39] as "custom hierarchical architecture", and later in [40] as "MUX-based architecture". Figure 3.7 shows flat and hierarchical Daisy-chained networks for the same set of instruments used earlier for the SIB-based network example, i.e., I_1, I_2, and I_3 in Figure 3.1. In our work, in order to be able to make a fair comparison between network types w.r.t. OAT, we assume additional flip-flops on bypass paths, represented by empty boxes in Figure 3.7. Such bypass flip-flops prevent long combinatorial paths, which can limit the clocking speed. On the other hand, shifting data through bypass flip-flops adds to the shift overhead. It should be noted that in SIB-based networks, the placement



(a) Flat architecture



(b) Hierarchical architecture

Figure 3.7. Flat and hierarchical Daisy-chained 1687 networks

of SIB's S flip-flop after the mux (Figure 2.7(a)) prevents formation of long combinatorial paths, without the need for extra bypass flip-flops.

In Daisy-chained networks, multiplexers are used to switch instrument shift-registers on and off the scan path. These multiplexers are controlled by ScanMux control bits (such as C_2, C_3, and C_4 in Figure 3.7(a)) placed on a separate branch of the scan path (the configuration path). To select between the two branches, other ScanMux control bits (such as C_1 in Figure 3.7(a)) are used.

In Figure 3.7(b), instruments I_2 and I_3 are placed in a deeper hierarchical level, which allows saving access time by removing their associated bypass flip-flops and ScanMux control bits from the scan path when these instruments are not being accessed. ScanMux control bits C_1 in Figure 3.7(a), as well as C_1 and C_4 in Figure 3.7(b), as well as their associated muxes can be seen as doorways to another hierarchical level, and will therefore be referred to as *doorway* ScanMux control bits henceforth.

The flat and hierarchical Daisy-chained networks shown in Figure 3.7 can be seen analogous to their SIB-based counterparts in having corresponding instruments placed in similar hierarchical levels. In Chapter 4, we will use this analogy to construct Daisy-chained networks from their optimized SIB-based counterparts.

In the following, OAT calculation algorithms are presented for the concurrent, sequential, and generic schedules. To use these algorithms, we model

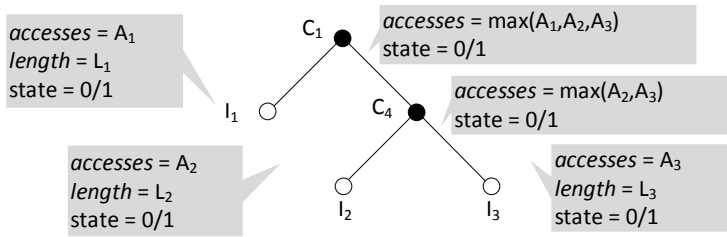


Figure 3.8. Tree representation for the network in Figure 3.7(b)

the given Daisy-chained network as a tree in which each internal node corresponds to a doorway ScanMux control bit, and each leaf node corresponds to an instrument. We clarify this with the help of the example tree shown in Figure 3.8 which models the network in Figure 3.7(b). Each node in the tree is associated with a *state* attribute which when set to 0, signifies that the node's corresponding instrument/segment is bypassed, and when set to 1 signifies that the corresponding instrument/segment is on the scan path. Each leaf node has two other attribute/value pairs: *accesses*, marking the number of accesses, and *length*, marking the length of the shift-register for the node's corresponding instrument. Each internal node, has also an *accesses* attribute whose value is the maximum among the values for *accesses* found in that node's subtree.

3.3.1. CONCURRENT SCHEDULE

For the concurrent schedule, the OAT calculation steps are captured by Algorithm 3.2. As input, the algorithm receives a tree representation of a Daisy-chained network (similar to the tree in Figure 3.8) whose root node is *root*. In the algorithm, each access to the instruments (Lines 2–4) comprises of

1. resetting the variable SL, which stores the number of clock cycles needed to shift data through the scan path,

2. a call to TraverseConcDC() (Line 3), which updates SL and returns the number of remaining accesses, and

3. adding the counted number of clock cycles to OAT (Line 4) which involves shifting SL bits followed by performing update and capture operations (represented by T_{FSM}).

Algorithm 3.2: Daisy-chained network, concurrent schedule

Input: A tree T modeling the Daisy-chained network, whose root node is *root*
Output: OAT

```
1 while root.accesses > −1 do
2 |   SL := 0                                  // Scan path length for the current access.
3 |   root.accesses := TraverseConcDC(root)
4 |   OAT := OAT + SL + T_FSM
5 end
```

The algorithm terminates when there are no more accesses to be performed and the last responses are also shifted out (i.e., $root.accesses \leq -1$).

Function TraverseConcDC() receives a tree *node* (corresponding to a segment in the Daisy-chained network) as input, and by recursively calling itself

1. calculates the number of clock cycles needed to shift data for the current access (stored in SL), and

2. calculates and updates the remaining number of accesses for each instrument/segment in the *node*'s subtree.

If the doorway ScanMux control bit for the segment represented by *node* contains a logic zero (Line 3), the multiplexer control path (i.e., the ScanMux control bits path) is selected and should be configured such that the instruments/segments with remaining accesses are placed on the scan path while the rest are bypassed. This reconfiguration involves shifting one bit per each ScanMux control bit in the segment (Line 2 and Line 4), and updating the *node*'s state to select the instrument path (Line 5). If, however, the instrument path in the current segment is selected (Line 8), for every child node (instrument/segment) on the path which has remaining accesses (Line 10), if the child node corresponds to

- an instrument, the algorithm reduces the remaining number of accesses by one and adds the number of required clock cycles for shifting the data through the instrument's shift-register to SL (Lines 11–13),

- a segment, the algorithm calls itself recursively (Line 16).

When there are no more accesses to be performed (Line 18) the corresponding instrument/segment is bypassed (Line 19). After analysis of each child node, the *remaining* variable is set to the maximum number of accesses among the child nodes analyzed so far (Line 21), and is returned after all child nodes are considered (Line 28).

Function TraverseConcDC(*node*)

```
1  remaining := −1                                  // # of remaining accesses in node's subtree
2  SL := SL + 1                                     // +1 for the doorway ScanMux control bit
3  if node.state = 0 then
4  |   SL := SL + |node.children|
5  |   node.state := 1
6  |   remaining := node.accesses
7  end
8  else
9  |   foreach child ∈ node.children do
10 |   |   if child.accesses > −1 then
11 |   |   |   if |child.children| = 0 then
12 |   |   |   |   child.accesses = child.accesses − 1
13 |   |   |   |   SL = SL + child.length
14 |   |   |   end
15 |   |   |   else
16 |   |   |   |   child.accesses := TraverseConcDC (child)
17 |   |   |   end
18 |   |   |   if child.accesses < 0 then
19 |   |   |   |   node.state := 0
20 |   |   |   end
21 |   |   |   remaining := max{remaining, child.accesses}
22 |   |   end
23 |   |   else
24 |   |   |   SL := SL + 1                         // +1 for the bypass flip-flop
25 |   |   end
26 |   end
27 end
28 return remaining
```

3.3.2. SEQUENTIAL SCHEDULE

For the sequential schedule, the OAT calculation can be performed by traversing the tree and calculating the required number of clock cycles needed for network configuration and instrument access, at each of the leaf nodes. Such tree traversal is shown in Function TraverseSeqDC(), which as input receives an internal tree node and calculates the number of clock cycles required to sequentially access the instruments in the segment represented by that subtree.

Function TraverseSeqDC() is initially called with the TAP as parameter (TAP being the root node of the tree). Before calling Function TraverseSeqDC(), the global variables SL and OAT should be set to zero. When an instrument in a given segment is being accessed, the rest of the instruments

Function TraverseSeqDC(*node*)

```
1  SL := SL + |node.children|
2  foreach child ∈ node.children do
3  |   OAT := OAT + SL + 1 + T_FSM
4  |   if |child.children| > 0 then
5  |   |   TraverseSeqDC (child)
6  |   end
7  |   else
8  |   |   OAT := OAT + (child.length + SL + T_FSM) · (child.accesses + 1)
9  |   end
10 end
11 SL := SL − |node.children|
```

(or subsegments) in that segment are bypassed which means that their corresponding bypass flip-flops are on the scan path. Variable SL (Line 1) serves two purposes:

- during the configuration step, it represents the number of ScanMux control bits on the configuration path, which is equal to the number of direct child nodes (i.e., $|node.children|$) in the subtree of *node*,
- when performing accesses, it represents the $|node.children| - 1$ bypass flip-flops plus one for doorway ScanMux control bit on the scan path.

It can be seen that in both cases, SL should be increased by $|node.children|$.

For each child node (Line 2), a configuration step is considered (Line 3) to put the node's corresponding instrument/segment on the scan path and put the other instruments/segments in bypass. In Line 2, SL represents the number of ScanMux control bits that should be configured to switch the instruments on and off the scan path, $+1$ represents the doorway ScanMux control bit, and T_{FSM} is the number of clock cycles spent on update and capture operations. If the node is an internal node (Line 4) the function calls itself recursively, otherwise the number of clock cycles needed to access the instrument corresponding to this leaf node is added to OAT (Line 8). Before return, SL is reset to its previous value (Line 11). In the OAT calculation (Line 8), it is considered that each instrument i is accessed $A_i + 1$ times ($+1$ for shifting out the last responses), and that for each access $SL + L_i$ bits should shifted followed by performing update and capture operations (denoted by T_{FSM}).

3.3.3. GENERIC SCHEDULES

The OAT calculation for a given generic schedule follows the same flowchart presented in Section 3.2.2.3. The difference is that to perform a CSU, Func-

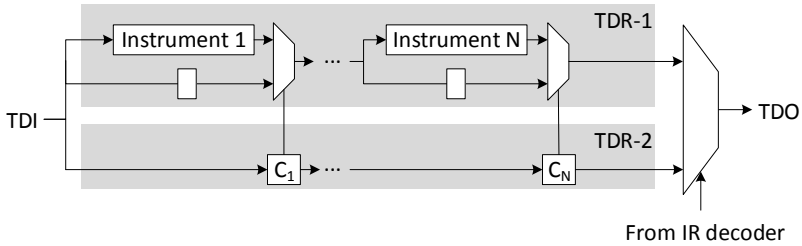


Figure 3.9. In the Remote network type, one TDR is used for instruments (TDR-1) and another TDR for ScanMux control bits (TDR-2)

tion TraverseConcDC() is called.

3.4. REMOTE NETWORKS

In this network type, there is one TDR for ScanMux control bits and one TDR for instruments (Figure 3.9). When the scan path needs to be reconfigured, the TDR with control bits is accessed (i.e., TDR-2). After the scan path is reconfigured, the TDR with the instruments (i.e., TDR-1) is selected to access the instruments. In this network type, since ScanMux control bits are not on the same scan path as the instruments, it is possible to *pipeline* the instrument data through the bypass flip-flops, and therefore effectively reduce the time wasted in the bypass flip-flops. Here, pipelining refers to filling the bypass flip-flops with instrument data for the next access instead of filling them with fill bits or dummy bits. The overhead reduction can be understood by referring to the work in [35] in which it is shown how pipelining of data through bypass flip-flops in a daisy-chained scan path results in extremely low test time overhead. This is in contrast to the work in [30], which shows that time is wasted in passing the bypass flip-flops. The key difference between [30] and [35] in their assumptions on bypass flip-flops is that in [35] it is assumed that bypass flip-flops are dedicated to testing, whereas in [30] the bypasses are functional flip-flops converted to scan registers. Since the contents of functional flip-flops change during an execution step (application of stimuli), it is in general not possible to pipeline the test patterns through them. Such wasted time in passing the bypass flip-flops was also present in the Daisy-chained networks discussed in Section 3.3, in spite of assuming dedicated bypass flip-flops. The reason was that the doorway ScanMux control bits were on the same scan path as the instruments, which required programming them with the correct value for every access. This constraint is, however, not present in the network

shown in Figure 3.9 since ScanMux control bits are placed on a separate TDR, and therefore, it is possible to reduce the access time overhead by pipelining the instrument data through the bypass flip-flops.

The OAT calculation for the networks such as the one in Figure 3.9 can be done similar to the test application time calculation in [30] for concurrent schedule, and to the test application time calculation in [35] for sequential schedule. However, as mentioned above, the calculations in [30] for the concurrent schedule are done under the assumption that time is wasted while shifting through the bypass flip-flops—which is not the case in the architecture presented here. Moreover, for Remote networks, we additionally need to take into account the switching between the two TDRs (needed to perform the network reconfigurations). The above differences make the OAT calculation for Remote network different from the calculations in both [30] and [35]. Therefore, in the following, we present the complete OAT calculations for the Remote networks.

Below we detail the OAT calculations for concurrent, sequential, and generic schedules. In all cases, it is assumed that instrument data is pipelined through the bypass flip-flops, and that initially TDR-1 is selected (Figure 3.9).

3.4.1. CONCURRENT SCHEDULE

We start by the concurrent schedule in which accesses to all instruments start at the same time. When there are no more accesses to be performed to a particular instrument, the scan path is configured such that this instrument is bypassed. OAT consists of the time it takes to setup the network by configuring the ScanMux control bits (T_{setup}), and the time it takes to perform the required number of accesses (T_{access}):

$$\text{OAT} = T_{setup} + T_{access} \tag{3.3}$$

Next, we derive the formulas for T_{setup} and T_{access}. Assume that there are N instruments, and A_i is the number of accesses to be performed on instrument i ($1 < i < N$). Moreover, assume that the instruments are ordered on the scan path such that $A_1 > A_2 > \cdots > A_N$. In the concurrent schedule, the network is reconfigured each time the access to an instrument is completed. Hence, there are N reconfigurations, and for each reconfiguration, we need to switch to TDR-2, shift in the configuration data into the ScanMux control bits, and switch back to TDR-1. The total required setup time for these reconfigurations is captured in the following:

$$T_{setup} = N \cdot (\mathrm{T_{switch}} + N + \mathrm{T_{switch}}) \tag{3.4}$$

where the first N represents the required number of reconfigurations, T_{switch} represents the time to switch TDRs (taking the TAP controller state machine from shifting data, to loading an instruction and back to the shifting data state), and the second N represents bits that are shifted in through ScanMux control bits. The reason that T_{switch} is considered two times is that we need to switch from TDR-1 to TDR-2 for reconfiguration, and back from TDR-2 to TDR-1 for accessing the instruments.

We now derive T_{access}. Initially all instruments are included in the scan path and $A_N + 1$ accesses are performed until the access to instrument N (which has the least number of accesses) is complete and the last responses are shifted out. The time it takes to perform $A_N + 1$ accesses is calculated as follows:

$$T_N = (\sum_{i=1}^{N} L_i + T_{FSM}) \cdot (A_N + 1) - T_{FSM} \tag{3.5}$$

where L_i is the length of the shift-register for instrument i, and T_{FSM} represents the clock cycles needed to perform the update and capture operations. The reason that one T_{FSM} is reduced from the calculated time is that the update and capture operations for the last access to instrument N are included in the time T_{switch} for the next network reconfiguration (i.e., in Eq. (3.4)).

At this point, instrument N should be bypassed, which requires one reconfiguration (considered in T_{setup}). Under the assumption of pipelining data in the bypass flip-flop for instrument N, performing the remainder of accesses for instrument $N-1$ takes the following time:

$$T_{N-1} = 1 + (\sum_{i=1}^{N-1} L_i + T_{FSM}) \cdot (A_{N-1} - A_N) - T_{FSM} \tag{3.6}$$

where 1 represents flushing the pipeline after the last access (i.e., one extra clock cycle is needed to shift the captured responses out completely through the bypass flip-flop for instrument N). In the same manner, we get the following time for performing the remainder of accesses for instrument 1:

$$T_1 = (N-1) + (L_1 + T_{FSM}) \cdot (A_1 - A_2) - T_{FSM} \tag{3.7}$$

where $(N-1)$ represents flushing the pipeline after the last access through the bypass flip-flops for instruments 2 to N. Finally, by summing up the access time for individual instruments, we can write T_{access} as:

$$T_{access} = \sum_{j=1}^{N} T_j \tag{3.8}$$

where T_j is (by assuming $A_{N+1} = -1$):

$$T_j = (N - j) + (\sum_{i=1}^{j} L_i + \mathrm{T_{FSM}}) \cdot (A_j - A_{j+1}) - \mathrm{T_{FSM}} \tag{3.9}$$

The above OAT calculation is performed under the assumption that no two instruments have the same number of accesses (i.e., $A_1 > A_2 > \cdots > A_N$). When there are instruments with the same number of accesses, since accessing them starts and ends at the same time, they share the same network reconfiguration step, and also the flushing of the pipeline will be performed once for all of them. Moreover, if instruments do not appear on the scan path in the assumed order, it can happen that two instruments are active with some bypass flip-flops in between them on the scan path. In this case, instrument data cannot (in general) be pipelined through those bypass flip-flops. The reason is that the captured responses from the instruments at the beginning of the path might break the scan vectors which are pipelined for the instruments further down the scan path. For these cases, to take the time wasted in the bypass flip-flops—that appear between active (i.e., not bypassed) instruments on the scan path—into account, Eq. (3.9) should be modified as:

$$T_j = R_b + R_e + (\sum_{i=1}^{j} L_i + R_m + \mathrm{T_{FSM}}) \cdot (A_j - A_{j+1}) - \mathrm{T_{FSM}} \tag{3.10}$$

where R_b represents the number of bypass flip-flops on the scan path preceding the first currently active instrument, R_e represents the number of bypass flip-flops on the scan path after the last currently active instrument, and R_m represents the number of bypass flip-flops that appear between the currently active instruments. Eq. (3.10) shows that the bypass flip-flops represented by R_m contribute to the access time (as overhead) for every access, whereas those represented by R_b and R_e only increase the time once per reconfiguration (to flush the pipelined data). To perform the parametric analysis (Section 3.5) and experiments (Chapter 4), we implemented an algorithm based on the formulas presented in this section, that calculates R_b, R_e, and R_m values based on the placement of the currently active instruments on the scan path, and therefore takes into account the time wasted passing through the bypass flip-flops.

3.4.2. SEQUENTIAL SCHEDULE

In the sequential schedule, instruments are accessed one at a time, and the accesses for each instrument are completed before accessing any other instrument. The order of performing accesses has no impact on OAT, which can be written as:

$$\mathrm{OAT} = T_{setup} + T_{access} \tag{3.11}$$

The total time needed for reconfigurations can be written as:

$$T_{setup} = N \cdot (\mathrm{T_{switch}} + N + \mathrm{T_{switch}}) \tag{3.12}$$

where the first N represents the required number of reconfigurations, $\mathrm{T_{switch}}$ represents the time to switch TDRs (taking the TAP controller state machine from shifting data, to loading an instruction and back to the shifting data state), and the second N represents bits that are shifted in through ScanMux control bits. The reason that $\mathrm{T_{switch}}$ is considered two times is that we need to switch from TDR-1 to TDR-2 for reconfiguration, and back from TDR-2 to TDR-1 for accessing the instruments.

Assuming that T_i is the time it takes to complete A_i accesses for instrument i, we have:

$$T_{access} = \sum_{i=1}^{N} T_i \tag{3.13}$$

where

$$T_i = N - 1 + (L_i + \mathrm{T_{FSM}}) \cdot (A_i + 1) - \mathrm{T_{FSM}} \tag{3.14}$$

In Eq. (3.14), $N - 1$ represents the bypass flip-flops that should be flushed after the last access to instrument i, L_i is the length of the shift-register for instrument i, A_i is the number of accesses for instrument i, and $\mathrm{T_{FSM}}$ represents the clock cycles needed to perform the update and capture operations. Similar to Eq. (3.10), one $\mathrm{T_{FSM}}$ is reduced from the T_i as it is included in $\mathrm{T_{switch}}$ for the next network reconfiguration.

3.4.3. GENERIC SCHEDULES

Applying generic schedules to the Remote network type is different from that for SIB-based and Daisy-chained networks in the following ways:

- it is not possible to reconfigure the Remote network type while accessing instruments, and
- reconfiguration time is the same for all instruments, as all control bits are placed in the same level in their associated TDR (namely, TDR-2 in Figure 3.9).

These differences make application of generic schedules to Remote Networks rather straightforward as the applied schedule is similar to the given schedule extended with reconfiguration steps between the sessions (which always take the same number of clock cycles for the same network).

Therefore, to apply a given generic schedule to a Remote network, it suffices to break the given schedule into sessions, and perform one reconfiguration step between each session. The access time for each session is calculated according to Eq. (3.10) for the instruments active in that session. The OAT is the sum of access times for sessions plus the time for reconfiguration steps.

3.5. PARAMETRIC ANALYSIS

So far in this chapter, we have discussed and presented algorithms for OAT calculation for three network types, namely, SIB-based, Daisy-chained, and Remote networks. We showed that OAT has three components: instrument data, shift overhead, and TAP overhead. In this section, we study how each of these OAT components varies with parameters such as the number of instruments and (where applicable) number of hierarchical levels. For each of the studied parameters, we consider the concurrent and sequential schedules. Besides reporting the OAT components, we report the overhead percentage calculated as:

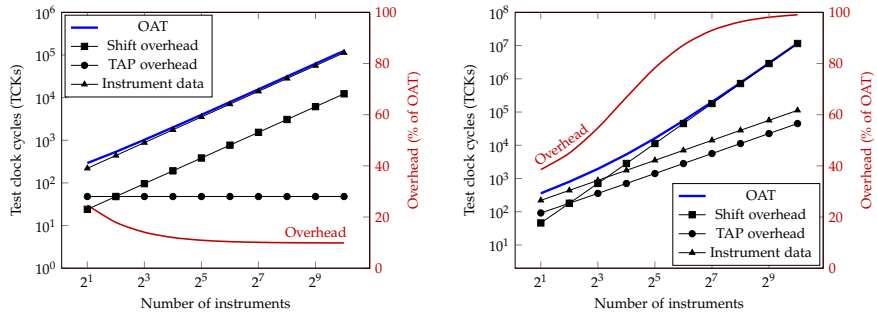
$$\frac{\text{Shift overhead} + \text{TAP overhead}}{\text{Instrument data} + \text{Shift overhead} + \text{TAP overhead}} \times 100 \tag{3.15}$$

The observations from this analysis will be used in Chapter 4 for designing reconfigurable networks optimized w.r.t. OAT. To perform this study, we have implemented the presented algorithms, and have additionally instrumented them to report each of the OAT components separately. Throughout the thesis, for OAT computation, it is assumed that T_{FSM} is four TCKs and T_{switch} is 19 TCKs. They are both reported as TAP overhead. Appendix A presents complete results from this analysis, as well as additional charts for easier comparison between the results for the concurrent and sequential schedules.

3.5.1. INCREASING THE NUMBER OF INSTRUMENTS

We begin our parametric analysis with observing how OAT components vary with the number of instruments. For this purpose, we assume that initially we have a network for two instruments, and increase the number of instruments from two to 1024 in powers of two. We assumed that each instrument has a shift-register of length 10 flip-flops and is accessed 10 times ($L = 10$, $A = 10$).

Figure 3.10 shows the result of OAT calculation for flat SIB-based networks, for concurrent and sequential schedules. Each plot has two y-axes, where the y-axis on the left is used for OAT and its components, and the y-axis on the right is used for the total overhead percentage in OAT. The following observations can be made for the flat SIB-based networks:



(a) Concurrent schedule (b) Sequential schedule

Figure 3.10. The effect of increase in number of instruments on OAT and its components, in SIB-based networks

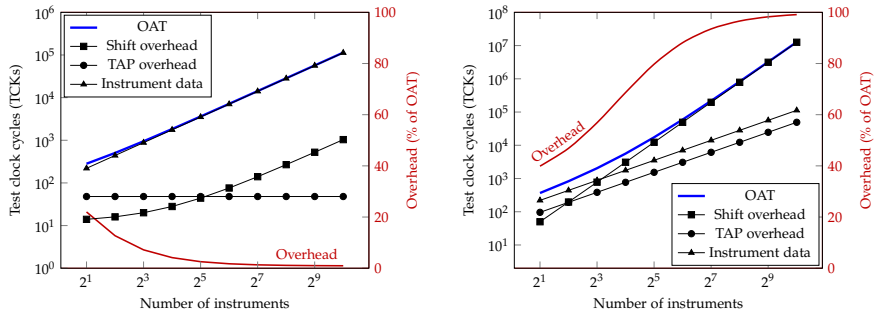
- For the concurrent schedule, the overhead percentage decreases with an increase in the number of instruments. The reason is that instrument data and shift overhead increase linearly with the number of instruments, whereas TAP overhead remains constant (due to the number of accesses remaining constant and the access schedule being concurrent). Consequently, overhead percentage decreases.

- For the sequential schedule, the overhead percentage increases with the number of instruments. The reason is that the instrument data and TAP overhead grow linearly with the number of instruments whereas the shift overhead grows quadratically with the number of instruments (as can be seen from the greater slope of the shift overhead plotted on a logarithmic scale).

- For the same number of instruments, the sequential schedule results in higher OAT. As the instrument data is the same for both schedules, the lower OAT for the concurrent schedule is the result of lower overhead (which is also reflected in the overhead ratio). The lower overhead for the concurrent schedule is due to many accesses sharing the same SIB programming data and TAP operations.

- For more than four instruments, the dominant overhead component for both schedules is the shift overhead.

- For the same network, the TAP overhead varies considerably between the concurrent and sequential schedules.



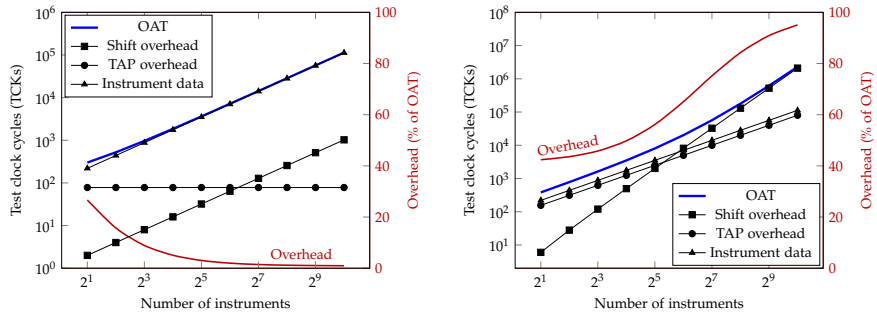
(a) Concurrent schedule (b) Sequential schedule

Figure 3.11. The effect of increase in number of instruments on OAT and its components, in Daisy-chained networks

Figure 3.11 shows the result of OAT calculation for flat Daisy-chained networks, for concurrent and sequential schedules. Similar to the SIB-based networks, here again the sequential schedule results in higher OAT compared to the concurrent schedule, for a given number of instruments. Comparing the SIB-based networks with Daisy-chained networks when the concurrent schedule is applied shows a noticeably lower shift overhead for the Daisy-chained networks. This lower shift overhead is due to that in the (flat) SIB-based networks, the SIBs are always on the scan path and data should be shifted through them for every access. In contrast, in the (flat) Daisy-chained networks, when all instruments are being accessed concurrently, only one doorway Scan-mux control bit is on the scan path (such as C_1 in Figure 3.7(a)) that contributes to the shift overhead.

Figure 3.12 shows the result of OAT calculation for Remote networks, for concurrent and sequential schedules. For concurrent schedule, the OAT is very similar to the Daisy-chained network and again noticeably smaller than OAT for SIB-based networks. However, for the sequential schedule, OAT is considerably lower compared with SIB-based and Daisy-chained networks. This lower OAT can be attributed to the pipelining of instrument data through the bypass flip-flops, thus lowering the shift overhead. Although OAT is lower for the case of sequential schedule, up to about 92 percent of OAT is still the shift overhead (see Table A.3 in Appendix A for the details).

Finally, a general observation is that the TAP overhead does not change considerably between different network types, but varies significantly between different access schedules.



(a) Concurrent schedule (b) Sequential schedule

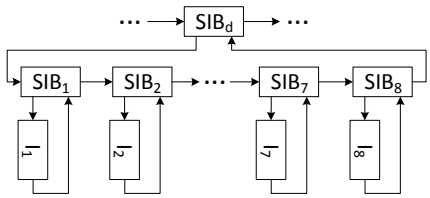
Figure 3.12. The effect of increase in number of instruments on OAT and its components, in Remote networks

3.5.2. INCREASING THE NUMBER OF HIERARCHICAL LEVELS

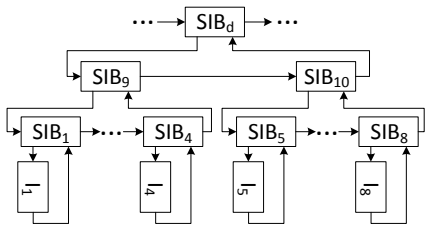
Another important parameter to consider is the use of hierarchy in the network design. There are, however, many ways to create hierarchy for a given set of instruments. To keep this parametric analysis manageable, only a very small subset of all possible networks are considered. We start with a flat network (i.e., only a single level of hierarchy) and observe how OAT changes with increasing the number of hierarchical levels.

For the SIB-based networks, we show how we add a level of hierarchy by using the example network in Figure 3.13(a). In the figure, SIB_d is a doorway SIB having eight instrument SIBs connected to its host port. To add a level of hierarchy, we (1) divide the instrument SIBs into two groups, (2) connect each group to the host port of an additional doorway SIB, and (3) connect these two additional doorway SIBs to the host port of doorway SIB_d. The resulting network is shown in Figure 3.13(b). For this experiment, we start with a flat network of 1024 instruments ($L = 10$ and $A = 10$), and increase the hierarchical levels from one (i.e., the flat network itself) to 10 levels. At the 10^{th} level, each doorway SIB has two instrument SIBs connected to its host port. For the initial flat network, as there is no doorway SIB, we simply consider TAP to be SIB_d.

The OAT calculation results are presented in Figure 3.14 for the concurrent and sequential schedules. As expected and can be seen from the graphs in Figure 3.14, the instrument data is independent from the access schedule and number of hierarchical levels. From Figure 3.14(a) (for the concurrent access schedule) it is seen that an increase in the number of hierarchical levels increases OAT. The reason for the increase are the extra SIBs on the scan path.



(a) A network segment comprising eight instruments



(b) Adding a hierarchical level to the segment shown in Figure 3.13(a)

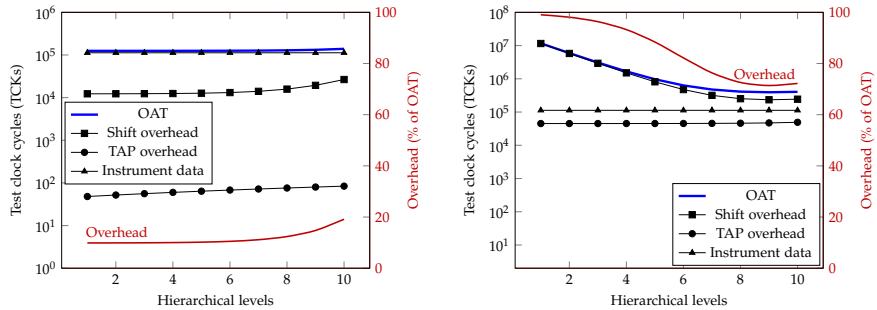
Figure 3.13. Adding hierarchy to a SIB-based network segment

This observation cannot be generalized, though, as in this case, all instruments have the same number of accesses and therefore accessing them starts and finishes at the same time. As a consequence, any extra SIB on the scan path becomes just another contributor to overhead.

In contrast to the increasing trend in shift overhead for the concurrent schedule, Figure 3.14(b) shows an opposite trend for the sequential access. Here, the shift overhead is reduced approximately 50 times going from one level to 10 hierarchical levels. The resulting impact on overhead ratio is a reduction of about 25 percent. The reason for the reduction is that the use of hierarchy allows for exclusion of inactive subsegments (comprising of both SIBs and instruments) from the scan path. Removing SIBs from the scan path obviates the need for shifting data through them, which results in the reduced overhead.

Here, an important observation is that for both concurrent and sequential schedules, the TAP overhead is not affected noticeably by increase in hierarchical levels. Therefore, since instrument data is independent of network architecture, for both schedules, the increase and decrease in OAT can be attributed to the contribution of shift overhead.

Figure 3.15 shows how hierarchical levels are added to a network segment that has a Daisy-chained type. Figure 3.16 presents the OAT calculation results. The same observations made for the SIB-based network can be made here, as well.



(a) Concurrent schedule (b) Sequential schedule

Figure 3.14. The effect of increase in hierarchical levels on OAT and its components, in SIB-based networks

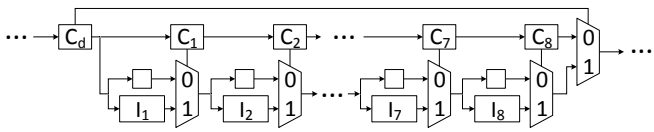
In this work, we have not considered the use of hierarchy for the Remote networks.

3.5.3. VARYING THE INSTRUMENT PROPERTIES

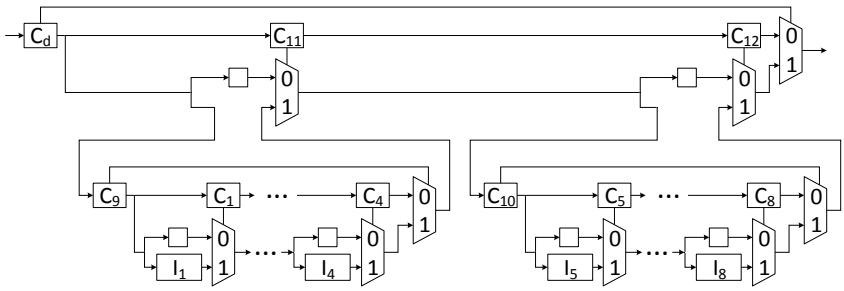
So far in our parametric analysis, we have assumed that each instrument has a shift-register of length 10 flip-flops and is accessed 10 times (i.e., $L = 10$ and $A = 10$). In the final part of our analysis, we focus our attention to these two properties.

An increase in the shift-register length increases instrument data, which in turn increases OAT. However, the shift-register length has no impact on any of the overhead types, which can be explained as follows: all network types considered in this section were designed such that each instrument can be switched on and off the scan path independently from other instruments. Moreover, in the considered access schedules, each instrument was switched on the scan path only when it was being accessed, otherwise, we had to use dummy bits (fill bits) for instruments that were on the scan path but were not accessed. As a result, the length of instrument shift-registers did not contribute to the overhead. It can then be concluded that if we increase the length of shift-registers, instrument data and OAT will increase but the overhead remains constant, leading to a decrease in the overhead ratio.

An increase in the number of accesses, however, increases instrument data and both overhead types. This can, for example, be seen from Line 10 in Function TraverseSeqSIB(). In the same function, we can see that the overhead due to opening levels of hierarchy is not affected by the number of accesses (Line 3). However, for large number of accesses, the overhead contributed by the accesses will be much larger than the overhead due to opening levels of



(a) A network segment comprising eight instruments

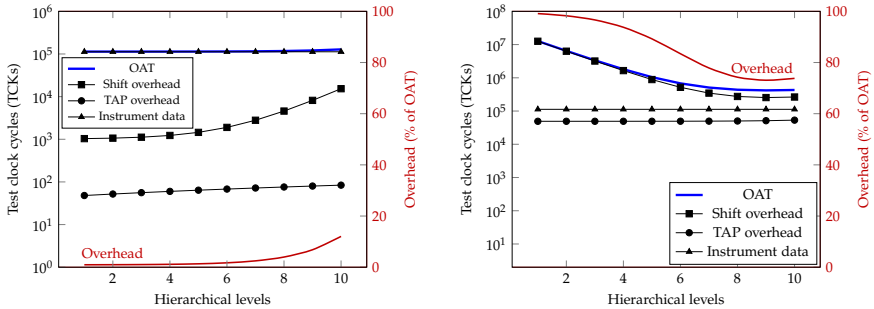


(b) Adding a hierarchical level to the segment shown in Figure 3.15(a)

Figure 3.15. Adding hierarchy to a Daisy-chained network segment

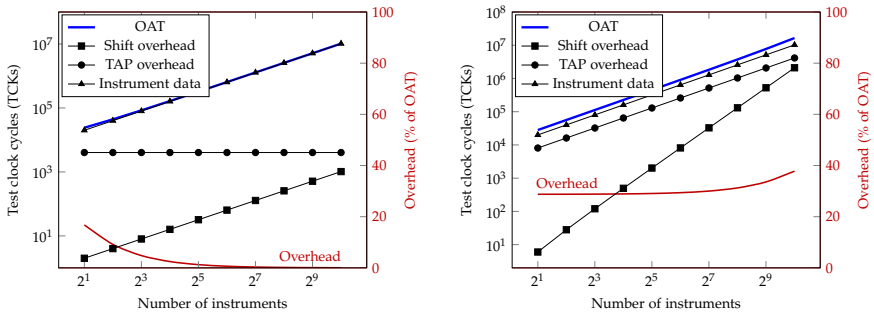
hierarchy. Therefore, it can be concluded that a large increase in number of accesses, will result in (almost) similar increase in all OAT components (i.e., instrument data and overhead types). Consequently, the overhead ratio will remain almost the same.

For the Remote network, an increase in number of accesses reveals an interesting property of this network type. Eq. (3.14) shows that for each instrument, it takes $N-1$ clock cycles to flush the pipeline, N being the number of instruments. Therefore, $N \times (N-1)$ cycles will be wasted in total. For large number of instruments, say 1024, and low number of accesses, say 10, these wasted cycles constitute a large part of OAT. However, for large number of accesses, the same number of wasted cycles is a significantly smaller part of OAT. This can be seen from Figure 3.17, which repeats the experiment of increasing the number of instruments presented earlier for Remote networks, but this time with $A = 1000$ instead of $A = 10$. Comparison of plots in this figure with those in Figure 3.12 shows that in case of the sequential schedule, the overhead ratio drops from about 95 percent to about 40 percent. Note that with increasing the number of accesses from 10 to 1000, instrument data and TAP overhead have increased while the shift overhead has remained exactly the same.



(a) Concurrent schedule (b) Sequential schedule

Figure 3.16. The effect of increase in hierarchical levels on OAT and its components, in Daisy-chained networks



(a) Concurrent schedule (b) Sequential schedule

Figure 3.17. The effect of increase in number of instruments having large number of accesses, on OAT and its components, in Remote networks

3.6. CHAPTER CONCLUSIONS

In this chapter, overall access time (OAT) analysis and calculation methods were presented for three 1687 network types, namely, SIB-based networks, Daisy-chained networks, and Remote networks. The analysis identified three components for OAT in the studied networks: instrument data, shift overhead, and TAP overhead. Instrument data is independent from network type, network architecture, and access schedule, and is a function of number of accesses to each instrument and the length of shift-registers. Therefore, in order to reduce OAT, it is the overhead that should be reduced. The parametric analysis presented in this chapter showed the following regarding the over-

head. For the SIB-based and Daisy-chained networks that have large number of instruments, the shift overhead is the dominant overhead type. Moreover, it was observed that the shift overhead varies significantly with the network type, network architecture, and access schedule. For the Remote networks, it was observed that pipelining of instrument data lowers the shift overhead. A general observation regarding the TAP overhead was that it does not change considerably between different network types and architectures, but varies significantly between different access schedules. In Chapter 4, we use these observations to design reconfigurable networks that are optimized with respect to OAT.

Part II

Design

4

Design of Optimized 1687 Networks

The parametric analysis presented in Chapter 3 showed that the network type (e.g., SIB-based) and architecture (e.g., levels of hierarchy), as well as access schedule, can have a significant impact on OAT. When the access schedule is known, one can choose a network type and architecture such that OAT is minimized for that schedule. However, in reality, it can happen that the access schedule changes after the design is fixed, where a change might happen in the number of accesses to each instrument or the in the way that instruments are accessed together (i.e., concurrency in the schedule). More concretely, it can happen that different schedules, each involving only a subset of instruments, are applied to the network at different points throughout the chip's life cycle. The following example helps in clarifying this case of multiple access schedules. An MBIST instrument might be accessed (1) during yield learning for a new process to choose the most suitable algorithms, (2) during wafer sort and package test to detect defective devices and perform repair, (3) in the burn-in process to cause activity in the chip and to detect infant mortality [41, 42], (4) during PCB bring-up [8], (5) during PCB assembly manufacturing test [8], and (6) during power-on self-test and other in-field tests. Also, the number of accesses to a given instrument typically varies throughout the life cycle of a chip. For example, during yield learning, an embedded memory might be tested several times by running multiple MBIST algorithms. Another example is reading out the memory contents for diagnostic purposes [43]. In both examples, many accesses might be needed. In contrast, during manufacturing tests, an embedded memory might be tested only by accessing the associated MBIST engine a few times to setup the algorithm, start the MBIST, check for its completion, and read the results.

In this chapter, we present methods for designing networks that are optimized with respect to OAT. We begin by considering the case where the

network is designed only for one access schedule. Lately, there has been work on optimized design of SIB-based networks [44], which we will discuss in Section 4.1.1 and Section 4.1.3. Later, in Section 4.2, we present methods for optimizing the network when it is subjected to different access schedules. Finally, in Section 4.3, we make an evaluation of the proposed design methods in the context of late schedule changes or more generally, *unknown* schedules.

4.1. SINGLE ACCESS SCHEDULE

A general observation from the analysis in Section 3.5 was that TAP overhead did not vary significantly with network type and architecture, but varied considerably between different access schedules. As in this section, we assume that the access schedule is given, to reduce OAT, we focus only on the reduction of shift overhead. In general, in any given schedule, some instruments might be accessed more than other instruments. As each access contributes to shift overhead, instruments with higher number of accesses might have a larger contribution to shift overhead than those with smaller number of accesses. Therefore, where possible, design of 1687 networks should be such that length of scan path is shorter for instruments with higher number of accesses. This shortening of scan path length for frequently accessed instruments might come at the cost of longer scan path for less frequently accessed instruments.

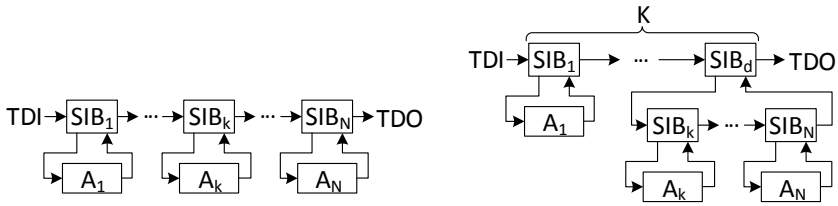
We define the problem of network design for a given schedule as follows: Given a set I of instruments, where for each instrument i ($i \in I$) the number of accesses A_i is provided, and an access schedule, a 1687 network should be designed such that shift overhead is minimized. For the access schedules, we consider the concurrent, sequential and generic schedules.

4.1.1. THE CONCURRENT SCHEDULE

In this section, we present methods for designing 1687 networks that are optimized w.r.t. OAT for the concurrent schedule. We consider SIB-based, Daisy-chained, and Remote networks.

4.1.1.1. SIB-BASED NETWORKS

Figure 4.1(a) shows N instruments in a single-level (i.e., flat) SIB-based network. Figure 4.1(b) shows the same instruments in a two-level design. The instruments are ordered so that $A_1 \geq \ldots \geq A_K \geq \ldots \geq A_N$. In both networks, by closing the instrument SIBs whose corresponding instruments are not accessed anymore (say instruments K through N), the scan path becomes shorter for the instruments that are still accessed (say instruments 1 through $K-1$). For the flat architecture, however, this leaves the closed instrument



(a) In a single-level network (b) In a two-level network

Figure 4.1. N instruments in single-level and two-level networks

SIBs themselves on the scan path, contributing to shift overhead for each subsequent access. By using hierarchical designs, such as the two-level design shown in Figure 4.1(b), it is possible to reduce the shift overhead due to the instrument SIBs (for instruments K through N) by excluding these SIBs from the scan path.

In the concurrent schedule, accesses to all instruments start as soon as possible. Before accessing instruments in the network shown in Figure 4.1(a), all the SIBs should be opened. This is done by shifting N bits to program the SIBs. As was mentioned in Chapter 3, these N bits are considered overhead since they are not part of the instrument data. Furthermore, each of the N SIBs that are on the active scan path must be programmed for every access. Since A_1 is the maximum number of accesses among the instruments, for the concurrent schedule, a total of A_1 accesses is performed and $(A_1 + 1) \cdot N$ clock cycles are spent in total on shifting these SIB control bits. Therefore, the shift overhead for the network shown in Figure 4.1(a) is calculated as:

$$O = N + (A_1 + 1) \cdot N \tag{4.1}$$

To access the instruments in the network shown in Figure 4.1(b), K bits should be shifted in to open the SIBs at the first level of hierarchy, namely, SIB_1 to $SIB_{K\text{-}1}$ and SIB_d. Subsequently, control bits are shifted in to open SIB_K through SIB_N (as well as to keep SIB_1 to $SIB_{K\text{-}1}$ open), together with the first input data for instruments corresponding to SIB_1 through $SIB_{K\text{-}1}$. Therefore, $N + 1$ control bits are shifted in besides the instrument data. Now that SIBs at the second level are open, A_K more accesses are performed to all instruments. Accessing the instruments A_K times, requires shifting $(A_K + 1) \cdot (N + 1)$ control bits. At this point, no more input data exists for the instruments for SIB_K through SIB_N, and therefore, SIB_d is closed to shorten the scan path for the rest of instruments. Once SIB_d is closed, the rest of input data (i.e. those left from A_1) are left to be applied. This requires $(A_1 - A_K - 1) \cdot K$ more control bits to be shifted in. Therefore, the total shift overhead for the

Algorithm 4.1: Method for the concurrent schedule

```
1  l := 1                                          // Initially the design has one level
2  I := {A_1, A_2, ..., A_N}                       // Initially I contains all the instruments
3  while |I| > 2 do
4  |   Starting from A_2, find K that satisfies Eq. (4.3) for the instruments in I
5  |   if there is no such K then
6  |   |   break                                   // No reduction is possible
7  |   end
8  |   I_l := First K − 1 instruments              // Current level gets the first K-1 instruments in I
9  |   I := I \ I_l                                // The used instruments are removed from I
10 |   l := l + 1                                  // A new level is added for the rest of the instruments
11 end
12 I_l := I                                        // The last level contains the remainder of the instruments
```

design in Figure 4.1(b) is calculated as:

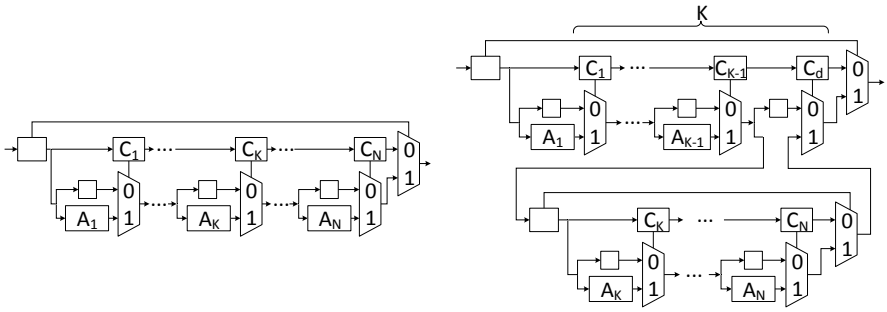
$$O = K + (N+1) + (A_K + 1) \cdot (N+1) + (A_1 - A_K - 1) \cdot K \tag{4.2}$$

Based on these calculations, it can be concluded that if Eq. (4.3) is satisfied for the set of N instruments shown in Figure 4.1, the design in Figure 4.1(b) will result in less shift overhead, at the cost of the additional $\mathrm{SIB_d}$.

$$\begin{aligned} K + (N+1) + (A_K + 1) \cdot (N+1) + (A_1 - A_K - 1) \cdot K \\ < N + (A_1 + 1) \cdot N \end{aligned} \tag{4.3}$$

Based on this observation, Algorithm 4.1 is presented for the construction of SIB-based networks that are optimized w.r.t. OAT for the concurrent schedule.

In Algorithm 4.1, l is the hierarchical level number, which starts at one (Line 1) and is incremented (Line 10) for each successful introduction of a new hierarchical level. Initially, set I contains N instruments, represented by their number of accesses. Instruments in I are arranged in descending order based on their number of accesses (Line 2). If K can be found such that Eq. (4.3) is satisfied for instruments in I (Line 4), the first $K-1$ instruments should remain on the hierarchy level specified by l. Therefore, these instruments are stored in set I_l for that level (Line 8), and are removed from set I (Line 9). The rest of instruments are moved to the next level of hierarchy (they remain in I for further processing). This continues until there are only two instruments in I (Line 3), or no K can be found to satisfy Eq. (4.3) (Line 6). The outcome of Algorithm 4.1 is a list of instrument sets, named $I_1, I_2, \ldots, I_l$, where I_1 contains the instruments on the first level, I_2 contains the instruments for the second level, and so on. It should be noted that when the observation regarding



(a) In a single-level network (b) In a two-level network

Figure 4.2. N instruments in single-level and two-level Daisy-chained networks, analogous to the networks in Figure 4.1

Eq. (4.3) is applied on Line 4, A_1 in Eq. (4.3) refers to the first element in the current set of instruments stored in I. Furthermore, adding hierarchy levels is done by adding a doorway SIB such as SIB_d in Figure 4.1(b). There will be at most one doorway SIB at each level of hierarchy in the constructed network.

Algorithm 4.1 is a greedy method, which we presented in [45]. Later, [44] presented a construction method (for the concurrent schedule) for SIB-based networks that results in minimal OAT. The experimental results in [44] showed up to 12.7 percent reduction in shift overhead compared to the results achieved by Algorithm 4.1.

4.1.1.2. DAISY-CHAINED NETWORKS

To construct Daisy-chained networks that are optimized w.r.t OAT for the concurrent schedule, the same design method presented for the SIB-based networks is applicable. That is, one could derive a condition similar to Eq. (4.3) for the Daisy-chained networks and use Algorithm 4.1. However, our early experiments showed that a Daisy-chained network constructed using analogy from the architecture of its optimized SIB-based counterpart, results in low shift overhead. Figure 4.2 shows flat and hierarchical Daisy-chained networks for the same N instruments used in the SIB-based networks in Figure 4.1, where each of the flat and hierarchical Daisy-chained networks can be seen analogous to the flat and hierarchical SIB-based networks, respectively. For the experiments presented in this thesis, we use the analogy from optimized SIB-based networks to construct Daisy-chained networks.

4.1.1.3. REMOTE NETWORKS

For Remote networks, to minimize shift overhead for the concurrent schedule, it suffices to order the instruments on the scan path based on the number of accesses. Such ordering removes the R_m parameter from Eq. (3.10), thus effectively reducing shift overhead.

4.1.2. THE SEQUENTIAL SCHEDULE

This section presents the design of 1687 networks with the objective of shift overhead reduction for the sequential schedule. We consider SIB-based, Daisy-chained, and Remote networks.

4.1.2.1. SIB-BASED NETWORKS

In sequential schedules, instruments are accessed one at a time. Therefore, the total shift overhead will be the sum of the shift overheads due to accessing each of instruments. As mentioned earlier in this chapter, the instrument with the largest number of accesses can have the largest contribution to the shift overhead. Such instruments should be placed on a short scan path. In a hierarchical network, instruments with large number of accesses should be placed on a level close to the TAP to avoid many SIBs on their scan paths. Also, instruments with smaller number of accesses should be placed on a level farther from the TAP so that their instrument SIBs do not add to the length of scan path to the instruments that are more frequently accessed. To develop an algorithm for constructing a SIB-based network with such placement of instruments, we have taken inspiration from Huffman tree construction, which is a method for constructing labeled trees of symbols, used in variable-length coding [46]. The basic idea in Huffman tree construction is that symbols with higher frequency of occurrence (weight) are assigned code words of shorter length. To construct such a tree, symbols with larger weights are placed closer to the root of the tree.

In the following, we explain Huffman tree construction by using the example of seven symbols having weights of 1, 1, 1, 1, 5, 8, and 25. These symbols are represented by their weights in Figure 4.3(a). Figure 4.3(g) presents the Huffman tree constructed for binary encoding of these symbols. In Figure 4.3(g), the root of tree is the node marked with weight 42. The leaf nodes are the seven symbols, and the internal nodes are marked with the combined weight of the symbols in their subtrees, hence weight 42 for the root node. Each edge is labeled with either '0' or '1'. To encode a symbol, the tree is traversed from the root node towards the leaf representing that symbol, and the labels of edges on the path are recorded as the code word. It can be seen that the symbol with weight 25 has received the shortest length binary code word

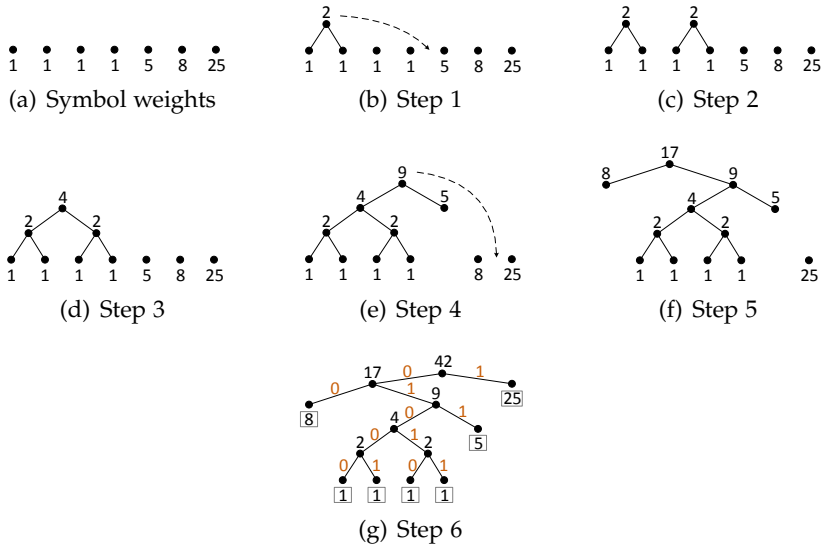
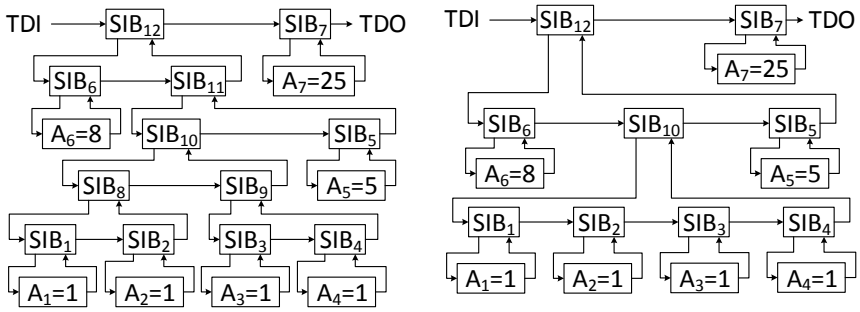


Figure 4.3. Steps in Huffman tree construction

"1", whereas a symbol with weight 1 has received the code word of "01000". In the following, construction of this tree will be explained with the help of Figure 4.3. Initially, the symbols are sorted in the ascending order based on their weights and the first two symbols are *combined*, as shown in Figure 4.3(b). The combination can be viewed as a symbol having weight of 2. This *combined* symbol must be placed such that the list is kept sorted, as shown by the dashed arrow in Figure 4.3(b). Again, starting from the beginning of the list, the first two symbols, which both have weight 1, are combined (Figure 4.3(c)). The process of combining the first two symbols in the list and repositioning will continue until only one symbol is left (the symbol with weight 42 in our example), in which case the construction algorithm terminates.

In construction of a SIB-based network, analogy can be made between weight of a symbol (in a Huffman tree) and the number of accesses for an instrument. That is, instruments that are accessed more frequently, can be placed in the network such that number of SIBs on their scan path (analogous to the length of the code word for the symbol) becomes smaller than the number of SIBs on the scan path to less frequently accessed instruments.

Figure 4.4(a) shows a network constructed by such analogy for a set I of instruments with the following number of accesses: $I = \{1, 1, 1, 1, 5, 8, 25\}$. As can be seen, each instrument's number of accesses has determined the hierarchical level at which that instrument is placed. For example, Instrument



(a) Output of Algorithm 4.2 (b) Output of Algorithm 4.3

Figure 4.4. Example 1687 networks

7 with the highest number of accesses (i.e., A_7=25) is placed such that it can be accessed with only two SIBs on the scan path. In Figure 4.3 we saw how symbols were grouped to create combined symbols. The same grouping idea applies here by placing two instruments on the host port of a newly added doorway SIB. For example, in Figure 4.4(a) Instrument 1 and Instrument 2 are added to the host port of SIB_8, which can be seen as a *combined* instrument with two accesses (i.e., A_8=2). The same grouping idea can be performed on an instrument and a combined instrument. For example, Instrument 5 is grouped with a combined instrument represented by SIB_{10}. If the instruments in Figure 4.4(a) were arranged in flat architecture, the shift overhead would be 350 clock cycles for the sequential schedule, while the shift overhead for the design in Figure 4.4(a) is 244 cycles. Therefore, reduction of shift overhead is achieved at the cost of five additional doorway SIBs (SIB_8 through SIB_{12}).

We will now present Algorithm 4.2, which uses the idea in Huffman tree construction to construct a network out of a given set of instruments, such that the shift overhead is minimized for the sequential schedule. As input, the algorithm receives a set of instruments represented by their number of accesses (Line 1). The assumption is that each instrument has a dedicated instrument SIB. The algorithm performs the grouping of instruments iteratively until only two instruments are left in I. In each iteration (Lines 2–6), two instruments with the lowest number of accesses are selected (Line 3) and grouped to create a combined instrument (Line 4). The combined instrument is represented by set X containing all instruments grouped together in it. This combined instrument is then treated as an instrument having $A_X = \sum_{i \in X} A_i$ number of accesses. The instruments used to create X are then replaced by X in I (Line 5).

Figure 4.4(a) is the output of Algorithm 4.2 run for $I = \{1, 1, 1, 1, 5, 8, 25\}$.

Algorithm 4.2: Construction for sequential schedule

```
1 I := {A_1, A_2, ..., A_N}
2 while |I| > 2 do
3 |   Find A_i and A_j that are smaller than all other items in I
4 |   Combine the two instruments i and j to form X
5 |   Replace A_i and A_j with A_X in I
6 end
```

Algorithm 4.3: Complete method for sequential schedule

```
1 Run Algorithm 4.2
2 for each SIB_d do
3 |   ShiftOverhead := shift overhead of the network
4 |   Remove SIB_d
5 |   NewShiftOverhead := shift overhead of the network
6 |   if NewShiftOverhead > ShiftOverhead then
7 |   |   Restore SIB_d
8 |   end
9 end
```

It is possible to further reduce the shift overhead of 244 clock cycles in the network in Figure 4.4(a) by removing SIB_8, SIB_9 and SIB_{11}. For the resulting network shown in Figure 4.4(b) the shift overhead for sequential schedule is 215 clock cycles. The reason for this possibility of further reduction in shift overhead is that in the analogy to Huffman tree construction, there is no counterpart for the shift overhead coming from opening the SIBs before the first access to a given instrument. Moreover, a Huffman tree is a binary tree (in which each internal node has two child nodes) whereas we are not limited to this constraint in constructing a SIB-based network. An optimization step can therefore follow the network construction. The optimization step analyzes a network and finds doorway SIBs that should be removed to further reduce the shift overhead. The complete method for the sequential schedule is thus as suggested in Algorithm 4.3.

The basic idea in Algorithm 4.3 is to construct an initial network, using Algorithm 4.2, and examine the effect of removal of each of the doorway SIBs in that network (Line 4) on the total shift overhead. Removal of a doorway SIB is done by replacing the doorway SIB by the network segment on its host port. To this end, Algorithm 4.3 compares the shift overhead before (Line 3) and after (Line 5) removal of each of the doorway SIBs, and restores the removed SIB (Line 7) should the shift overhead increase after removal of the SIB (Line 6).

4.1.2.2. DAISY-CHAINED NETWORKS

To construct Daisy-chained networks optimized for sequential schedule, we again use the analogy from an optimized SIB-based network.

4.1.2.3. REMOTE NETWORK

For the Remote networks, not much can be done to lower the shift overhead under the sequential schedule. Eq. (3.14) shows the only contributer to shift overhead to be the flushing of $N-1$ bypass flip-flops when access to one instrument is complete. This number is not dependent on the order of the instruments and therefore, reordering of instruments on the scan path is not going to affect it. On the other hand, shift overhead can be negligible (as was shown in the parametric analysis in Section 3.5.3) when the number of accesses is relatively large.

4.1.3. GENERIC SCHEDULES

In this section, we describe heuristics for designing 1687 networks that are optimized for a given generic schedule. Here again, we consider SIB-based, Daisy-chained, and Remote networks.

4.1.3.1. SIB-BASED NETWORKS

In this section, we describe a heuristic method for optimized design of SIB-based networks for a given generic schedule. Prior work [44] presented heuristics for design of SIB-based networks that are optimized for *hybrid* access schedules. Hybrid access schedules[1] can be seen as a succession of concurrent schedules, in which a session ends when all accesses that have started in that session are complete. This is in contrast to generic schedules in which a new session begins as soon as accesses to any instrument is complete.

Our proposed heuristic method recursively decomposes the given schedule into sequential and concurrent blocks and constructs network segments upon each return from a (recursive) function call. We will explain this shortly with the help of the example schedule in Figure 4.5 for eleven instruments, where the number of accesses for each instrument is reported in Table 4.1. However, before proceeding to explain the proposed method, we need to clarify some of the used principles and assumptions.

First, to identify sequential and concurrent blocks in the schedule, we need to determine the amount of concurrent accesses between different instruments (as specified by the given schedule). To quantify such amount of concurrency between an instrument i and an instrument j, we define a concurrency ratio

[1] Also referred to as session-based schedules in SoC test scheduling terminology [37].

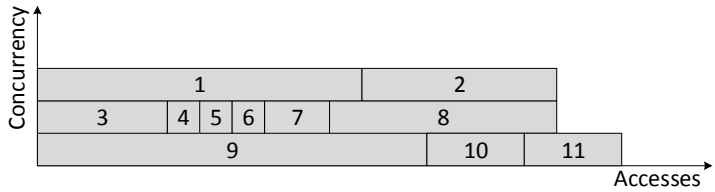


Figure 4.5. The given generic schedule for the set of instruments in Table 4.1

Table 4.1. Number of accesses for the set of instrument in Figure 4.5

Instrument	1	2	3	4	5	6	7	8	9	10	11
Accesses	50	30	20	5	5	5	10	35	60	15	15

as:

$$cr_{i,j} = A_{ij}/A_i \tag{4.4}$$

where A_{ij} is the total number of accesses shared between instruments i and j and A_i is the total number of accesses for instrument i. We consider accessing instruments i and j concurrent if $cr_{i,j} \geq 0.5$. Based on the concurrency ratio, a concurrency matrix CR is created for all instruments whose elements are $cr_{i,j}$. For example, the CR matrix for the schedule in Figure 4.5 is as follows:

$$CR = \begin{bmatrix} 1 & 0 & 0.4 & 0.1 & 0.1 & 0.1 & 0.2 & 0.1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0.3 & 0.5 & 0.2 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0.1 & 0.9 & 0 & 0 & 0 & 0 & 0 & 1 & 0.4 & 0.4 & 0.1 \\ 0.8 & 0.2 & 0.3 & 0.1 & 0.1 & 0.1 & 0.2 & 0.2 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0.3 & 0 & 0 & 0 & 0 & 0 & 0.3 & 0 & 0 & 1 \end{bmatrix}$$

In the above matrix, the number in row i and column j is $cr_{i,j}$. Note that this matrix is not symmetric.

In the following, we explain Function ConstructForGeneric(), which is the main algorithm in the proposed method. Before the function is called, set of instruments I should be initialized with a list of instruments sorted in the descending order based on the number of accesses. For our example, $I = \{9, 1, 8, 2, 3, 10, 11, 7, 4, 5, 6\}$. Moreover, the function uses a set C as a global variable, to keep track of the instruments considered concurrently accessed. It

should be noted that an instrument i is considered concurrent with all instruments in the set C if $\forall c \in C,\ cr_{i,c} \geq 0.5$. Set C is initially empty. By definition, we consider an instrument to be concurrently accessed with an empty set C.

Next, to represent how in Function ConstructForGeneric() we construct the network segments for the identified blocks, we use the two following functions:

- `SIB(S)` receives a set S of network segments as input and adds those segments in series to the host port of a newly added SIB.
- `ConstructForSequential(S)` receives a set S of segments as input and uses Algorithm 4.3 to create an optimized network for sequential access out of those segments. Algorithm 4.3 receives a set of instruments as input whereas `ConstructForSequential()` receives a set of segments. To use Algorithm 4.3, `ConstructForSequential()` creates virtual instruments as placeholders for each segment where the number of accesses for each of those virtual instruments is the largest number of accesses found among the instruments inside each segment.

Given the above assumptions, we now proceed to describe the operation of Function ConstructForGeneric() with the help of our example schedule (Figure 4.1). When Function ConstructForGeneric() is called for this schedule, the first instrument in set I is instrument 9. As C is initially empty, condition in Line 3 passes and instrument 9 is added to set C. When in Line 6 the function calls itself recursively for the first time, the next instrument in set I passing the condition in Line 3 is instrument 1, which is also added to C. When the function calls itself recursively for the second time, instrument 3 will pass the condition (since $cr_{3,9} = 1$ and $cr_{3,1} = 1$) and is added to C. At this point, $C = \{9, 1, 3\}$. When the function calls itself recursively for the third time, no other instrument passes the condition and the function returns an *empty* netlist (Line 18). Upon returning from this third call, s_i is set to the netlist segment shown in Figure 4.6(a) (Line 7). Subsequently, s_i is added to S (Line 9) and instrument 3 is removed from C (Line 14), thus $C = \{9, 1\}$. Now the function is back to the second call, and the next item in I, which is instrument 4, is selected. For instrument 4, the same procedure as the one for instrument 3 is repeated. The same goes for instruments 5, 6, and 7 as well. As there are no more instruments to have concurrency with $C = \{9, 1\}$, the function gets to Line 21 and returns an optimized network for the segments in S (Figure 4.6(a)–Figure 4.6(e)), which is the segment in Figure 4.6(i). So far, the function has detected that instruments 3, 4, 5, 6, and 7 are each accessed concurrently with instruments 9 and 1, but accessed sequentially with each other, and has constructed a network segment for them, accordingly.

Function ConstructForGeneric()

```
1  S := ∅
2  foreach i ∈ I do
3  |  if ∀c ∈ C, cr_{i,c} ≥ 0.5 then
4  |  |  C := C ∪ {i}
5  |  |  I := I \ {i}
6  |  |  s := ConstructForGeneric ()
7  |  |  s_i := SIB({i})
8  |  |  if s is empty then
9  |  |  |  S := S ∪ s_i
10 |  |  end
11 |  |  else
12 |  |  |  S := S ∪ SIB({s, s_i})
13 |  |  end
14 |  |  C := C \ {i}
15 |  end
16 end
17 if S = ∅ then
18 |  return empty
19 end
20 else
21 |  return ConstructForSequential (S)
22 end
```

After this return from the second recursive call, the calling function gets to Line 7 for instrument 1, and creates the segment in Figure 4.6(f). This time, since the returned netlist is not empty, Line 12 is executed, generating a new segment (Figure 4.6(j) minus SIB_{11}) which after Line 21 is as shown in Figure 4.6(j).

After returning from the first recursive call, similar to the procedure for instrument 1, segment in Figure 4.6(k) is constructed after adding instrument 9. At this point, $C = \emptyset$ again, set S contains the segment in Figure 4.6(k), and the next instrument selected is instrument 8. In the same manner explained so far, segments shown in Figure 4.6(h), Figure 4.6(l), and Figure 4.6(m) are constructed after three recursive calls.

At this point, again the algorithm is back from the first recursive call. Now, $C = \emptyset$ again, and set S contains the segments in Figure 4.6(k) and Figure 4.6(m). The only remaining instrument in I is instrument 11 for which a segment as shown in Figure 4.6(g) is constructed and added to S. At this point, set S contains the segments shown in Figure 4.6(k), Figure 4.6(m), and Figure 4.6(g). Through a final call to `ConstructForSequential()`, the network shown in Figure 4.7(a) is constructed. This network is them optimized

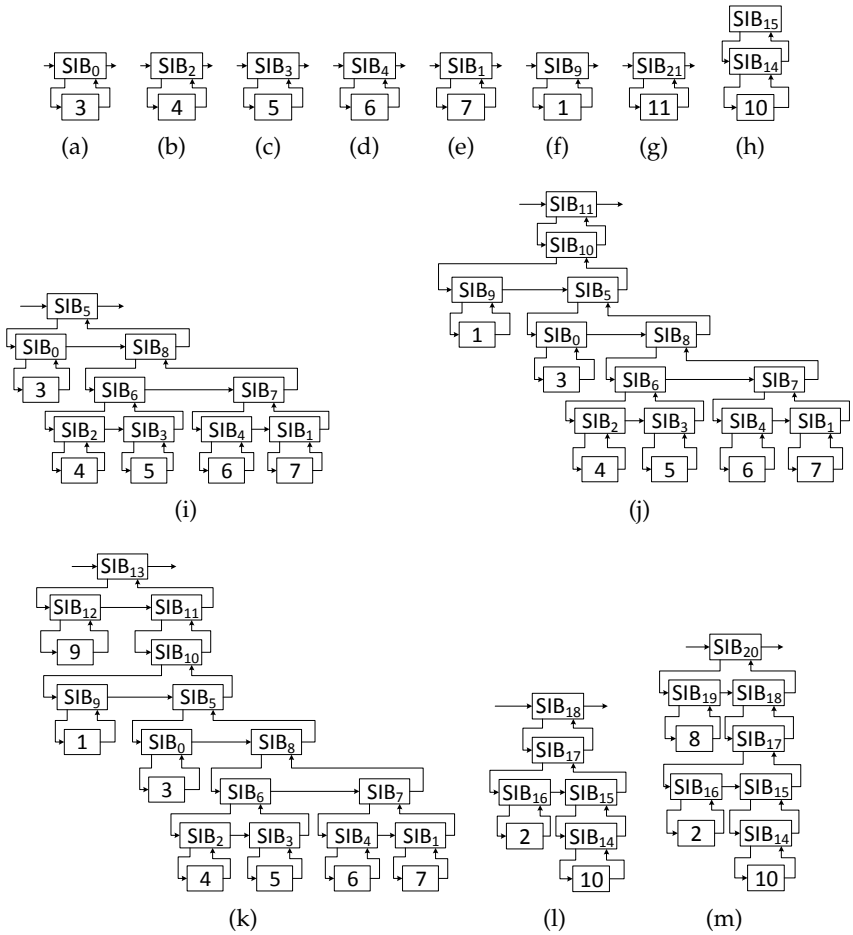
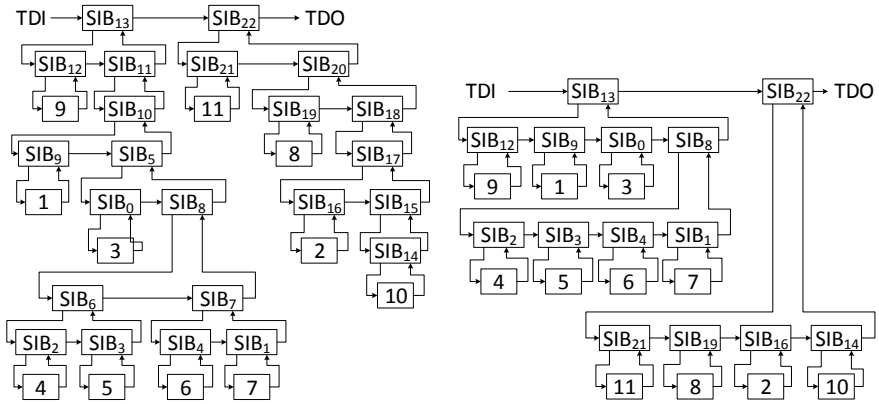


Figure 4.6. Segments generated through the operation of Function ConstructForGeneric() for the schedule in Figure 4.5

by removing redundant SIBs, resulting in the network shown in Figure 4.7(b). The removal of redundant SIBs is performed similarly to how Algorithm 4.3 removed extra SIBs.

4.1.3.2. DAISY-CHAINED NETWORKS

As was the case for the concurrent and sequential access schedules, here again we use analogy from optimized SIB-based networks to construct Daisy-chained networks.



(a) Before running Algorithm 4.3 (b) After running Algorithm 4.3

Figure 4.7. Resulting network for the schedule in Figure 4.5

4.1.3.3. REMOTE NETWORKS

As was discussed earlier on network design for the concurrent schedule, the ordering of instruments on the scan path can affect shift overhead through reducing the R_m variable in Eq. (3.10). In this section, we present a greedy heuristic for ordering of instruments on the scan path. The basic idea is that the instruments that have the most number of concurrent accesses together are placed immediately after each other on the scan path, so that R_m is minimized. The algorithm starts by placing the instrument with the highest number of accesses on the scan path. The next instrument is selected such that it has the highest number of concurrent accesses with all previously placed instruments, and so on.

4.1.4. EXPERIMENTS

To evaluate the effectiveness of the proposed design methods in reducing the shift overhead, we implemented the corresponding algorithms and carried out a number of experiments. To run the algorithms, sets of instruments were needed, and since there were no benchmarks available for IEEE 1687[2], we chose to create such sets based on the ITC'02 benchmark set [48]. The ITC'02 set consists of 12 benchmark SoCs. For each SoC, list of modules (i.e., cores) are given, and for each module, number of I/O terminals and internal scan-chains, as well as number of patterns to apply to those terminals and

[2]Recently, a set of benchmarks are presented for experimenting with 1687 networks [47].

Table 4.2. Benchmarks used for the experiments with a single schedule

Benchmark name	Instrument data	Number of instruments	Length of shift-registers			Number of accesses		
			min	ave.	max	min	ave.	max
a586710	838530522	26	34	1545	2626	2945	166573	1914433
d281	1496291	48	7	48	233	26	907	2048
d695	704057	157	1	52	320	12	89	234
f2126	5330439	34	20	447	1000	103	339	422
g1023	736216	63	9	81	377	15	133	1024
h953	1197178	44	9	125	348	9	169	341
p22810	7784963	254	1	117	400	1	352	12324
p34392	16403755	103	4	224	806	27	1620	12336
p93791	30083283	586	1	166	538	11	356	6127
q12710	31801946	21	413	1245	3784	852	1160	1314
t512505	165400967	126	2	607	1669	3	1035	3370
u226	252929	30	3	42	97	15	589	2666

scan-chains are specified. Appendix D details how we have extracted sets of instruments based on the ITC'02 SoCs.

Table 4.2 lists some properties of the extracted instrument sets corresponding to each of the SoCs. The first column presents the SoC name from the ITC'02 set. The second column presents instrument data for each set calculated by using Eq. (3.2). The third column lists the number of instruments included in each set. Columns 4–6 present the minimum, average, and maximum length found among instrument shift-registers in the set. Finally, columns 7–9 present the minimum, average, and maximum number of accesses found among instruments in the set.

In the following, experimental results are presented for each network type separately. To perform the experiments, we have used the OAT calculation algorithms presented in Chapter 3. As the basic optimization idea (for lowering OAT) in all algorithms is to reduce the shift overhead, to avoid clutter in presentation of the results, we only report the **ratio of shift overhead to instrument data**. The complete data including all OAT components in TCKs (as well as ratios) are presented in Appendix C. We note that the instrument data is the amount of data that should be transported to instruments no matter what network type, network architecture, or schedule is used. Therefore, the shift overhead to instrument data ratio shows how much overhead is introduced, e.g., by the network design. Regarding the TAP overhead, it can be seen in the results reported in Appendix C that the TAP overhead does not vary noticeably between different networks, as was also observed in the analysis presented in Section 3.5.

For all networks, four schedules are considered: sequential schedule, concurrent schedule, a generic schedule in which 10 percent of instruments are accessed concurrently (denoted as **G10** in the presentation of the experimen-

tal results), and a generic schedule in which 25 percent of instruments are accessed concurrently (denoted as **G25** in the presentation of the experimental results).

4.1.4.1. SIB-BASED NETWORKS

For SIB-based networks, the following explains the design methods compared against each other in the experiments:

- Networks represented by F in the results have flat architectures, and are used as a baseline for measuring the reduction in shift overhead achieved via other methods.
- Networks represented by H are constructed by Algorithm 4.2 (i.e., they are optimized for the sequential schedule)[3].
- Networks represented by HPO are constructed by Algorithm 4.3 (i.e., designed by Algorithm 4.2 with post optimization).
- Networks represented by OC are constructed by Algorithm 4.1 (i.e., they are optimized for the concurrent schedule).
- Networks represented by OG10 are optimized by using Function ConstructForGeneric() for the G10 schedule described earlier.
- Networks represented by OG25 are optimized by using Function ConstructForGeneric() for the G25 schedule described earlier.

Table 4.3 presents the experimental results for the constructed SIB-based networks. As was mentioned earlier, we use the ratio of shift overhead to instrument data as the basis for comparison of the constructed networks for each benchmark. In the table, for each benchmark, and for each schedule, the lowest ratio is marked in bold face, signifying that the corresponding design method performs better than the other methods for that schedule.

From the results for the sequential schedule, it can be seen that the HPO network results in the lowest ratio for all benchmarks. The largest improvement over the F network is seen for the p93791 benchmark and the smallest belongs to the q12710 benchmark. Comparing these benchmarks (Table 4.2) reveals that compared to q12710, the p93791 benchmark has many instruments—thus many SIBs and much shift overhead—while having similar instrument data, leading to a smaller ratio. For these two benchmarks, if we consider the overhead percentage (by using Eq. (3.15)), we observe that for p93791, the F network shows 80 percent overhead whereas the HPO network

[3]Letter H was chosen as this algorithm is based on the Huffman method.

shows 13 percent overhead. For the q12710 benchmark, the F network shows 1.9 percent overhead and the HPO network shows 0.9 percent overhead. It can be concluded that a reduction in the ratio of shift overhead to instrument data is a good indicator of how much the total overhead percentage is reduced.

Regarding the concurrent schedule, it is the OC network that outperforms the others w.r.t. the ratio. The largest improvement over the F network is seen for p22810 and the smallest (actually no improvement) is seen for q12710. The large improvement for p22810 comes from the fact that there are few instruments with many accesses in this benchmark and many instruments with few accesses. In a flat network, all SIBs are always on the scan path and contribute to overhead for all accesses performed only on those few instrument with many accesses. Therefore, by placing some of the SIBs in another hierarchical level, the shift overhead can be reduced significantly. In case of q12710, the reason that the OC network does not reduce the shift overhead much compared to the F network (or other networks), is that the shift overhead for the flat network (F) constitutes only a very small fraction of the OAT. The overhead numbers in clock cycles are reported in Appendix C. Here, we explain how this overhead is calculated to better relate it to the properties of this benchmark. For q12710, there are (21 instruments and thus) 21 instrument SIBs in the F network constructed for this benchmark. Since the largest number of accesses is 1314, the shift overhead for the concurrent schedule amount to $21_{\text{setup}} + (1314 + 1) \times 21 = 27636$ clocks, which compared to the instrument data is negligible. This low overhead leaves little room for improvement for the OC (or any other) design method.

For the OG10 and OG25 networks, it can be seen that with the exception of the a586710 benchmark, they result in the lowest ratios for their corresponding schedule.

Table 4.3. Experimental results: shift overhead to instrument data ratio in SIB-based benchmark networks

Benchmark name	Instrument data	Design method	Ratio of shift overhead to instrument data			
			Sequential	Concurrent	G10	G25
a586710	838530522	F	0.134	0.059	0.071	0.060
		H	0.019	0.010	0.014	0.011
		HPO	**0.018**	0.008	**0.011**	**0.008**
		OC	0.019	**0.008**	0.011	0.008
		OG10	0.024	0.010	0.013	0.011
		OG25	0.023	0.010	0.013	0.010

continues on next page

Table 4.3. Experimental results: shift overhead to instrument data ratio in SIB-based benchmark networks

Benchmark name	Instrument data	Design method	Ratio of shift overhead to instrument data			
			Sequential	Concurrent	G10	G25
d281	1496291	F	1.399	0.066	0.371	0.140
		H	0.278	0.059	0.187	0.110
		HPO	**0.268**	0.050	0.171	0.094
		OC	0.643	**0.031**	0.188	0.072
		OG10	0.318	0.051	**0.098**	0.062
		OG25	0.415	0.037	0.128	**0.047**
d695	704057	F	3.155	0.053	0.245	0.116
		H	0.282	0.043	0.145	0.099
		HPO	**0.277**	0.032	0.127	0.080
		OC	1.319	**0.022**	0.195	0.083
		OG10	0.357	0.033	**0.048**	0.039
		OG25	0.538	0.028	0.070	**0.036**
f2126	5330439	F	0.074	0.003	0.025	0.011
		H	0.022	0.004	0.015	0.011
		HPO	**0.021**	0.004	0.014	0.010
		OC	0.050	**0.002**	0.020	0.009
		OG10	0.021	0.004	**0.009**	0.007
		OG25	0.028	0.003	0.011	**0.005**
g1023	736216	F	0.723	0.088	0.174	0.114
		H	0.119	0.030	0.073	0.052
		HPO	**0.117**	0.026	0.068	0.046
		OC	0.243	**0.015**	0.086	0.042
		OG10	0.138	0.026	**0.038**	0.030
		OG25	0.152	0.020	0.048	**0.025**
h953	1197178	F	0.276	0.013	0.073	0.030
		H	0.064	0.014	0.044	0.030
		HPO	**0.062**	0.011	0.039	0.025
		OC	0.121	**0.007**	0.050	0.023
		OG10	0.071	0.013	**0.024**	0.017
		OG25	0.087	0.011	0.030	**0.015**
p22810	7784963	F	2.927	0.402	0.442	0.411
		H	0.143	0.030	0.061	0.046
		HPO	**0.140**	0.026	0.054	0.039
		OC	0.609	**0.015**	0.079	0.035
		OG10	0.197	0.020	**0.026**	0.021
		OG25	0.254	0.018	0.034	**0.021**
p34392	16403755	F	1.048	0.077	0.143	0.092
		H	0.110	0.026	0.054	0.037
		HPO	**0.107**	0.023	0.050	0.033
		OC	0.275	**0.013**	0.078	0.029
		OG10	0.142	0.020	**0.028**	0.022
		OG25	0.175	0.017	0.036	**0.020**

continues on next page

Table 4.3. Experimental results: shift overhead to instrument data ratio in SIB-based benchmark networks

Benchmark name	Instrument data	Design method	Ratio of shift overhead to instrument data			
			Sequential	Concurrent	G10	G25
p93791	30083283	F	4.077	0.119	0.162	0.132
		H	0.120	0.016	0.046	0.032
		HPO	**0.119**	0.013	0.041	0.028
		OC	1.435	**0.008**	0.066	0.028
		OG10	0.234	0.010	**0.013**	**0.011**
		OG25	0.395	0.009	0.022	0.011
q12710	31801946	F	0.016	0.001	0.008	0.003
		H	0.007	0.002	0.005	0.004
		HPO	**0.007**	0.001	0.005	0.003
		OC	0.012	**0.001**	0.008	0.003
		OG10	0.007	0.001	0.004	0.003
		OG25	0.007	0.001	**0.004**	**0.002**
t512505	165400967	F	0.099	0.003	0.009	0.005
		H	0.009	0.002	0.005	0.003
		HPO	**0.009**	0.001	0.004	0.002
		OC	0.030	**0.001**	0.005	0.003
		OG10	0.013	0.001	**0.002**	0.002
		OG25	0.018	0.001	0.003	**0.001**
u226	252929	F	2.103	0.316	0.718	0.367
		H	0.464	0.140	0.275	0.184
		HPO	**0.440**	0.117	0.234	0.157
		OC	0.648	**0.081**	0.276	0.150
		OG10	0.503	0.126	**0.196**	0.149
		OG25	0.616	0.094	0.229	**0.118**

4.1.4.2. DAISY-CHAINED NETWORKS

The Daisy-chained networks used for the experiments are constructed by analogy from the SIB-based networks constructed for the same benchmarks and the same schedules (see Section 4.1.4.1).

Table 4.4 presents the experimental results for Daisy-chained benchmark networks. As was mentioned earlier, we use the ratio of shift overhead to instrument data as the basis for comparison of the constructed networks for each benchmark. In the table, for each benchmark, and for each schedule, the lowest ratio is marked in bold face, signifying that the corresponding design method performs better than the other methods for that schedule. The same observations as those for SIB-based networks can be made here as well. There some exceptions though. For example, the OG10 network does not yield the lowest overhead ratio for the q12710 benchmark in case of the G10 schedule. In this case, however, the resulting overhead of 112901 TCKs is very close to that of the best among networks (namely, OG25) being 110621 TCKs (see

Appendix C for the actual overhead numbers).

Table 4.4. Experimental results: shift overhead to instrument data ratio in Daisy-chained benchmark networks

Benchmark name	Instrument data	Design method	Ratio of shift overhead to instrument data			
			Sequential	Concurrent	G10	G25
a586710	838530522	F	0.134	0.056	0.069	0.057
		H	0.019	0.007	0.011	0.008
		HPO	**0.018**	0.005	**0.008**	**0.005**
		OC	0.019	**0.005**	0.009	0.006
		OG10	0.024	0.008	0.011	0.008
		OG25	0.023	0.007	0.011	0.007
d281	1496291	F	1.400	0.038	0.350	0.115
		H	0.278	0.031	0.166	0.084
		HPO	**0.268**	0.022	0.150	0.068
		OC	0.644	**0.003**	0.168	0.046
		OG10	0.319	0.023	**0.077**	0.036
		OG25	0.415	0.009	0.107	**0.021**
d695	704057	F	3.190	0.035	0.250	0.114
		H	0.286	0.023	0.129	0.082
		HPO	**0.280**	0.012	0.109	0.062
		OC	1.341	**0.002**	0.188	0.068
		OG10	0.361	0.014	**0.032**	0.021
		OG25	0.545	0.009	0.055	**0.019**
f2126	5330439	F	0.074	0.001	0.024	0.009
		H	0.022	0.002	0.014	0.009
		HPO	**0.021**	0.002	0.013	0.008
		OC	0.050	**0.000**	0.019	0.008
		OG10	0.021	0.002	**0.008**	0.005
		OG25	0.028	0.001	0.010	**0.003**
g1023	736216	F	0.728	0.079	0.169	0.108
		H	0.120	0.020	0.065	0.042
		HPO	**0.118**	0.017	0.060	0.037
		OC	0.247	**0.005**	0.082	0.034
		OG10	0.139	0.017	**0.030**	0.021
		OG25	0.153	0.010	0.041	**0.016**
h953	1197178	F	0.278	0.007	0.069	0.025
		H	0.065	0.008	0.039	0.024
		HPO	**0.063**	0.005	0.035	0.020
		OC	0.122	**0.001**	0.046	0.017
		OG10	0.071	0.007	**0.020**	0.011
		OG25	0.087	0.005	0.026	**0.009**
p22810	7784963	F	2.935	0.393	0.439	0.407
		H	0.144	0.020	0.052	0.036
		HPO	**0.141**	0.016	0.044	0.029
		OC	0.614	**0.005**	0.072	0.027
		OG10	0.198	0.010	**0.016**	0.012
		OG25	0.255	0.009	0.025	**0.011**

continues on next page

Table 4.4. Experimental results: shift overhead to instrument data ratio in Daisy-chained benchmark networks

Benchmark name	Instrument data	Design method	Ratio of shift overhead to instrument data			
			Sequential	Concurrent	G10	G25
p34392	16403755	F	1.049	0.068	0.135	0.084
		H	0.110	0.016	0.045	0.028
		HPO	**0.107**	0.013	0.041	0.024
		OC	0.276	**0.004**	0.070	0.020
		OG10	0.142	0.011	**0.019**	0.013
		OG25	0.175	0.007	0.027	**0.010**
p93791	30083283	F	4.088	0.113	0.165	0.132
		H	0.121	0.009	0.040	0.026
		HPO	**0.120**	0.006	0.035	0.021
		OC	1.442	**0.001**	0.062	0.023
		OG10	0.235	0.003	**0.007**	**0.005**
		OG25	0.397	0.002	0.016	0.005
q12710	31801946	F	0.016	0.000	0.008	0.003
		H	0.007	0.001	0.005	0.003
		HPO	**0.007**	0.001	0.005	0.003
		OC	0.012	**0.000**	0.007	0.003
		OG10	0.007	0.001	0.004	0.002
		OG25	0.007	0.000	**0.003**	**0.001**
t512505	165400967	F	0.100	0.002	0.009	0.004
		H	0.009	0.001	0.004	0.002
		HPO	**0.009**	0.000	0.003	0.002
		OC	0.030	**0.000**	0.005	0.002
		OG10	0.013	0.001	**0.001**	0.001
		OG25	0.018	0.000	0.002	**0.001**
u226	252929	F	2.106	0.257	0.674	0.310
		H	0.465	0.081	0.231	0.127
		HPO	**0.441**	0.058	0.189	0.100
		OC	0.651	**0.022**	0.231	0.093
		OG10	0.504	0.066	**0.151**	0.092
		OG25	0.617	0.035	0.184	**0.061**

4.1.4.3. REMOTE NETWORKS

For Remote networks, the presented optimization methods were based on ordering the instruments on the scan path, based on their number of accesses. However, the way that the instruments were extracted from the ITC'02 benchmarks made many instruments with exactly the same number of accesses appear next to each other in the set of instruments for each benchmark. This initial ordering would give the impression that the ordering of instruments has little (or no) effect on the shift overhead. Therefore, we used two different randomizations of the set of input instruments for each benchmark to create Remote networks used as the baseline for the comparison. The following explains the design methods compared against each other in the experiments:

- Networks represented by Random 1 and Random 2 in the results have different random ordering of instruments on their scan path.
- Networks represented by OC have their instruments sorted on the number of accesses.
- Networks represented by OG10 and OG25, have their instruments ordered as explained in Section 4.1.3.3 for the G10 and G25 schedules, respectively.

Table 4.5 presents the experimental results for the constructed Remote networks. The reported shift overhead ratio is used to compare these design methods. In the table, for each benchmark, and for each schedule, the lowest ratio is marked in bold face, signifying that the corresponding design method performs better than the other methods for that schedule. Whenever, for a schedule, more than one ratio is marked in boldface, all those boldfaced ratios have been the same before rounding.

From the results, it can be seen that in case of sequential schedule, the network design has no effect on the shift overhead. This observation is in line with the discussion presented in Section 4.1.2.3 on the effect of pipelining of instrument data on lowering shift overhead.

For the rest of schedules, generally the network optimized for each schedule results in the lowest shift overhead. The following can be mentioned as examples deviating from this general observation: OG25 performs worse than OG10 for d695, g1023, h953, and t512505 benchmarks for the G25 schedule.

For every schedule, and for all benchmarks, the overhead for the optimized network is lower than the overhead from the randomly placed instruments.

Table 4.5. Experimental results: shift overhead to instrument data ratio in Remote benchmark networks

Benchmark name	Instrument data	Design method	Ratio of shift overhead to instrument data			
			Sequential	Concurrent	G10	G25
a586710	838530522	Random 1	**0.000**	0.010	0.010	0.010
		Random 2	**0.000**	0.018	0.018	0.018
		OC	**0.000**	**0.000**	0.001	0.000
		OG10	**0.000**	**0.000**	**0.001**	0.000
		OG25	**0.000**	**0.000**	0.001	**0.000**
d281	1496291	Random 1	**0.003**	0.033	0.210	0.094
		Random 2	**0.003**	0.034	0.176	0.103
		OC	**0.003**	**0.000**	0.089	0.032
		OG10	**0.003**	0.017	**0.023**	0.035
		OG25	**0.003**	0.001	0.024	**0.010**

continues on next page

Table 4.5. Experimental results: shift overhead to instrument data ratio in Remote benchmark networks

Benchmark name	Instrument data	Design method	Ratio of shift overhead to instrument data			
			Sequential	Concurrent	G10	G25
d695	704057	Random 1	**0.070**	0.029	0.223	0.114
		Random 2	**0.070**	0.032	0.234	0.118
		OC	**0.070**	**0.003**	0.194	0.092
		OG10	**0.070**	0.018	**0.126**	**0.074**
		OG25	**0.070**	0.009	0.155	0.078
f2126	5330439	Random 1	**0.000**	0.001	0.011	0.007
		Random 2	**0.000**	0.000	0.011	0.006
		OC	**0.000**	**0.000**	0.010	0.005
		OG10	**0.000**	0.001	**0.002**	0.003
		OG25	**0.000**	0.001	0.002	**0.001**
g1023	736216	Random 1	**0.011**	0.057	0.098	0.077
		Random 2	**0.011**	0.047	0.095	0.068
		OC	**0.011**	**0.002**	0.060	0.028
		OG10	**0.011**	0.009	**0.042**	**0.024**
		OG25	**0.011**	0.003	0.054	0.026
h953	1197178	Random 1	**0.003**	0.005	0.038	0.018
		Random 2	**0.003**	0.006	0.038	0.021
		OC	**0.003**	**0.000**	0.030	0.014
		OG10	**0.003**	0.003	**0.012**	**0.009**
		OG25	**0.003**	0.002	0.018	0.009
p22810	7784963	Random 1	**0.017**	0.368	0.379	0.377
		Random 2	**0.017**	0.201	0.218	0.211
		OC	**0.017**	**0.001**	0.072	0.030
		OG10	**0.017**	0.003	0.068	0.029
		OG25	**0.017**	0.002	**0.068**	**0.029**
p34392	16403755	Random 1	**0.001**	0.034	0.067	0.040
		Random 2	**0.001**	0.043	0.075	0.052
		OC	**0.001**	**0.000**	0.049	0.013
		OG10	**0.001**	0.002	**0.029**	0.010
		OG25	**0.001**	0.001	0.029	**0.008**
p93791	30083283	Random 1	**0.023**	0.080	0.131	0.100
		Random 2	**0.023**	0.042	0.104	0.065
		OC	**0.023**	**0.001**	0.073	0.031
		OG10	**0.023**	0.010	**0.056**	0.028
		OG25	**0.023**	0.003	0.062	**0.027**
q12710	31801946	Random 1	**0.000**	0.000	0.002	0.002
		Random 2	**0.000**	0.000	0.003	0.002
		OC	**0.000**	**0.000**	0.002	0.002
		OG10	**0.000**	0.000	**0.000**	0.001
		OG25	**0.000**	0.000	0.000	**0.000**

continues on next page

Table 4.5. Experimental results: shift overhead to instrument data ratio in Remote benchmark networks

Benchmark name	Instrument data	Design method	Ratio of shift overhead to instrument data			
			Sequential	Concurrent	G10	G25
t512505	165400967	Random 1	**0.000**	0.002	0.007	0.003
		Random 2	**0.000**	0.002	0.007	0.004
		OC	**0.000**	**0.000**	0.004	0.001
		OG10	**0.000**	0.000	**0.003**	**0.001**
		OG25	**0.000**	0.000	0.003	0.001
u226	252929	Random 1	**0.007**	0.134	0.248	0.164
		Random 2	**0.007**	0.216	0.373	0.235
		OC	**0.007**	**0.000**	0.094	0.049
		OG10	**0.007**	0.113	**0.029**	0.129
		OG25	**0.007**	**0.000**	0.062	**0.028**

4.1.4.4. COMPARISON BETWEEN NETWORK TYPES

Comparing the results for SIB-based and Daisy-chained networks shows that except for the sequential schedule, the Daisy-chained networks result in lower hardware overhead. In many cases, the difference in OAT between these network types is not significant, but in case of u226 benchmark, the Daisy-chained networks result in noticeably lower ratio. The reason for the lower overhead in Daisy-chained networks for access schedules with (some degree of) concurrency can be explained as follows. In Daisy-chained networks, the shift overhead is mostly contributed to by the bypass flip-flops, which are used when their corresponding instrument is not being accessed. Therefore, when there is concurrency in the schedule, these bypass flip-flops will be used less, hence lower shift overhead. This is in contrast to SIB-based networks in which a SIB contributes to overhead when its corresponding instrument is being accessed.

Comparing the results for the Remote network type with the results for the SIB-based and Daisy-chained network types shows the following. For the sequential schedule (per benchmark), the Remote network type shows significantly lower shift overhead. The reason can be attributed to the pipelining of instrument data in the bypass flip-flops, which can be utilized greatly for the sequential schedule. For the concurrent schedule, again it is the Remote network type that shows better or similar results. For the G10 and G25 schedules, no conclusive remark can be made as to which network types performs better.

4.2. MULTIPLE ACCESS SCHEDULES

In Section 4.1, we proposed methods for designing 1687 networks optimized for one given access schedule. In this section, we address the problem of designing 1687 networks for multiple access schedules. We assume the following are given:

- a set of access schedules, denoted by S, in which for each schedule $s \in S$, a weight W_s is specified. The weight (W_s) is assigned by the designer as a relative metric for the importance of access time reduction for that schedule as compared with the other schedules, and
- a set I of instruments in which for each instrument $i \in I$ the length of its interface shift-register (L_i) and the number of accesses ($A_{i,s}$) at each schedule s are provided.

4.2.1. NETWORK DESIGN METHODS

In general, we consider two possibilities for addressing the problem of network design for multiple access schedules. The first possibility, is to design a single network such that its performance with respect to OAT is optimized by considering all given schedules. It might happen that optimizing for one access schedule counters optimizations done for another schedule. For example, an instrument might have a high number of accesses in one schedule (relative to the number of accesses for other instruments in that same schedule) and low number of accesses in another schedule (again, relative to the number of accesses for other instruments in that other schedule). Clearly, such cases will lead to a trade-off between the OATs for each schedule. To make such trade-off, we use the Huffman tree inspired network construction algorithm (Algorithm 4.3). However, instead of using the number of accesses for each instrument as the base for placement of instruments, we assign an attribute, *weighted number of accesses* ($A_{i,w}$), to each instrument. This weighted number of accesses ($A_{i,w}$) captures both the number of accesses for an instrument in each access schedule ($A_{i,s}$) and the relative weight of the access schedules (W_s), and is calculated as:

$$A_{i,w} = \sum_{s \in S} (A_{i,s} \times W_s) \tag{4.5}$$

The idea is to design a network which performs reasonably well for all the given access schedules, by considering the relative weight assigned to each schedule.

The second possibility is that a dedicated network is designed and optimized for each access schedule. Each network is then connected to the TAP

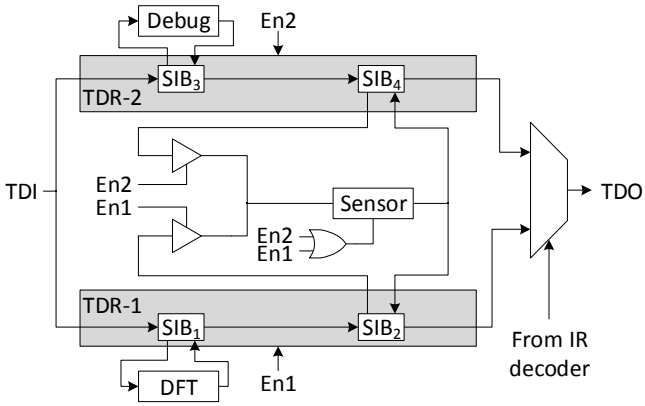


Figure 4.8. The sensor instrument is shared by two networks (TDRs)

through a dedicated TDR. The instruments whose interface shift-register is to be accessed through multiple schedules (i.e., multiple TDRs) can be shared among the corresponding networks by using, for example, a scheme similar to the one shown in Figure 4.8. In the presented scheme, tristate buffers are used to control to which network the shared instrument shift-register is connected. The enable signals in this scheme (i.e., En1 and En2) are applied from the TAP circuitry. That is, given that no two such TDRs are active at the same time, the same enable signals that are applied to the TDRs are used to connect the shared instrument shift-registers to the scan path which belongs to the active TDR. The two networks in Figure 4.8 are designed for two schedules where the Sensor instrument is used in both schedules, while the DFT and the Debug instruments are each accessed only in one of the schedules (hence each accessible only through one of the TDRs). Although the Sensor instrument is shared by both networks, a SIB is dedicated to it in each network.

Based on the above-mentioned possibilities, we consider the following alternatives for experimenting with optimizing networks for multiple access schedules:

- N: A non-reconfigurable network (see Section 2.2)
- F_{sib}: A flat SIB-based network consisting of all instruments used in all given access schedules
- F_{dc}: A flat Daisy-chained network consisting of all instruments used in all given access schedules
- H_{sib}: A hierarchical SIB-based network constructed by using Algorithm 4.3, where $A_{i,w}$ is used as input (see Eq. (4.5))

- H_{dc}: The Daisy-chained counterpart of H_{sib}
- M_{sib}: Multiple SIB-based networks each optimized for a given schedule
- M_{dc}: The Daisy-chained counterpart of M_{sib}
- R: A Remote network consisting of all instruments used in all given access schedules, with no optimization.

4.2.2. EXPERIMENTS

In this section, we present the experiments we performed to compare the considered design methods for multiple access schedules (listed in Section 4.2.1). The comparison is with respect to OAT and hardware overhead.

To perform the experiments, a set of instruments and access schedules were needed. We considered a total of 100 instruments each having a shift-register of length 20 flip-flops, as well as eight different access schedules. Table 4.6 lists the considered set of instruments and access schedules. In Table 4.6, column 1 lists that there are five types of instruments, column 2 lists how many of each type of instrument are considered, and columns 3–10 list the number of accesses for each instrument type for each access schedule. In Table 4.6, under the headers for columns 3–10, the access schedules as well as the weights assigned to them (within parentheses), are presented. The instruments in the benchmarks are listed randomly so that instruments with similar number of accesses do not appear beside each other on the scan path. This is particularly relevant to the R (i.e., the Remote) network.

For the experiments, we calculated OAT for each of the access schedules listed in Table 4.6, by the use of the algorithms proposed in Chapter 3. For the non-reconfigurable network, OAT is calculated using Eq. (2.1), where we have considered (for $N = 100$ instruments each having $L = 20$):

$$l = \sum_{i=1}^{N} L_i = \sum_{i=1}^{100} 20 = 2000 \tag{4.6}$$

and assumed $T_a = 4$, which is reported as TAP overhead. In Eq. (2.1), when accessing instruments according to the concurrent schedule, we have:

$$p = \max_{1 \le i \le N} \{A_i\} \tag{4.7}$$

where N is the number of instruments and A_i is the number of accesses for instrument i. In Eq. (2.1), when accessing instruments according to the sequential schedule, we have:

$$p = \sum_{i=1}^{N} A_i \tag{4.8}$$

Table 4.6. Benchmarks used for the experiments with multiple schedules

Instruments		Access schedules and their assigned weights (in parentheses)							
Type	Count	S1 (1) Seq.	S2 (100) Conc.	S3 (1) Conc.	S4 (1) Seq.	S5 (100) Conc.	S6 (1) Seq.	S7 (10) Conc.	S8 (10) Conc.
1	20	100	10	10	10	10	10	10	10000
2	20	10000	0	0	0	0	0	0	10000
3	10	100	10	10	10	10	10	10	10000
4	40	100000	10	10	100000	100	10	10	10000
5	10	10	100	10	10	10	100000	100000	10000

Conc. denotes the concurrent schedule, and **Seq.** denotes the sequential schedule.

Table 4.7. OAT calculation results when the networks are optimized for S1–S8

Design method	Scaled† weighted OAT ($OAT_s \times W_s$)								Sum
	S1	S2	S3	S4	S5	S6	S7	S8	
N	8,423,014	20,240	22	8,016,804	20,240	2,005,405	2,004,020	200,420	20,690,166
F_{sib}	521,197	4,621	19	496,060	10,021	124,097	304,158	210,422	1,670,594
F_{dc}	521,207	4,463	18	496,070	9,593	124,107	295,151	200,521	1,651,132
H_{sib}	155,788	3,947	19	147,619	9,852	33,031	224,171	215,724	790,152
H_{dc}	155,790	3,786	19	147,621	9,423	33,033	215,164	205,825	770,659
M_{sib}	147,054	3,798	19	138,621	9,468	30,834	215,165	210,422	755,381
M_{dc}	147,056	3,632	18	138,624	9,035	30,836	206,157	200,521	735,879
R	100,900	4,387	18	96,030	9,535	24,037	286,153	200,421	721,481

†Scaled by 1000.

where N is the number of instruments and A_i is the number of accesses for instrument i.

4.2.2.1. COMPARISON WITH RESPECT TO OAT

In this section, we report and compare the design methods presented in Section 4.2.1 w.r.t OAT. The networks (where applicable) are designed by taking into account all schedules and their assigned weights. As the baseline for comparison, we use the non-reconfigurable scan network (denoted by N).

Table 4.7 presents the results of the experiment in details. In the table, the first column lists the examined design methods. Columns 2–9 list for each access schedule the product of the OAT and the assigned weight for that schedule. Column "Sum" presents sum of the values in columns 2–9, to be used as the comparison metric.

From the "Sum" column of Table 4.7 it can be seen that for N, F_{sib}, and F_{dc}, the sum is at least two times larger than the sum reported for the rest of the networks. The R network shows the best sum among all, which makes it specifically interesting given that it has a fixed architecture (which does not change with the applied schedule). M_{sib} and M_{dc} also show a low sum as well as good results for the individual schedules, as they consist of a set of

Table 4.8. Hardware overhead

Design method	Cell density	Setup slack (ns)	Hold slack (ns)	Area (μm^2)	Area difference with N	Area increase	Equivalent gate count
N	0.93	2.469	0.228	84208	0	0.0%	0
F_{sib}	0.92	2.455	0.221	89513	5305	6.3%	2551
F_{dc}	0.92	2.421	0.247	89782	5575	6.6%	2680
H_{sib}	0.92	2.136	0.246	92691	8484	10.1%	4079
H_{dc}	0.93	1.863	0.224	96174	11966	14.2%	5753
M_{sib}	0.92	0.819	0.223	138750	54542	64.8%	26222
M_{dc}	0.92	0.282	0.22	148437	64229	76.3%	30879
R	0.92	2.654	0.223	89688	5480	6.5%	2635

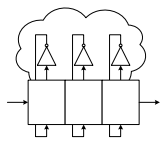


Figure 4.9. An example inverter array used as an instrument for validation of the network design implementation

networks each optimized for one of the access schedules. H_{sib} and H_{dc} also show good results.

4.2.2.2. COMPARISON WITH RESPECT TO HARDWARE OVERHEAD

To give an idea of the hardware overhead associated with each of the considered network design methods, we implemented and synthesized a number of designs using 65 nm technology, and performed place & route. Each design included a 1687 network based on one of the considered design methods, the 1149.1 circuitry, and the instrument shift-registers. We considered each instrument to be simply an array of inverters where each inverter was connected to an I/O pair in the corresponding instrument shift-register (Figure 4.9).

To have a fair comparison regarding the required area, we instructed the place & route tool to fit each of the designs in a square-shaped layout with standard cell density of 0.85. The area of the generated layout is then used as the basis for hardware overhead comparison. The achieved standard cell density is also reported to observe how close the achieved densities are. Ideally, the achieved densities should be the same to ensure a fair comparison. Since it is desired to calculate the part of overhead which is associated only with the network components, we should remove the part of area associated with the TAP circuitry (for one TDR) and instrument shift-registers, as this part of overhead is the same for all networks. To calculate the 1687 network overhead, we subtracted the area of the layout generated for the N network (which uses

no network reconfiguration components) from area of layouts generated for each of the other networks. Appendix B presents schematic and details our implementation of the SIB component (Figure B.3).

To validate correctness of the implemented designs, we generated a number of patterns in form of scan vectors to be applied through the TAP terminals during post-layout simulation. Each pattern consisted of random stimuli for instruments (i.e., inverter arrays) interleaved with configuration bits for the ScanMux control bits, and the expected responses. Through these post-layout simulations we established that the designs work as expected at the chosen frequency of 100 MHz.

Table 4.8 presents the data obtained from the hardware implementation experiment. The second column shows the cell density after the place & route. Columns three and four present the setup slack and hold slack in nanoseconds, respectively. Column five reports the area in square micrometers. To see how much area is consumed only by the 1687 network(s), we subtracted the area for the N network (i.e., the non-reconfigurable network) from the area for each of the designs, and reported the resulting area difference in column six. For each design, the area percentage increase due to the 1687 network components is reported in column seven. Lastly, to give an idea of the equivalent overhead in other technologies, column eight reports the area difference with N in number of two-input NAND gates.

The following should be noted from the results presented in Table 4.8:

- The setup times are positive indicating that the networks can be clocked at 100 MHz. Moreover, except for M_{sib} and M_{dc}, the rest of the designs can be clocked at even higher frequencies as indicated by larger setup time margins.
- The hold time slacks are positive for all of the designed layouts, which indicates that hold violations are fixed by the tool. Fixing these violations is done by adding buffers, which slightly increase the hardware overhead. Although such increase is not necessarily the same for all the layouts, the comparison is still fair as such overhead can also be seen as complexities associated with certain design methods.
- The hardware overhead for M_{sib} and M_{dc} is relatively high. On the other hand, H_{sib}, H_{dc}, and R show relatively lower overhead.

4.3. THE *UNKNOWN* SCHEDULES

In choosing a network design method, a determining factor is the amount of overhead introduced into OAT by the designed network. The amount of overhead is, however, available only for access schedules known at design time. It

might happen that a network that has low time overhead for some known access schedules, results in high overhead in the context of a new schedule. It is therefore interesting to know how largely the time overhead might vary when a network is used according to new (i.e., previously *unknown*) access schedules. However, judging which network is better according to the amount of variation in time overhead, might be misleading. For example, if variation in time overhead has been due to change in number of accesses, there is a chance that the overhead percentage has not varied much, due to similar change in other OAT components. Therefore, we propose instead to observe the variability in the overhead percentage (Eq. (3.15)) as an indicator for predictability of the network when new schedules are applied.

Based on the above, in order to evaluate how largely the overhead percentage might vary when new access schedules are applied to a network, we decided to perform the following experiment.

We considered the same benchmark sets reported in Table 4.6, and assumed that at design time we only know one access schedule s_k ($1 \leq k \leq 8$) for which we design networks using each of the design methods specified in Section 4.2.1 (except for M_{sib} and M_{dc} that are not applicable for a single schedule). Next, for each of the networks, we computed the overhead percentage for each of the access schedules s_i ($1 \leq i \leq 8, i \neq k$) by using Eq. (3.15). To measure the variability in overhead percentage, we calculated the standard deviation of the computed percentages. The smaller the standard deviation, the more stable the overhead percentage for new access schedules.

Table 4.9 presents the results of this experiment. Where applicable, the numbers in boldface are the largest standard deviation found for a given design method, and are used for the purpose of comparison. The R network shows the least value, i.e., 0.10. In this case, no optimization was done (due to that the given schedule was sequential, see Section 4.1.2.3) and therefore the placement of instruments on the scan path was the random order provided by the benchmark set. Consequently, the benefits of pipelining of instrument data is only partially exploited (see Section 3.4.1). Based on this observation, it can be expected that even for future schedules, an overhead percentage similar to the overhead percentages for currently known schedules is observed. In this regard, the average overhead percentages for the R and H_{sib} networks that have shown the smallest variabilities are 14 percent and 26 percent, respectively.

To evaluate this proposed predictability metric, we present another experiment, in which we apply new schedules to these two networks and study the changes in the overhead percentage. In this experiment, the instruments originally described in the schedule S1 are accessed according to partially concurrent schedules in groups of 5, 10, 20, 50, and 100 concurrently accessed instruments.

Table 4.9. Variability of overhead percentage in different design methods

Design method	Schedule optimized for	Standard deviation
F_{sib}	†	0.36
F_{dc}	†	0.38
H_{sib}	S1	**0.22**
	S2	0.19
	S3	0.17
	S4	0.19
	S5	0.19
	S6	0.21
	S7	0.21
	S8	0.17
H_{dc}	S1	**0.23**
	S2	0.21
	S3	0.19
	S4	0.21
	S5	0.21
	S6	0.23
	S7	0.23
	S8	0.19
R	S1‡	**0.10**
	S2	0.10
	S3	0.09
	S4‡	0.09
	S5	0.10
	S6‡	0.09
	S7	0.07
	S8	0.09

†The F_{sib} and F_{dc} networks have fixed architectures.

‡No optimization done for the sequential schedule (Section 4.1.2.3).

The result of overhead computation for this experiment is presented in the chart in Figure 4.10. From the results, it can be seen that the overhead percentage for the R network varies between 6 to 17 percent, and for the H_{sib} network between 7 and 43 percent. These results show that the overhead percentage varies less for the R network (in comparison to the H_{sib} network) for these new schedules. This observation is in line with the expectation of less variability in overhead percentage from the R network.

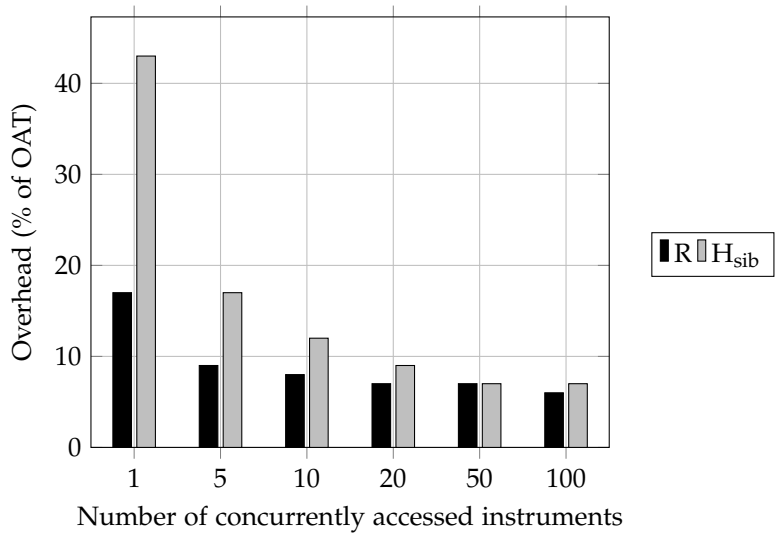


Figure 4.10. Change in overhead percentage as concurrency increases

4.4. CHAPTER CONCLUSIONS

In this chapter, we presented design methods for constructing 1687 networks optimized with respect to OAT. We considered two optimization problems: when the objective is to optimize only for one given access schedule, and when the objective is to optimize for multiple access schedules. Moreover, we considered the case that instruments in a network might be accessed according to a schedule not known at design time. As a predictor of how much overhead percentage might vary for a new schedule, we used the standard deviation of the overhead percentages for already known access schedules. The smaller the standard deviation, the more stable the overhead percentage.

We presented experimental results for each of the considered optimization problems, as well as to evaluate the stability of the overhead percentage for new access schedules. The experimental results showed that the Remote network type performs reasonably well when considering OAT, hardware overhead, and stability of overhead percentage. It must be emphasized that the good performance of the Remote networks is achieved under the assumption that the instrument data can be pipelined in the bypass flip-flops, which (at least) requires support from the retargeting tools. To avoid such dependency on pipelining, the hierarchical SIB-based and Daisy-chained networks can be used, considering that they showed relatively low hardware overhead, as well as low OAT in the experimental results.

Part III

Operation

In Part II, we presented methods for designing 1687 networks such that they were optimized with respect to OAT, when access schedules were given. In this part, we present methods for reducing OAT when the network is given, by focusing on the retargeting process. More specifically:

- In Chapter 5, OAT reduction in the complete retargeting flow (from PDL to bit vectors) will be regarded as a scheduling problem, for which we discuss opportunities for optimization.
- In Chapter 6, we focus on a key step in the retargeting flow, the *retargeting step* (Section 2.3.2.2), which is the generation of scan vectors to carry out a given iApply group. For a retargeting step, we present a method for reducing the solution space without removing the optimal (w.r.t. application time) set of vectors.

5

The Retargeting Flow

So far in this thesis, the assumption has been that the access schedule is given as an abstract model of how many times and in what combinations the instruments are to be accessed. As our focus was on the access time minimization, we did not model the wait cycles in our access schedules. In this chapter, however, we focus on how the complete schedule (including instrument accesses, wait cycles, and concurrency) can be inferred from the given PDL code during the retargeting process, and will point out possibilities for reduction in schedule application time. In this chapter, we use the term schedule application time (as contrasted to OAT), since the complete schedule contains wait cycles, in addition to the (read/write) accesses to instruments.

We start the discussion in this chapter by presenting a retargeting flow (Section 5.1) that performs the basic tasks such as dealing with procedure calls and merge blocks (see Section 2.3.2.2). In Section 5.2, we discuss the shortcomings of the related work [49, 50] on scheduling for 1687 networks. Moreover, we point out opportunities in the basic flow that can be exploited for optimized retargeting with respect to application time. Prior work on retargeting [13, 39, 40, 51, 52, 53, 54] has not considered the PDL-level scheduling with the aim of reducing the application time. We discuss these works in Chapter 6.

5.1. BASIC RETARGETING FLOW

In this section, we begin by explaining how concurrency can be captured in a PDL script (Section 5.1.1), and continue by presenting the three tasks that should be done in a basic retargeting flow, namely, flattening (Section 5.1.2), merging (Section 5.1.3), and translation (Section 5.1.4).

5.1.1. PDL AND CONCURRENCY

As was mentioned in Section 2.3.2.2, in a given PDL, instrument accesses are specified via iRead, iWrite, and iScan commands, which are setup commands. The setup commands take effect at the first subsequent iApply command, and together are referred to as an iApply group. From the iApply groups in a given PDL, it can be inferred how many times each instrument is accessed. The accesses are not necessarily sequential as the PDL developers can add concurrency to the schedule by

- placing multiple setup commands (for different instruments) under the same iApply group, and
- placing multiple procedure calls inside a merge block (to instruct the retargeting tool to execute them in parallel)[1]. IEEE 1687 has mechanisms for further control over concurrency inside a merge block, such as describing a resource as exclusively accessible.

In the retargeting process, through which PDL is translated into low-level TAP operations or bit-vectors, the schedule is inferred. It should be noted that the resulting schedule is not unique and depends on the optimizations performed by the retargeting tools. What matters is that the sequence of carried out action commands (such as iApply) should be as specified in the PDL script.

Figure 5.1 shows a basic PDL retargeting flow by illustrating a retargeting scenario for two PDL procedures called from within a merge block:

```
iProc ParentProc {
    iMerge -begin
        iCall Proc1();
        iCall Proc2();
    iMerge -end
}
```

In Figure 5.1, the frames labeled as Proc1() and Proc2() represent procedure bodies, the frame labeled ParentProc() is a higher (e.g., chip-level) procedure in which Proc1() and Proc2() are called from within a merge block, the boxes labeled as s represent setup commands such as iRead and iWrite, and the dashed boxes represent *critical sections* that cannot be run together due to resource conflicts. In PDL, critical sections begin by an iTake command specifying a resource to be taken exclusively, and end by an iRelease command

[1]The standard does not put any requirement on the tools to perform merging. That is, the tool might simply execute all procedures sequentially.

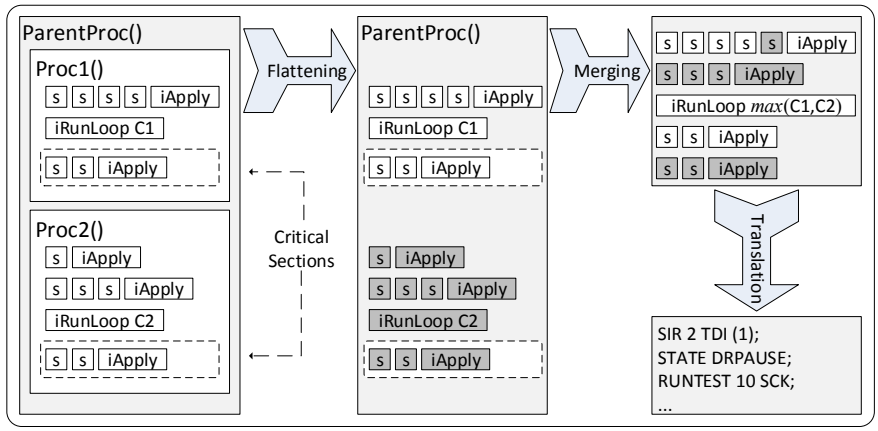


Figure 5.1. Example showing the basic PDL retargeting flow

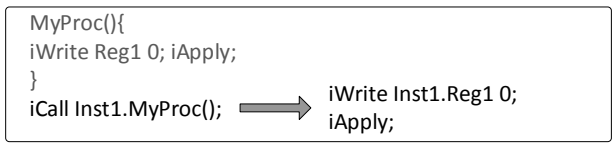


Figure 5.2. An example, showing the basic idea of flattening

specifying that the taken resource is no more exclusively required. In the following, and with the help of Figure 5.1, each step in this basic flow will be elaborated on.

5.1.2. FLATTENING THE PDLS

Flattening is replacing a call to a procedure with the body of that procedure, and adding the name of the instrument on which the procedure is called, to the names of terminals/registers inside the called procedure by using a dot notation. Flattening is similar to the inline expansion in languages such as C/C++, in which the call to a function is replaced with the body of the called function. Figure 5.2 shows the basic idea of flattening, where the call iCall Inst1.MyProc() is replaced by the body of MyProc, and the name of the register Reg1 is changed to Inst1.Reg1. Flattening is illustrated in Figure 5.1 by changing the color of commands inside Proc2, and removing the frames presenting the procedure bodies.

5.1.3. MERGING THE RESULTING FLATTENED PDL

Merging is the selection of setup commands inside different procedures that are called from within merge blocks, and grouping them under the same iApply group. Figure 5.1 shows a possible merging of the Proc1() and Proc2(). In this case, merging also involves replacing multiple iRunLoop commands with one iRunLoop whose parameter is the maximum value found among the parameters for each of those iRunLoops. As can be seen in Figure 5.1, the iApply commands inside the critical sections are not merged.

5.1.4. TRANSLATION OF THE MERGED PDL INTO CHIP-LEVEL VECTORS

In the translation stage, a number of retargeting steps are performed (see Section 2.3.2.2). That is, the iRead/iWrite commands inside iApply groups are translated into chip-level scan vectors to be applied from the TAP. Each vector consists of bits to be shifted in serially (taken from iWrite commands) and expected responses (taken from iRead commands) to be compared against the bits shifted out. Alternatively, the result of translation can be other description languages, such as Serial Vector Format (SVF) used for describing TAP operations [55], as shown in Figure 5.1.

5.2. ENHANCING THE BASIC FLOW

The merging step in the basic retargeting flow discussed above can be seen as test scheduling with resource constraints. Power constraints can also be taken into account during the PDL merging. In its current state, however, PDL does not support specification of power consumption (or in general any such parameters). In case such constraints are needed to be taken into account during retargeting (e.g., due to power budget limitation when running BISTs), one can annotate the PDL code with specially formatted comments to be extracted and considered by the retargeting tools.

The problem of finding the schedule with the shortest application time, during PDL retargeting can be seen as the SoC test scheduling with precedence relationship and resource and power constraints, which is formulated in [56] and is shown to be NP-complete. Prior work has considered test scheduling with resource and power constraints for IEEE 1687 [49, 50]. However, none of the heuristics suggested in [49, 56, 50] are applicable in a PDL retargeting scenario involving both scan vectors and wait cycles, since their proposed heuristics require the length of all tests to be known (in the same unit, i.e., either in time [56] or in number of accesses [49, 50]) prior to scheduling. This prior knowledge is not available in PDL retargeting, since the length of a wait cycle is specified in number of clock cycles, whereas the actual length of

an iApply group (in clock cycles) is not known before the translation is performed [49]. Therefore, on the one hand, for ensuring that the resource and power constraints are met during the PDL merging, the start and finishing time of all tests/accesses should be known, while on the other hand, during merging, the length (in clock cycles) of iApply groups are not known, and therefore, cannot be safely merged with those PDL codes that have iRunLoop commands.

In the basic flow, this problem can be circumvented, e.g., by separating iRunLoop and iApply commands into (a number of) separate blocks and performing the scheduling on each block separately. This way, however, not all the potential concurrency is exploited, as demonstrated by the motivational example presented in Section 5.2.2. Considering the above, in the following, the considerations for running the iRunLoop and iApply commands concurrently are detailed.

5.2.1. RUNNING IAPPLY AND IRUNLOOP COMMANDS CONCURRENTLY

In the simple merging scenario described in Section 5.1, the two iRunLoop commands (i.e., `iRunLoop C1` and `iRunLoop C2`) were merged into one command with the largest number of clock cycles found among them. This way, shifting out the result of BIST from one instrument should unnecessarily wait until the other instrument completes its test. These wasted cycles could have been used to apply other tests, to increase concurrency.

To perform iApply commands concurrently with iRunLoop commands, there are two requirements: (1) being able to de-select an instrument while it is still running, and (2) knowing how many clock cycles it takes to perform a (flattened and merged) iApply group. In the following we will explain the reasons for each of these requirements.

Considering the first requirement as a network design guideline, the 1687 network should be designed such that it is possible to keep an instrument active but de-selected (e.g., by connecting each instrument to the network through a SIB). The reason for this is that, for example, when a BIST instrument whose control inputs are directly on the scan path is running, its operation might be interfered with if scan vectors are shifted in and applied to other instruments over the same scan path. However, if the BIST instrument is kept active but off the scan path, the clocks that are applied to perform other tests, can be applied to this BIST instrument, as well.

As for the second requirement, one way to obtain the information regarding the actual number of clock cycles is to integrate the merging and translation steps in the PDL retargeting flow, such that for every command that is being merged, the translation is performed to provide the number of required clocks. Knowing the number of spent clock cycles, the merging algorithm can

Table 5.1. Scheduling results for the u226 benchmark

Power budget	The resulting schedule length in TCKs		Reduction (%) in application time
	The basic flow	The modified flow	
∞	490709	354313	27.7
9	493673	357277	27.6
3	720662	577829	19.8

decide if enough clock cycles have passed for the previously issued iRunLoop command.

5.2.2. THE U226 BENCHMARK EXAMPLE

In this section, with the help of an example, we demonstrate the benefits of running the BISTs concurrently with the iApply groups.

Among the ITC'02 benchmark SoCs introduced in Section 4.1.4, u226 and d281 have BISTs (see Appendix D for details). The u226 SoC is particularly suitable for this example as it has relatively long BIST runs. For this example, we assumed that the network is SIB-based and has a flat architecture (see Figure 3.1(a)). Moreover, we assumed that the BIST is running on a system clock 10 times faster than TCK. Additionally, we assumed that each of the 34 instruments has a power consumption of one unit, and considered the given power budget to be infinity (ideally allowing for fully concurrent schedule), nine units (ideally allowing for about 25 percent concurrency) and three units (ideally allowing for about 10 percent concurrency). For each of the given power budgets, we considered two cases: (1) when the wait cycles cannot be merged with scan vector applications (referred to as the basic flow), and (2) when the wait cycles can be merged with scan vector applications (referred to as the modified flow). In the latter case, BISTs are started as soon as resources are available, without interrupting the operation of other BISTs.

We assumed that the accesses for each instrument is specified in a procedure, and that all procedures are called within a merge block. To perform the scheduling, we developed a simple algorithm that selected the procedures (corresponding to the instruments) from a given ordered list. As the order that the scheduler selects the procedures affects the results, we used genetic algorithms [57] to guide the scheduler. We skip the details on the scheduler in this thesis.

Table 5.1 presents the result of scheduling for this example. For the basic flow, the scheduler was not allowed to run BISTs at the same time as performing the accesses to instruments. In contrast, for the modified flow, which used integrated merging and translation, the scheduler could freely choose access

procedures to run concurrently with BISTs, as long as the power constraint was not violated. From the table, it can be seen that the modified flow has resulted in up to 28 percent reduction in the schedule application time.

For this relatively small example, the run-time for the scheduler guided by genetic algorithm was up to eight hours for the cases reported in Table 5.1. The reason for the long run-time is that the translation step is performed for every access (i.e., iApply group) every time the scheduler runs and the scheduler is called many times by the genetic algorithm. Clearly, for real life problems, this method of optimization is not practical and there is, therefore, a need for more efficient approaches.

5.3. CHAPTER CONCLUSIONS

In this chapter, we detailed a basic PDL retargeting flow consisting of a number of steps, namely flattening, merging, and translation, and showed that the merging step can be seen as power- and resource-aware scheduling, which is a known NP-complete problem. We noted that lengths of some PDL commands are specified in time units and are thus known during merging, whereas length (in time units) of some other commands will not be known until translation step is completed. Therefore, the scheduling approaches in prior work are not applicable to this problem. With the help of an example, we showed the benefits of a modified retargeting flow in which the merging and translation steps are integrated. The modified flow managed to reduce the schedule application time (in TCKs) by up to 28 percent, via increasing concurrency in the schedule while satisfying constraints. The long run-time for the modified flow reveals the need for more efficient approaches for the problem of optimized PDL retargeting.

In Chapter 6, we present a method for performing the retargeting step optimally. It is important to note that optimal retargeting steps (w.r.t. vector application times for each step) do not necessarily result in the optimum schedule application time for the completely retargeted PDL. The reason is that each retargeting step is affected by the condition that the previous retargeting step has left the network in. In explanation, we can draw parallels to a greedy method that chooses what seems best at the moment without considering the big picture. Therefore, to perform retargeting such that the schedule application time is minimal, the scheduler should also guide the retargeting step with enforcing constraints on particular control bits in the network. Needless to say, this further complicates the retargeting process.

6

Optimal Retargeting Step

As was discussed in Chapter 5, one of the main tasks in retargeting is the translation of given PDL scripts into bit vectors (Section 5.1.4). The translation involves a number of retargeting steps, each translating an iApply group into bit vectors. As we discussed in Chapter 5, the retargeting step might be performed as part of the solution space exploration as well. Therefore, it is important to increase the run-time efficiency of the retargeting step. Moreover, it is important to generate the vectors such that the application time is minimized (effectiveness).

There have been a number of works addressing retargeting for 1687 networks [13, 39, 40, 51, 52, 53, 54]. The work in [13] was an early work to motivate the benefits of describing instrument operations in a high-level language and having it automatically translated to bit vectors. The work in [52] presents a study on the use of retargeting tools for the specific case of 3D stacked ICs. The works in [51, 53, 54] present retargeting for Level-1 PDL. Among the aforementioned works, the only works that have so far addressed minimization of application time for the generated scan vectors are [39, 40], which are discussed in Section 6.1.

In this chapter, we improve upon the prior work [39, 40] by presenting a method for reducing the solution space in the process of searching for the optimal vector in a retargeting step. In this chapter, we use the term optimal (solution) for a set of scan vectors that result in the shortest possible application time in terms of clock cycles. Briefly, the proposed method analyzes the given network, and computes the largest number of CSU operations required to take the network from any configuration to any configuration. In this thesis, we refer to the computed number as *upper-bound*. The upper-bound can then be used by any of the approaches presented in [39, 40] for the scan vector generation, to guarantee the optimality of the results. It should be noted that

in computation of the upper-bound, the assumption is that the instruments that are requested to be accessed in the given retargeting step (i.e., iApply group) are not mutually inaccessible (e.g., are not on different inputs to a multiplexer). If that is the case, the retargeting tool should issue a warning to the user and break that iApply group into smaller iApply groups.

The rest of this chapter is organized as follows. In Section 6.1, we review the related work and discuss how the proposed method can improve both efficiency and effectiveness of the retargeting step as presented by those works. In Section 6.2, we use an example to, among other things, show that minimum number of CSUs does not necessarily result in the minimum application time in clock cycles. Section 6.3 presents the core technique in computation of the upper-bound. As the core technique is not directly applicable to large networks, in Section 6.4 we present a number of reduction techniques that break the network into a number of smaller segments so that the core technique can be applied to each segment separately. To compute the upper-bound for the original network, the upper-bounds computed by the core technique for smaller segments are then combined together in the appropriate way. Section 6.5 presents some experimental results for a number of benchmarks.

6.1. PRIOR WORK

Verification and pattern generation (retargeting) for reconfigurable scan networks were presented in [39]. The work in [39] models general reconfigurable scan networks using a structural SAT[1] model in which each control bit in the network is represented by a Boolean variable. The model can therefore capture any arbitrary configuration of the network. In a typical retargeting step, several configuration cycles should be performed to take the network from an initial configuration to a target configuration (in which the shift-registers of the required instruments become part of the active TDI to TDO scan path). Therefore, to capture all the configuration cycles, the SAT model is unrolled over a number of time frames. Each of the time frames corresponds to a CSU, which is considered an atomic operation in [39]. That is, each individual clock cycle spent on shifting input data (or performing capture and update operations) is not considered to be a separate configuration step, rather the whole cycle of capturing, shifting, and updating is seen as one step. The state of each bit in each time frame is then used to form a scan vector that should be shifted in and applied (by going through the update phase) for the transition from a frame to the next one. A sequence of such scan vectors is what a retargeting tool computes for taking the circuit from its current configuration to a target configuration.

[1] Boolean Satisfiability Problem

Using the above-mentioned scheme requires the algorithm to receive as input the number of times it should unroll the model (i.e., the number of allowed CSUs). The choice of the number of CSUs has a crucial impact on the resulting solution (i.e., the generated scan vectors). If the allowed number of CSUs is too small, the target configuration might be unreachable from the current configuration (i.e., no feasible solution). Moreover, given that some solutions might be better than the others w.r.t. clock cycles, a too small value for the number of CSUs might exclude those better solutions from the solution space. Therefore, finding the upper-bound on the number of CSUs is essential for effective retargeting (i.e., generating scan vectors which are optimal w.r.t. access time). On the other hand, if the number of allowed CSUs is too large, the generated model becomes unnecessarily large resulting in decreased runtime efficiency, yet with no guarantee on optimality.

The work in [39] does not present an upper-bound derivation method for the number of required time frames and assumes that the user specifies a maximum allowable number of frames. Moreover, the generated scan vectors are not optimal regarding instrument access time. To address these issues, [40] presents an upper-bound for the number of time frames. The calculation of upper-bound on the number of frames, as presented in [40] can be explained as follows. The total access time is formulated as:

$$t = 2n + \sum_{i=1}^{n} L_i \tag{6.1}$$

where n is the number of frames, 2 represents the TAP overhead for each frame, and L_i represents the length of the scan path for frame i. The upper-bound for n, denoted by n_{bound}, is presented as:

$$n_{\text{bound}} < \lceil \texttt{Cycles}_n / 2 \rceil \tag{6.2}$$

where $\texttt{Cycles}_n$ is the minimum access time achievable with n frames. According to [40], finding the global minimum is an iterative process in which after finding an initial solution, the bound is calculated and iteratively lowered as we find solutions with smaller access times (i.e., smaller than $\texttt{Cycles}_n$ which was originally found).

Given that in real-life circuits, the access time might be in the order of thousands of clock cycles, the bound calculated using Eq. (6.2) will not be helpful in practice. The reason is that, as discussed in [40], finding the optimal solution is NP-hard, hence requiring heavy computations to search the solution space, which is limited by the upper-bound on the number of frames. If this upper-bound is very high (that is, hundreds or even thousands of frames), the time that it takes to find the optimal solution will be extremely long.

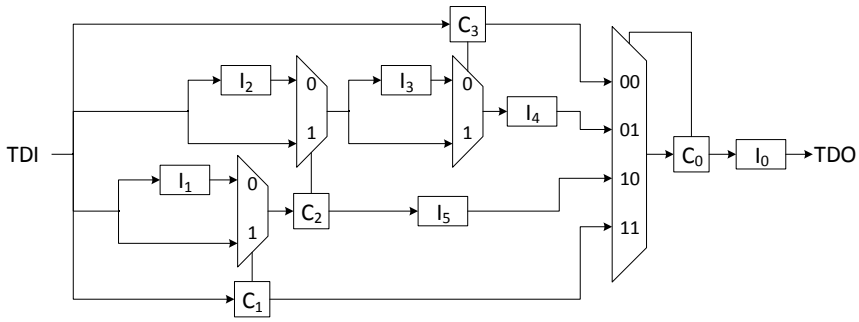


Figure 6.1. A 1687 network used in the discussion in Section 6.2

Therefore, the authors of [40] propose a heuristic for retargeting, which initially searches for the minimum number of CSUs required to get a solution, and from that point continues the search for a better solution by allowing a limited number of extra CSUs. There are two drawbacks with the heuristic proposed in [40], both negatively impacting the run-time efficiency. Firstly, searching for the minimum number of required frames (i.e., CSUs) involves multiple calls to the SAT solver, each with an incremented number of allowed CSUs. Secondly, allowing extra CSUs after an initial solution found (hoping to reach a local minimum) might be unnecessary if the solution already found is the globally minimum solution.

In this thesis, we detail an upper-bound computation method which is applicable to arbitrarily designed 1687 networks, and results in a bound low enough for real-life retargeting applications. By using the proposed upper-bound, the model can be initially unrolled as many times as the upper-bound, for which the SAT solver is called only once (in contrast to the heuristic method described above). Therefore, the run-time efficiency of the retargeting tool increases while guaranteeing optimality of the generated vectors.

6.2. MOTIVATIONAL EXAMPLE

In this section, with the help of an example, we show that a solution with minimum number of CSUs is not necessarily the optimal solution w.r.t. the number of clock cycles. We also show that the bound calculated by using Eq. (6.2) can be large even for a very small example network. Moreover, by varying the length of instrument shift-registers, we show that the computed upper-bound is a function of length of instrument shift-registers as well.

Figure 6.1 shows a network of six instruments. Lengths of instrument shift-registers in this network are shown in Table 6.1 for three instances A, B, and C. The difference between instances is only in the length of shift-registers, as

Table 6.1. Shift-registers' length for the instruments in Figure 6.1

	Length of instrument shift-registers					
	I_0	I_1	I_2	I_3	I_4	I_5
Instance A	20	50	100	20	20	5
Instance B	20	50	**70**	20	20	5
Instance C	**50**	50	100	20	20	5

Numbers in boldface denote where the instances differ.

marked in boldface in the table. Assume that initially all control bits are set to zero, and that we aim to access instrument I_4. Accessing I_4 can be done by setting C_0 to "01". This, however, will not necessarily lead to minimum access time for I_4 since instruments I_2 and I_3 are then on the scan path to I_4. Therefore, it might be better to first switch I_2 and I_3 off the scan path before setting C_0 to "01". The reason for saying "might be" is that in this example, I_0 is always on the scan path and for each access to the network, dummy bits should be shifted through it. If length of I_0 is comparable to the length of the shift-registers for I_2 and I_3, its contribution to overhead cancels out the benefit from switching I_2 and I_3 off the scan path. To see how the length of shift-registers affect the search for the optimal way to access I_4, in the following, we will examine the three instances more closely. In this chapter, similar to [40], we assume it takes two clock cycles to perform update and capture.

6.2.1. INSTANCE A

The length of shift-registers for this instance are reported in the corresponding row in Table 6.1. Assuming that initially all control bits are set to zero and the goal is to perform a read/write operation on I_4, we calculate the access time for different configuration alternatives of the network. First, we consider the case where the only configuration performed is setting C_0 to "01". Here, two CSUs are needed and access time is calculated as the sum of number of clock cycles needed to (1) configure C_0 in the first CSU and (2) perform one read/write on I_4 in the second CSU. The number of clock cycles for the first CSU is 1 (for C_3) + 2 (for C_0 which is a two-bit register) + 20 (for I_0) + 2 (to perform the update and capture operations). The number of clock cycles for the second CSU is 160 (for instruments I_2, I_3, I_4, and I_0) + 2 (for C_0) + 2 (for the update and capture operations). In total, it takes 189 clock cycles to perform these two CSUs (marked on the plot shown in Figure 6.2). Alternatively, since C_3 is initially on the scan path, it can also be set to '1' in the first CSU. In this case, I_3 will not be on the scan path in the second CSU and it thus takes 169 clock cycles in total to perform the two CSUs (also marked in Figure 6.2).

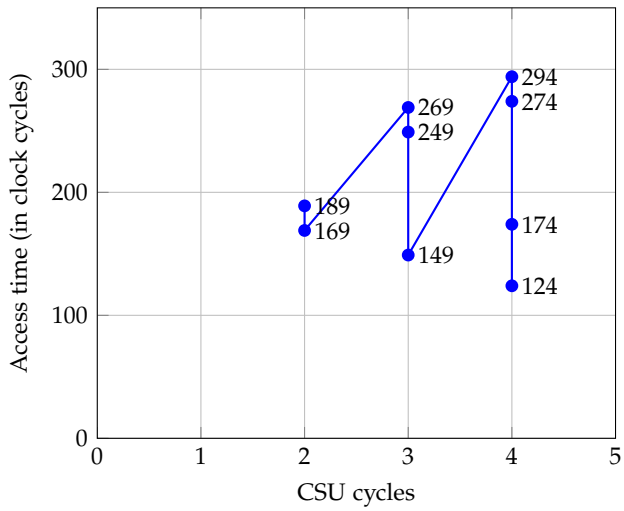


Figure 6.2. Access time vs number of allowed CSUs for Instance A

The two alternatives discussed above used two CSUs to access I_4. That is, if we limit the retargeting tool to unroll the model twice, the pseudo-Boolean optimization explores the above solutions and picks the one with the lowest access time, i.e., the one with 169 clock cycles. In the following, we explore alternative configurations with more than two CSUs.

If instead of switching C_0 to "01", we set it first to "10", we gain access to C_2 and can switch I_2 off the scan path before performing the read/write operation on I_4. In this case, we use three CSUs and the access time is calculated as 149 clock cycles in total. If we allow the retargeting algorithm to use three CSUs, all the solutions marked with two and three CSUs on the plot are explored and the minimum which is 149 will be chosen.

If we switch I_1 off the scan path before configuring C_2, access time might be further reduced. In this case, four CSUs are required in total and the access time is calculated as 124 clock cycles. The plot in Figure 6.2 shows access time for other solutions obtainable by using four CSUs, as well.

For this example, allowing further increase in CSUs will not yield lower access time, but will result in growingly complex models that lower the efficiency of the retargeting algorithm. In this regard, for this instance of the problem, the bound calculation in [40] (see Eq. (6.2)) calculates the bound on the number of CSUs as $\lceil 169/2 \rceil = 85$. Since there are five control bits, unrolling the model 85 times would result in a model with 5×85 decision variables, which should be compared to 5×4 variables when the model is unrolled only four times.

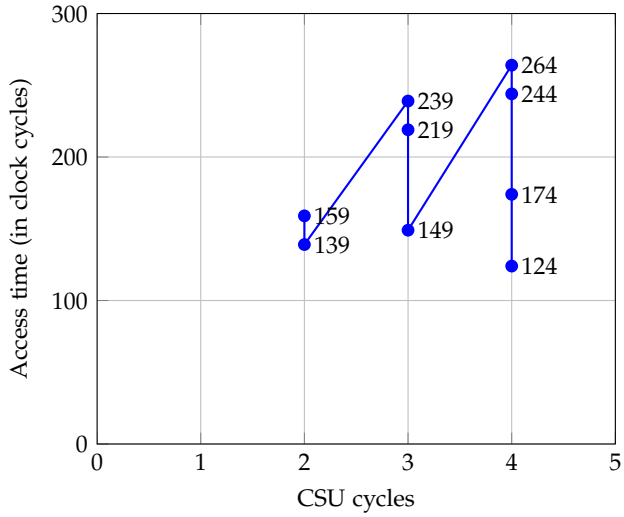


Figure 6.3. Access time vs number of allowed CSUs for Instance B

6.2.2. INSTANCE B

Figure 6.3 shows how the solution space would look like if the length of shift-register for I_2 was 70 instead of 100. It is interesting to see that access time does not decrease when three CSUs are allowed but decreases when four CSUs are allowed. This entails that a heuristic searching the solution space by incrementing the bound on CSUs gets stuck at a local minimum. If, however, the search algorithm is aware of a bound on the number of CSUs, it can do enough unrollings of the model and let the pseudo-Boolean optimization find the minimal access time (as well as the right number of CSUs).

6.2.3. INSTANCE C

Figure 6.4 shows how the solution space would look like if the length of shift-register for I_0 was 50 instead of 20. In this case, the overhead caused by shifting dummy bits through the shift-register for I_0, cancels out any potential benefit from using more CSUs used for removing I_2 and I_3 from the scan path to I_4.

It is important to note that in this example, if the aim was to access I_2 instead of I_4, the optimal solution would be obtained by using a different number of CSUs. The same can be said for other starting configurations (i.e., other than all control bits set to zero). In this work, however, our aim is to find an upper-bound on the number of CSUs that enables reaching the optimal solution for any retargeting step, regardless of the starting configuration and

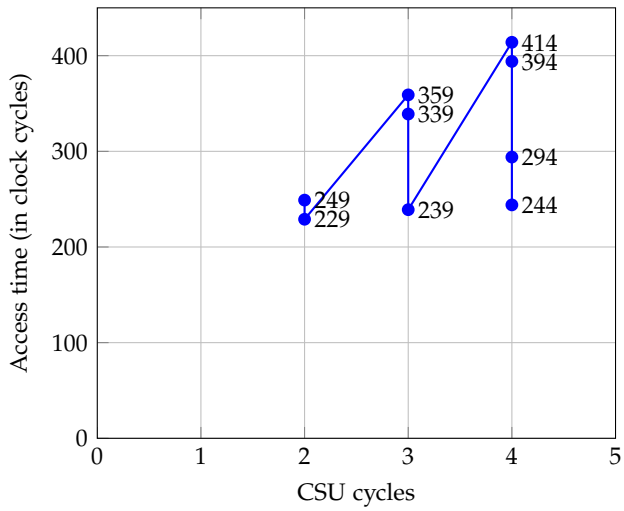


Figure 6.4. Access time vs number of allowed CSUs for Instance C

the set of instruments to be accessed. As was mentioned earlier, the proposed method is applicable when no pair of instruments in the given set are mutually inaccessible. Therefore, in the following section, we propose a method which computes the upper-bound on the number of CSUs as the maximum number of CSUs needed to take the network from any initial configuration to any target configuration. Note that the retargeting algorithm should unroll the model one extra time to account for the actual read/write operation.

6.3. UPPER-BOUND COMPUTATION CORE (UCC)

As was mentioned earlier, we aim to provide a method for computation of an upper-bound on the number of CSUs for a given network. In this section, we explain our generalized Upper-bound Computation Core (UCC) and discuss how its output can be used for optimal retargeting.

6.3.1. THE CORE: UCC

UCC consists of two steps: (1) modeling the network with a finite state machine (FSM), and (2) computation of the upper-bound. In the following sections, each of these steps is detailed. We will use the example network in Figure 6.5 to describe UCC.

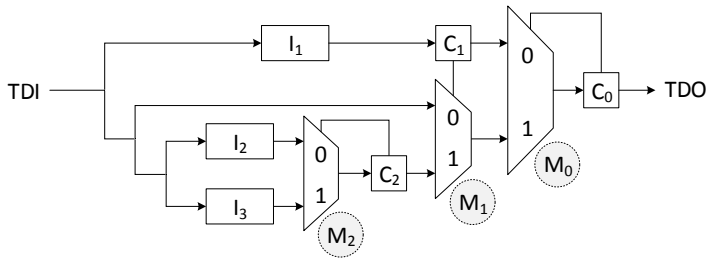


Figure 6.5. Example network used to describe UCC (Section 6.3)

6.3.1.1. MODELING WITH AN FSM

The network in Figure 6.5 has three one-bit mux controllers C_0, C_1, and C_2 and thus has eight possible configurations. The FSM in Figure 6.6 models the network in Figure 6.5, where each state (encoded as the bit sequence $C_2C_1C_0$) represents one of the eight configurations, and each edge models a transition between two states. Transitions which are from a state to itself are not considered in the model. The labels l_i beside transition arrowheads represent the number of clock cycles needed to perform the transition. The required number of clock cycles is calculated as the sum of length (in number of flip-flops) of components on the active scan path (namely, shift-registers and control bits) plus the number of clock cycles needed to perform capture and update operations. Table 6.2 lists the components that are active in each of the states, as well as length of scan path (in number of flip-flops) for each state. In the table, L_i represents the length of shift-register for instrument I_i. As an example, l_0, which corresponds to state 000, is calculated as length of the scan path for state 000 plus two clock cycles (following the assumption in [40]) for capture and update operations. It is worth noting that not all transitions are bidirectional, and that length of a transition is not necessarily equal to the length of the transition in the opposite direction.

6.3.1.2. COMPUTING THE UPPER-BOUND

The FSM in Figure 6.6 can be used to calculate the number of CSUs needed to transition from each of the states to any other state. The number of CSUs is equal to the number of transitions between two states. There might be multiple paths for transitioning between a pair of states. For example, both paths marked with P_1 and P_2 on the FSM in Figure 6.6 can be taken to change the state from 000 to 011, where P_1 takes l_0 clock cycles and P_2 takes l_0+l_2 clock cycles. We are, however, only interested in the number of transitions for the path that uses fewer clock cycles (which is not necessarily the path with fewer number of transitions, as we noted in Section 6.2). Therefore, if we find

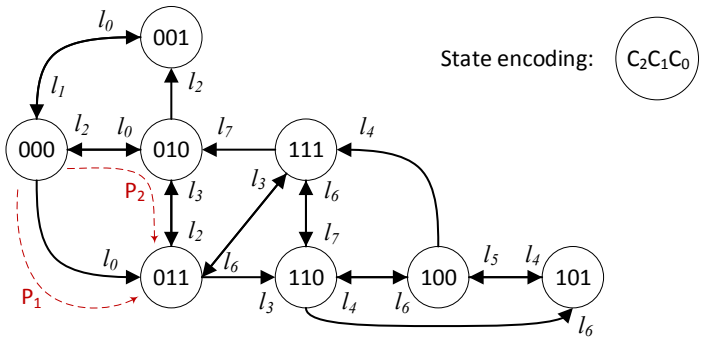


Figure 6.6. FSM showing the transitions for the network in Figure 6.5. Labels beside each arrowhead represent the number of clock cycles needed to perform each transition.

Table 6.2. Paths corresponding to each state

State	Active components	Length of scan path
000	I_1, C_1, C_0	$2 + L_1$
001	C_0	1
010	I_1, C_1, C_0	$2 + L_1$
011	I_2, C_2, C_0	$2 + L_2$
100	I_1, C_1, C_0	$2 + L_1$
101	C_0	1
110	I_1, C_1, C_0	$2 + L_1$
111	I_3, C_2, C_0	$2 + L_3$

the shortest path between any two states s_1 and s_2, and compute the number of transitions (a.k.a. number of hops) needed to achieve that shortest path, we will know how many CSUs are needed for the transition from s_1 to s_2 to achieve the optimal scan vectors. The upper-bound, i.e., the number of CSUs which allows to take the network from any state to any state with the smallest number of clock cycles, can then be computed as the maximum among the number of hops corresponding to each pairwise shortest path.

Assuming a length of 20 flip-flops for instrument shift-registers I_1–I_3, Table 6.3 presents the pairwise shortest path computed between the pairs of states. The first column lists the source states and the first row lists the target states. Table 6.4 presents the number of transitions corresponding to the shortest path between each pair of states in the FSM (Figure 6.6). Based on Table 6.4, the upper-bound on the number of CSUs is found to be four.

Table 6.3. Pairwise shortest paths among the states in Figure 6.6 ($L_i = 20$)

State	000	001	010	011	100	101	110	111
000	0	24	24	24	72	72	48	48
001	3	0	27	27	75	75	51	51
010	24	24	0	24	72	72	48	48
011	48	48	24	0	48	48	24	24
100	72	72	48	48	0	24	24	24
101	75	75	51	51	3	0	27	27
110	72	72	48	48	24	24	0	24
111	48	48	24	24	48	48	24	0

Table 6.4. Number of transitions (hops) corresponding to the pairwise shortest paths among the states in Figure 6.6

State	000	001	010	011	100	101	110	111
000	0	1	1	1	3	3	2	2
001	1	0	2	2	4	4	3	3
010	1	1	0	1	3	3	2	2
011	2	2	1	0	2	2	1	1
100	3	3	2	2	0	1	1	1
101	4	4	3	3	1	0	2	2
110	3	3	2	2	1	1	0	1
111	2	2	1	1	2	2	1	0

6.3.1.3. OPTIMAL RETARGETING FOR SMALL NETWORKS

The pairwise shortest paths information, obtained as described in previous section, can be used to *directly* generate the optimal scan vectors needed for retargeting. That is, instead of using the upper-bound to unroll a SAT model, and solving the resulting pseudo-Boolean optimization, one can use the shortest paths information to find what configuration steps should be taken for taking a network from its current configuration to a target configuration optimally. Since in many target configurations a superset of the desired instruments might be accessible, an approach merely based on the shortest paths information should choose the smallest among the shortest paths from current configuration to all those target configurations. Moreover, the length of the scan path for those configurations should also be taken into account. The reason, as was discussed in Section 6.2, is that the actual goal in retargeting is performing read/write operations on the instruments. Therefore, for optimal retargeting, not only the transition time between states should be taken into account, but the time it takes to perform (at least) one read/write should also

Figure 6.7. A network consisting of N isolated segments.

be considered.

This method of retargeting is, however, only applicable to small networks for which the pairwise shortest paths can be computed efficiently. For large networks, the computation time and memory requirements makes the use of this method inefficient.

6.3.1.4. PESSIMISM IN THE UCC RESULTS

There are two types of transitions that might increase the upper-bound unnecessarily. The first type are transitions that do not change the set of active components, such as transition from state 001 to state 101. The second type are transitions that do change the set of active components, but the new set is achievable via other transitions with smaller number of CSUs and less than or equal number of clock cycles. For example, states 000 and 100 activate the same set of components, but it takes fewer clock cycles to go from 001 to 000 than from 001 to 100. These two transition types make the computed upper-bound slightly pessimistic. In Section 6.5, we present the computed upper-bound both before and after the removal of such pessimism from the results.

6.4. HANDLING LARGE NETWORKS

The method we described in Section 6.3 is not directly applicable to large networks as the number of states in the FSM model grows exponentially w.r.t. the number of control bits. In this section, we describe three techniques (referred to as *reduction* here) that can help in handling large networks. Due to the lack of space, we only detail the implementation of the decomposition technique. We conclude this section by explaining how these reduction techniques are used in a complete upper-bound computation flow.

6.4.1. REDUCTION THROUGH DECOMPOSITION

Figure 6.7 shows a network consisting of N segments S_1–S_N. Each of these segments is connected to the rest of the network exclusively via a scan-in/scan-out pair. In this work, any such segment is referred to as an *isolated* segment. In the network in Figure 6.7, a CSU applied to any of these N segments is also applied to the other $N-1$ segments at the same time. The reason is that the

serial data goes through all the segments and the control signals are applied to (the currently active path) in each of them at the same time. Therefore, the segment requiring maximum number of CSUs determines the upper-bound. That is, the technique described in Section 6.3 can be applied to each segment S_i individually to compute the upper-bound for that segment (denoted as $u_{b,i}$), and the upper-bound for the whole network, denoted by U, can be calculated as:

$$U = \max_{i=1}^{N} u_{b,i} \tag{6.3}$$

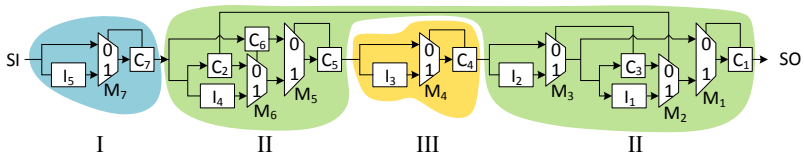
Through decomposition, the worst-case complexity of upper-bound computation for the original network is reduced to the complexity of upper-bound computation for the segment containing the highest number of control bits.

6.4.1.1. IMPACT OF DECOMPOSITION ON UPPER-BOUND

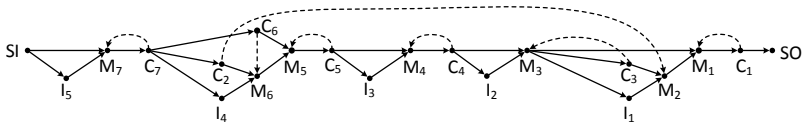
The upper-bound computed via decomposition might be slightly higher than what would be computed if UCC was directly applied to the original network (and therefore, higher than what is actually needed for optimal retargeting). The reason can be explained by referring to the motivational example network in Figure 6.1, which can be seen as combination of two isolated segments: s_1 containing instrument I_0, and s_2 containing the rest of components. We observed for Instance C of that example that an increase in the length of I_0 (from 20 to 50) caused a decrease in the number of CSUs needed for optimal access to I_4 (from 4 to 2). Seen the other way around, going from Instance C to Instance A, which decreases the length of I_0, causes an increase in the number of CSUs needed for optimal retargeting. The same effect is present in decomposition as it removes other segments from each other's scan path. This increased number of required CSUs calculated for each isolated segment, might make the upper-bound computed by the use of decomposition slightly pessimistic.

6.4.1.2. PERFORMING DECOMPOSITION

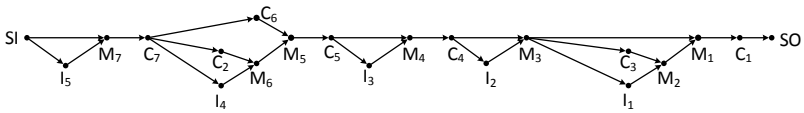
We will now use the example network in Figure 6.8(a) to explain how to distinguish isolated segments. In this figure, the network components belonging to different isolated segments are marked with colored areas. For more clarity, each of the three isolated segments is also marked with Roman numerals. Compared to the conceptual illustration of isolated segments presented in Figure 6.7, in which it is clear where on the scan path an isolated segment begins and ends, it is less straightforward to identify all isolated segments in the network in Figure 6.8(a). Given the exponential complexity of the presented UCC technique w.r.t. number of control bits, it is crucial to identify more (and consequently smaller) isolated segments in a given network.



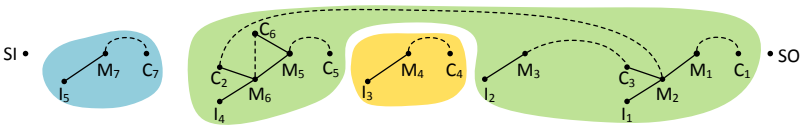
(a) Example network



(b) Corresponsing directed graph with control signals denoted by dashed edges



(c) Corresponsing directed graph without control signals (scan path-only graph)



(d) Corresponsing undirected graph after removing output edges of all vertices on the chain of idoms

Figure 6.8. Decomposition example

In the following, a two-step procedure for identification of isolated segments is presented. In the first step, we identify network segments connected to each other in series on the scan path (hereinafter *candidate segments*). In the second step, based on the control dependencies between these candidate segments, we group them to form isolated segments.

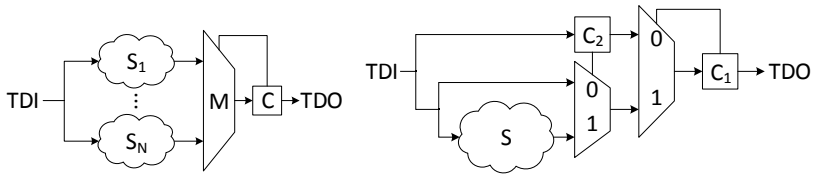
STEP 1 The graph in Figure 6.8(b) models the network in Figure 6.8(a), where the control signals are represented by dashed lines. In identification of candidate segments, we use the concept of *graph dominators*. In a directed graph, vertex v_1 dominates vertex v_2 if all the paths going through v_2 pass first through v_1. For example, in Figure 6.8(b), vertex SI dominates all vertices in the network. However, SI is only *immediate* dominator (called *idom*) to I_5 and M_7. There are efficient algorithms to find idoms for vertices in a graph [58].

Dominators help to identify where on the scan path a candidate segment starts and ends. For example, C_7 marks where isolated segment I finishes

and isolated segment II begins on the scan path. If, however, we apply the concept of dominators directly to the complete network graph in Figure 6.8(b), we fail to identify segment III as an isolated segment. Therefore, we instead apply the graph dominators algorithm to a scan path-only copy of the graph (in which control signals are removed) shown in Figure 6.8(c). Based on the results, we create a chain of idoms for the scan path-only graph by going from the scan-out (SO) towards the scan-in (SI). The chain will be as SI $\Rightarrow$ M_7 $\Rightarrow$ C_7 $\Rightarrow$ M_5 $\Rightarrow$ C_5 $\Rightarrow$ M_4 $\Rightarrow$ C_4 $\Rightarrow$ M_3 $\Rightarrow$ M_1 $\Rightarrow$ C_1 $\Rightarrow$ SO, which reads as SO is immediately dominated by C_1, which is in turn immediately dominated by M_1, and so on. The vertices on this chain mark entry and exit points of candidate segments.

STEP 2 The key to grouping candidate segments into isolated segments is detecting control dependencies between those candidate segments. That is, if there is a control signal connecting two candidate segments, those segments should be grouped and analyzed as one isolated segment. To detect such dependencies, we use a copy of the network graph in which the output edges of all vertices on the chain of idoms are removed, as shown in Figure 6.8(d). To calrify this, we note that the chain of idoms was obtained from the scan path-only graph. Therefore, if after removing the output edges of all vertices on the chain of idoms, two candidate segments are still connected, they are connected via a control signal. Moreover, this graph is converted into an undirected graph, as the aim is to find control dependencies irrespective of the order that candidate segments appear on the scan path. To identify which of the candidate segments should be grouped together, we use the concept of *connected components* in graph theory. A connected component in an undirected graph is a set of vertices in which any two vertices are connected (either directly or indirectly). It should be noted that a "component" in graph theory is a set of vertices, and in our problem maps to an isolated segment, and not to a network component. After applying the connected components algorithm, the isolated segments are identified as marked with the colored areas in Figure 6.8(d). The algorithm also identifies SI and SO as isolated segments, which we ignore. It can be seen that via these two steps, we successfully identified isolated segments in the network in Figure 6.8(a).

In this example, there were no instruments in the chain of dominators, as there was no instrument directly on the scan path between scan-in vertex SI and scan-out vertex SO. When there are instruments on the chain, they can be ignored, because if we form separate isolated segments for them, the upper-bound for that segment is zero (simply because there are no control bits in such an isolated segment).



(a) Type I structure (b) Type II structure

Figure 6.9. Example structures for the "lookup" technique

6.4.2. REDUCTION THROUGH "LOOKUP"

Another technique for handling upper-bound computation for large networks is to recognize structures for which we know how to calculate the upper-bound. In this thesis, we present two such structures shown in Figure 6.9:

- Type I structure (Figure 6.9(a)): In this structure, the each of the segments S_1–S_N is isolated (in the sense defined in Section 6.4.1). As in any retargeting step, only one of the inputs to mux M can be active, only one of the segments S_1–S_N is required to be configured. Therefore, in computation of the upper-bound for the complete structure, it suffices to consider only the segment that requires the largest number of CSUs. For the Type I structure, the upper-bound (for the whole structure) can be computed as:

$$1 + \max_{i=1}^{N} u_{b,i} \tag{6.4}$$

  where $u_{b,i}$ is the upper-bound computed for segment S_i, and 1 represents the CSU needed to configure mux M itself.

- Type II structure (Figure 6.9(b)): For this structure, irrespective of the current configuration of the network, it takes maximum two CSUs to program C_1 and C_2 such that segment S becomes accessible. Therefore, the upper-bound for the whole structure is the upper-bound for segment S plus two.

6.4.3. REDUCTION THROUGH REWRITING

The idea in rewriting is to create a network which is equivalent to the original network w.r.t. the upper-bound on the number of CSUs, but can be handled by the other reduction techniques (such as decomposition and lookup mentioned above). An example of rewriting is presented in Figure 6.10 where the network to the left is rewritten by duplicating control bit C_0 along with its associated mux. The resulting network (to the right) can then be reduced by

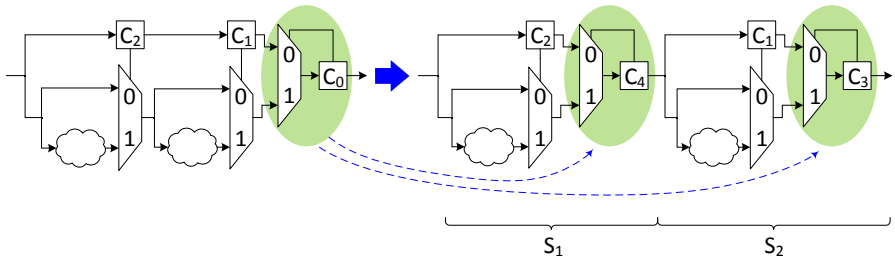


Figure 6.10. An example rewriting technique

using the technique in Section 6.4.1, as each of the segments marked by S_1 and S_2 are isolated. Note that although the functionality of the rewritten network is different from the original network, the upper-bounds of both networks are equal.

6.4.4. THE COMPLETE UPPER-BOUND COMPUTATION FLOW

In the following, we describe our complete upper-bound computation flow, which is based on the use of UCC (described in Section 6.3) and the reduction techniques described earlier in this section. Initially, the rewriting method is used to create a new network that has the same upper-bound as the original network. The computation of upper-bound starts by applying decomposition, which identifies one or more isolated segments. The lookup technique is then applied to each of these segments. If the lookup does not recognize any known structures, UCC is performed on the segment. However, if the lookup recognizes a structure, it calls the decomposition technique on the isolated segments existing within the recognized structure.

In other words, after performing the initial rewriting, the upper-bound computation consists of a number of calls between the decomposition and lookup methods. When an isolated segment is not recognized by the lookup function, UCC is applied to it. The upper-bound computed for each segment is then used to compute the upper-bound for the whole network by using the formulas described for each of the reduction techniques.

6.5. EXPERIMENTS

We implemented the proposed upper-bound computation method and applied it to a number of benchmark circuits. The considered benchmarks are divided into three groups. The first and second groups are introduced in [39] and reused in [40]. The instruments used to construct these networks are extracted from ITC'02 [48] benchmark set as explained in Appendix D.1. The

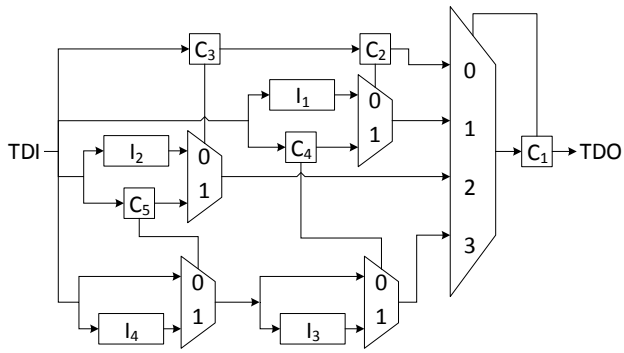


Figure 6.11. N1

networks in the first group are SIB-based and those in the second group are Daisy-chained (referred to as MUX-based in [40]). Appendix D.2 provides details on the architecture of these networks. Our initial experiments showed that the networks in the first and second group were completely reducible by the proposed reduction techniques. Therefore, there was a need to new benchmark networks that exercise the UCC technique, as well. That is why we constructed a third group of benchmarks, consisting of the following networks:

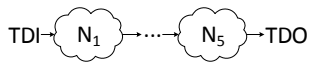
- a group of networks, referred to as N1–N5, that are not reducible by the reduction techniques presented in this thesis, and therefore, UCC should be applied to the complete network. Figure 6.11 shows the smallest (w.r.t. the number of control bits) in this group. See Appendix D.2.2 for the rest of the networks in this group.
- the C1–C2 networks (Figure 6.12), which are constructed by combining the N1–N5 networks such that the combination network exercises the reduction techniques, as well as the UCC technique.

For all benchmarks, the length of instrument shift-registers is assumed to be 20 flip-flops. Moreover, following the assumption in [40], the number of clock cycles needed for performing capture and update operations is assumed to be two clock cycles.

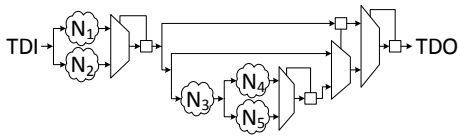
The results obtained by evaluating the techniques proposed in this chapter are summarized in Table 6.5. The first two columns of the table list the names and the total number of control bits for each benchmark network. The third column reports the maximum number of control bits required to model irreducible sections within the network. This information is important since the

Table 6.5. Experimental results

Benchmark	# of control bits		Upper-bound, before & after pessimism removal		Reductions run-times (milliseconds)			UCC run-time (milliseconds)			
	Total	Max seen by UCC			Rewriting	Decomposition	Lookup	FSM generation	Shortest-path computation		Pessimism removal
									Dijkstra	Floyd-Warshall	
The following are SIB-based networks from [39]:											
a586710	39	0	3	3	2.7	12.4	391.7	0	0	0	0
d281	58	0	2	2	3.4	20.5	843.1	0	0	0	0
d695	167	0	2	2	8.7	72.9	8371.8	0	0	0	0
f2126	40	0	2	2	2.4	8.6	409.0	0	0	0	0
g1023	79	0	2	2	4.2	40.8	1554.8	0	0	0	0
h953	54	0	2	2	2.8	17.8	791.8	0	0	0	0
p22810	282	0	3	3	20.7	333.9	27295.7	0	0	0	0
p34392	122	0	3	3	7.2	83.7	3824.8	0	0	0	0
p93791	620	0	3	3	64.1	1176.8	211621.3	0	0	0	0
q12710	25	0	2	2	1.2	5.2	167.7	0	0	0	0
t512505	159	0	2	2	8.5	182.3	6416.0	0	0	0	0
u226	49	0	2	2	2.6	17.6	533.6	0	0	0	0
The following are MUX-based networks from [39]:											
a586710	47	0	6	6	10.4	96.8	407.7	0	0	0	0
d281	67	0	4	4	14.1	210.0	907.6	0	0	0	0
d695	178	0	4	4	30.4	2120.2	10210.5	0	0	0	0
f2126	45	0	4	4	8.1	100.9	424.7	0	0	0	0
g1023	94	0	4	4	19.9	392.2	1741.5	0	0	0	0
h953	63	0	4	4	12.6	186.0	816.2	0	0	0	0
p22810	311	0	6	6	88.0	7340.9	37807.9	0	0	0	0
p34392	142	0	6	6	38.1	969.6	4389.2	0	0	0	0
p93791	653	0	6	6	226.1	54435.0	311911.3	0	0	0	0
q12710	30	0	4	4	4.7	41.7	171.7	0	0	0	0
t512505	191	0	4	4	56.7	1797.1	8532.1	0	0	0	0
u226	59	0	4	4	10.5	141.8	605.8	0	0	0	0
The following networks are constructed for the current work:											
N1	6	6	5	4	0.4	0.4	0.7	0.6	1.1	0.2	0.2
N2	7	7	12	8	0.5	0.4	0.7	1.2	3.1	1.1	1.0
N3	11	11	8	8	0.6	0.6	1.9	37.3	629.1	2334.9	1284.2
N4	12	12	7	6	1.3	0.7	0	127.0	4932.0	39822.8	21927.2
N5	15	15	11	8	1.4	0.8	1.5	1857.0	488686.0	16425900.0	1828870.0
C1	51	15	12	8	7.0	2.8	15.5	1982.6	493760.5	16421300.0	1852375.0
C2	55	15	14	11	7.4	12.6	48.9	2026.8	509139.0	16434350.0	1852115.0



(a) C1



(b) C2

Figure 6.12. Two networks constructed by combining N1–N5 networks

number of control bits significantly impacts the run-time of UCC. It can be observed that for those benchmarks constructed to be irreducible, namely, N1–N5, the proposed reduction techniques do not succeed to reduce the number of control bits. On the other hand, if the reductions are successfully applied, such as for the set of SIB-based and Daisy-chained (MUX-based) benchmarks, the generation of an FSM and the application of UCC can be completely omitted. The reason is that for networks in the first and the second groups, the reduction techniques reduce the networks into a number of isolated segments each containing only one instrument shift-register. As was mentioned in Section 6.4.1, the upper-bound for an isolated segment containing only instrument shift-registers is zero—hence no need for applying UCC.

The computed upper-bounds are listed in columns four and five, before and after pessimism removal, respectively. The computed upper-bound denotes the maximum number of CSUs needed to reconfigure the network by using the minimum number of clock cycles. Pessimism removal is only used in the UCC technique and therefore has no effect on the results for the first and second groups of networks. Comparing the results in columns four and five for the third group of benchmarks shows that the pessimism removal can have a significant effect on the efficiency of retargeting. For example, for network C1, the upper-bound is reduced from 12 to eight, which translates into 51×4 less variables for the retargeting tool to deal with. In order to perform the actual read/write operation an additional CSU is required (see Section 6.2).

The described reduction techniques, such as rewriting, decomposition, and lookup, are evaluated in the columns six to eight. The reported run-times are the total sum over all the application cases of each of these techniques for each of the benchmarks. Applying the reduction techniques to the largest among SIB-based and Daisy-chained (MUX-based) benchmarks (i.e., p93791) requires up to more than a total of six minutes of run-time. For the third

group of benchmarks the run-time of the reduction techniques is negligible.

The run-time for generating the FSM after reduction is listed in column nine. As was explained in Section 6.3.1.2, to compute the upper-bound from the generated FSM, the shortest path between each pair of states should be computed. To do so, we evaluated two well-known shortest path computation algorithms, namely, Dijkstra and Floyd-Warshall. The Dijkstra algorithm finds the shortest path between a given source state and all target states, and is therefore run once for each state in the FSM. The run-time reported for Dijkstra algorithm in column 10, is the sum of the run-times for each source state. The Floyd-Warshall algorithm is, on the other hand, an all-pairs shortest-paths algorithm and finds the shortest path between all pairs of states in the FSM in one run. The run-time for the Floyd-Warshall algorithm is reported in column 11. The observation from our experiments is that the Dijkstra algorithm performed especially well on large FSMs (namely, for benchmarks N_3–N_5 and consequently C1–C2), whereas the alternative Floyd-Warshall algorithm required slightly less runtime on small FSMs. In general, the Floyd-Warshall algorithm has higher complexity (compared to running Dijkstra once for each source state) when the FSM is a sparse graph. Both algorithms delivered the same results. The last column in the table reports the time it took to perform the pessimism removal in the UCC technique. Finally, as was mentioned earlier in this section, there is no UCC run-time required for the benchmarks in the second and third groups.

6.6. CHAPTER CONCLUSIONS

For the problem of optimal retargeting for 1687 networks, the shrinking of the solution space is highly important in order to ensure efficient generation of the optimal scan vectors. This can be done by providing bounds on how many CSU operations have to be considered in the retargeting step. To provide such bounds, in this chapter, we proposed a method for the computation of upper-bound on the number of CSUs. The proposed method uses a number of techniques that make it applicable to a range of complex and large 1687 networks. By applying the approach to a set of benchmarks, it is shown that the method is able to efficiently provide tight bounds for complex and large benchmark networks.

Part IV

Application

7

IEEE 1687 Networks for Fault Monitoring

As was mentioned in Chapter 1, due to phenomena such as soft errors, intermittent faults, and aging, there is a need for in-field monitoring of the operation of SoCs. In-field monitoring can be done by embedding instruments into the SoCs for detecting errors[1] or measuring health related parameters, such as voltage droop, current, temperature, etc. Such instruments can be connected to a *fault manager* that makes decisions based on the collected error statuses. The fault manager can be implemented in an on-chip or off-chip processor. In either case, there should exist a network for connecting the fault manager to the on-chip monitoring instruments. It is important that the latency in a fault monitoring network is kept low, as the earlier the fault manager gets aware of errors in the system, the faster it launches recovery actions. Moreover, the latency should be deterministic to let designers of a system assess its reliability. In this thesis, we consider two types of latency in the fault monitoring network:

- *fault detection time*: the time interval between detection of an error by a monitoring instrument and when the fault manager gets aware of presence of an error in the system, and
- *fault localization time*: the time it takes between detection of a fault by the fault manager and when the fault manager identifies the faulty resource and extracts the error code reported by the respective monitoring instrument.

A fault monitoring network can be stand-alone, or part of an existing functional infrastructure such as network-on-chip or system bus. There are advan-

[1] In this thesis, we use the terms fault and error interchangeably even though in practice these two concepts are different, i.e., an error is a manifestation of a fault.

tages and drawbacks with using an existing infrastructure for the additional purpose of fault monitoring. The advantage is that no extra hardware cost is incurred. One drawback is that adding traffic of fault monitoring information may impact the performance of the system, as it might be difficult at design time to estimate the timing and the amount of traffic information that is to be generated from occurrence of errors. Another drawback is that the predictability of the fault monitoring system is reduced, as the traffic on the functional network might also affect the latency of the fault monitoring information. To be on the safe side, the network might be over-designed to ensure that performance is kept high, which is however costly. With a stand-alone network, the advantage is twofold: it does not impact the performance of the system, and simplifies achieving a deterministic fault detection and localization time. The downside of using a stand-alone network is adding extra hardware cost, if it is added only for the purpose of fault management. However, many ICs are already equipped with stand-alone networks that are, e.g., accessed via the TAP, to enable test, diagnosis, configuration, etc. This makes the reuse of such networks for fault monitoring and error handling during operation attractive.

There have been a number of works on networks for transporting monitoring data (for transient faults, timing errors, power estimation, etc.) using a dedicated infrastructure [59, 60, 1, 22, 23]. The works in [1, 22, 23] stand out as they rely on reusing the existing 1687 network for monitoring purposes. The assumption in these works is that the IC is to be equipped with embedded monitoring instruments that can detect errors and raise error flags, and that these on-chip monitoring instruments are to be interfaced to a 1687 network. In this thesis, we follow these assumptions and additionally assume that the monitoring instruments produce error codes according to the type of the detected errors. We propose a scheme where the 1687 network is self-reconfigured (while maintaining standard compliance) to automatically include the instrument registers containing error codes in its scan path. The proposed scheme enables very fast error detection, and achieves significantly faster fault localization compared with [1, 22, 23].

We begin the discussion in this chapter by reviewing the related work (Section 7.1). We describe the hardware structure of our proposed self-reconfiguring networks in Section 7.2 along with an example illustrating how the fault localization is done. In Section 7.3, we present fault detection and localization time analysis for two cases: when a single fault occurs, and when multiple faults occur concurrently. Section 7.4 presents a method for optimal design of self-reconfiguring networks, and Section 7.5 focuses on the fault manager and details a hardware module that greatly facilitates the extraction of error data from the network during the localization process. In Section 7.6, we compare the fault detection and localization times of the proposed self-reconfiguring scheme against previous IEEE 1687-based fault management schemes. Finally,

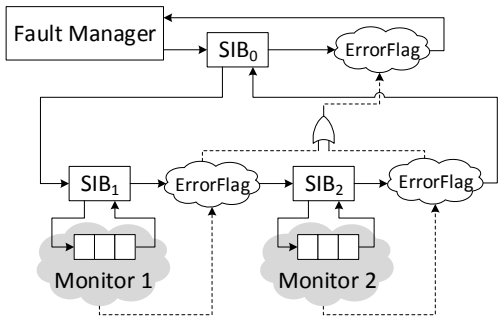


Figure 7.1. A simplified representation of the basic idea in [1]

in Section 7.7, we discuss practical issues regarding the implementation of the proposed self-reconfiguring networks.

7.1. PRIOR WORK

In this section, prior work on fault management using 1687 networks as the fault monitoring infrastructure is discussed.

Hierarchical 1687 networks have been used in fault management schemes to connect instruments to a fault manager [1, 22, 23]. In [1], methods for optimized design and calculation of error localization time are presented for their proposed fault management scheme. The work in [22] extends [1] by elaborating on how the fault manager can react faster to new faults while the instrument access network is in use for other purposes and how multiple faults can be addressed, but presents no time analysis method or experimental results for such cases. In [23], a simulation-based platform for experimenting with fault injection and fault management is elaborated, but no time analysis or network optimization method is presented.

Along with the 1687 network, [1, 22, 23] use a fully combinational error flag propagation network which propagates error flags to the highest hierarchical level of the 1687 network. A simplified representation of the hierarchical networks used in [1] is shown in Figure 7.1 where the error flag propagation network is marked by the dashed lines. The advantage is that by reading the ErrorFlag register in the highest level the fault manager gets informed of any error in the system without checking each and every instrument. To guide fault localization, [1, 22, 23] added ErrorFlags at every level, resulting in dramatic increase in fault localization time. Also, fault localization in [1, 22, 23] involves a number of CSUs to open hierarchical levels, each CSU performed over a scan path longer than the scan path for the previous CSU, increasing

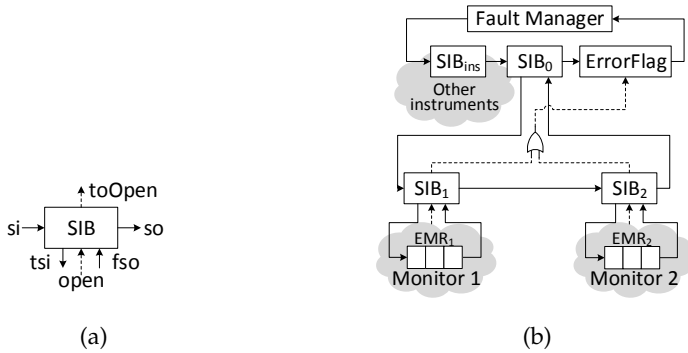


Figure 7.2. (a) Symbol for the *modified* SIB, and (b) An example self-reconfiguring network (the dashed line represents the error flag propagation network)

the fault localization time. In this regard, recall from the time analysis for the example network in Figure 3.1(b) how for accessing the instruments in the second hierarchical level, the SIBs in the first level had to be programmed for every CSU (see Section 3.2.1.2).

In this thesis, to address the fault localization time, we consider a fault management scheme similar to those in [1, 22, 23] and propose self-reconfiguration. We show that by adding self-reconfiguration it is possible to reduce the fault localization time significantly while keeping conformity to the IEEE 1687 rules.

7.2. SELF-RECONFIGURING NETWORK

In this section, we describe the hardware structure of the self-reconfiguring networks (Section 7.2.1), as well as how to detect and localize errors in the proposed structure (Section 7.2.2).

The basic idea in self-reconfiguration is that when a fault is detected by a fault monitor, the corresponding error code register is automatically included in the active scan path so that its contents can be readily shifted out and analyzed. Such scheme, improves the speed of fault localization via (1) avoiding to open levels of hierarchy one level at a time, and (2) using only one single-bit ErrorFlag register instead of placing multiple such registers at each hierarchical level.

7.2.1. HARDWARE STRUCTURE

In this work, we assume a hierarchical 1687 network interfacing all embedded instruments (test, debug, fault monitors, etc.) in a system to a Fault Manager, which has the purpose of detecting and localizing errors that may occur in different components of the system over time, such that it can initiate necessary fault handling actions. The novelty of this work relies on the fact that part of the hierarchical 1687 network has the feature of self-reconfiguration.

Figure 7.2(b) shows an example of a self-reconfiguring network. Among all the instruments, we assume that there is a set of fault monitoring instruments. In the top level of the hierarchical network, the fault monitoring instruments are connected through a dedicated SIB, denoted with SIB_0, while all the other instruments (test, debug, etc.) are connected through another SIB, denoted with SIB_{ins}. The top level also includes a one bit shift-register (ErrorFlag) to indicate if any errors are detected by any of the fault monitoring instruments.

We assume that a fault monitoring instrument has a *fault flag* output terminal that is set to logic '1' in case a fault is detected. The *fault flag* stays active until it is acknowledged via a *clear flag* input terminal. The fault flag signal will be used as an input to reconfigure the network, such that an access to the fault monitoring instrument is enabled. Furthermore, the fault flag signal is propagated across the hierarchical levels and is finally captured by the ErrorFlag register in the top level of the hierarchical network.

Additionally, we assume that a fault monitoring instrument produces an error-code which is parallel-loaded during the capture phase into an error-code/mask register (EMR) interfacing the instrument to the 1687 network. An EMR is assumed to have capture and update features (similar to standard 1149.1 TDRs) and it contains an error-code field (written by the fault monitor) and a mask field (written by the Fault Manager). Error masking is used to stop a permanent fault from constantly raising the fault flag. To be compliant with the IEEE 1687 standard, error masking should be enabled by default at reset to disable self-reconfiguration of the network. When the EMR of a fault monitoring instrument is selected and data is shifted through it, the *clear flag* is asserted to indicate that the fault from the fault monitor has been acknowledged. In Figure 7.2(b), the 3-bit registers, namely EMR_1 and EMR_2, are the EMRs associated to Monitor 1 and Monitor 2, respectively.

To enable self-reconfiguration, we propose a *modified* SIB, which is the core component in a self-reconfiguring network. A *modified* SIB, while being IEEE 1687 compliant, can additionally be opened asynchronously via a dedicated terminal. The symbol shown in Figure 7.2(a) will be used in this thesis to represent a *modified* SIB. In Section 7.7, we detail the circuitry of the proposed *modified* SIB.

All fault monitoring instruments in the network are connected to the top-

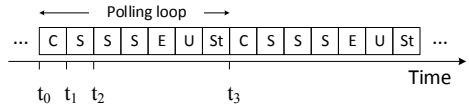
level SIB_0 through a network of *modified* SIBs. The main difference between a regular SIB and a *modified* SIB is the pair of terminals "open" and "toOpen". The "open" terminal of a *modified* SIB is connected either to (1) the fault flag of the monitoring instrument—see SIB_1 and SIB_2 in Figure 7.2(b)—or (2) the ORed output of the "toOpen" terminals of all *modified* SIBs attached to it (placed one hierarchical level below). When the "open" terminal is asserted (pulled high), it changes the state of the SIB to opened only if the SIB is not already part of an active scan path. The signal from the "open" terminal is gated internally using (an inverted copy of) the select signal to make sure that the state of the SIB does not change (from closed to opened) when it is part of an active scan path (see Figure 7.9 for details on the *modified* SIB). The "toOpen" terminal propagates the internally gated signal (from the "open" terminal) via an OR gate either to (1) the *modified* SIB in the hierarchical level above, or (2) the ErrorFlag register in the top level—see Figure 7.2(b). Note that when the fault flag has managed to propagate to the ErrorFlag register, all the *modified* SIBs on the path from the fault monitor raising the flag to the top level SIB_0 are properly configured, i.e. the network has self-reconfigured.

A requirement for a *modified* SIB (as well as for SIB_0 and SIB_{ins}) is to have its shift (S) flip-flop placed after the hierarchical mux (similar to what is shown in Figure 2.7(a)). Such placement, while being fully standard compliant, ensures that during shifting, the state of the SIB is always shifted out first. This is required by the fault-localization method to determine the current configuration of the network.

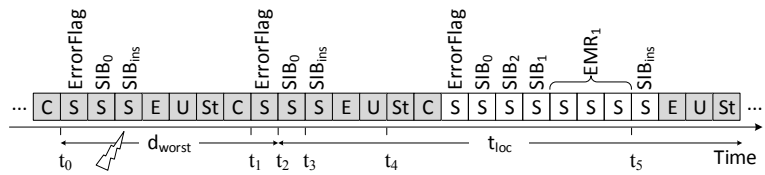
7.2.2. FAULT DETECTION AND LOCALIZATION METHOD

In this section, we explain the fault detection and localization method with the help of the example network shown in Figure 7.2(b) and the timelines shown in Figure 7.3. The following scenarios are considered: (1) no error has occurred, (2) an error occurs when the Fault Manager is not localizing another error, (3) an error occurs when the Fault Manager is localizing another error, and (4) two errors occur in a short span of time when the Fault Manager is not localizing another error.

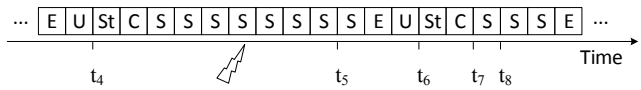
For the first scenario, when no error occurs, the Fault Manager constantly checks the status of the system by polling the value captured by the ErrorFlag register. The Fault Manager does the polling via looping constantly through the *Capture*, *Shift*, *Exit1*, *Update*, and *Select* states in the DR branch of the TAP controller state machine. Since SIB_0 is closed when no errors are being localized, the polling takes seven test clock cycles (TCK)—the interval between t_0 and t_3 in Figure 7.3(a)—as three shifts are required: for SIB_{ins}, SIB_0, and ErrorFlag. The value of the fault flag raised by monitoring instruments is captured at t_1 into ErrorFlag register and can be observed at t_2 (see Figure 7.3(a)). The



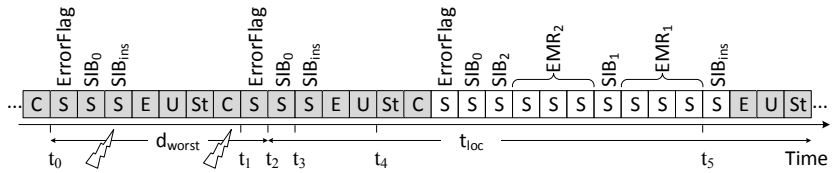
(a) **C**: Capture, **S**: Shift, **E**: Exit, **U**: Update, **St**: Select



(b)



(c)



(d)

Figure 7.3. The detection and localization method: (a) constant polling to detect a fault, (b) an error is detected and localized, (c) another error happens when the previous one is being localized, and (d) when two faults are detected together.

polling continues as long as the shifted out bit corresponding to the ErrorFlag register is a '0'. During polling, zeros are shifted in to keep SIB_0 and SIB_{ins} closed.

For the second scenario (see Figure 7.3(b)), consider that a fault happens at the interval between t_0 and t_1 and is reported by Monitor 1. The reason we chose this interval is that no matter when in this interval a fault occurs, it will not be captured until t_1 and will therefore not be detected until shifted out at t_2. We refer to the interval between t_0 and t_2 (which is eight TCKs long) as the worst-case fault detection time (when no other error is being localized) and denote it by d_{worst}. When the value shifted out at t_2 (which belongs to

the ErrorFlag) is a '1', the localization procedure is launched by shifting a '1' into SIB_0 at t_3 which takes effect at the following Update phase (t_4). Once SIB_0 is open, as the rest of the network is already self-reconfigured, the Fault Manager starts shifting out data from the network (while shifting in zeros to close the SIBs and reset EMRs on the active scan path) to localize the fault: The first two bits shifted out are the contents of ErrorFlag and SIB_0. The third bit is the contents of SIB_2 for which a value of zero indicates that SIB_2 is closed and the fault is not reported from the network segment connected to the host port of SIB_2. The next bit is the contents of SIB_1 which is '1' meaning that SIB_1 is open and the fault is reported from the segment connected to it, i.e., Monitor 1 in this example. The next three bits are the contents of the 3-bit EMR_1 which interfaces Monitor 1 to the 1687 network. At this point, i.e., at t_5, the error is localized and the error information is retrieved. In this work, however, we include in the localization time (denoted by t_{loc}) the next four TCKs needed to shift-in one more zero for SIB_{ins} and take the TAP controller state machine back to the capture phase. The worst-case error detection and localization time can then be written as:

$$t_{worst} = d_{worst} + t_{loc} \tag{7.1}$$

where d_{worst} is the worst-case fault detection time (when no other error is being localized), and t_{loc} is the fault localization time.

In practice, for the above scenario, d_{worst} should be extended to include the time that it takes a fault flag signal to propagate from the fault monitoring instrument to the ErrorFlag. We denote this propagation delay by δ and note that if the fault monitor signals the error later than t_0-δ, it is not captured at t_0. Therefore, d_{worst} should be written as t_2-t_0+δ which is equal to $8/f_{TCK} + \delta$ where f_{TCK} is the maximum frequency that the TAP can be operated at.

For the third scenario, when an error happens while the Fault Manager is localizing a previous error, consider Figure 7.3(c) as continuation of the timeline in Figure 7.3(b). As discussed for the second scenario, at t_4 SIB_0 is opened which puts SIB_1 and SIB_2 on the active scan path. SIB_0 is closed at t_6 meaning that between t_4 and t_6 SIB_2 is selected (though closed) and, therefore, cannot be opened by a fault flag signal from Monitor 2. That is, any fault reported by Monitor 2 after t_4, is captured at t_7 and detected at t_8. Since SIB_2 is closed, the fault flag from Monitor 2 is not acknowledged and therefore remains active until SIB_2 is opened and the error code from Monitor 2 is captured into EMR_2.

For the last scenario, consider the timeline in Figure 7.3(d), where both monitors detect faults in the interval between t_0 and t_1. In this case, both faults are detected at t_2. In comparison to the scenario for one fault (see Figure 7.3(b)), the localization procedure takes a longer time as this time SIB_2 is also opened and EMR_2 is also included in the scan path.

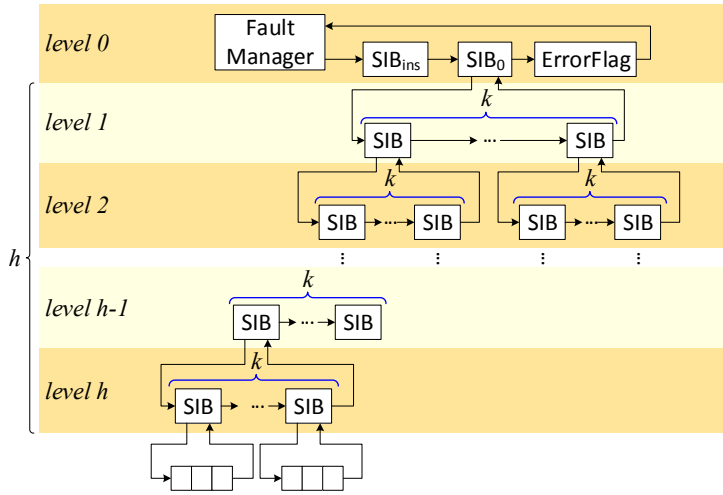


Figure 7.4. A balanced tree hierarchical network

As a final note in this section, we observe from Figure 7.3(b) and Figure 7.3(d) that the shaded states are traversed no matter how many faults are being detected and localized. We denote this constant overhead of 18 TCKs by J_{OH}, and write Eq. (7.1) as:

$$t_{worst} = J_{OH} + t_s \tag{7.2}$$

where t_s denotes the number of shift cycles in t_{loc} and varies with the number of faults being localized.

7.3. TIME ANALYSIS

In this section, we present analyses for the worst-case error detection and localization time (t_{worst}) in a self-reconfiguring network, for two cases: when a single fault occurs (Section 7.3.1), and when multiple faults occur such that they are all detected by the Fault Manager at the same time (Section 7.3.2).

As shown in Eq. (7.2), t_{worst} has a constant part J_{OH} and a variable part t_s. For the analyses we focus on calculating t_s.

We present time analyses for balanced tree networks. Figure 7.4 illustrates a network connecting $N = k^h$ instruments that resembles a k-ary tree whose root is SIB_0. In this network, each doorway SIB has k SIBs directly connected to its host port, and there are $h+1$ levels, where the instruments are interfaced through the SIBs in the lowest level.

7.3.1. SINGLE FAULT

Given the network in Figure 7.4, assume that only one monitor has raised a fault flag causing all SIBs on its hierarchy to change state to opened, and that SIB_0 is also opened by the fault manager. In this case, the number of SIBs on the scan path is calculated as follows. There are h opened doorway SIBs at each of the hierarchical levels 0 to $h-1$, and one opened instrument SIB at level h, which is connected to the EMR of the monitor. Each of the h opened doorway SIBs has k SIBs on its host port. Therefore, including SIB_0 and SIB_{ins}, there are $s = 2 + k \times h$ SIBs on the path to each monitoring instrument. The total shift time t_s is therefore the sum of s and the length of the EMR (denoted by L) plus one for ErrorFlag:

$$t_s = 2 + k \times h + L + 1 = 3 + k \times \log_k N + L \tag{7.3}$$

7.3.2. MULTIPLE FAULTS

Assume that F faults ($F \leq N$) are to be localized at the same time (see the discussion on Figure 7.3(d)). To calculate t_s, we consider monitors detecting these faults to be spread in the network such they cause maximum possible number of SIBs to be opened (maximizing the length of the active scan path, thus leading to the longest localization time). As an example, when $F = k$ faults happen in the system monitored via the network in Figure 7.4, the localization time is maximized when each of these k faults happen in the subtree of each of the k SIBs in level 1. Another observation is that for localization of $F \geq 1$ faults, the SIB at level 0 is opened, for $F \geq k$, all SIBs in level 1 are opened, for $F \geq k^2$, all SIBs in level 2 are opened, and so on. The number of these SIBs, which are on the scan path to all F monitors (i.e., shared by all of them), is captured by:

$$\sum_{i=0}^{r} k^i \tag{7.4}$$

where r is the number of upper levels in which all the SIBs are open:

$$r = \lceil \log_k F \rceil \tag{7.5}$$

Next, to calculate the number of SIBs exclusively on the path to each of the F monitors, we note that $h - r$ remaining lower levels are open exclusively for each fault, each having k SIBs. Therefore, the total number of SIBs exclusively opened for the F monitors is:

$$F \times k \times (h - r) \tag{7.6}$$

To sum up, the total number of SIBs on the scan path for the F faults is:

$$s = 1 + \sum_{i=0}^{r} k^i + F \times k \times (h - r) \tag{7.7}$$

where 1 is for SIB_{ins}. To calculate t_s, we need to add to this number of SIBs, the total length of EMRs on the scan path (i.e., $F \times L$) as well as one for the ErrorFlag, as follows:

$$t_s = 1 + \sum_{i=0}^{r} k^i + F \times k \times (h - r) + F \times L + 1 \tag{7.8}$$

7.4. NETWORK DESIGN

In this section, we describe a method for designing a self-reconfiguring network for N instruments, such that t_{worst} for a single fault is minimized. As t_{worst} has a constant part J_{OH} and a variable part t_s (see Eq. (7.2)), minimizing t_{worst} reduces to minimizing t_s.

Given an arbitrary number of instruments N, it might not be possible to construct a balanced k-ary tree for $k \neq N$. In such cases, a straightforward way to construct the network is to create a balanced tree for $k^{\lceil \log_k N \rceil}$ instruments. Following from Eq. (7.3), the total shift time for such a tree can be written as:

$$t_s = 3 + k \times \lceil \log_k N \rceil + L \tag{7.9}$$

As t_s in Eq. (7.9) is not continuous, to minimize t_s, we assume it to be a continuous function, thus transforming Eq. (7.9) to Eq. (7.3). To find k that minimizes t_s, we set the first derivative of t_s w.r.t. k to zero:

$$t_s = 3 + \ln N \times \frac{k}{\ln k} + L \;\Rightarrow\; t_s' = \ln N \frac{\ln k - 1}{(\ln k)^2} \tag{7.10}$$

$$t_s' = 0 \;\Rightarrow\; \ln k = 1 \;\Rightarrow\; k = \mathrm{e} \tag{7.11}$$

Given that $\mathrm{e} \approxeq 2.72$, we can choose either $k = 2$ or $k = 3$. However, solving the relaxation of an optimization problem does not necessarily result in the optimal solution for the original problem. Therefore, based on the results of the relaxation, we describe a straightforward method (Section 7.4.1), as well as a heuristic (Section 7.4.2) that use $k = 2$ and $k = 3$ for minimization of t_s.

7.4.1. PRUNED TREES

Generally, given an arbitrary number of instruments N where N is not a power of two or three, it is not possible to construct a balanced tree. In such cases, a straightforward way to construct the network is to create a balanced tree for $k^{\lceil \log_k N \rceil}$ instruments, where $k = 2$ results in a binary tree and $k = 3$ results in a ternary tree, and prune the tree (after placing the N instruments at the leaf nodes). Pruning can be done by removing the internal nodes to which one or no instrument is connected. After pruning, one can compare the results from the pruned binary and ternary trees and pick the better one.

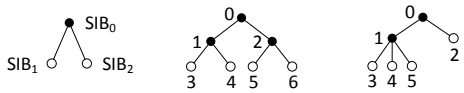


Figure 7.5. Alternative representation of networks, where filled circles represent the SIBs which are not directly connected to instruments, empty circles represent SIBs connected to instruments, and edges represent the hierarchical relations: (a) representation of network in Figure 7.2(b), (b) and (c) networks for four instruments

7.4.2. HEURISTICS

In the following, we present a network construction method that by mixing binary and ternary subtrees yields similar or better results compared with each of the pruned binary and ternary tree alternatives.

Let us now switch to a simpler network representation which is more suitable for the discussion in this section. The tree in Figure 7.5(a) captures the hierarchical relation (and not the data connections) between the SIB components in the network shown in Figure 7.2(b). The instruments are not shown (as the length of instruments' shift-registers has no effect on SIB shifting overhead) and those SIBs directly connected to instruments are represented by empty circles. In Figure 7.5(a), node SIB_0 is parent to sibling leaf nodes SIB_1 and SIB_2. When a parent SIB is opened, its children are on the scan path no matter if they are opened or closed. In other words, when a node is on the scan path, all its siblings are also on the scan path.

As the proposed network construction method is based on bundling instruments in groups of three, we would first like to make an observation for when the remaining number of instruments is one, i.e., when $N \bmod 3 = 1$. Figure 7.5(b) and Figure 7.5(c) show two networks constructed for four instruments. When in the network in Figure 7.5(b) a fault is detected at instrument connected to the SIB at node 3, that SIB (node 3) as well as the SIB at node 1 are opened. This means that in total five SIBs are on the path (namely, nodes 0–4) and it therefore takes five clock cycles to read their status. In this case, as all instruments have the same number of SIBs on their scan path, the average-case and the worst-case fault localization time is the same for all of them. This, however, is not the case for the network represented in Figure 7.5(c) in which for the instrument connected to node 2 three shift clocks are needed while for those connected to nodes 3–5 six shift clocks are needed—averaging to $(3 \times 6 + 1 \times 3)/4 = 5.25$ clock cycles. It can be seen from this example that

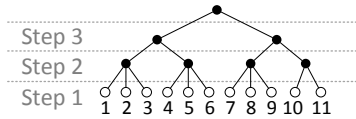


Figure 7.6. Representation of a self-reconfiguring network constructed for 11 instruments

the network represented by Figure 7.5(b) results in both better average-case and worst-case shifting time.

Based on the above observation, we propose the following construction algorithm. For given N instruments, we bundle the instruments into clusters of three instruments each. When N is a multiple of three, we will have $c = N/3$ clusters. If N is not a multiple of three, one or two instruments will remain. Following the observation made for Figure 7.5(b), when the number of remaining instruments is one, we make $c = \lfloor N/3 \rfloor - 1$ three-instrument clusters plus two two-instrument clusters. If, however, the number of remaining instruments is two, we make $c = \lfloor N/3 \rfloor$ three-instrument clusters plus one two-instrument cluster. Assuming each cluster to be an instrument, the same procedure described above can be applied to the created clusters, creating clusters of clusters until the network is complete. Figure 7.6 shows this procedure for 11 instruments. In the first step, as $N = 11$, we make $\lfloor 11/3 \rfloor = 3$ three-instrument clusters plus one two-instrument cluster. In the second step, as $N = 3 + 1 = 4$, we make $\lfloor 4/3 \rfloor - 1 = 0$ three-instrument clusters plus two two-instrument clusters. Finally, in the third step, as $N = 2$, we make $\lfloor 2/3 \rfloor = 0$ three-instrument clusters plus one two-instrument cluster.

7.5. FAULT MANAGER

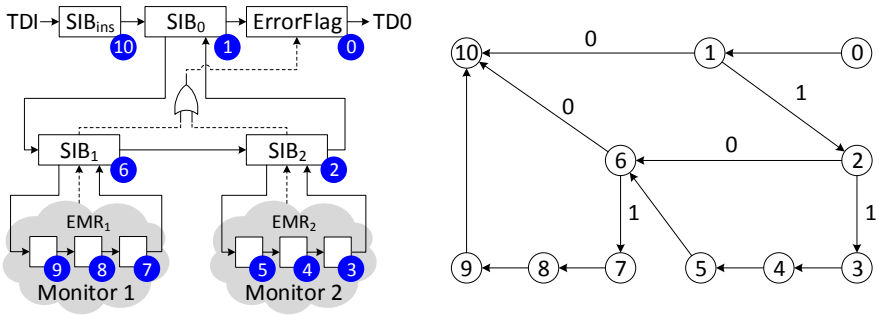
In this section, we elaborate on the tasks of the Fault Manager unit and propose one possible implementation.

To operate the on-chip monitoring instruments and take necessary actions upon detection of a fault, the Fault Manager unit should perform the following tasks:

1. activation of the monitoring instruments (after reset) by clearing their mask bits,

2. polling ErrorFlag and launching the localization process in case faults are reported in the monitored system,

3. analysis of the bit sequence shifted out from TDO during localization, to determine which monitoring instrument has raised the error flag and



(a) An self-reconfiguring network

(b) An FSM that identifies which bit is being shifted out next

Figure 7.7. Detecting the current network configuration based on the values being shifted out can be done by an FSM.

to store the reported error code,

4. taking necessary actions based on the error code reported by the monitoring instrument that has raised the flag, and

5. disabling a fault monitor that keeps raising the fault flag (either due to a permanent fault or due to that the monitor itself is defective) by setting its mask bit.

Except for the first item in the above list, the way each of these tasks is carried out affects the fault detection and localization time. This effect is particularly more dramatic for the third item above as that task might involve processing long sequences of hundreds of bits. If the analysis of the shifted out bit sequence (during localization) is done after the shifting is complete, the analysis time is added to the fault localization time, which can increase the localization time significantly. Moreover, for such post processing, the bit sequence should be stored first, which requires allocation of buffers of adequate length. If, on the other hand, the processing is done at the same time as the bit sequence is shifted out, the need for the buffer is obviated and the analysis can overlap in time with the shift-out. This, however, enforces certain constraints on the amount of time that processing of each bit can take maximum, otherwise shifted out data might be lost. Let us take a closer look at how this analysis can be done by taking the example network in Figure 7.2(b), presented again in Figure 7.7(a).

Assume that a fault is detected by a monitoring instrument, the network has performed self-reconfiguration accordingly, the Fault Manager has detected the fault by reading the ErrorFlag, and has subsequently opened SIB_0

Table 7.1. Storing the FSM in Figure 7.7(b) in memory as a state transition table

	Next state	
Current state	TDO = 0	TDO = 1
0 →	1	1
1 →	10	2
2 →	6	3
3 →	4	4
	⋮	

to start the localization process. The numbered circles next to the components in Figure 7.7(a) denote the order that those bits appear at TDO during the shift out (under the assumption that all components are part of the active scan path). The FSM in Figure 7.7(b) shows how by looking at the values shifted out, the Fault Manager can discover the current configuration of the network, identify the faulty resources, and collect the error codes. Values read at TDO, when corresponding to SIBs, are used to determine to which component the next bit in the sequence belongs. These values are therefore used to label the transitions in the FSM for the SIBs, that is, where more than one output transition from a given state exists. By using such an FSM, the Fault Manager can be guided during the localization process to detect the current configuration of the network and to collect the bits corresponding to each error code. For example, assuming that Monitor 2 has detected the fault, SIB_2 is automatically opened and the bits corresponding to EMR_2 are included in the scan path. In this case, upon reading a value of '1' for bit number 2 at TDO, the Fault Manager enters state 3 in the FSM. From this point, Fault Manager should start collecting bits 3–5 as the error code and store the collected code upon leaving state 5.

The FSM shown in Figure 7.7(b) is not complete as it does not capture the actions that should be taken for each fault that is localized. We will shortly elaborate on this issue. For the moment, we should note that a standard way of implementing an FSM in software is by using a state transition table. For the simple FSM in Figure 7.7(b), one such table looks like what is shown in Table 7.1. For such a table, the current state is just a pointer to a row (i.e., it is not stored in the table), and each row contains pointers to the next state based on the value observed at TDO. This way, the next state can be computed instantly for each observed bit at TDO.

In order for the Fault Manager to process the localization bit sequence, it should either be running on a faster clock (compared to TCK) so that it does not fall behind in case it needs to perform other tasks while performing the

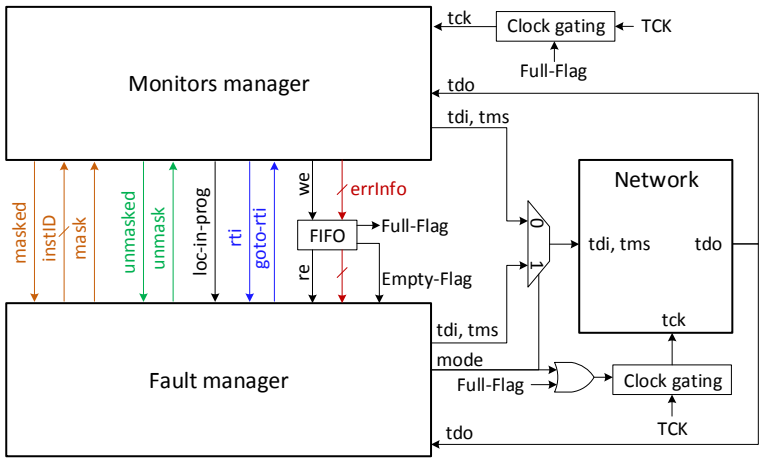


Figure 7.8. The interfaced between the proposed Monitors Manager, Fault Manager, and the network

localization, or allocate buffers for temporary storage of the shifted-out bit sequence. To avoid these two limitations, namely, the faster clock and buffer allocation, as well as to avoid storing a table such as Table 7.1, we propose and detail a hardware module that runs on same clock as the network (i.e., TCK) and performs all tasks related to the monitoring instruments. More specifically, the proposed module performs tasks 1,2,3, and 5 mentioned in the beginning of this section, and therefore, relieves the Fault Manager from having anything to do with the monitoring network, which is the self-reconfigurable network connected to SIB_0. We will refer to this hardware module as *Monitors Manager* and will show that its area in hardware is lower than the area of memory that a software based solution would require.

7.5.1. MONITORS MANAGER'S INTERFACE

Figure 7.8 shows how the proposed Monitors Manager is interfaced to the 1687 Network and the Fault Manager. The assumption is that the Fault Manager is implemented as software running on an on-chip microprocessor (or a micro-controller).

When the **mode** signal is set to 0, the Monitors Manager is connected to the TAP controller in the network. After the reset, the Monitors Manager module waits (while keeping the network's TAP controller state machine in the Test-Logic-Reset state) until it receives the **unmask** signal from the Fault Manager. It then starts opening the SIBs in the network level by level until the EMRs are part of the scan path. It will then clear all mask bits while

closing all the SIBs. This is rather straightforward as the proposed network construction method in Section 7.4 places all the EMRs at the same level, making it relatively easy to embed this unmasking feature into the Monitors Manager module. Monitors Manager signals the completion of the unmasking through asserting the **unmasked** signal.

After the initial unmasking, if the goto-rti signal is active, Monitors Manager keeps the network's TAP controller state machine in the Run-test/Idle state, otherwise it starts the fault detection and localization process. The purpose of goto-rti is to signal Monitors Manager to stop the fault detection and localization process and take the network's TAP controller state machine back to the Run-test/Idle state. This way, Fault Manager can take over (by switching the **mode** to 1) and access the other instruments in the network (i.e., those connected to SIB_{ins}, in order to take actions based on the detected errors).

When a fault is detected and localized by Monitors Manager, the instrument ID and the error code (i.e., contents of the corresponding EMR except for the mask bit) is pushed into the FIFO. Fault Manager is notified of existence of errors in the system by polling the **Empty-flag** of the FIFO. Alternatively, the **Empty-flag** can be interfaced as an interrupt signal. The **Full-flag** of the FIFO is used to *freeze* the operation of Monitors Manager in case Fault Manager has fallen behind in reading the error information from the FIFO. The freezing halts the clock to both Monitors Manager and the network, preventing the loss of the error information that is to be reported by Monitors Manager. This way, the errors that are detected and whose code is being shifted out stay in the scan path waiting to be shifted out, and new error flags will propagate and be detected in the next round of localization.

When a monitoring instrument keeps raising the error flag—either due to a permanent fault or due to that the monitoring circuitry itself is defective—the Fault Manager can mask that instrument by placing its ID number on **instID** and asserting the **mask** signal. As will be detailed shortly, this masking is only possible when the requested instrument keeps raising the error flag and is thus part of the active scan path after self-reconfiguration. Once Monitors Manager has set the mask bit in the EMR corresponding to the specified instrument, the **masked** signal is asserted.

Finally, the Monitors Manager module asserts the **loc-in-prog** signal whenever it detects a fault and starts the localization process. This signal can help Fault Manager in certain cases. For example, if a masking request is not acknowledged and localization is not in progress either, it means that the instrument that Fault Manager is trying to mask has cleared its error flag before being masked (i.e., the detected error has not been permanent).

7.5.2. INTERNAL OPERATION OF MONITORS MANAGER

Internally, the Monitors Manager module is a state machine. In our experiments, we constructed this module automatically based on the description of the self-reconfigurable part of the network. In this section, we explain how such a state machine performs the fault localization as well as the masking/unmasking tasks.

7.5.2.1. PERFORMING THE INITIAL UNMASKING

As was mentioned earlier, the network design method in Section 7.4 constructs the network such that all instruments are placed in the same depth (hierarchical level) in the tree. Based on this, the following is done to perform the unmasking:

- one CSU is applied to open each hierarchical level. For each CSU, a counter is loaded with the number of SIBs currently on the scan path (i.e., total number of SIBs on all currently opened hierarchical levels) plus one for the ErrorFlag. This counter is decremented with every clock cycles while a '1' is shifted in from **tdi**, until the counter reaches zero. Then a '0' is shifted in for the SIB_{ins} to keep it closed, followed by an update.

- once all hierarchical levels are opened in the previous step, one final CSU is needed for clearing the mask bits and closing the SIBs. This time, the counter is loaded with the total length of the scan path and is decremented with every clock cycles while a '0' is shifted in from **tdi**, until the counter reaches zero. After this, an update is performed and the network's TAP controller state machine is taken back to the Run-Test/Idle state.

Implementing the above steps as a state machine is pretty straightforward and we skip detailing it further.

7.5.2.2. PERFORMING THE LOCALIZATION AND FAULT MASKING

Before delving into the details in this section, we should mention that for the sake of simplicity in presentation, we disregard the mandatory half-cycle tristate delay element that is present at TDO [29]. In our implementation, we have taken that delay into account.

The Monitors Manager starts the localization process after detecting that the ErrorFlag is set to one. If it is detected that the bit corresponding to ErrorFlag is '1', in the same CSU, a '1' is shifted in to open SIB_0. As it is rather straightforward, we skip detailing the detection part of the FSM in Monitors

Table 7.2. The localization state transition table for the network in Figure 7.7(a)

Current state	Next state		Output signal assignments			Other actions
	tdo==0	tdo==1	mask==0	mask==1 && instID==1	mask==1 && instID==2	
0 →	1	1	tdi=0; tms=0; masked=0;			
1 →	10	2				
2 →	6	3				
3 →	4	4				EMR=tdo;
4 →	5	5				EMR=(EMR<<1)|tdo;
5 →	6	6	errInfo=(2<<2)|EMR; we=1;	errInfo=(2<<2)|EMR; we=1;	tdi=1; masked=1;	
6 →	10	7	tdi=0; we=0;			
7 →	8	8				EMR=tdo;
8 →	9	9				EMR=(EMR<<1)|tdo;
9 →	10	10	errInfo=(1<<2)|EMR; we=1;	tdi=1; masked=1;	errInfo=(1<<2)|EMR; we=1;	

Manager. We, however, detail how our proposed hardware implementation performs localization, while carrying out the instrument masking requests.

One important assumption behind our implementation is that if a monitoring instrument is requested to be masked, it should be already part of the self-reconfigured active scan path. That is, it should have raised the error flag. This assumption is justified by noting that if an instrument has raised the error flag before but not in the current localization round, the corresponding fault has not been a permanent one in the first place.

We use the state transition table presented as Table 7.2 to explain how the FSM in Monitors Manager performs the localization and masking for the example network in Figure 7.7(a). In practice, we have not used such a table in our implementation and have directly implemented the FSM in a hardware description language. The table, however, makes it easier to explain the localization process, and helps in getting a more realistic view of the memory usage for a software implementation of the Monitors Manager module. In the table, we have used the C language expressions to explain the low-level hardware operations.

In Table 7.2, the current state of the FSM is a pointer to a row in the state transition table. The next state is determined solely by the value observed at the **tdo** terminal. For example, in state 1, if **tdo** value is zero, the next state will be 10 otherwise 2. The output signals, on the other hand, are determined by the values present at the **mask** and **instID** input terminals. Once an output signal is assigned it will retain its value until next time it is assigned a new value. That is, empty cells in Table 7.2 are in fact repetitions of the closest non-empty cell above them, and are left empty to reduce clutter. At the beginning of the localization process (namely, state 0) the **tdi**, **tms**, and **masked** outputs are all set to zero.

As an example, assume that both instruments in the network in Figure 7.7(a) have raised the error flag, but Fault Manager has requested Monitor 2 to be

fault masked. That is, **mask**==1 and **instID**==2. In this case, once the FSM is in state 3, it starts buffering the bits corresponding to the EMR for Monitor 2—regardless of the **mask** and **instID** inputs—and it is only in state 5 when masking conditions are tested. If the monitoring instrument is not to be masked, its ID and error code are concatenated to form the error information and are pushed into the FIFO via the FIFO input terminal **errInfo** and by asserting the write-enable (**we**) signal. Regarding the concatenation expression errInfo=(2<<2)|EMR;, the first 2 is the instrument ID for the current instrument and the second 2 is a two-bit left shift. The reason for the two-bit left shift is that in our example, the EMR has three bits, one of which is the mask bit, which is not needed to be included in the error information. If, on the other hand, the instrument is to be masked, a '1' is placed on the **tdi** output (which is connected to the *tdi* input of the network) to be shifted in for the mask bit, and the **masked** output is set to '1'. In state 6, the **tdi** and **we** outputs are set back to '0'. As Monitor 1 has also raised the error flag, the **tdo** input will have the value of '1' and the next state is set to 7. In state 7, similar to state 3 discussed above, the Monitors Manager starts to buffer the error code and finally in state 9, the collected error code is pushed into the FIFO.

7.6. COMPARISON WITH SIMILAR APPROACHES

We have compared our proposed self-reconfiguring 1687 network with the work presented in [1] (which uses a regular 1687 network for monitoring) with regards to t_{worst} (see Eq. (7.1)). In this section, we present the results of the comparison for two cases: when one fault occurs (discussed in Section 7.3.1), and when multiple faults are detected by the Fault Manager at the same time (discussed in Section 7.3.2).

7.6.1. FOR A SINGLE FAULT

For the construction of the proposed self-reconfiguring network, we compared four alternatives: (1) using the network construction method presented in Section 4.1.2, which was for regular (i.e., non-self-reconfiguring) SIB-based 1687 networks optimized for sequential access schedules (denoted by HPO), (2) a binary tree with pruning, (3) a ternary tree with pruning, and (4) the construction method proposed in Section 7.4. To calculate the number of SIBs on the scan path for the self-reconfiguring networks, pre-order tree traversal is employed. A fixed number of $J_{OH} + 2 = 18 + 2$ TCKs is added to the calculated shift time to account for the constant overhead (see Section 7.2.2), ErrorFlag, and SIB_{ins}. Moreover, another three TCKs are added to account for the length of the fault monitor's EMR (i.e., $L = 3$). We have chosen $L = 3$ for a fair comparison with the work in [1].

Table 7.3. t_{worst} for a single fault (in TCKs)

Number of instruments	[1]	Self-reconfigurable networks			
		HPO	binary tree	ternary tree	proposed method
25	90	35	34	33	33
50	118	37	36	35	35
100	158	39	38	38	37
200	206	42	40	39	39
500	266	52	42	42	42
1000	326	53	44	44	44

Table 7.3 shows the results of comparison with the approach proposed in [1]. From the results, it can be seen that by using the proposed self-reconfiguration scheme (regardless of the considered network tree construction method), at least 2.6x reduction in localization time is achieved compared to [1]. The reason for this improvement can be attributed to opening many hierarchical levels in a single CSU and having only one single-bit ErrorFlag register directly on the scan path.

Among the construction methods examined for the self-reconfiguring network, the one described in Section 7.4 performs up to 17% better than the method in Section 4.1.2, and results in better or equal t_{worst} compared to binary and ternary trees.

7.6.2. FOR MULTIPLE FAULTS

The work in [1] has not presented analysis and results on multiple faults. On the other hand, our calculations for multiple faults (see Section 7.3.2) are for balanced *k*-ary trees, and cannot be directly used for the network structures presented in [1]. Therefore, to perform the comparison, we used constraint programming (by using the constraints formulation in [1]) to get the optimal network architecture for the number of instruments suitable for our analysis (see below), and developed time analysis for multiple faults for the networks presented in [1] based on their time analysis for a single fault and our analysis for multiple faults. In developing the time analysis, whenever the optimal architecture, computed for N instruments by the constraint programming solver, allowed for more monitoring instruments than those actually requested, say $N' > N$, we assume that the network has N' instruments.

For the number of instruments, we chose numbers which are powers of three resulting in networks resembling balanced ternary trees, as the presented time analysis applies to banalced trees only. For each of these networks, we calculated t_{worst} for one to 10 faults. For each pair of network and number of faults, we calculated t_{worst} using Eq. (7.2) where $J_{OH} = 18$ and t_s

Table 7.4. t_{worst} for multiple faults (in TCKs)

# instruments	Number of faults									
	1	2	3	4	5	6	7	8	9	10
27	33	42	51	57	63	69	75	81	87	90
	90	126	162	162	162	162	162	162	162	162
81	36	48	60	69	78	87	96	105	114	120
	146	202	258	314	370	426	426	426	426	426
243	39	54	69	81	93	105	117	129	141	150
	218	346	474	542	610	678	746	814	882	950
729	42	60	78	93	108	123	138	153	168	180
	298	486	674	862	954	1046	1138	1230	1322	1414
2187	45	66	87	105	123	141	159	177	195	210
	394	662	930	1118	1306	1494	1682	1870	2058	2246
Average ratio	5.6	6.3	6.7	6.8	6.7	6.6	6.4	6.2	6.1	6.0

The shaded numbers are calculated based on the approach in [1].

is calculated using Eq. (7.8). The results are presented in Table 7.4 where the shaded rows present the numbers obtained for the network structure type proposed in [1]. The last row, presents the average improvement ratio achieved over [1] in case of multiple concurrent faults which ranges between 5.6 to 6.8 times improvement. As was the case for single faults, the reason for this improvement can be attributed to opening many hierarchical levels at once and having only one ErrorFlag register directly on the scan path.

7.7. PRACTICAL ISSUES

To validate our proposed self-reconfiguring networks and also give an idea of the hardware overhead associated with the Monitors Manager, we implemented such networks for the number of instruments presented in Table 7.4, and performed synthesis and place & route (optimized for a 100MHz TCK) using 65nm technology. The target cell density was chosen as 70 percent, which was achieved for the target 100MHz clock frequency. Through implementation and post-layout simulations we established the practicality of the proposed self-reconfigurable networks and the Monitors Manager. In this section, we discuss the implementation of the *modified* SIB, report and discuss the hardware overhead associated with the Monitors Manager module, and report the measured delay for the error flag propagation network.

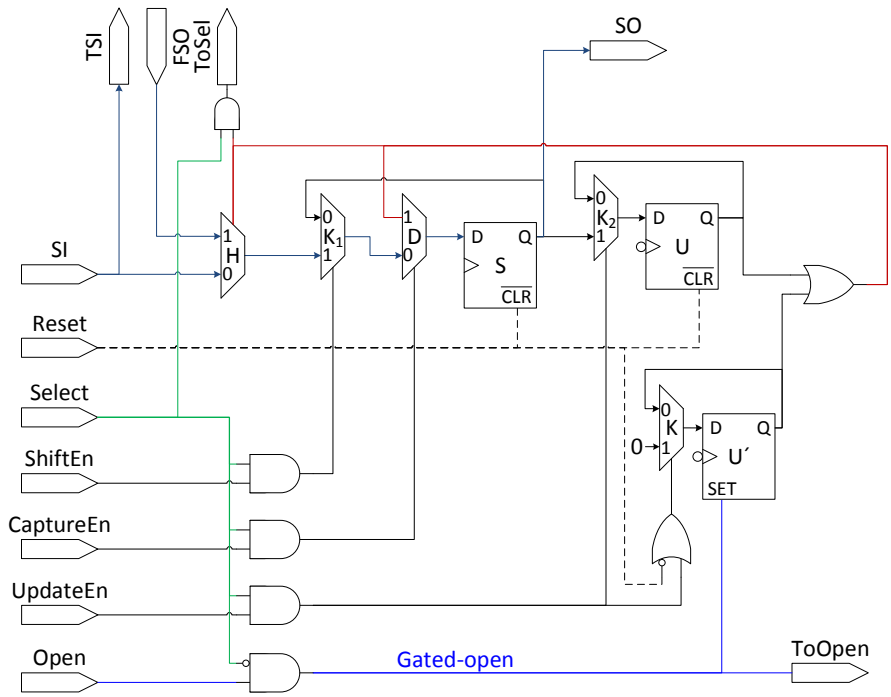


Figure 7.9. Schematic of the proposed *modified* SIB

7.7.1. *MODIFIED* SIB

Figure 7.9 shows our implementation of the proposed *modified* SIB. Before discussing Figure 7.9, we should mention that depending on the available standard cell library, simpler designs with the same functionality might be possible, and that our implementation is affected by our ASIC vendor's library.

In Figure 7.9, the clock signal is not shown to avoid clutter. The **Reset** signal is the synchronous active-low reset from Test-Logic-Reset state in the TAP controller state machine. The self-reconfigurability revolves around the U′ flip-flop, which is D-type with asynchronous active-high **set**. The **set** input of U′ is connected to a gated copy of the **Open** signal of the SIB. The **Open** signal is gated via the **Select** signal so that the self-reconfiguration only happens when the SIB is not selected (i.e., not part of the active scan path). The **Q** output of U′ is used to open the SIB—i.e., to include the segment connected between **TSI** and **FSO** terminals in the scan path. As is required by the localization method described in Section 7.2.2, the output of U′ is captured into the S flip-flop when the TAP controller state machine goes through the capture

phase. U′ is cleared when the TAP controller state machine goes through the **Update** phase or through the Test-Logic-Reset state during initialization.

The area increase due to the extra components in the *modified* SIB as compared with a regular SIB with and without the diagnostics mux (i.e., mux D in Figure B.3) is 49 percent and 75 percent, respectively.

7.7.2. MONITORS MANAGER

In the post-layout simulations, we performed the following:

- Instructed the Monitors Manager to perform the initial unmasking,
- inserted faults into the system by raising fault flags associated with some of the monitors, and observed that the Monitors Manager correctly identifies the IDs of the associated instruments and inserts the right ID and error code into the FIFO, and
- inserted a permanent fault into the system by constantly raising a fault flag, and instructed the Monitors Manager to mask the corresponding monitor.

To justify the hardware overhead associated with the Monitors Manager and its associated circuitry such as FIFO, clock gating logic, etc., we report the standard cell area occupied by these modules, and make a rough comparison with the SRAM area required for a "partial" software implementation of the Monitors Manager module. The partial implementation only performs the state transitions in the FSM for the localization task, without performing any actions (such as writing the error code in another memory location or performing specific tasks depending on the observed error code). Even by comparing against a partial software implementation, we demonstrate that for larger networks, the area taken by a hardware implementation of the Monitors Manager module, is lower than the area taken by the SRAM bit cells that are required to implement this module in software.

Table 7.5 presents the hardware area, as well as SRAM area estimation for a software implementation of Monitors Manager, for the same networks used in Section 7.6.2 (Table 7.4). In Table 7.5, the second and the third columns present the standard cell area taken by the 1687 network and the Monitors Manager, respectively, in square micrometers. Columns four to eight present how we have estimated the equivalent SRAM area required for a "partial" software implementation of the Monitors Manager. This estimation is based on assuming a state transition table similar to the one presented in Table 7.1. Column four shows the number of states in the localization state machine. Column five shows the number of memory locations required for the storage of the state machine (two locations per state). Column six presents the

Table 7.5. Hardware area and estimates for SRAM area that would be used by the software-based approach

Number of instruments	Hardware area in μm^2		SRAM memory required for software-based localization				
	1687 network	Monitors Manager	number of states	memory locations	Bits per location	Total bits	Total area
27	8779	4511	123	246	8	1968	982
81	26639	6691	366	732	16	11712	5844
243	77390	12646	1095	2190	16	35040	17485
729	231541	29793	3282	6564	16	105024	52407
2187	703424	81175	9843	19686	16	314976	157173

number of bits required per memory location. When the number of memory locations are less than 256, we considered that eight-bit memory cells can be used, otherwise 16-bit cells. Column seven shows the total number of bits, and column eight presents the total area, assuming 0.499 μm^2 per cell [61]. In this computation of the total area, we have only considered the area taken by the 6T SRAM bit-cells required for the localization process, and have disregarded the impact of these additional cells on the area taken by the decoding circuitry, sense amplifiers, etc. Comparison of the area reported in the third and the eighth columns shows that the area taken by a hardware implementation of the Monitors Manager is less than the partial software implementation for 243 and higher number of instruments, and very close to the partial implementation for 81 instruments. It is only for the case of 27 monitoring instruments that a hardware implementation shows higher overhead compared to its partially implemented software counterpart. It should be noted that a full software implementation of all tasks performed by Monitors Manager will result in higher SRAM area.

7.7.3. PROPAGATION DELAYS

In Section 7.2.2, we noted that the delay in the error flag propagation network (denoted by δ) should in practice be considered in d_{worst}. To give an idea about how large that delay might be, we report it for the largest design with 2187 instruments. The delay between each of the 2187 instrument fault flags and the parallel input of the ErrorFlag register (i.e., through all seven hierarchical levels) is reported by the place & route tool to be at least 1.73ns, on average 2.1ns, and at most 2.62ns, thus shorter than one TCK period, which is 10ns for the 100MHz target.

As an example, assuming an on-chip Monitors Manager that can operate the network at 100MHz, the worst-case localization time (in seconds) for the case of one fault happening in a network with 2187 instruments (see Table 7.4) is calculated as $45 \times \frac{1}{100 \times 10^6} + 2.62 \times 10^{-9}$ which is 452.62ns.

7.8. CHAPTER CONCLUSIONS

In this chapter, we showed how fault localization time can be reduced by using a segment insertion bit (SIB) that enables self-reconfiguration of 1687 networks. We presented timing analysis for single and multiple concurrent faults, as well as a construction method for the proposed self-reconfiguring network. Moreover, we detailed a hardware module for performing the localization, which discovers the configuration of the network after self-reconfiguration and extracts the error information in real time. We validated the idea of self-reconfiguring networks through post-layout simulations of a number of such networks. We compared the proposed scheme with a previous similar work and observed at least 2.6 times reduction in localization time for a single fault and 5.6 times reduction in case of multiple faults.

8

Conclusions and Future Work

In this chapter, we present a summary for each of the contributions of the work, as well as future research directions.

8.1. THESIS CONCLUSIONS

In this section, for each of the four parts in this thesis, we recapitulate the main contributions and observations.

8.1.1. ANALYSIS

In this thesis, we presented time analysis and overall access time (OAT) calculation algorithms for three 1687 network types, which we referred to as SIB-based, Daisy-chained, and Remote networks. The algorithms covered concurrent, sequential, and generic instrument access schedules. The analysis showed that OAT has three main components: instrument data, shift overhead, and TAP overhead. Using the OAT calculation algorithms, we presented a parametric analysis to identify possibilities for reduction of OAT. An important observation was that the use of hierarchical architectures and the concurrency in the access schedule have no influence on the instrument data, but can vary the overhead components. More specifically, For the SIB-based and Daisy-chained network types, it was observed that the TAP overhead was affected significantly by the change in the concurrency in the access schedule, but not considerably with the network architecture. On the other hand, the shift overhead was affected significantly with both access schedule and the network architecture. For the Remote networks, the use of pipelining of instrument data was shown to help reduce the overhead significantly. In case of sequential schedule, pipelining showed to be very effective in reducing the

shift overhead. For the concurrent and generic schedules, the reductions in shift overhead were dependent on the order of instruments on the scan path. The observations from the time analysis were used to devise network optimization methods.

8.1.2. DESIGN

Based on the observations from the time analysis, we proposed methods for designing 1687 networks that are optimized with respect to overall access time both for one given access schedule, and multiple given access schedules. Here again, the considered network types were SIB-based, Daisy-chained, and Remote networks. The basic idea behind optimization for the SIB-based and Daisy-chained networks was the use of hierarchy to reduce the shift overhead. For the Remote networks, however, the idea was to reorder the instruments on the scan path, such that less time is wasted in the bypass flip-flops. The experimental results showed the optimization methods to be highly effective in reducing the OAT. A comparison between different network types showed that when there is the possibility of pipelining instrument data (for example, when the retargeting tools support it) the Remote network type seems the best choice given its low hardware overhead, the relatively low OAT observed for it in the experiments, as well as its predictability w.r.t. changes in overhead percentage when subjected to new schedules. The hierarchical SIB-based and Daisy-chained networks also showed relatively low hardware overhead, as well as reasonable performance regarding OAT.

8.1.3. OPERATION

Operating the 1687 networks requires sophisticated design automation tools. One of the most essential tasks for such tools is to perform the retargeting: translation of human-readable access procedures described at instrument terminals into bit vectors (or other description languages) applicable at the chip terminals. In this thesis, we outlined the steps in the retargeting process, namely, flattening, merging, and translation, and discussed optimization potentials. As a key operation in the translation is a retargeting step, we presented a method for reducing the solution space to help in performing the retargeting step optimally.

8.1.4. APPLICATION

The 1687 networks find application in testing, debugging, monitoring, and so. In this thesis, we focused on the fault monitoring application and proposed to add self-reconfiguration to 1687 networks in order to reduce the fault localization time. For self-reconfiguring networks, we presented time analysis for single and multiple concurrent faults, as well as a method for constructing

these networks, given an arbitrary number of instruments, such that the localization time is minimized. Comparison with fault monitoring approaches based on 1687 networks showed that self-reconfiguring networks can achieve significantly lower fault localization time. Finally, we detailed the practical issues in implementation and operation of these networks.

8.2. FUTURE WORK

At the time of writing of this thesis, there is ongoing work to use interfaces other than TAP to connect to the on-chip instrument access networks. A new interface might bring new challenges for which there would be a need to revisit the problems addressed in this thesis.

Regarding the specific topics covered in this thesis, the following could be directions for future work.

8.2.1. ANALYSIS AND DESIGN

In the work presented in this thesis, it was either assumed that the network is given with the objective being optimized retargeting, or that the schedule is given with the objective being optimized network design. As future work, a co-optimization problem can be considered such that for a given set of instruments, where for each instrument the PDL procedures are given, a network is designed and retargeting performed such that application time of the retargeted PDL is minimized.

Also, in our work on optimized network design, we did not investigate possible combinations of the considered network types (namely, SIB-based, Daisy-chained, and Remote networks).

Moreover, in our work, we did not consider the use of hierarchy in Remote networks. Given the relatively good performance of this network type w.r.t. OAT and hardware overhead, it could be beneficial to consider the use of hierarchy in this network type, as well. This requires development of OAT calculation method for this new architecture.

8.2.2. OPERATION

As was mentioned in Section 6.4.1.1, there might be cases where the computed upper-bound is pessimistic. That is, the computed value is higher than what is actually needed for optimal retargeting. These cases should be investigated and addressed in our proposed upper-bound computation method. Additionally, the proposed upper-bound computation method can be further developed to recognize more structures for lookup and rewriting. Finally, in computing the upper-bound in this work, we made no assumptions on the initial and the target configurations. The benefit of this relaxation is that the

upper-bound computation needs to be done only once at the beginning of the retargeting process. The resulting upper-bound can then be used for all retargeting steps in that retargeting process. On the other hand, if the initial and target configurations are considered in the computation of the upper-bound, the computation should be performed once for each retargeting step. In this case, the result will be a tighter bound tailored to that step, which increases the retargeting efficiency. Therefore, the trade-off between (1) saving time by running the upper-bound computation once at the beginning of the retargeting process, and (2) saving time by faster retargeting steps should be investigated.

Finally, as both IEEE Std 1687 and IEEE Std 1149.1-2013 support the use of broadcast, it is necessary to consider this feature in the process of retargeting.

8.2.3. APPLICATION

For the self-reconfiguring networks presented in Chapter 7, further work could be done to enable faster interruption of an ongoing localization process, in order to take necessary action in case there are errors that should be addressed urgently by the Fault Manager unit.

Part V

Appendix

Appendix A

Additional Graphs from the Parametric Analysis

The results for the parametric analysis, presented in Section 3.5, were plotted as separate charts for the sequential and concurrent schedules. In this appendix, we provide the results as separate tables per analysis, where each table details the OAT and its components, namely, instrument data, TAP overhead, and shift overhead, for the sequential and concurrent schedules.

Moreover, to facilitate side by side comparison of OAT components between the sequential and concurrent schedules, in this appendix, we present new plots in which the results are presented as stacked bars. Each stacked bar presents OAT as a sum of its three components on a logarithmic y-axis.

A.1. INCREASING THE NUMBER OF INSTRUMENTS

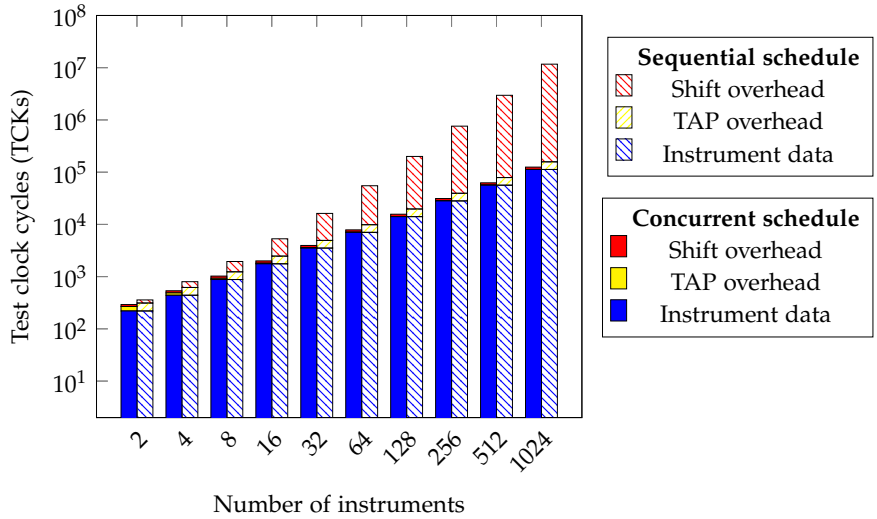


Figure A.1. The effect of increase in number of instruments on OAT and its components, in SIB-based networks

Table A.1. The effect of increase in number of instruments on OAT and its components, in SIB-based networks

Number of instruments	Instrument data	Sequential schedule			Concurrent schedule		
		TAP overhead	Shift overhead	OAT	TAP overhead	Shift overhead	OAT
2	220	92	46	358	48	24	292
4	440	180	180	800	48	48	536
8	880	356	712	1948	48	96	1024
16	1760	708	2832	5300	48	192	2000
32	3520	1412	11296	16228	48	384	3952
64	7040	2820	45120	54980	48	768	7856
128	14080	5636	180352	200068	48	1536	15664
256	28160	11268	721152	760580	48	3072	31280
512	56320	22532	2884096	2962948	48	6144	62512
1024	112640	45060	11535360	11693060	48	12288	124976

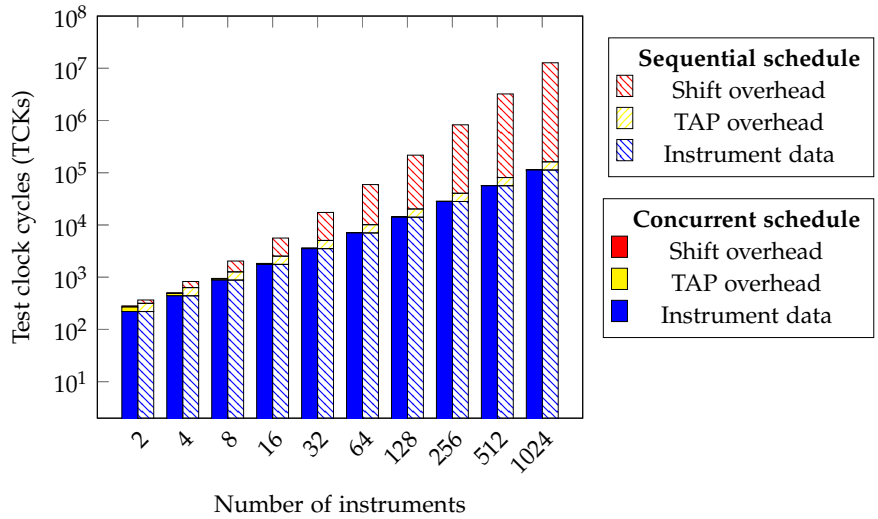


Figure A.2. The effect of increase in number of instruments on OAT and its components, in Daisy-chained networks

Table A.2. The effect of increase in number of instruments on OAT and its components, in Daisy-chained networks

Number of instruments	Instrument data	Sequential schedule			Concurrent schedule		
		TAP overhead	Shift overhead	OAT	TAP overhead	Shift overhead	OAT
2	220	96	50	366	48	14	282
4	440	192	196	828	48	16	504
8	880	384	776	2040	48	20	948
16	1760	768	3088	5616	48	28	1836
32	3520	1536	12320	17376	48	44	3612
64	7040	3072	49216	59328	48	76	7164
128	14080	6144	196736	216960	48	140	14268
256	28160	12288	786688	827136	48	268	28476
512	56320	24576	3146240	3227136	48	524	56892
1024	112640	49152	12583936	12745728	48	1036	113724

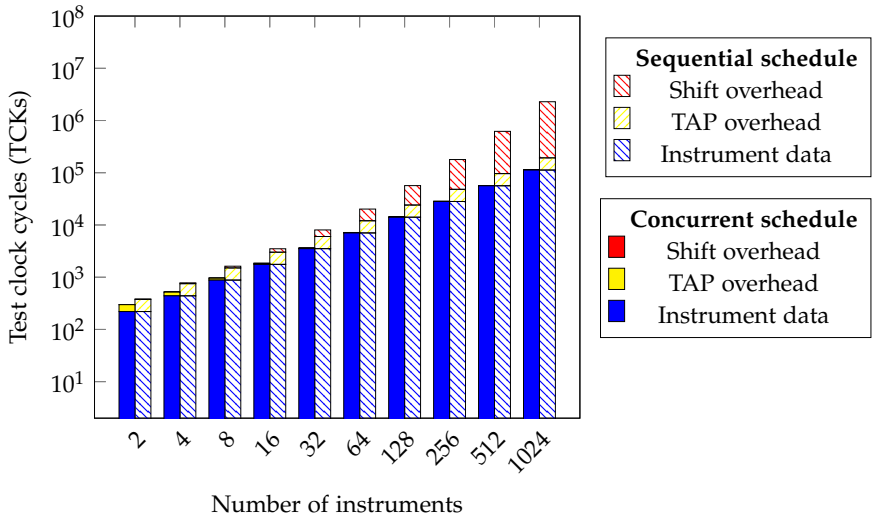


Figure A.3. The effect of increase in number of instruments on OAT and its components, in Remote networks

Table A.3. The effect of increase in number of instruments on OAT and its components, in Remote networks

Number of instruments	Instrument data	Sequential schedule			Concurrent schedule		
		TAP overhead	Shift overhead	OAT	TAP overhead	Shift overhead	OAT
2	220	156	6	382	78	2	300
4	440	312	28	780	78	4	522
8	880	624	120	1624	78	8	966
16	1760	1248	496	3504	78	16	1854
32	3520	2496	2016	8032	78	32	3630
64	7040	4992	8128	20160	78	64	7182
128	14080	9984	32640	56704	78	128	14286
256	28160	19968	130816	178944	78	256	28494
512	56320	39936	523776	620032	78	512	56910
1024	112640	79872	2096128	2288640	78	1024	113742

A.2. INCREASING THE NUMBER OF HIERARCHICAL LEVELS

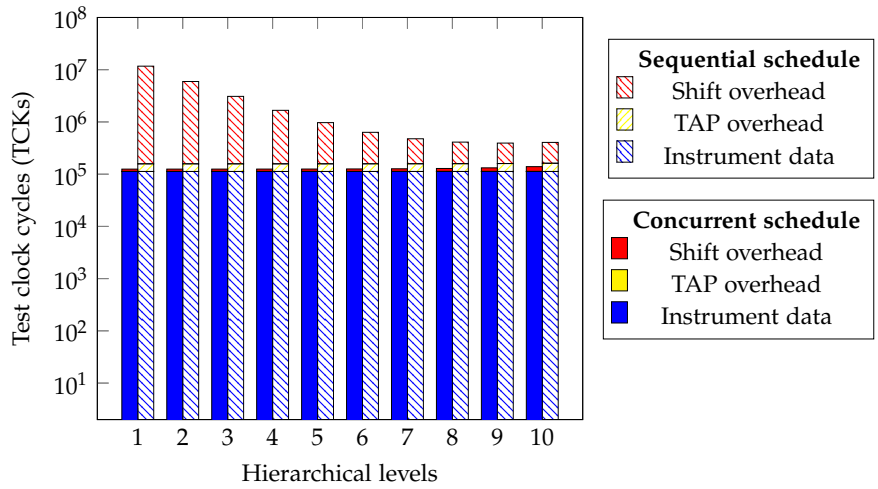


Figure A.4. The effect of increase in hierarchical levels on OAT and its components, in SIB-based networks

Table A.4. The effect of increase in hierarchical levels on OAT and its components, in SIB-based networks

Hierarchical Levels	Instrument data	Sequential schedule			Concurrent schedule		
		TAP overhead	Shift overhead	OAT	TAP overhead	Shift overhead	OAT
1	112640	45060	11535360	11693060	48	12288	124976
2	112640	45068	5790726	5948434	52	12314	125006
3	112640	45084	2929690	3087414	56	12368	125064
4	112640	45116	1510482	1668238	60	12478	125178
5	112640	45180	812258	970078	64	12700	125404
6	112640	45308	474690	632638	68	13146	125854
7	112640	45564	317826	476030	72	14040	126752
8	112640	46076	252162	410878	76	15830	128546
9	112640	47100	233986	393726	80	19412	132132
10	112640	49148	243714	405502	84	26578	139302

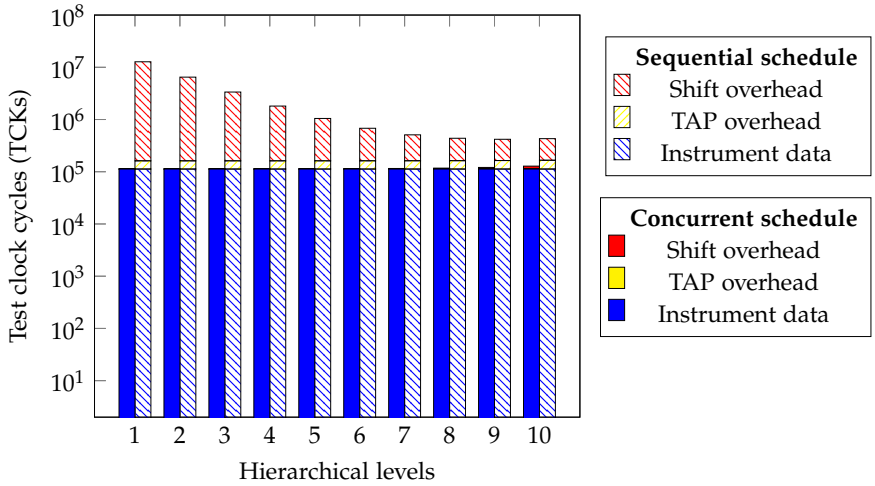


Figure A.5. The effect of increase in hierarchical levels on OAT and its components, in Daisy-chained networks

Table A.5. The effect of increase in hierarchical levels on OAT and its components, in Daisy-chained networks

Hierarchical Levels	Instrument data	Sequential schedule			Concurrent schedule		
		TAP overhead	Shift overhead	OAT	TAP overhead	Shift overhead	OAT
1	112640	49152	12583936	12745728	48	1036	113724
2	112640	49160	6317062	6478862	52	1063	113755
3	112640	49176	3195930	3357746	56	1118	113814
4	112640	49208	1647698	1809546	60	1229	113929
5	112640	49272	885986	1047898	64	1452	114156
6	112640	49400	517698	679738	68	1899	114607
7	112640	49656	346498	508794	72	2794	115506
8	112640	50168	274690	437498	76	4585	117301
9	112640	51192	254466	418298	80	8168	120888
10	112640	53240	264194	430074	84	15335	128059

A.3. VARYING THE INSTRUMENT PROPERTIES

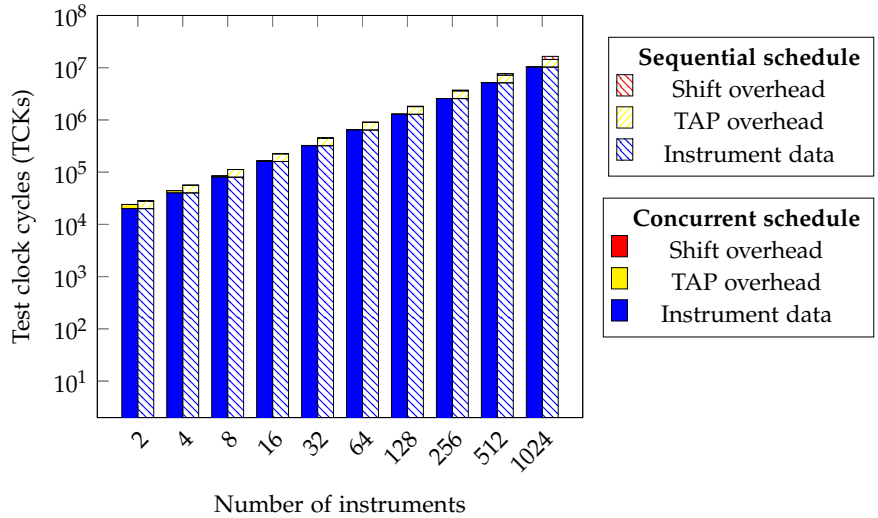


Figure A.6. The effect of increase in number of instruments having large number of accesses, on OAT and its components, in Remote networks

Table A.6. The effect of increase in number of instruments having large number of accesses, on OAT and its components, in Remote networks

Number of instruments	Instrument data	Sequential schedule			Concurrent schedule		
		TAP overhead	Shift overhead	OAT	TAP overhead	Shift overhead	OAT
2	20020	8076	6	28102	4038	2	24060
4	40040	16152	28	56220	4038	4	44082
8	80080	32304	120	112504	4038	8	84126
16	160160	64608	496	225264	4038	16	164214
32	320320	129216	2016	451552	4038	32	324390
64	640640	258432	8128	907200	4038	64	644742
128	1281280	516864	32640	1830784	4038	128	1285446
256	2562560	1033728	130816	3727104	4038	256	2566854
512	5125120	2067456	523776	7716352	4038	512	5129670
1024	10250240	4134912	2096128	16481280	4038	1024	10255302

Appendix B

Detailed Circuit Schematics

In this appendix, the RTL circuitry for a number of components discussed throughout this thesis is presented.

B.1. 1149.1-STYLE TDR

Figure B.1 shows the circuitry for a typical TDR cell. In shift mode, i.e., when the ShiftEn is set to logic '1', serial data is shifted in on the rising edge of the TCK clock, through the SI terminal, passes through mux K_1, mux C, and the S/C flip-flop, and finally is shifted out from the SO terminal. In the parallel load mode, the CaptureEn signal is set to '1' and the data present at the PI terminal is captured into the S/C flip-flop on the rising edge of TCK. To latch the contents of the cell to appear at the PO terminal, the UpdateEn is set to '1' and on the falling edge of the clock, the value shifted into the S/C flip-flop is copied into the U flip-flop.

Figure B.2 shows how two of these TDR cells are used to form a two-bit TDR. The globally routed ShiftEn, CaptureEn, and UpdateEn signals are gated via the Select signal. The scan path is formed via connecting the TDR cells SO to SI, and the parallel (i.e., the PI and PO) terminals are formed by connecting each bit of these signals directly to the PI and PO terminals of the corresponding TDR cell.

B.2. SIB

Figure B.3 shows a possible implementation of a SIB, which matches the simplified schematic presented in Figure 2.7(a).

The control signals (namely, ShiftEn, CaptureEn, and UpdateEn) are gated

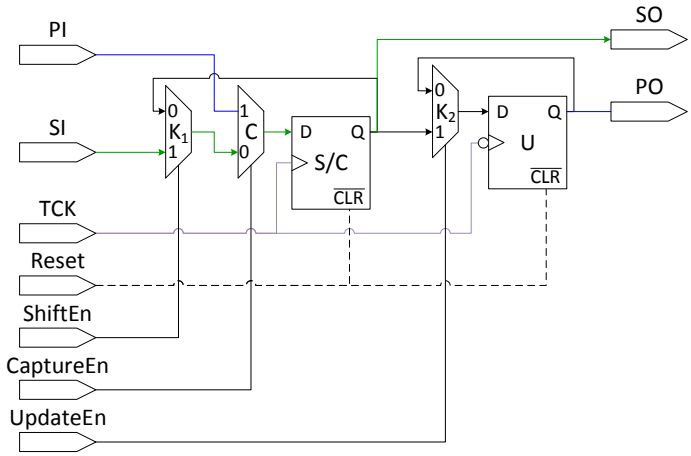


Figure B.1. TDR cell

by the Select signal. The S flip-flop operates at the rising edge of the SCK clock. When not in the shift mode (i.e., when the ShiftEn is set to '0'), it retains its currently stored value via feedback through the keeper mux K_1. In the shift mode (i.e., when the ShiftEn is set to '1'), new values are shifted in the S flip-flop through the K_1 and D muxes. Mux D is not required for the operation of the SIB and is only used for diagnostic purposes, as it captures the current status of the SIB (i.e., the value stored in the U flop) to be shifted out. The U flip-flop operates at the falling edge of the SCK clock. When not in the update mode (i.e., when the UpdateEn is set to '0'), it retains its currently stored value via feedback through the keeper mux K_2. In the update mode (i.e., when the UpdateEn is set to '1'), the U flip-flop gets the value currently stored at the S flip-flop. When the U flip-flop stores a '0', input 0 of the host port mux H is selected and, therefore, in the shift mode, serial data goes directly from the input terminal SI through the SIB, excluding the network segment connected to the host port terminals, namely, TSI and FSO terminals. When the U flip-flop stores a '1', input 1 of the host port mux H is selected, the network segment is selected via the ToSel terminal, and serial data goes through the segment connected between the TSI and TSO terminals.

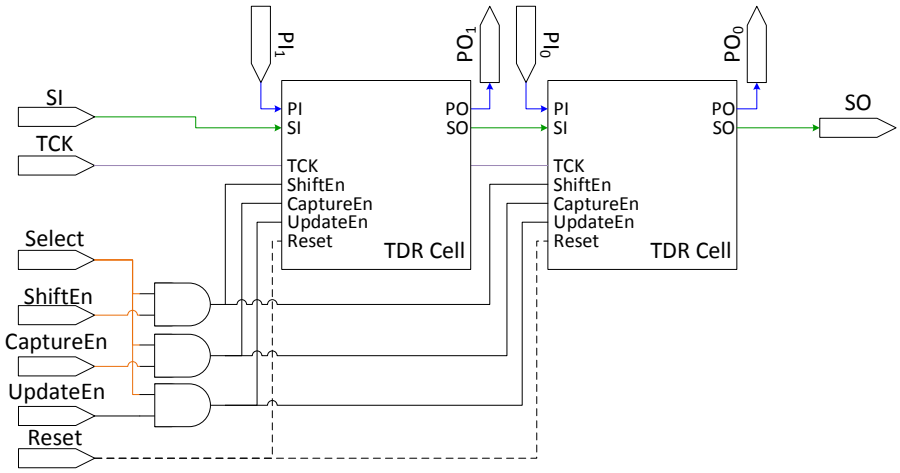


Figure B.2. A two-bit TDR

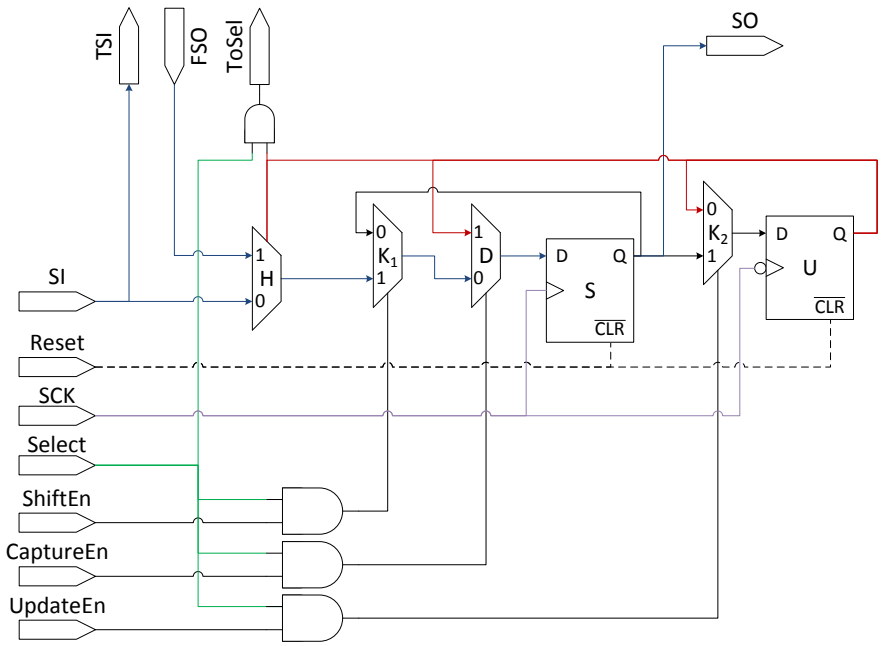


Figure B.3. A possible implementation of a SIB

Appendix C

Detailed Experimental Results for Chapter 3

In Section 4.1.4, experimental results were presented for network optimization for single access schedules. In the results, only the ratio of the shift overhead to instrument data were presented. In this appendix, the complete experimental results is presented, including the overhead numbers in TCKs and as ratios to instrument data.

Table C.1 and Table C.2 present the results for the experiments on designing optimized SIB-based networks. Table C.3 and Table C.4 present the results for the experiments on designing optimized Daisy-chained networks. Table C.5 and Table C.6 present the results for the experiments on designing optimized Remote networks.

Table C.1. Experimental results detailing OAT components for the SIB-based benchmark networks (part I)

Benchmark name	Number of instruments	Lower-bound	Construction method	Schedule															
				Sequential				Concurrent				G10				G25			
				TAP overhead		Shift overhead		TAP overhead		Shift overhead		TAP overhead		Shift overhead		TAP overhead		Shift overhead	
				TCKs	ratio	TCKs	ratio	TCKs	ratio	TCKs	ratio	TCKs	ratio	TCKs	ratio	TCKs	ratio	TCKs	ratio
a586710	26	838530522	F	17323740	0.021	112604310	0.134	7657740	0.009	49775310	0.059	9165816	0.011	59577804	0.071	7704892	0.009	50081798	0.060
			H	17323836	0.021	16101012	0.019	7657744	0.009	8656064	0.010	9165892	0.011	11372326	0.014	7704940	0.009	9062952	0.011
			HPO	17323796	0.021	15425286	0.018	7657740	0.009	6499401	0.008	9165868	0.011	8990986	0.011	7704920	0.009	6894427	0.008
			OC	17323752	0.021	16096799	0.019	7657740	0.009	6466970	0.008	9165836	0.011	9624305	0.011	7704904	0.009	7112381	0.008
			OG10	17323788	0.021	19736648	0.024	7657740	0.009	8761383	0.010	9165852	0.011	11094998	0.013	7704924	0.009	8979522	0.011
			OG25	17323780	0.021	19577739	0.023	7657740	0.009	8436583	0.010	9165844	0.011	11159716	0.013	7704916	0.009	8675358	0.010
d281	48	1496291	F	174420	0.117	2093040	1.399	8200	0.005	98400	0.066	46232	0.031	554784	0.371	17488	0.012	209856	0.140
			H	174604	0.117	415386	0.278	8216	0.005	87828	0.059	46428	0.031	279586	0.187	17552	0.012	164166	0.110
			HPO	174528	0.117	400920	0.268	8208	0.005	74396	0.050	46356	0.031	256047	0.171	17524	0.012	140520	0.094
			OC	174444	0.117	961397	0.643	8200	0.005	46258	0.031	46288	0.031	281508	0.188	17508	0.012	107341	0.072
			OG10	174488	0.117	476501	0.318	8208	0.005	75569	0.051	46284	0.031	145954	0.098	17516	0.012	93180	0.062
			OG25	174456	0.117	620573	0.415	8204	0.005	55354	0.037	46268	0.031	191865	0.128	17508	0.012	70289	0.047
d695	157	704057	F	56588	0.080	2221079	3.155	944	0.001	37052	0.053	4388	0.006	172229	0.245	2080	0.003	81640	0.116
			H	57208	0.081	198798	0.282	964	0.001	30074	0.043	4616	0.007	102226	0.145	2184	0.003	69350	0.099
			HPO	56860	0.081	194819	0.277	956	0.001	22492	0.032	4520	0.006	89599	0.127	2124	0.003	56272	0.080
			OC	56620	0.080	928564	1.319	944	0.001	15176	0.022	4440	0.006	137119	0.195	2104	0.003	58360	0.083
			OG10	56760	0.081	251100	0.357	952	0.001	23446	0.033	4432	0.006	34079	0.048	2116	0.003	27764	0.039
			OG25	56680	0.081	378891	0.538	952	0.001	19698	0.028	4420	0.006	49217	0.070	2100	0.003	25248	0.036
f2126	34	5330439	F	46356	0.009	394026	0.074	1696	0.000	14416	0.003	15972	0.003	135762	0.025	6852	0.001	58242	0.011
			H	46484	0.009	116558	0.022	1712	0.000	23720	0.004	16064	0.003	81326	0.015	6904	0.001	59316	0.011
			HPO	46436	0.009	113363	0.021	1708	0.000	19867	0.004	16040	0.003	75248	0.014	6884	0.001	51405	0.010
			OC	46364	0.009	267715	0.050	1696	0.000	12382	0.002	15980	0.003	109055	0.020	6860	0.001	50329	0.009
			OG10	46416	0.009	113600	0.021	1708	0.000	20184	0.004	16012	0.003	49341	0.009	6880	0.001	37898	0.007
			OG25	46388	0.009	150955	0.028	1704	0.000	15906	0.003	15988	0.003	60604	0.011	6872	0.001	24976	0.005
g1023	63	736216	F	33796	0.046	532287	0.723	4104	0.006	64638	0.088	8156	0.011	128457	0.174	5320	0.007	83790	0.114
			H	34040	0.046	87288	0.119	4112	0.006	21932	0.030	8324	0.011	53872	0.073	5380	0.007	37952	0.052
			HPO	33932	0.046	86321	0.117	4112	0.006	19508	0.026	8260	0.011	50283	0.068	5352	0.007	33891	0.046
			OC	33836	0.046	178986	0.243	4104	0.006	10803	0.015	8184	0.011	63675	0.086	5360	0.007	31152	0.042
			OG10	33888	0.046	101238	0.138	4104	0.006	19382	0.026	8184	0.011	28108	0.038	5348	0.007	21961	0.030
			OG25	33856	0.046	111712	0.152	4104	0.006	14551	0.020	8180	0.011	35619	0.048	5340	0.007	18627	0.025
h953	44	1197178	F	30036	0.025	330396	0.276	1372	0.001	15092	0.013	7900	0.007	86900	0.073	3212	0.003	35332	0.030
			H	30204	0.025	76972	0.064	1388	0.001	16206	0.014	7976	0.007	52370	0.044	3252	0.003	35636	0.030
			HPO	30124	0.025	74371	0.062	1380	0.001	12734	0.011	7948	0.007	46591	0.039	3236	0.003	30463	0.025
			OC	30064	0.025	145243	0.121	1372	0.001	8657	0.007	7904	0.007	60005	0.050	3216	0.003	27048	0.023
			OG10	30104	0.025	84521	0.071	1380	0.001	15651	0.013	7916	0.007	28724	0.024	3224	0.003	20205	0.017
			OG25	30080	0.025	103693	0.087	1376	0.001	12783	0.011	7912	0.007	35949	0.030	3224	0.003	17467	0.015

G10: a generic schedule in which 10 percent of instruments are accessed concurrently.
G25: a generic schedule in which 25 percent of instruments are accessed concurrently

F: these networks have a flat architecture.
H: these networks are constructed by Algorithm 4.2 (i.e., they are optimized for the sequential schedule).
HPO: these networks are constructed by Algorithm 4.3 (i.e., designed by Algorithm 4.2 with post optimization).
OC: these networks are optimized for the concurrent schedule.
OG10 and **OG25**: these networks are optimized for the G10 and G25 schedules, respectively.

Table C.2. Experimental results detailing OAT components for the SIB-based benchmark networks (part II)

Benchmark name	Number of instruments	Lower-bound	Construction method	Schedule															
				Sequential				Concurrent				G10				G25			
				TAP overhead		Shift overhead		TAP overhead		Shift overhead		TAP overhead		Shift overhead		TAP overhead		Shift overhead	
				TCKs	ratio	TCKs	ratio	TCKs	ratio	TCKs	ratio	TCKs	ratio	TCKs	ratio	TCKs	ratio	TCKs	ratio
p22810	254	7784963	F	358868	0.046	22788118	2.927	49304	0.006	3130804	0.402	54160	0.007	3439160	0.442	50364	0.006	3198114	0.411
			H	359876	0.046	1116126	0.143	49312	0.006	235688	0.030	54332	0.007	475228	0.061	50456	0.006	355214	0.046
			HPO	359372	0.046	1093240	0.140	49308	0.006	199204	0.026	54256	0.007	420091	0.054	50404	0.006	303267	0.039
			OC	358964	0.046	4738139	0.609	49304	0.006	114948	0.015	54252	0.007	618247	0.079	50460	0.006	274211	0.035
			OG10	359116	0.046	1535350	0.197	49304	0.006	157762	0.020	54192	0.007	200489	0.026	50388	0.006	166117	0.021
			OG25	359044	0.046	1973604	0.254	49304	0.006	143137	0.018	54188	0.007	267930	0.034	50380	0.006	159874	0.021
p34392	103	16403755	F	667896	0.041	17198322	1.048	49352	0.003	1270814	0.077	91060	0.006	2344795	0.143	58864	0.004	1515748	0.092
			H	668300	0.041	1797300	0.110	49364	0.003	418674	0.026	91152	0.006	882548	0.054	58928	0.004	605506	0.037
			HPO	668140	0.041	1755469	0.107	49360	0.003	372409	0.023	91116	0.006	821852	0.050	58912	0.004	538916	0.033
			OC	667956	0.041	4512864	0.275	49352	0.003	220040	0.013	91100	0.006	1287559	0.078	58908	0.004	480595	0.029
			OG10	668024	0.041	2332982	0.142	49356	0.003	335487	0.020	91084	0.006	452798	0.028	58884	0.004	364874	0.022
			OG25	667980	0.041	2865517	0.175	49352	0.003	273673	0.017	91076	0.006	588389	0.036	58884	0.004	321807	0.020
p93791	586	30083283	F	837196	0.028	122649214	4.077	24516	0.001	3591594	0.119	33332	0.001	4883138	0.162	27204	0.001	3985386	0.132
			H	839532	0.028	3619582	0.120	24532	0.001	470982	0.016	33488	0.001	1391680	0.046	27292	0.001	971414	0.032
			HPO	838340	0.028	3584330	0.119	24528	0.001	382085	0.013	33428	0.001	1240833	0.041	27260	0.001	828431	0.028
			OC	837288	0.028	43177252	1.435	24516	0.001	231491	0.008	33424	0.001	1990219	0.066	27296	0.001	838038	0.028
			OG10	837548	0.028	7032340	0.234	24516	0.001	301894	0.010	33368	0.001	394691	0.013	27224	0.001	336884	0.011
			OG25	837416	0.028	11889613	0.395	24516	0.001	266745	0.009	33360	0.001	659514	0.022	27220	0.001	338180	0.011
q12710	21	31801946	F	97584	0.003	512316	0.016	5264	0.000	27636	0.001	50904	0.002	267246	0.008	20328	0.001	106722	0.003
			H	97660	0.003	216078	0.007	5276	0.000	49182	0.002	50984	0.002	170774	0.005	20356	0.001	117300	0.004
			HPO	97632	0.003	209616	0.007	5272	0.000	40321	0.001	50956	0.002	158247	0.005	20344	0.001	105011	0.003
			OC	97592	0.003	373693	0.012	5264	0.000	26957	0.001	50916	0.002	242288	0.008	20332	0.001	110282	0.003
			OG10	97624	0.003	215323	0.007	5276	0.000	40180	0.001	50944	0.002	124456	0.004	20348	0.001	81809	0.003
			OG25	97600	0.003	226380	0.007	5268	0.000	32449	0.001	50916	0.002	122211	0.004	20336	0.001	55979	0.002
t512505	126	165400967	F	522236	0.003	16450434	0.099	13488	0.000	424872	0.003	49816	0.000	1569204	0.009	26452	0.000	833238	0.005
			H	522732	0.003	1564180	0.009	13508	0.000	277222	0.002	49928	0.000	758102	0.005	26568	0.000	481708	0.003
			HPO	522468	0.003	1529037	0.009	13496	0.000	200126	0.001	49884	0.000	653567	0.004	26520	0.000	390404	0.002
			OC	522328	0.003	5025190	0.030	13488	0.000	140559	0.001	49840	0.000	880210	0.005	26560	0.000	441703	0.003
			OG10	522408	0.003	2199495	0.013	13496	0.000	219823	0.001	49860	0.000	353794	0.002	26488	0.000	276864	0.002
			OG25	522356	0.003	3038852	0.018	13492	0.000	180553	0.001	49844	0.000	468715	0.003	26492	0.000	237739	0.001
u226	30	252929	F	70916	0.280	531870	2.103	10672	0.042	80040	0.316	24220	0.096	181650	0.718	12368	0.049	92760	0.367
			H	71028	0.281	117318	0.464	10680	0.042	35528	0.140	24320	0.096	69606	0.275	12448	0.049	46600	0.184
			HPO	70992	0.281	111236	0.440	10676	0.042	29620	0.117	24284	0.096	59172	0.234	12416	0.049	39671	0.157
			OC	70924	0.280	163927	0.648	10672	0.042	20504	0.081	24228	0.096	69791	0.276	12380	0.049	37879	0.150
			OG10	70972	0.281	127209	0.503	10676	0.042	31813	0.126	24292	0.096	49523	0.196	12412	0.049	37760	0.149
			OG25	70952	0.281	155725	0.616	10672	0.042	23771	0.094	24264	0.096	57968	0.229	12396	0.049	29858	0.118

G10: a generic schedule in which 10 percent of instruments are accessed concurrently.
G25: a generic schedule in which 25 percent of instruments are accessed concurrently

F: these networks have a flat architecture.
H: these networks are constructed by Algorithm 4.2 (i.e., they are optimized for the sequential schedule).
HPO: these networks are constructed by Algorithm 4.3 (i.e., designed by Algorithm 4.2 with post optimization).
OC: these networks are optimized for the concurrent schedule.
OG10 and **OG25**: these networks are optimized for the G10 and G25 schedules, respectively.

Table C.3. Experimental results detailing OAT components for the Daisy-chained benchmark networks (part I)

Benchmark name	Number of instruments	Lower-bound	Construction method	Schedule															
				Sequential				Concurrent				G10				G25			
				TAP overhead		Shift overhead		TAP overhead		Shift overhead		TAP overhead		Shift overhead		TAP overhead		Shift overhead	
				TCKs	ratio	TCKs	ratio	TCKs	ratio	TCKs	ratio	TCKs	ratio	TCKs	ratio	TCKs	ratio	TCKs	ratio
a586710	26	838530522	F	17323840	0.021	112604986	0.134	7657752	0.009	47358892	0.056	9165884	0.011	57538783	0.069	7704932	0.009	47677357	0.057
			H	17323936	0.021	16101432	0.019	7657752	0.009	6239588	0.007	9166068	0.011	9333489	0.011	7705000	0.009	6658478	0.008
			HPO	17323896	0.021	15425674	0.018	7657744	0.009	4082915	0.005	9165984	0.011	6951856	0.008	7704952	0.009	4489866	0.005
			OC	17323852	0.021	16097415	0.019	7657744	0.009	4050481	0.005	9165944	0.011	7585079	0.009	7704920	0.009	4707740	0.006
			OG10	17323888	0.021	19736944	0.024	7657748	0.009	6344916	0.008	9165972	0.011	9055744	0.011	7704960	0.009	6574895	0.008
			OG25	17323880	0.021	19578067	0.023	7657748	0.009	6020112	0.007	9165936	0.011	9120551	0.011	7704936	0.009	6270730	0.007
d281	48	1496291	F	174608	0.117	2095344	1.400	8228	0.005	57189	0.038	46340	0.031	524061	0.350	17632	0.012	172388	0.115
			H	174792	0.117	416070	0.278	8232	0.006	46358	0.031	46660	0.031	248311	0.166	17584	0.012	125204	0.084
			HPO	174716	0.117	401531	0.268	8216	0.005	32912	0.022	46500	0.031	224422	0.150	17564	0.012	101584	0.068
			OC	174632	0.117	962969	0.644	8204	0.005	4766	0.003	46496	0.031	250983	0.168	17612	0.012	68883	0.046
			OG10	174676	0.117	477157	0.319	8232	0.006	34173	0.023	46404	0.031	114602	0.077	17572	0.012	54284	0.036
			OG25	174644	0.117	621442	0.415	8216	0.005	13954	0.009	46440	0.031	160722	0.107	17604	0.012	31832	0.021
d695	157	704057	F	57212	0.081	2245728	3.190	976	0.001	24406	0.035	4808	0.007	175770	0.250	2396	0.003	80496	0.114
			H	57832	0.082	201238	0.286	984	0.001	16310	0.023	4772	0.007	90871	0.129	2288	0.003	57382	0.082
			HPO	57484	0.082	197267	0.280	976	0.001	8712	0.012	4580	0.007	76916	0.109	2196	0.003	43673	0.062
			OC	57244	0.081	943869	1.341	948	0.001	1385	0.002	4816	0.007	132367	0.188	2312	0.003	48144	0.068
			OG10	57384	0.082	254215	0.361	980	0.001	10133	0.014	4652	0.007	22587	0.032	2180	0.003	15067	0.021
			OG25	57304	0.081	383669	0.545	988	0.001	6368	0.009	4748	0.007	39033	0.055	2252	0.003	13396	0.019
f2126	34	5330439	F	46488	0.009	395182	0.074	1704	0.000	3322	0.001	16056	0.003	128902	0.024	6892	0.001	48717	0.009
			H	46616	0.009	116914	0.022	1720	0.000	12590	0.002	16140	0.003	73897	0.014	6936	0.001	49550	0.009
			HPO	46568	0.009	113711	0.021	1728	0.000	8752	0.002	16100	0.003	67764	0.013	6936	0.001	41696	0.008
			OC	46496	0.009	268573	0.050	1700	0.000	1249	0.000	16056	0.003	101908	0.019	6900	0.001	40690	0.008
			OG10	46548	0.009	113936	0.021	1732	0.000	9121	0.002	16068	0.003	42032	0.008	6916	0.001	28101	0.005
			OG25	46520	0.009	151399	0.028	1724	0.000	4824	0.001	16048	0.003	53186	0.010	6912	0.001	15312	0.003
g1023	63	736216	F	34044	0.046	536256	0.728	4156	0.006	58048	0.079	8320	0.011	124672	0.169	5512	0.007	79744	0.108
			H	34288	0.047	88144	0.120	4120	0.006	14612	0.020	8432	0.011	48162	0.065	5416	0.007	31166	0.042
			HPO	34180	0.046	87135	0.118	4132	0.006	12212	0.017	8340	0.011	44100	0.060	5396	0.007	26990	0.037
			OC	34084	0.046	181627	0.247	4108	0.006	3463	0.005	8432	0.011	60286	0.082	5448	0.007	25154	0.034
			OG10	34136	0.046	102260	0.139	4112	0.006	12190	0.017	8276	0.011	22127	0.030	5388	0.007	15102	0.021
			OG25	34104	0.046	112965	0.153	4112	0.006	7405	0.010	8288	0.011	29894	0.041	5420	0.007	12130	0.016
h953	44	1197178	F	30208	0.025	332332	0.278	1400	0.001	8242	0.007	8036	0.007	82897	0.069	3344	0.003	30112	0.025
			H	30376	0.025	77506	0.065	1404	0.001	9109	0.008	8072	0.007	47230	0.039	3276	0.003	29231	0.024
			HPO	30296	0.025	74881	0.063	1396	0.001	5630	0.005	8036	0.007	41381	0.035	3268	0.003	23897	0.020
			OC	30236	0.025	146521	0.122	1376	0.001	1540	0.001	8016	0.007	55149	0.046	3268	0.003	20733	0.017
			OG10	30276	0.025	85036	0.071	1408	0.001	8655	0.007	7980	0.007	23542	0.020	3256	0.003	13725	0.011
			OG25	30252	0.025	104360	0.087	1392	0.001	5778	0.005	8016	0.007	30788	0.026	3260	0.003	11083	0.009

G10: a generic schedule in which 10 percent of instruments are accessed concurrently.
G25: a generic schedule in which 25 percent of instruments are accessed concurrently

F: these networks have a flat architecture.
H: these networks are constructed by Algorithm 4.2 (i.e., they are optimized for the sequential schedule).
HPO: these networks are constructed by Algorithm 4.3 (i.e., designed by Algorithm 4.2 with post optimization).
OC: these networks are optimized for the concurrent schedule.
OG10 and **OG25**: these networks are optimized for the G10 and G25 schedules, respectively.

Table C.4. Experimental results detailing OAT components for the Daisy-chained benchmark networks (part II)

Benchmark name	Number of instruments	Lower-bound	Construction method	Schedule															
				Sequential				Concurrent				G10				G25			
				TAP overhead		Shift overhead		TAP overhead		Shift overhead		TAP overhead		Shift overhead		TAP overhead		Shift overhead	
				TCKs	ratio	TCKs	ratio	TCKs	ratio	TCKs	ratio	TCKs	ratio	TCKs	ratio	TCKs	ratio	TCKs	ratio
p22810	254	7784963	F	359880	0.046	22852634	2.935	49416	0.006	3060554	0.393	55032	0.007	3418574	0.439	51096	0.007	3167654	0.407
			H	360888	0.046	1120894	0.144	49320	0.006	158592	0.020	54444	0.007	403001	0.052	50476	0.006	280507	0.036
			HPO	360384	0.046	1097997	0.141	49316	0.006	122133	0.016	54316	0.007	345562	0.044	50436	0.006	227344	0.029
			OC	359976	0.046	4776223	0.614	49308	0.006	37887	0.005	54656	0.007	564101	0.072	50572	0.006	209230	0.027
			OG10	360128	0.046	1542692	0.198	49312	0.006	81594	0.010	54380	0.007	127860	0.016	50432	0.006	90721	0.012
			OG25	360056	0.046	1983821	0.255	49312	0.006	66929	0.009	54404	0.007	197512	0.025	50444	0.006	85797	0.011
p34392	103	16403755	F	668304	0.041	17208931	1.049	49424	0.003	1118051	0.068	91400	0.006	2209427	0.135	59144	0.004	1370771	0.084
			H	668708	0.041	1798906	0.110	49372	0.003	264150	0.016	91228	0.006	739110	0.045	58964	0.004	453666	0.028
			HPO	668548	0.041	1757097	0.107	49376	0.003	217915	0.013	91184	0.006	678153	0.041	58940	0.004	386993	0.024
			OC	668364	0.041	4520192	0.276	49356	0.003	65586	0.004	91168	0.006	1143806	0.070	59044	0.004	330506	0.020
			OG10	668432	0.041	2335212	0.142	49384	0.003	181223	0.011	91164	0.006	309437	0.019	58928	0.004	213178	0.013
			OG25	668388	0.041	2868318	0.175	49376	0.003	119441	0.007	91200	0.006	445139	0.027	58984	0.004	170545	0.010
p93791	586	30083283	F	839536	0.028	122992610	4.088	24612	0.001	3402513	0.113	35192	0.001	4955128	0.165	28408	0.001	3959576	0.132
			H	841872	0.028	3630952	0.121	24548	0.001	268303	0.009	33584	0.001	1196574	0.040	27320	0.001	773686	0.026
			HPO	840680	0.028	3595527	0.120	24552	0.001	179364	0.006	33524	0.001	1043326	0.035	27280	0.001	628169	0.021
			OC	839628	0.028	43372336	1.442	24520	0.001	28961	0.001	34272	0.001	1865944	0.062	27560	0.001	690434	0.023
			OG10	839888	0.028	7057402	0.235	24524	0.001	101548	0.003	33920	0.001	206725	0.007	27308	0.001	139943	0.005
			OG25	839756	0.028	11935243	0.397	24524	0.001	66719	0.002	34136	0.001	488164	0.016	27496	0.001	164027	0.005
q12710	21	31801946	F	97664	0.003	512757	0.016	5272	0.000	4601	0.000	50936	0.002	255753	0.008	20384	0.001	87717	0.003
			H	97740	0.003	216266	0.007	5288	0.000	26151	0.001	51052	0.002	159198	0.005	20380	0.001	98070	0.003
			HPO	97712	0.003	209798	0.007	5292	0.000	17296	0.001	51000	0.002	146643	0.005	20380	0.001	85803	0.003
			OC	97672	0.003	374030	0.012	5268	0.000	3897	0.000	50960	0.002	230732	0.007	20364	0.001	91068	0.003
			OG10	97704	0.003	215509	0.007	5300	0.000	17182	0.001	50972	0.002	112901	0.004	20392	0.001	62633	0.002
			OG25	97680	0.003	226575	0.007	5280	0.000	9423	0.000	50948	0.002	110621	0.003	20384	0.001	36850	0.001
t512505	126	165400967	F	522736	0.003	16466310	0.100	13600	0.000	301242	0.002	50268	0.000	1465451	0.009	26840	0.000	721612	0.004
			H	523232	0.003	1566408	0.009	13516	0.000	150239	0.001	50032	0.000	641760	0.004	26628	0.000	359237	0.002
			HPO	522968	0.003	1531227	0.009	13504	0.000	73158	0.000	49956	0.000	536272	0.003	26592	0.000	267478	0.002
			OC	522828	0.003	5035085	0.030	13492	0.000	13561	0.000	50344	0.000	770397	0.005	26768	0.000	324039	0.002
			OG10	522908	0.003	2202122	0.013	13528	0.000	93157	0.001	49988	0.000	236954	0.001	26600	0.000	153972	0.001
			OG25	522856	0.003	3043104	0.018	13516	0.000	53908	0.000	49964	0.000	351768	0.002	26592	0.000	115328	0.001
u226	30	252929	F	71032	0.281	532770	2.106	10680	0.042	65042	0.257	24272	0.096	170380	0.674	12412	0.049	78465	0.310
			H	71144	0.281	117722	0.465	10692	0.042	20515	0.081	24492	0.097	58489	0.231	12504	0.049	32184	0.127
			HPO	71108	0.281	111628	0.441	10684	0.042	14590	0.058	24396	0.096	47786	0.189	12460	0.049	25208	0.100
			OC	71040	0.281	164693	0.651	10676	0.042	5475	0.022	24300	0.096	58379	0.231	12428	0.049	23457	0.093
			OG10	71088	0.281	127574	0.504	10688	0.042	16794	0.066	24384	0.096	38132	0.151	12464	0.049	23268	0.092
			OG25	71068	0.281	156158	0.617	10680	0.042	8752	0.035	24340	0.096	46538	0.184	12460	0.049	15406	0.061

G10: a generic schedule in which 10 percent of instruments are accessed concurrently.
G25: a generic schedule in which 25 percent of instruments are accessed concurrently

F: these networks have a flat architecture.
H: these networks are constructed by Algorithm 4.2 (i.e., they are optimized for the sequential schedule).
HPO: these networks are constructed by Algorithm 4.3 (i.e., designed by Algorithm 4.2 with post optimization).
OC: these networks are optimized for the concurrent schedule.
OG10 and **OG25**: these networks are optimized for the G10 and G25 schedules, respectively.

Table C.5. Experimental results detailing OAT components for the Remote benchmark networks (part I)

Benchmark name	Number of instruments	Lower-bound	Construction method	Schedule															
				Sequential				Concurrent				G10				G25			
				TAP overhead		Shift overhead		TAP overhead		Shift overhead		TAP overhead		Shift overhead		TAP overhead		Shift overhead	
				TCKs	ratio	TCKs	ratio	TCKs	ratio	TCKs	ratio	TCKs	ratio	TCKs	ratio	TCKs	ratio	TCKs	ratio
a586710	26	838530522	Random 1	17324620	0.021	1326	0.000	7657872	0.009	8593024	0.010	9166714	0.011	8285687	0.010	7705552	0.009	8743872	0.010
			Random 2	17324620	0.021	1326	0.000	7657872	0.009	14909332	0.018	9166714	0.011	15141104	0.018	7705552	0.009	14998273	0.018
			OC	17324620	0.021	1326	0.000	7657872	0.009	170	0.000	9166714	0.011	887143	0.001	7705552	0.009	372013	0.000
			OG10	17324620	0.021	1326	0.000	7657872	0.009	170	0.000	9166714	0.011	798792	0.001	7705552	0.009	327838	0.000
			OG25	17324620	0.021	1326	0.000	7657872	0.009	170	0.000	9166714	0.011	810572	0.001	7705552	0.009	257164	0.000
d281	48	1496291	Random 1	176048	0.118	4560	0.003	8468	0.006	49348	0.033	48204	0.032	313745	0.210	19482	0.013	140207	0.094
			Random 2	176048	0.118	4560	0.003	8468	0.006	51241	0.034	48204	0.032	262691	0.176	19482	0.013	153560	0.103
			OC	176048	0.118	4560	0.003	8468	0.006	490	0.000	48204	0.032	133007	0.089	19482	0.013	47597	0.032
			OG10	176048	0.118	4560	0.003	8468	0.006	26142	0.017	48204	0.032	33937	0.023	19482	0.013	52442	0.035
			OG25	176048	0.118	4560	0.003	8468	0.006	1308	0.001	48204	0.032	36424	0.024	19482	0.013	15546	0.010
d695	157	704057	Random 1	61922	0.088	49141	0.070	1246	0.002	20357	0.029	11098	0.016	157239	0.223	6960	0.010	80349	0.114
			Random 2	61922	0.088	49141	0.070	1246	0.002	22670	0.032	11098	0.016	164868	0.234	6960	0.010	82744	0.118
			OC	61922	0.088	49141	0.070	1246	0.002	2102	0.003	11098	0.016	136886	0.194	6960	0.010	65027	0.092
			OG10	61922	0.088	49141	0.070	1246	0.002	12387	0.018	11098	0.016	88684	0.126	6960	0.010	51774	0.074
			OG25	61922	0.088	49141	0.070	1246	0.002	6661	0.009	11098	0.016	109421	0.155	6960	0.010	55117	0.078
f2126	34	5330439	Random 1	47508	0.009	2278	0.000	1794	0.000	2896	0.001	17160	0.003	57969	0.011	7410	0.001	34709	0.007
			Random 2	47508	0.009	2278	0.000	1794	0.000	2548	0.000	17160	0.003	57650	0.011	7410	0.001	32418	0.006
			OC	47508	0.009	2278	0.000	1794	0.000	124	0.000	17160	0.003	53081	0.010	7410	0.001	26014	0.005
			OG10	47508	0.009	2278	0.000	1794	0.000	2809	0.001	17160	0.003	8356	0.002	7410	0.001	14251	0.003
			OG25	47508	0.009	2278	0.000	1794	0.000	2896	0.001	17160	0.003	12101	0.002	7410	0.001	3147	0.001
g1023	63	736216	Random 1	35934	0.049	7875	0.011	4576	0.006	42196	0.057	10916	0.015	72406	0.098	8002	0.011	56961	0.077
			Random 2	35934	0.049	7875	0.011	4576	0.006	34896	0.047	10916	0.015	69982	0.095	8002	0.011	50287	0.068
			OC	35934	0.049	7875	0.011	4576	0.006	1251	0.002	10916	0.015	44194	0.060	8002	0.011	20802	0.028
			OG10	35934	0.049	7875	0.011	4576	0.006	6711	0.009	10916	0.015	30938	0.042	8002	0.011	17349	0.024
			OG25	35934	0.049	7875	0.011	4576	0.006	2486	0.003	10916	0.015	39692	0.054	8002	0.011	19385	0.026
h953	44	1197178	Random 1	31528	0.026	3828	0.003	1640	0.001	6056	0.005	9834	0.008	45031	0.038	5014	0.004	21736	0.018
			Random 2	31528	0.026	3828	0.003	1640	0.001	7315	0.006	9834	0.008	45638	0.038	5014	0.004	25481	0.021
			OC	31528	0.026	3828	0.003	1640	0.001	482	0.000	9834	0.008	36503	0.030	5014	0.004	16228	0.014
			OG10	31528	0.026	3828	0.003	1640	0.001	3589	0.003	9834	0.008	14642	0.012	5014	0.004	11027	0.009
			OG25	31528	0.026	3828	0.003	1640	0.001	2486	0.002	9834	0.008	22042	0.018	5014	0.004	11311	0.009

G10: a generic schedule in which 10 percent of instruments are accessed concurrently.
G25: a generic schedule in which 25 percent of instruments are accessed concurrently

Random 1 and **Random 2**: these networks have different random ordering of instruments on their scan path.
OC: these networks are optimized for the concurrent schedule.
OG10 and **OG25**: these networks are optimized for the G10 and G25 schedules, respectively.

Table C.6. Experimental results detailing OAT components for the Remote benchmark networks (part II)

Benchmark name	Number of instruments	Lower-bound	Construction method	Schedule															
				Sequential				Concurrent				G10				G25			
				TAP overhead		Shift overhead		TAP overhead		Shift overhead		TAP overhead		Shift overhead		TAP overhead		Shift overhead	
				TCKs	ratio	TCKs	ratio	TCKs	ratio	TCKs	ratio	TCKs	ratio	TCKs	ratio	TCKs	ratio	TCKs	ratio
p22810	254	7784963	Random 1	367500	0.047	128778	0.017	50286	0.006	2865282	0.368	66778	0.009	2947241	0.379	60326	0.008	2935895	0.377
			Random 2	367500	0.047	128778	0.017	50286	0.006	1565327	0.201	66778	0.009	1696358	0.218	60326	0.008	1643714	0.211
			OC	367500	0.047	128778	0.017	50286	0.006	10775	0.001	66778	0.009	556744	0.072	60326	0.008	233209	0.030
			OG10	367500	0.047	128778	0.017	50286	0.006	20781	0.003	66778	0.009	526938	0.068	60326	0.008	229290	0.029
			OG25	367500	0.047	128778	0.017	50286	0.006	15979	0.002	66778	0.009	526279	0.068	60326	0.008	227297	0.029
p34392	103	16403755	Random 1	671394	0.041	21115	0.001	49994	0.003	557844	0.034	95054	0.006	1092873	0.067	62592	0.004	654717	0.040
			Random 2	671394	0.041	21115	0.001	49994	0.003	713477	0.043	95054	0.006	1233923	0.075	62592	0.004	855546	0.052
			OC	671394	0.041	21115	0.001	49994	0.003	2958	0.000	95054	0.006	796301	0.049	62592	0.004	218279	0.013
			OG10	671394	0.041	21115	0.001	49994	0.003	28599	0.002	95054	0.006	471450	0.029	62592	0.004	155992	0.010
			OG25	671394	0.041	21115	0.001	49994	0.003	16497	0.001	95054	0.006	479190	0.029	62592	0.004	128039	0.008
p93791	586	30083283	Random 1	857116	0.028	686206	0.023	25362	0.001	2415387	0.080	60474	0.002	3942795	0.131	43840	0.001	3018774	0.100
			Random 2	857116	0.028	686206	0.023	25362	0.001	1257836	0.042	60474	0.002	3142521	0.104	43840	0.001	1951933	0.065
			OC	857116	0.028	686206	0.023	25362	0.001	22344	0.001	60474	0.002	2189237	0.073	43840	0.001	925660	0.031
			OG10	857116	0.028	686206	0.023	25362	0.001	315236	0.010	60474	0.002	1686133	0.056	43840	0.001	843575	0.028
			OG25	857116	0.028	686206	0.023	25362	0.001	92342	0.003	60474	0.002	1863336	0.062	43840	0.001	811629	0.027
q12710	21	31801946	Random 1	98294	0.003	861	0.000	5362	0.000	2923	0.000	51506	0.002	52702	0.002	20996	0.001	53301	0.002
			Random 2	98294	0.003	861	0.000	5362	0.000	2463	0.000	51506	0.002	94962	0.003	20996	0.001	52657	0.002
			OC	98294	0.003	861	0.000	5362	0.000	83	0.000	51506	0.002	77606	0.002	20996	0.001	49811	0.002
			OG10	98294	0.003	861	0.000	5362	0.000	3013	0.000	51506	0.002	2148	0.000	20996	0.001	28728	0.001
			OG25	98294	0.003	861	0.000	5362	0.000	3103	0.000	51506	0.002	6794	0.000	20996	0.001	2171	0.000
t512505	126	165400967	Random 1	526516	0.003	31626	0.000	14470	0.000	279886	0.002	55838	0.000	1186138	0.007	32130	0.000	561442	0.003
			Random 2	526516	0.003	31626	0.000	14470	0.000	294679	0.002	55838	0.000	1236396	0.007	32130	0.000	600080	0.004
			OC	526516	0.003	31626	0.000	14470	0.000	4922	0.000	55838	0.000	613362	0.004	32130	0.000	241702	0.001
			OG10	526516	0.003	31626	0.000	14470	0.000	21393	0.000	55838	0.000	427718	0.003	32130	0.000	179643	0.001
			OG25	526516	0.003	31626	0.000	14470	0.000	9265	0.000	55838	0.000	510145	0.003	32130	0.000	212907	0.001
u226	30	252929	Random 1	71932	0.284	1770	0.007	10770	0.043	33893	0.134	25200	0.100	62828	0.248	13152	0.052	41602	0.164
			Random 2	71932	0.284	1770	0.007	10770	0.043	54605	0.216	25200	0.100	94263	0.373	13152	0.052	59484	0.235
			OC	71932	0.284	1770	0.007	10770	0.043	116	0.000	25200	0.100	23842	0.094	13152	0.052	12395	0.049
			OG10	71932	0.284	1770	0.007	10770	0.043	28655	0.113	25200	0.100	7292	0.029	13152	0.052	32720	0.129
			OG25	71932	0.284	1770	0.007	10770	0.043	116	0.000	25200	0.100	15568	0.062	13152	0.052	7092	0.028

G10: a generic schedule in which 10 percent of instruments are accessed concurrently.
G25: a generic schedule in which 25 percent of instruments are accessed concurrently

Random 1 and **Random 2**: these networks have different random ordering of instruments on their scan path.
OC: these networks are optimized for the concurrent schedule.
OG10 and **OG25**: these networks are optimized for the G10 and G25 schedules, respectively.

Appendix D

Benchmarks

In this appendix, we present details on the benchmarks used throughout the thesis. Section D.1 provides information on how we have extracted instrument sets from the ITC'02 benchmarks used in Chapter 4, Chapter 5, and Chapter 6. Section D.2 provides information on how we have constructed circuits from ITC'02 benchmarks for the purpose of upper-bound computation in Chapter 6, as well as larger illustrations of the N and C benchmarks introduced in Chapter 6.

D.1. ITC'02 BENCHMARKS

In this section, we detail how we have extracted instruments from the ITC'02 benchmark set [48]. The extracted instrument sets are used in the experiments presented in Chapter 4, Chapter 5, and Chapter 6.

The ITC'02 set consists of 12 benchmark SoCs. For each SoC, list of modules (i.e., cores) are given, and for each module, number of I/O terminals and internal scan-chains, as well as number of patterns to apply to those terminals and scan-chains are specified. Moreover, for two of the SoCs, namely, d281, u226, tests are specified for which TAM USE and SCAN USE properties are set to zero. We interpreted these tests to be of BIST type, and the test length property to be in number of system clock cycles.

We created a set of instruments for each of the available SoCs, based on [39], as explained below:

- We considered the set of input terminals for each module (including bidirectional terminals) as an instrument with a shift-register length (L) equal to the number of input terminals, and number of accesses (A) equal to the number of patterns specified for the input terminals.

Table D.1. Properties of instrument sets extracted from ITC'02 benchmark set

Benchmark name	Instrument data	Number of instruments	Length of shift-registers			Number of accesses		
			min	ave.	max	min	ave.	max
a586710	838530522	26	34	1545	2626	2945	166573	1914433
d281	1496291	48	7	48	233	26	907	2048
d695	704057	157	1	52	320	12	89	234
f2126	5330439	34	20	447	1000	103	339	422
g1023	736216	63	9	81	377	15	133	1024
h953	1197178	44	9	125	348	9	169	341
p22810	7784963	254	1	117	400	1	352	12324
p34392	16403755	103	4	224	806	27	1620	12336
p93791	30083283	586	1	166	538	11	356	6127
q12710	31801946	21	413	1245	3784	852	1160	1314
t512505	165400967	126	2	607	1669	3	1035	3370
u226	252929	30	3	42	97	15	589	2666

- We considered the set of output terminals for each module (including bidirectional terminals) as an instrument with a shift-register length (L) equal to the number of output terminals, and number of accesses (A) equal to the number of patterns specified for the output terminals.
- We considered each internal scan-chain as an instrument with a shift-register length (L) equal to the length of that scan-chain, and number of accesses (A) equal to the number of patterns specified for that scan-chain.

Table D.1 lists some properties of the extracted instrument sets corresponding to each of the SoCs. The first column presents the SoC name from the ITC'02 set. The second column presents instrument data for each set calculated by using Eq. (3.2). The third column lists the number of instruments included in each set. Columns 4–6 present the minimum, average, and maximum length found among instrument shift-registers in the set. Finally, columns 7–9 present the minimum, average, and maximum number of accesses found among instruments in the set.

In the experiment presented in Chapter 5, there was a need to a set of instruments to experiment with concurrent access to instruments while executing wait cycles. The u226 benchmark was particularly suitable for this purpose as it contained information on BIST instruments. Table D.2 details the list of instruments extracted from the u226 benchmark SoC. Note that the length of instrument shift-register for the BIST instruments is considered to be zero, as the presented experiment disregard the few accesses required to start the BIST instruments and to check the results when the instrument is done.

Table D.2. Test specifications of the network assumed for U226

Instrument	BIST/Access	Shift-register Length	Test length	Number of accesses
1	Access	3		2666
2	Access	17		2666
3	Access	3		2666
4	Access	17		2666
5	Access	3		2666
6	Access	17		2666
7	Access	52		76
8	Access	52		76
9	Access	52		76
10	Access	52		76
11	Access	52		76
12	Access	52		76
13	Access	52		76
14	Access	52		76
15	Access	52		76
16	Access	52		76
17	Access	52		76
18	Access	52		76
19	Access	52		76
20	Access	52		76
21	Access	52		76
22	Access	52		76
23	Access	52		76
24	Access	52		76
25	Access	52		76
26	Access	52		76
27	Access	97		76
28	Access	64		76
29	Access	17		15
30	Access	10		15
31	BIST		1363968	
32	BIST		1363968	
33	BIST		1363968	
34	BIST		1048576	

D.2. BENCHMARK CIRCUITS

In this section, details will be provided on the benchmark networks used in Chapter 6. The networks presented in Section D.2.1 are taken from literature, and the ones presented in Section D.2.2 are introduced by us for experimenting with our upper-bound computation method.

D.2.1. BENCHMARKS FROM LITERATURE

This section presents some details on the benchmark networks taken from [39, 40]. The instruments used to construct these networks are extracted from the ITC'02 benchmark set in the same way as explained in Section D.1. The networks are constructed such that there is a one-to-one correspondence between the hierarchical levels in the networks and the hierarchy in the original SoC from the benchmark. As an example, Figure D.1 outlines the hierarchical relation between modules (cores) in the p34392 SoC, and Figure D.2 shows (partially) SIB-based and Daisy-chained networks constructed from the instruments extracted from the p34392 SoC. In case of p34392, Module 1 has only one internal scan-chain specified in its description. That is why, only one instrument is labeled as scan-chain. For modules that have multiple scan-chains, a separate instrument is considered for each scan-chain.

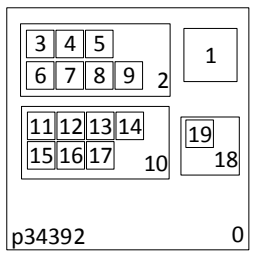
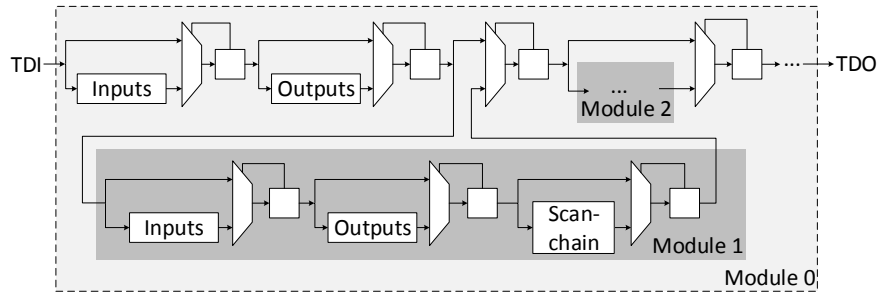
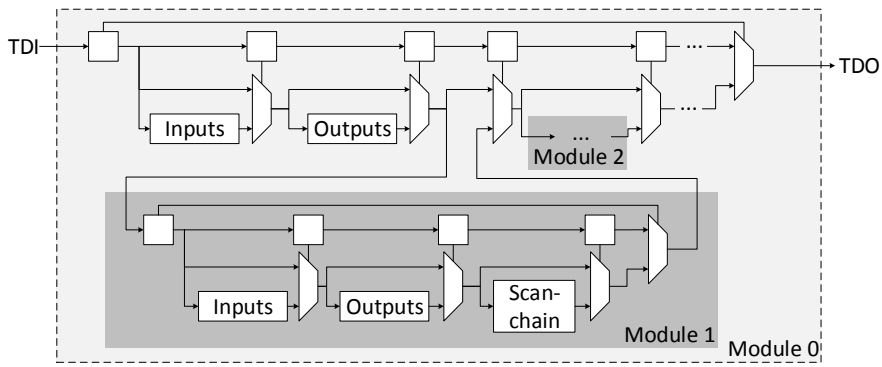


Figure D.1. Overview of hierarchical modules in the p34392 SoC



(a) SIB-based (here, the internals of a SIB module are shown)



(b) Daisy-chained (referred to as MUX-based in [40])

Figure D.2. The two variants of p34392 benchmark

D.2.2. N1–N5

The N1–N5 networks are constructed in such a way that the reduction techniques introduced in this thesis cannot reduce them into smaller segments. For the experiments presented in this thesis, the length of all instrument shift-registers are considered to be 20 flip-flops.

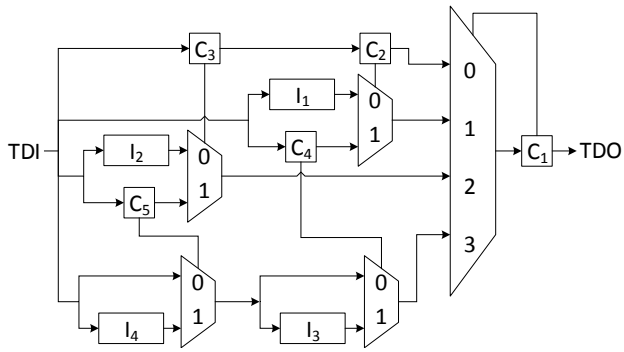


Figure D.3. N1

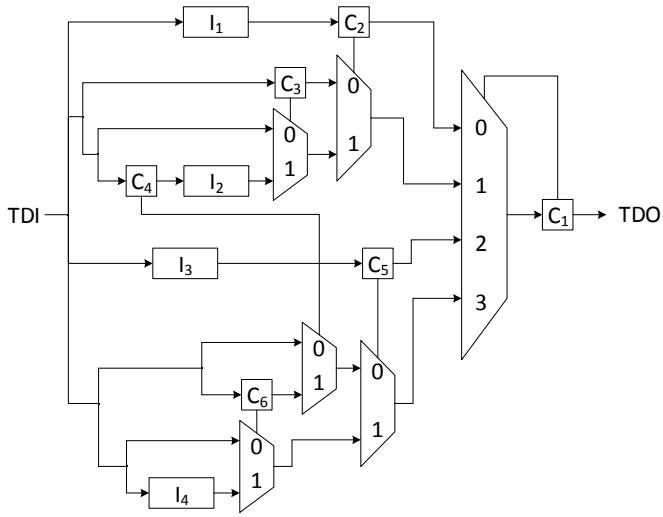


Figure D.4. N2

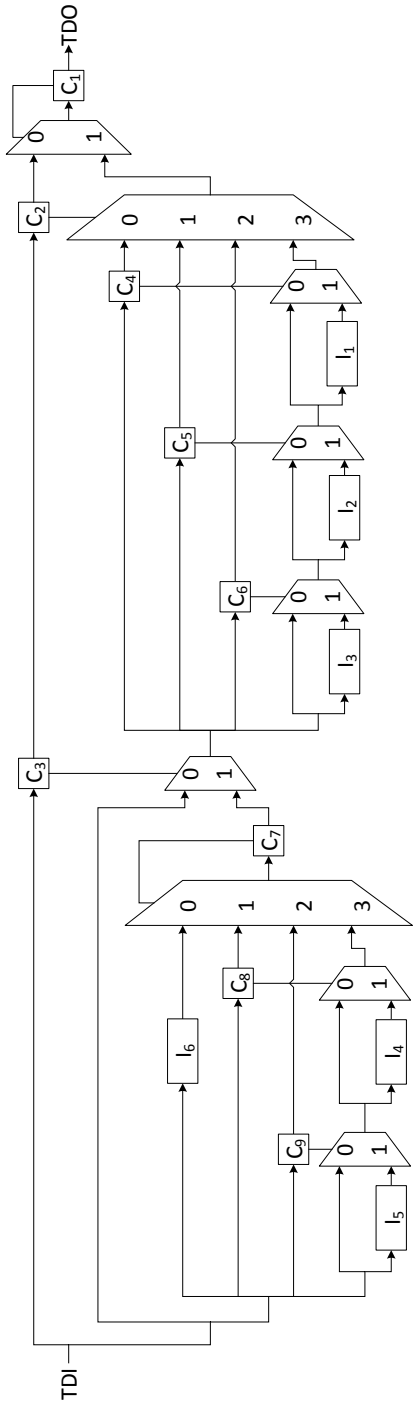


Figure D.5. N3

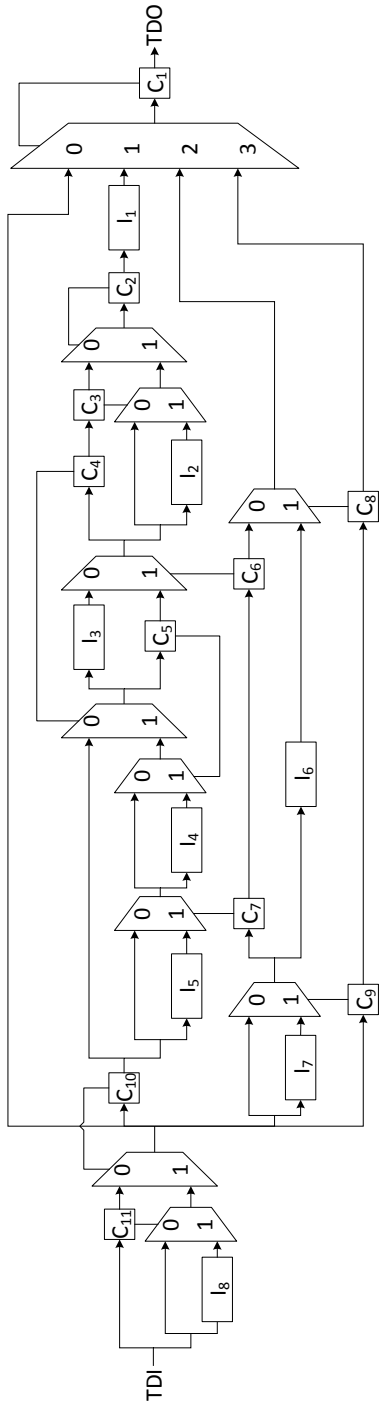

TDO
TDI
C1
C2
C3
C4
C5
C6
C7
C8
C9
C10
C11
I1
I2
I3
I4
I5
I6
I7
I8
0
1
2
3

Figure D.6. N4

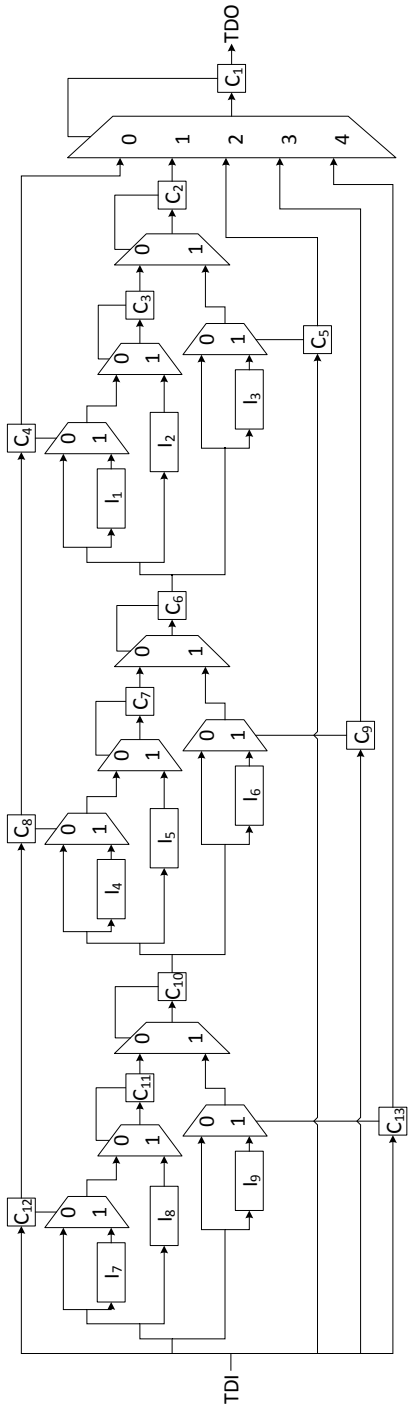

TDO
TDI
C1
C2
C3
C4
C5
C6
C7
C8
C9
C10
C11
C12
C13
I1
I2
I3
I4
I5
I6
I7
I8
I9
0
1
2
3
4

Figure D.7. N5

References

[1] A. Jutman, S. Devadze, and K. Shibin, "Effective scalable IEEE 1687 instrumentation network for fault management," *IEEE Design & Test*, vol. 30, no. 5, pp. 26–35, Oct 2013.

[2] K.-J. Lee, "Chapter 10 - boundary scan and core-based testing," in *VLSI Test Principles and Architectures*, L.-T. Wang, C.-W. Wu, and X. Wen, Eds. San Francisco: Morgan Kaufmann, 2006, pp. 557–618.

[3] F. Poehl, F. Demmerle, J. Alt, and H. Obermeir, "Production test challenges for highly integrated mobile phone socs—a case study," in *European Test Symposium (ETS)*, 2010, pp. 17–22.

[4] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 305–316, Sept 2005.

[5] C. Constantinescu, "Trends and challenges in VLSI circuit reliability," *IEEE Micro*, vol. 23, no. 4, pp. 14–19, July 2003.

[6] A. W. Strong, E. Y. Wu, R.-P. Vollertsen, J. Sune, G. La Rosa, T. D. Sullivan, and S. E. Rauch III, *Reliability wearout mechanisms in advanced CMOS technologies*. John Wiley & Sons, 2009.

[7] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10–16, 2005.

[8] Z. Conroy, J. Grealish, H. Miles, A. J. Suto, A. Crouch, and S. Meyers, "Board assisted-BIST: long and short term solutions for testpoint erosion – reaching into the DFx toolbox," in *International Test Conference (ITC)*, 2012.

[9] "IEEE standard testability method for embedded core-based integrated circuits," *IEEE Std 1500-2005*, 2005.

[10] J. Rearick, B. Eklow, K. Posse, A. Crouch, and B. Bennetts, "IJTAG (Internal JTAG): A step toward a DFT standard," in *International Test Conference (ITC)*, 2005.

[11] H. Park, J. Xu, K. Kim, and J. Park, "On-chip debug architecture for multicore processor," *Electronics and Telecommunications Research Institute (ETRI)*, vol. 34, no. 1, pp. 44–54, Feb. 2012.

[12] A. B. Kinsman, H. F. Ko, and N. Nicolici, "In-system constrained-random stimuli generation for post-silicon validation," in *International Test Conference (ITC)*, 2012.

[13] J. Rearick and A. Volz, "A case study of using IEEE P1687 (IJTAG) for high-speed serial I/O characterization and testing," in *International Test Conference (ITC)*, 2006.

[14] H. M. von Staudt and A. Spyronasios, "Using IJTAG digital islands in analogue circuits to perform trim and test functions," in *International Mixed-Signal Testing Workshop (IMSTW)*, 2015.

[15] Sun Microsystems, Inc. (2016, Mar.), "UltraSPARC T2^{TM} supplement to the UltraSPARC architecture 2007." [Online]. Available: http://www.oracle.com/technetwork/systems/opensparc/t2-14-ust2-uasuppl-draft-hp-ext-1537761.html

[16] M. Boule, J. s. Chenard, and Z. Zilic, "Debug enhancements in assertion-checker generation," *IET Computers Digital Techniques*, vol. 1, no. 6, pp. 669–677, Nov 2007.

[17] T. H. Kim, R. Persaud, and C. H. Kim, "Silicon odometer: an on-chip reliability monitor for measuring frequency degradation of digital circuits," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 4, pp. 874–880, April 2008.

[18] T. Wang, D. Chen, and R. Geiger, "Multi-site on-chip current sensor for electromigration monitoring," in *International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2011.

[19] A. L. Crouch, "IJTAG: the path to organized instrument connectivity," in *International Test Conference (ITC)*, 2007.

[20] "IEEE standard for access and control of instrumentation embedded within a semiconductor device," *IEEE Std 1687-2014*, 2014.

[21] "IEEE standard for test access port and boundary-scan architecture," *IEEE Std 1149.1-2013 (Revision of IEEE Std 1149.1-2001)*, 2013.

[22] K. Shibin, S. Devadze, and A. Jutman, "Asynchronous fault detection in IEEE P1687 instrument network," in *IEEE 23rd North Atlantic Test Workshop (NATW)*, 2014, pp. 73–78.

[23] K. Petersen, D. Nikolov, U. Ingelsson, G. Carlsson, F. Zadegan, and E. Larsson, "Fault injection and fault handling: an MPSoC demonstrator using IEEE P1687," in *IEEE International On-Line Testing Symposium (IOLTS), 2014*, 2014, pp. 170–175.

[24] P. B. Geiger and S. Butkovich, "Boundary-scan adoption - an industry snapshot with emphasis on the semiconductor industry," in *International Test Conference*, 2009.

[25] (2014, April) I2C-bus specification and user manual. [Online]. Available: http://cache.nxp.com/documents/user_manual/UM10204.pdf

[26] M. Ware, K. Rajamani, M. Floyd, B. Brock, J. C. Rubio, F. Rawson, and J. B. Carter, "Architecting for power management: the IBM® POWER7™approach," in *The Sixteenth International Symposium on High-Performance Computer Architecture (HPCA)*, 2010.

[27] A. J. S. Escobar, J. M. da Silva, and M. Correia, "An i2c based mixed-signal test and measurement infrastructure," in *International Mixed-Signals, Sensors and Systems Test Workshop (IMS3TW)*, 2014.

[28] (2016, May) UltraScale architecture system monitor. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug580-ultrascale-sysmon.pdf

[29] "IEEE standard test access port and boundary-scan architecture," *IEEE Std 1149.1-2001*, 2001.

[30] J. Aerts and E. J. Marinissen, "Scan chain design for test time reduction in core-based ICs," in *International Test Conference (ITC)*, 1998, pp. 448–457.

[31] S. P. Morley and R. A. Marlett, "Selectable length partial scan: a method to reduce vector length," in *International Test Conference (ITC)*, 1991, pp. 385–392.

[32] S. Narayanan and M. A. Breuer, "Reconfiguration techniques for a single scan chain," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 6, pp. 750–765, June 1995.

[33] L. D. Whetsel, "Hierarchical scan selection," Oct. 3 1989, US Patent 4,872,169.

[34] J. K. Ousterhout and K. Jones, *Tcl and the Tk toolkit*, 2nd ed. Upper Saddle River, NJ: Addison Wesley, 2010.

[35] T. Waayers, R. Morren, and R. Grandi, "Definition of a robust modular SOC test architecture; resurrection of the single TAM daisy-chain," in *International Test Conference (ITC)*, 2005.

[36] K. Posse, A. Crouch, J. Rearick, B. Eklow, M. Laisne, B. Bennetts, J. Doege, M. Ricchetti, and J.-F. Cote, "IEEE P1687: toward standardized access of embedded instrumentation," in *International Test Conference (ITC)*, 2006.

[37] V. Sheshadri, V. D. Agrawal, and P. Agrawal, "Power-aware SoC test optimization through dynamic voltage and frequency scaling," in *International Conference on Very Large Scale Integration (VLSI-SoC)*, 2013, pp. 102–107.

[38] G. L. Craig, C. R. Kine, and K. K. Saluja, "Test scheduling and control for VLSI built-in self-test," *IEEE Transactions on Computers*, vol. 37, no. 9, pp. 1099–1109, Sep 1988.

[39] R. Baranowski, M. Kochte, and H.-J. Wunderlich, "Modeling, verification and pattern generation for reconfigurable scan networks," in *International Test Conference (ITC)*, 2012.

[40] R. Baranowski, M. A. Kochte, and H.-J. Wunderlich, "Scan pattern retargeting and merging with reduced access time," in *European Test Symposium (ETS)*, 2013, pp. 39–45.

[41] K. Yamasaki, I. Suzuki, A. Kobayashi, K. Horie, Y. Kobayashi, H. Aoki, H. Hayashi, K. Tada, K. Tsutsumida, and K. Higeta, "External memory BIST for system-in-package," in *International Test Conference (ITC)*, 2005.

[42] A. Carbine and D. Feltham, "Pentium(R) Pro processor design for test and debug," in *International Test Conference (ITC)*, 1997, pp. 294–303.

[43] A. Margulis, D. Akselrod, M. Ricchetti, and E. Rentschler, "Evolution of graphics northbridge test and debug architectures across four generations of AMD ASICs," *IEEE Design & Test*, vol. 30, no. 4, pp. 16–25, Aug 2013.

[44] S. S. Nuthakki, R. Karmakar, S. Chattopadhyay, and K. Chakrabarty, "Optimization of the IEEE 1687 access network for hybrid access schedules," in *VLSI Test Symposium (VTS)*, 2016.

[45] F. Ghani Zadegan, U. Ingelsson, G. Carlsson, and E. Larsson, "Design automation for IEEE P1687," in *Design, Automation & Test in Europe Conference (DATE)*, 2011.

[46] R. P. Grimaldi, *Discrete and combinatorial mathematics*. Pearson Education, 2004, ch. 12, pp. 609–614.

[47] A. Tšertov, A. Jutman, S. Devadze, M. S. Reorda, E. Larsson, F. G. Zadegan, R. Cantoro, M. Montazeri, and R. Krenz-Baath, "A suite of IEEE 1687 benchmark networks," in *International Test Conference (ITC)*, 2016.

[48] E. J. Marinissen, V. Iyengar, and K. Chakrabarty, "A set of benchmarks for modular testing of SOCs," in *International Test Conference (ITC)*, 2002, pp. 519–528.

[49] F. Ghani Zadegan, U. Ingelsson, G. Asani, G. Carlsson, and E. Larsson, "Test scheduling in an IEEE P1687 environment with resource and power constraints," in *Asian Test Symposium (ATS)*, 2011, pp. 525–531.

[50] S. Keshavarz, A. Nekooei, and Z. Navabi, "Preemptive multi-bit IJTAG testing with reconfigurable infrastructure," in *International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2014, pp. 293–298.

[51] M. Portolan, B. Van Treuren, and S. Goyal, "Executing IJTAG: are vectors enough?" *IEEE Design & Test*, vol. 30, no. 5, pp. 15–25, Oct 2013.

[52] Y. Fkih, P. Vivet, B. Rouzeyre, M.-L. Flottes, G. Di Natale, and J. Schloeffel, "2D to 3D test pattern retargeting using IEEE P1687 based 3D DFT architectures," in *Computer Society Annual Symposium on VLSI (ISVLSI)*, 2014, pp. 386–391.

[53] M. Portolan, "A novel test generation and application flow for functional access to IEEE 1687 instruments," in *European Test Symposium (ETS)*, 2016.

[54] A. Ibrahim and H. G. Kerkhoff, "Analysis and design of an on-chip retargeting engine for IEEE 1687 networks," in *European Test Symposium (ETS)*, 2016.

[55] (1999, March) Serial Vector Format Specification. [Online]. Available: http://www.asset-intertech.com/eresources/svf-serial-vector-format-specification-jtag-boundary-scan

[56] V. Iyengar and K. Chakrabarty, "System-on-a-chip test scheduling with precedence relationships, preemption, and power constraints," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 9, pp. 1088–1094, sep 2002.

[57] Z. Michalewicz, *Genetic algorithms + data structures = evolution programs*, 3rd ed. Berlin : Springer, 1996.

[58] T. Lengauer and R. E. Tarjan, "A fast algorithm for finding dominators in a flowgraph," *Transactions on Programming Languages and Systems*, vol. 1, no. 1, July 1979.

[59] A. Bouajila, A. Lakhtel, J. Zeppenfeld, W. Stechele, and A. Herkersdorf, "A low-overhead monitoring ring interconnect for MPSoC parameter optimization," in *International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, 2012.

[60] S. Madduri, R. Vadlamani, W. Burleson, and R. Tessier, "A monitor interconnect and support subsystem for multicore processors," in *Design, Automation & Test in Europe Conference (DATE)*, 2009, pp. 761–766.

[61] TSMC (2016, Dec.), "65nm technology." [Online]. Available: http://www.tsmc.com/english/dedicatedFoundry/technology/65nm.htm

Index