MASTERC

SmartScan
full scan benefits for partial scan cost

Verhaegh, J.F.H.

Award date:
1994

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
SmartScan:

Full Scan benefits
for
Partial Scan cost

by
J.F.H. Verhaegh

Supervisor: Prof. Ir. M.T.M. Segers
Advisor: Ir. P.W.M. Merkus

October 1994
SmartScan:

Full scan benefits for partial scan cost

Abstract

To cope with testing problems for large and complex logic circuits, it is widely acknowledged that one has to partition the circuit into independently testable blocks, and apply structural Design-For-Testability (DFT) techniques. A widely adopted DFT technique is scan design. With this technique, all (full scan) or some (partial scan) of the memory elements in a circuit are replaced by their scannable variant, and these memory elements are connected in such a way that they form one or more scan paths. A scan path is a collection of memory elements that can be put in test mode, in which case they line up to form a shift register. The use of scan paths greatly improves the testability of a design.

In this report a special partial scan method, called smart scan, is described. With this partial scan method the benefits of full scan are obtained for the lower cost of partial scan. With smart scan the selection of flip-flops, which are to be replaced by their scannable variant, is such that the fault coverage, compared to full scan, remains equally high. In contradiction to most partial scan methods, which use a sequential test pattern generator, with smart scan use can be made of a combinatorial test pattern generator.

A partial scan tool called SmartScan has been developed which, according to the smart scan theory, selects memory elements which must be replaced by their scannable variant for test purposes. Results, coming from experiments with this tool, show that the smart scan test method is very promising. These experiments showed an average reduction of 50% in test time for smart scan compared to the full scan test method. The average reduction in number of flip-flops that need to become scannable for test purposes, when smart scan was used, was 15% compared to full scan. In some cases testing with smart scan resulted in a reduction of the fault coverage compared to full scan. This reduction can be ascribed to redundancy in the circuit under test. The test pattern generation times always remained acceptable with smart scan.

Keywords

scan design, full scan, partial scan, smart scan, scan path, design-for-testability, testability, SmartScan, retiming, BALLAST

© Philips Electronics N.V.
Preface

This work was performed in partial fulfilment of the requirements to become Master of Electrical Engineering at the Eindhoven University of Technology. It was performed in the business unit Electronic Design and Tools (ED&T) at the Philips Research Laboratories from February up to October 1994.

Acknowledgements

First of all I would like to thank both my mentor Paul Merkus and Erik Jan Marinissen for their valuable support during my work at Philips. Furthermore, I would like to thank Albert van der Werf for his useful support regarding the retiming theory and the tool ScanLab. Also, I would like to thank Krijn Kuiper, Hans Bouwmeester, Rudi Stans, Richard Morren, Steven Oostdijk, Robert Arons and the other members of the group for their support and guidance. A special word of thanks is due to my fellow students Andries Hogenhuis and Roy Theeuwen, for the many fruitful discussions concerning our projects, and their pleasant company during the months I spent at Philips ED&T. Finally, I would like to thank my supervisor Prof. Ir. René Segers for giving me the opportunity to carry out this project in his group.

Philips Research Laboratories,

Eindhoven, October 1994
# Contents

1 Introduction

2 IC design and testing
   2.1 IC design ........................................ 13
   2.2 IC test .......................................... 15
   2.3 Structure test .................................... 16

3 Scan test
   3.1 Design for testability ............................. 19
   3.2 Introduction to scan test .......................... 20
   3.3 Advantages and disadvantages of scan test ........ 21

4 Partial Scan
   4.1 Introduction ...................................... 23
   4.2 Scan flip-flop selection ........................... 23
   4.3 Partial Scan approaches ........................... 24
   4.4 Advantages and disadvantages of partial scan ..... 24

5 The smart scan theory
   5.1 Theory ........................................... 27
      5.1.1 Definition of a pipeline ....................... 27
      5.1.2 Circuits with only pipeline flip-flops ........ 27
      5.1.3 Circuits with pipeline and non-pipeline flip-flops . . . 28
   5.2 Test Pattern Generation ........................... 31
   5.3 Test vector application ............................ 32
   5.4 Smart scan compared to other partial scan methods and full scan . 33

6 SmartScan
   6.1 SmartScan and Inscan ............................. 35
   6.2 The development track of SmartScan ............... 36

7 ScanLab
   7.1 The basics of retiming synthesis ................... 37
      7.1.1 Introduction .................................. 37
      7.1.2 A basic rule of retiming synthesis ............ 37
      7.1.3 A simple example .............................. 38
      7.1.4 The delta time shape ......................... 40
   7.2 Retiming-based scan flip-flop selection ............ 42
   7.3 ScanLab based on Retlab ........................... 43
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.3.1</td>
<td>Introduction</td>
<td>43</td>
</tr>
<tr>
<td>7.3.2</td>
<td>The causality constraint</td>
<td>43</td>
</tr>
<tr>
<td>7.3.3</td>
<td>Mapping the constrained network on a graph</td>
<td>44</td>
</tr>
<tr>
<td>7.3.4</td>
<td>The ScanLab constraints</td>
<td>44</td>
</tr>
<tr>
<td>7.4</td>
<td>Mapping the graph back to a network</td>
<td>46</td>
</tr>
<tr>
<td>8</td>
<td><strong>Experimental SmartScan based on ScanLab</strong></td>
<td>47</td>
</tr>
<tr>
<td>8.1</td>
<td>Introduction</td>
<td>47</td>
</tr>
<tr>
<td>8.2</td>
<td>The flow in abstracted form</td>
<td>47</td>
</tr>
<tr>
<td>8.3</td>
<td>Building blocks of the flow in detail</td>
<td>49</td>
</tr>
<tr>
<td>8.3.1</td>
<td>Pattern generation on ‘fooling’ circuit</td>
<td>49</td>
</tr>
<tr>
<td>8.3.2</td>
<td>Building the SmartScanned netlist</td>
<td>51</td>
</tr>
<tr>
<td>8.3.3</td>
<td>Test protocol generation</td>
<td>53</td>
</tr>
<tr>
<td>8.3.3.1</td>
<td>Example of a test protocol file</td>
<td>53</td>
</tr>
<tr>
<td>8.3.3.2</td>
<td>Creation of the test protocol file</td>
<td>55</td>
</tr>
<tr>
<td>8.3.4</td>
<td>Test vector generation</td>
<td>60</td>
</tr>
<tr>
<td>8.3.5</td>
<td>Simulation of circuit test</td>
<td>61</td>
</tr>
<tr>
<td>8.4</td>
<td>The total experimental SmartScan flow in detail</td>
<td>61</td>
</tr>
<tr>
<td>9</td>
<td><strong>Experimental SmartScan compared to Full Scan</strong></td>
<td>63</td>
</tr>
<tr>
<td>9.1</td>
<td>Results on the ISCAS’89 benchmark circuits</td>
<td>63</td>
</tr>
<tr>
<td>9.1.1</td>
<td>Sequential redundancy</td>
<td>63</td>
</tr>
<tr>
<td>9.1.2</td>
<td>Drop in number of test patterns for smart scan compared to full scan</td>
<td>64</td>
</tr>
<tr>
<td>9.1.3</td>
<td>Further discussion of the results on the ISCAS’89 benchmarks</td>
<td>65</td>
</tr>
<tr>
<td>9.2</td>
<td>Results on the Melzonic benchmarks</td>
<td>67</td>
</tr>
<tr>
<td>9.3</td>
<td>Results on the Fadic123 benchmark</td>
<td>68</td>
</tr>
<tr>
<td>9.4</td>
<td>Conclusions on results with the experimental SmartScan Tool</td>
<td>69</td>
</tr>
<tr>
<td>10</td>
<td><strong>Implementation of ScanLab on the new datastructure</strong></td>
<td>71</td>
</tr>
<tr>
<td>10.1</td>
<td>The RetLab class structure</td>
<td>71</td>
</tr>
<tr>
<td>10.2</td>
<td>The SmartScan class structure</td>
<td>72</td>
</tr>
<tr>
<td>10.3</td>
<td>The SmartScan program flow</td>
<td>74</td>
</tr>
<tr>
<td>10.4</td>
<td>Mapping the constraints on the graph</td>
<td>77</td>
</tr>
<tr>
<td>10.5</td>
<td>Implementing the mapping of the SmartScan constraints on the graph</td>
<td>80</td>
</tr>
<tr>
<td>10.6</td>
<td>Updating the flip-flop insertion algorithm</td>
<td>82</td>
</tr>
<tr>
<td>10.6.1</td>
<td>The RetLab flip-flop insertion algorithm</td>
<td>82</td>
</tr>
</tbody>
</table>
10.6.2 The SmartScan flip-flop insertion algorithm. 85
10.6.3 Implementation of the SmartScan flip-flop insertion algorithm 87

11 Optimising SmartScan 91

11.1 Introduction 91

11.2 The BALLAST methodology for structured Partial Scan design 92

11.2.1 B-structures 92

11.2.2 Scan design using B-structures. 93

11.2.3 Implementation of the scan path. 94

11.3 The BALLAST method compared to the SmartScan method. 97

11.4 Optimising SmartScan for use of BALLAST scan path implementation method 98

11.4.1 Determining the distances in the SmartScanned circuit 98

11.4.1.1 Theory 98

11.4.1.2 Implementation 103

11.4.2 Adding the SmartScan properties to the netlist 106

11.4.2.1 Theory 106

11.4.2.2 Implementation 106

12 SmartScan compared to ScanLab 109

12.1 Introduction 109

12.2 Results on the ISCAS'89 benchmark circuits 109

12.3 Results on the Melzonic benchmarks 112

12.4 Results on the Fadic123 benchmark 113

12.5 Conclusions on the results achieved with SmartScan 114

13 Future optimisations 115

13.1 Future optimisations on SmartScan 115

13.1.1 Flip-flop placement 115

13.1.2 Recognition of shift registers which contain combinatorial elements 117

13.1.3 Optimisation on the SmartScan selection algorithm 118

13.2 Optimisation of InScan 119

13.2.1 One chain per distance 119

13.2.2 BALLAST scan path implementation 121

14 Conclusion 123

© Philips Electronics N.V.
1 Introduction

Integrated Circuits (ICs) have been growing continuously in number of transistors mounted on them and complexity. Due to this, the probability of both design errors and manufacturing faults has increased. Manufacturing faults result in defects on the IC, while design errors result in an undesired functionality of the chip. By checking the design at several stages of the design flow, design errors can be prevented. However, preventing manufacturing faults is impossible. Typically, the yield of the manufacturing process is between 40 and 80 percent. To ensure that only zero-defect ICs will be delivered at the market, every single IC must pass several tests. One such test is the structure test. By performing a structure test all structures, created on the silicon surface, are tested for correctness. Because of the enormous amount of possible faults and the limited accessibility of parts of the IC via its pins, structure testing can become very problematic.

The requirements that must be met for a structure test are strict. The test should be fast because it is performed on every individual IC and because test costs are strongly related to the time the IC is on the tester. The test must have a high fault coverage since customers do not accept faulty ICs. Furthermore, it is important that the test can be generated in a short period of time to prevent lengthy design times. These requirements can only be met if IC testability is already taken into account during the design phase. This is referred to as design for testability (DFT).

In general DFT techniques are used to improve the controllability and observability of the circuit. This way test patterns can more easily reach isolated parts of the IC and responses can be more easily captured. In this report we are concerned with the DFT technique that is based on scan test.

When the scan test DFT technique is used during the design phase the testability of the design improves drastically by creating an extra test mode for the design. In normal mode, the design operates just as it is supposed to, according to its specification. In test mode, the memory elements will all be connected into one or more scan paths. To achieve this, all memory elements must be replaced by versions that are equipped with such a test mode (this version is called scannable variant). A scan path is a shift register that is only active during test mode. Test patterns can be shifted in this register and applied to the rest of the circuit, which now only contains combinatorial logic. After applying the test patterns during normal mode, the responses can be captured at the primary outputs of the circuit and the inputs of the memory elements. Switched back in test mode, the captured responses in the memory elements can be shifted out of the scan path and gathered by the tester. In this way, test pattern generation has only to be done for the combinatorial part of the circuit, this eases the process of structure testing very much.

When scan test, also referred to as full scan, is used, all memory elements are replaced by a scannable variant. This variant has a few extra pins and is somewhat larger than the original memory element, due to the implementation of the test mode (typically by putting a multiplexer in front of the memory element). A lot of memory elements are used in the design, the
cost of this replacement concerning silicon area might be bigger than the designer is willing to accept. Also, the replacement could have impact on the timing behaviour of the design, since the scannable variant has typically a greater delay than the original memory element, due to its implementation with a memory element in the critical path. Therefore it is worthwhile to improve the concept of full scan, to minimise its disadvantages. This then leads to the concept of partial scan.

When applying partial scan, not all memory elements are replaced by variants with a test mode. This results in less silicon area overhead due to the implementation of scan paths. In general, this will however influence the benefits of full scan, e.g., the testability may decrease, which results in longer test pattern generation times and a lower fault coverage. The direction in which the semiconductor market moves is however towards a higher quality level. A partial scan method has to be developed which keeps the benefits of full scan but reduces its costs. This leads to smart scan.

In this report a scan method called smart scan is described. Smart scan is a special form of partial scan. When smart scan is applied, like with partial scan, not all memory elements are replaced by variants with a test mode. In the smart scan method these memory elements are however selected in such a way that the testability of the circuit does not decrease compared to full scan.

This report also deals with the development of SmartScan. SmartScan is a software tool that prepares a design for scan test, in particular smart scan. In contradiction to many other partial scan tools SmartScan thus performs partial scan while keeping all the benefits of full scan. First an experimental version of the SmartScan tool has been created to be able to test the theory of smart scan on real life circuits. After analysing the results achieved by this tool it has been decided to proceed with the further development of this tool. In order to have a solid foundation on which future developments can be build the heart of the experimental SmartScan tool, called ScanLab, was built on a new data structure. On this new data structure a first optimisation has been implemented.

This report comprises the following chapters:

Chapter 2 gives a brief introduction to the testing of ICs.

Chapter 3 addresses the subject of scan test.

Chapter 4 is a brief overview of the partial scan methods currently known.

Chapter 5 discusses the theory behind smart scan.

Chapter 6 is a brief overview of the SmartScan development track.

Chapter 7 describes the flip-flop selection tool called ScanLab, which forms the basis of SmartScan.

Chapter 8 gives an extensive treatment of how a flow of existing and developed tools is build up to make an experimental SmartScan version.
Chapter 9 compares the results achieved by making a circuit testable with the *smart scan* method to the *full scan* method.

Chapter 10 addresses the implementation of ScanLab on a new data structure. Parallel to this action the toolname ScanLab is changed into SmartScan.

Chapter 11 discusses the development and implementation of the first optimisation on SmartScan.

Chapter 12 compares the results achieved by ScanLab to the results achieved by SmartScan.

Chapter 13 introduces several methods to optimise SmartScan and the tools directly related to SmartScan.
2 IC design and testing

Before an IC can be delivered to a customer, its correctness must be determined. This implies that each individual IC must operate conform to its specifications. Since the design of ICs is done in a number of phases, it must be checked that for each phase no errors are introduced (an error meaning a difference between implementation and specification). In this chapter we will first discuss the IC design process in more detail. Secondly, we will take a look at the IC testing process, which is closely related to the design process.

2.1 IC design

In the IC design trajectory we distinguish several phases [Woudsma 90]. They range from requirement specification via function specification and structure specification to finally the layout. The layout is used in the manufacturing process to make the IC. These phases are depicted on the left hand side of figure 2-1.
**Requirement specification.** First, an informal description of an IC will be drawn up, using a natural language, specifying the desired requirements. This specification describes both what the IC should do, formulated in behavioural terms (the functional behaviour), and under which conditions (e.g., environmental and parametric constraints).

**Function specification.** The requirement specification is transformed into a function specification. Here one specifies what the architecture of an IC would be and which functions it should perform. Also, the necessary top level modules and their interactions are determined. This specification is formal. One could, for instance, think of a VHDL description of a design being the function specification.

**Structure specification.** The function specification is then transformed into a structure specification. In this phase the modules and interconnections resulting from the function specification are worked out in more detail to the level of basic building blocks. This models the electrical connectivity. An example of this specification is an EDIF netlist description of a design.

**Layout.** The structure specification is then transformed into a layout. This specifies the placement of the different building blocks, and maps these blocks and their interconnections to polygons, describing the topology of the chip. An example of this specification is GDS II.

**Product.** The layout is used to actually manufacture a silicon version of the IC in a foundry.

The trajectory of creating an IC is very complex and highly error prone. Therefore, precautions must be taken to guarantee that the delivered IC conforms to its specification. To accomplish this, verification is used during the design trajectory, and testing is used during the test trajectory (after production of the IC).

To make the distinction between verification and testing clear, we will describe these terms below:

- **Verification**
  Comparing the results of two successive phases with each other is called verification. It is denoted by the upwards pointing arrows in the design trajectory in figure 2-1. As can be seen in this figure, verification is not done after the transformation from requirement specification to function specification. This verification step can be done, but because the requirements are stated informally, it must be performed by hand.

  During the design trajectory, a higher-level description was transformed into a lower-level description. To verify whether this step was performed correctly, the opposite action is taken. That is, given the lower-level description, a higher-level description is extracted. This extracted higher-level description is then compared with the higher-level description used during the design trajectory. Inconsistencies indicate errors.

  For instance, a composition of transistors can be transformed into a composition of logic AND and OR gates by means of verification. This can then be compared to the
gate level description that we already had at this level in the design stage. The advantage of verification is that it can be done prior to the manufacturing of the chip, and that it doesn’t need any stimuli, i.e., it can be done exhaustively.

- **Testing**
  
  When we have created the layout of a design, the chip can be produced. During the production process, a substantial part of the ICs will become defective. Therefore, each individual IC must be checked to see if it operates according to its specification. During testing, certain stimuli are presented to the input pins of the IC and responses at the outputs are collected and checked against the expected behaviour. An advantage of testing is that it can be done for all design stages, which can be seen in the right hand side of figure 2-1, where the test trajectory is depicted. Disadvantages are that testing can only be done after the IC is manufactured, and that for large ICs it cannot be done exhaustively. This is because an exhaustive test means that we should test all possible states of an IC, and for every state we have to test all possible input combinations. For an IC with $N$ flip-flops and $P$ inputs this means that we have to test $2N \cdot 2^P$ different states. If there are only 50 flip-flops and 10 inputs, testing this very small chip at 100 MHz would already take about 350 years....

This document is only concerned with the second item, namely testing. Below we give more detail on the testing of ICs.

### 2.2 IC test

Since IC production yields are typically between 40% and 80%, tests should be applied to every IC produced, in order to meet quality requirements. Therefore, the IC design phases are followed by extensive testing. This testing can also be divided into several phases. In general we can state that every phase in the design trajectory has an equivalent phase in the test trajectory [Beenker 90, Claasen 89], see also figure 2-1. Each test phase has a different goal, but they all have in common that they increase the confidence that the IC has been manufactured according to the original specification. The following test phases are depicted on the right hand side of figure 2-1:

- **Layout testing.** Layout testing is not done very extensively. This originates from the fact that matching a layout with the specification is practically not possible at this moment. What can be and is done is checking if all layout masks were correctly aligned when producing the chip. This checking can be easily done right after production by checking if certain markers, that were present on each mask, are well aligned on the chip.

- **Structure testing.** Structure tests look for defects that result in an incorrect behaviour and are caused by the IC production process. It should be applied on every IC produced, because each single IC may contain structural faults. This is the main reason that for structure testing limitation of the test time is very important.

In the trade-off between test time and the possibility of an incorrect IC passing the test, normally a (high) number of test patterns are applied. These test patterns are generated by a test
pattern generator according to certain fault models. Fault models are used to define the meaning of "fault". Many real faults, such as shorts and opens, can be modelled by faults defined by such a fault model, but generally a fault model will never cover all possible faults. Examples of fault models are the stuck-at fault model and the bridging-fault fault model. Structure testing (only) requires knowledge of the structure. This implies that test patterns can be generated automatically, based upon a chosen fault model.

**Function testing.** Test patterns for function testing are usually produced by the designer of the IC. They include tests at the critical ends of the function specification, and will normally only be applied to a few samples of the ICs produced, because structure testing should already have proven that the IC structure corresponds with the structure specification. Hence, this test is only needed to assure that the ICs produced are in conformity with the function requirement. This kind of testing uses a non-systematic approach to produce the test patterns (mostly manually by the designer). Function testing requires knowledge of the function of the design. This implies that the designer must produce the test patterns.

**Application mode testing.** In this test the IC is installed in an application environment, or software is used to model such an environment. This test examines the correctness of the IC design in its application and such proves that the IC is suitable for such an application. Also, a characterisation test can now be performed. This test aims at varying the performance of the circuit under varying environmental and electrical conditions (for example temperature, voltage and humidity). During this phase the actual electrical specification of the circuit can be determined. Characterisation is also called "performance testing". Application mode testing requires knowledge of the application. One has to know the specific application to be able to build or simulate it.

The area of interest to us in this thesis is structure testing. Therefore, we look at this kind of testing more closely in the following section.

### 2.3 Structure test

The large density of modern circuits results in an enormous amount of possible fault cases. The main problem in structure testing is the question how to detect such faults, given the limited accessibility via the IC pins.

A naive, but straightforward, strategy for structure testing is exhaustive testing. Here all possible test patterns are applied and the responses are collected. These responses can then be compared to the expected responses. The major drawback of this method is that for larger circuits the test would take so much time that it is not suitable for practical purposes. Especially for structure testing this is unacceptable, because the structure test must be applied to every IC produced.

In the case of circuits containing memory elements, i.e., sequential circuits, the problem is even worse. For these circuits the output not only depend on the input but also on the current state of the circuit. In order to test a sequential circuit exhaustively, one has to traverse all internal states of the circuit and for each state all test patterns should be applied.
The intractability of the exhaustive test strategy has led to a search for other, practically more useful, strategies. The next chapter will give an introduction to scan test, one of the strategies that can be used to ease structure testing.
3 Scan test

The previous chapter showed that it is virtually impossible to perform exhaustive structure tests on large sequential designs. This is due to the fact that a lot of test patterns are needed to traverse all internal states of a circuit. Furthermore, the problem remained of how to access structures on the chip via the input pins. These facts result in increasing test complexity, which can be converted into costs associated with the testing process, such as the cost of test pattern generation, the cost of test equipment, and the cost related to the testing process itself, namely the time required to detect and/or isolate a fault. Because these costs can be high (and may even exceed design costs), it is important that they be kept within reasonable bounds. One way to accomplish this is by the process of design for testability (DFT).

3.1 Design for testability

Testability has been defined in the following way [Benn84]:

"An electronic circuit is testable if test-patterns can be generated, evaluated, and applied in such a way as to satisfy pre-defined levels of performance (e.g. detection, location, application) within a pre-defined cost budget and time scale".

Design for testability (DFT) can then be described as the design effort that is specifically employed to ensure that a device is testable.

There are two important attributes related to testability, namely controllability and observability. Controllability is the ability to establish a specific signal value at each node in a circuit by setting values on the circuit's inputs. Observability is the ability to determine the signal value at any node in a circuit by observing its outputs.

Structure testing mainly involves applying test patterns to the circuit that we are testing, and then observing the responses. If no specific DFT technique is used, the entire circuit can only be controlled through its input pins. It may be clear that for a large design (thousands of gates, or more) the controllability will soon become very poor, since the number of input pins is restricted (no more than several tens or hundreds) and structures on the IC may be fairly isolated (not directly accessible from the input pins). Also, the observability will become very poor because of the same reasons and the restricted number of output pins. The poor controllability and observability makes the process of creating test patterns very difficult. This may become so hard that it is not possible anymore to get a high fault coverage within reasonable time and costs. At this point, the decision must be made to either accept a lower fault coverage or apply DFT techniques to increase controllability and observability. Since a lower fault coverage often is not acceptable due to the high quality requirements, the choice then falls on DFT.

There are a number of DFT techniques. Most of them deal with either the re-synthesis of an existing design or the addition of extra hardware to the design. This means that they affect such factors as chip area, I/O pins, and circuit delay. Hence, a critical balance exists between...
the amount of DFT to use and the gain achieved. Test engineers and design engineers usually disagree about the amount of DFT hardware to include in a design. In this report, we will focus only on the DFT technique that is of importance to us, namely scan test. The remainder of this chapter contains an introduction to scan test, and discusses its advantages and disadvantages.

3.2 Introduction to scan test

One of the most popular structured DFT techniques is referred to as scan design [Fujiwara 85]. In figure 3-1 the difference between a non-scan design and a scan design is given. The classical Huffman model of a sequential circuit is shown in figure 3-1(a). This model separates the memory elements (registers) from the rest of the circuit, so that the remaining part of the circuit is combinatorial. The combinatorial logic has a number of primary inputs (PI) and a number of secondary inputs (SI, the outputs of the registers). The output of the combinatorial logic consists of primary outputs (PO) and secondary outputs (inputs to the registers). Since the total circuit is sequential, testing it may be complicated if the circuit is large.

![Figure 3-1: (a) Normal sequential circuit (b) Scan version of sequential circuit](image)

In figure 3-1(b) the scan version of the circuit is shown. The registers are now replaced by scan registers. A scan register is a register that can operate in two modes. In the normal mode, it acts as a normal register, just as the ones that were used in figure 3-1(a). In test mode, the scan register will clock its data in from the 'scan-in' input instead of its normal data input. The value will be present on the 'scan-out' output. The 'test' input determines the mode in which the scan register will operate. Hence, a scan register has three special pins associated with it besides the original pins of a register, namely the 'scan-in', the 'scan-out' and the 'test' pin. The scan registers can now be transformed into one or more scan paths, by connecting the 'scan-out' pin of one scan register with the 'scan-in' pin of the next scan register. This means that the registers now form a shift register when put in test mode. In figure
3-2 this is illustrated. Testing has now become much more easier. Since all memory elements can be easily controlled and observed via the scan path (in test mode), the inputs and outputs of scan registers can be treated as primary inputs and outputs of the combinatorial logic. This means that we are able to shift test patterns in the scan path and apply them to the combinatorial logic. The outputs of the combinatorial logic can be captured at the primary outputs of the circuit and at the inputs of the scan registers. Instead of performing a sequential test on the entire circuit, it now suffices to perform a combinatorial test on the combinatorial logic, together with a test to check that the shift register is operating correctly. These two tests are much easier and faster to generate than the sequential test. Also, the fault coverage will improve by using this method.

![Diagram of scan register concatenation into a shift register](image)

**Figure 3-2: Scan register concatenation into a shift register**

Testing a scan design in this way will be referred to as scan test in this report. In particular, the example above is an example of full scan, i.e., all memory elements are replaced by scan variants. In chapter 4 the situation will be discussed where not all memory elements are replaced by scan variants. For now, when we are discussing scan test, we always mean full scan.

### 3.3 Advantages and disadvantages of scan test

One might think that scan test is the preferred solution for testing a (complex) sequential circuit when reading the previous section. This is however not always true. There are a number of costs to be observed when using scan test. [Bennets 93] gives an overview of the advantages and disadvantages of scan test. Below, we give a summary of some solid arguments [Benn93] uses.

Arguments in favour of scan test are:
• Test pattern generation for the combinatorial parts of the circuit can be done fully automated. Furthermore, the pattern generation software will always find a test for a fault if there is one. Popular test pattern generation algorithms for combinatorial circuits include Podem [Goel 81] and Fan [Fuji83].

• The fault-simulation costs are lower because the fault simulator is needed only for the combinatorial parts of the circuit. The fault coverage should be 100% of all detectable faults targeted by the pattern generator.

• The design debug capabilities by using scan paths to explore the behaviour of the intended circuit are better.

• The design environment stays manageable because of the existence of design tools and rule checkers. Also, there is a strong belief that scan enforces well-behaved clock schemes. Such schemes can reduce timing problems in the final design. The final benefit of a controlled design environment is lower risk of a major design change (= quicker time to market).

• The ability to locate the cause of a defect has increased because of the partitioning through the scan path.

The disadvantages of scan test are:

• Scan test introduces extra silicon and pins. Pins cost money, especially if the need for scan causes an increase in package size.

• Scan memory elements are usually enhanced versions of regular memory elements. The enhancement is normally done by adding a multiplexer function to the front end of the memory element. This extra functionality can be seen to increase propagation delays - hence the potential impact on performance. There are ways to avoid this problem, but the preferred way is that the design library is enhanced to contain dedicated scan cells.

For a more complete discussion of these topics, the reader is referred to [Benn93].

It may be clear that the designer has to weigh these arguments against each other to decide whether or not he should use scan test for his design. In general, it can be stated that when extra silicon, pins and delays are not critical factors, there are no serious reasons why the designer should not choose for scan test. Applying scan test improves the quality of the product, therefore making it a sensible choice for application. When extra silicon, pins or delays are critical, the designer should see if the problem can be worked around in some way, because the advantages of using scan test are clear. Partial scan, which will be discussed further in chapter 4, is a way of reducing the costs that come with scan test, and may therefore be of interest to the designer.
4 Partial Scan

4.1 Introduction

In chapter 3, we have discussed a variant of scan test called full scan. In this method of making a circuit testable all D flip-flops will be replaced by scan flip-flops. In the last section of the previous chapter the advantages and disadvantages of full scan were outlined. If the disadvantages of full scan out-weigh the advantages, it could be interesting to use partial scan. With partial scan only a specific subset of the DFFs used in the design are replaced by SFFs. This has of course an impact on the (dis)advantages of scan test.

4.2 Scan flip-flop selection

Although it is known what partial scan is concerned with, it is still not discussed how it can be implemented. The key problem with partial scan is how to decide which DFFs should be replaced by SFFs, and which shouldn’t. There are many different approaches to how this decision can be made. This problem has been the topic of many journal and conference papers that have emerged over the last years. The various partial scan techniques described in these articles differ in the way they select scan flip flops. These can be categorised into three main methods - testability analysis based, structural analysis based, and test generation based.

In the partial scan methods based on testability analysis, testability heuristics such as controllability and observability, and sequential depth, are applied to the sequential circuit. Flip-flops which have the worst values as measured by the heuristic are selected as scan flip-flops. Trischler [Tris80] described a technique whereby flip-flops which were not easily observable or controllable were included in the incomplete scan path. The effectiveness of this method depends entirely on testability analysis, which may not accurately model the problems faced during test generation. However, this method gave good results for circuits which were not highly sequential, and for circuits in which the complete scan approach resulted in area overheads which were prohibitive.

The second class of heuristics for selection of scan flip-flops is based on a structural analysis of the sequential circuit. Typically, the circuit is collapsed into a directed graph, whose vertices represent flip-flops, Primary Inputs (PIs), or Primary Outputs (POs), and arcs represent combinatorial paths. The common ground for those heuristics is the recognition of the fact that the complexity of test generation grows exponentially with the length of the directed cycles in the graph, while it grows only linearly with the sequential depth. In [Chen 89], heuristics were used to first select a minimal set of flip-flops to eliminate cycles, and once the graph was acyclic, flip-flops were selected to reduce the sequential depth of the circuit (defined as the longest path from the graph input to the graph output). Thus, one can bound the longest test sequence generated by a sequential test generator. Other strategies tried to minimise the number of reconvergent paths of unequal lengths.
Some of the methods based on test generation are described in [Agra87] and [Ma88]. The methodology in [Agra87] relies heavily on the initial phase where functional test are used to eliminate many faults. In [Ma88], test were generated for a large number of faults in a sequential circuit. For each undetected fault, a set of flip-flops was found, such that making the flip-flops in that set observable or controllable made the fault detectable. The incomplete scan path the minimal subset of memory elements which influenced the easy detection of as many faults as possible. The two methods described above gave good coverage and reasonable test lengths, but they incorporated the cost of test generation as well as the cost of calculating minimal sets, and are thus computational intensive.

The fourth method for selection of scan flip-flops is based on a timing analysis of the sequential circuit. Flip-flops in critical timing paths may not be made scannable. Because the delay of a scan flip-flop is larger than the delay of the D flip-flop, which it replaces, making a D flip-flop in a critical timing path scannable can lead to a decrease in the maximum clock frequency at which the circuit can work. Making only the flip-flops not in critical timing paths scannable won’t have an impact on the clock frequency at which the circuit operates. Due to the fact that in this method no testability measures are used to select the flip-flops to be made scannable the testability can be very poor.

4.3 Partial Scan approaches

The present approaches to partial scan are based upon testability measures, test pattern generation or structural analysis. First, testability measures can be used to select scan flip-flops for the scan path. This method however requires a sequential test pattern generator to create test patterns for the remaining circuit, which is not fully combinatorial. Other similar techniques use heuristics and a cost function to select flip-flops. The effectiveness of these approaches highly depends on the quality of the testability measures with respect to the test pattern generator used. Other partial scan methods start with a set of functional vectors to detect as many faults as possible. Then combinatorial test pattern generation is used to find a set of flip-flops that provide sufficient extra controllability and observability to detect the remaining faults. These methods heavily rely on the initial set of functional vectors and are computationally very expensive. Some structural approaches have been proposed which seem more promising and give good results. In these approaches flip-flops are selected in such a way that the remaining circuit is acyclic. It has been shown that feedback loops are the major burden for sequential test pattern generation. If these cycles are broken by scan elements, then a combinational test pattern generator can be used to obtain the test vectors for the remaining circuit.

4.4 Advantages and disadvantages of partial scan

Most forms of partial scan have in common that they provide a trade-off between the benefits and costs of Full Scan. First the extra silicon area that partial scan introduces will be less than introduced by Full Scan. Not all flip-flops have to be replaced by their scannable variants. The extra wiring to connect the scan flip-flops in one or more scan chains will be less because the number of scan flip-flops is less compared to Full Scan. Also the wiring to con-
trol the scan multiplexers is less.

The total decrease in area when partialinstead of full scan is used depends on the percentage of the flip-flops that isn't made scannable. Looking at the total circuit area full scan introduces can be calculated as follows.

When 20% of the circuit area is used by flip-flops and the overhead on each flip-flop for adding a testmode is around 25% (including wiring) then the extra overhead, for testpurposes, on the total circuit area is 0.2*25% = 5%. When a Partail Scan algorithm selects only half of the memory elements to become scannable, only half of the flip-flops will have an overhead for test purposes. The total circuit area overhead will then also be halved compared to full scan. The total overhead on the circuit for test purposes will thus be around 0.5*0.2*25% = 2.5%, so a 2.5% drop in area cost will be achieved.

When not all flip-flops are made scannable, a sequential pattern generation tool may be needed. These tools can only produce a high fault coverage when they are allowed to run endlessly without restrictions on computing resources like CPU time and memory. Sequential pattern generation tools generally work with multiple time frame images. This means that for each time frame an image of the circuit in that time frame must be available. When for example n time frames are needed to generate test patterns the memory usage of such a tool can easily be n times as high as used by a combinatorial pattern generator. When the need for memory exceeds the amount of memory present disk space must be used as virtual memory. This can lead to many swap operations between disk and memory, causing unacceptable high CPU times.

The sequential ATPG tools that are now commercially available can only efficiently generate testpatterns for circuits of limited size. They work well on little theoretical examples but cannot efficiently be used for pattern generation on real life VLSI circuits. It is for this reason that, when Partial Scan is used, test pattern generation should only rely on combinatorial ATPG tools.

The problem with most Partial Scan methods is that they make a trade off between the benefits and the costs of full scan. When, for example, 10% of the flip-flops is made scannable, still a fault coverage of 80% can be reached. However, these kind of fault coverages cannot meet the quality demanded usually by the average customer, such as the consumer electronics industry.
5 The smart scan theory

This report deals with the development of SmartScan. SmartScan must become a tool that selects the flip-flops in a circuit that need to be scannable according to the smart scan method. In the introduction it was explained that smart scan is a special form of partial scan. In the smart scan method, like with partial scan, only a subset of the flip-flops in the circuit is made scannable for test purposes. As depicted in chapter 4 most partial scan methods lead to a decrease in circuit testability compared to full scan. In the smart scan method the flip-flops that need to be made scannable are selected in such a way that the testability, compared to full scan, will not decrease. In this chapter the basic ideas behind the smart scan method are explained.

5.1 Theory

5.1.1 Definition of a pipeline

The term 'pipeline' that is used in this thesis is defined as follows:

A pipeline is a set of memory elements that, when added to or deleted from a network, leaves the functionality of the network unchanged. Only the timing relation between the inputs and outputs is changed as follows. If a pipeline is added (deleted) to (from) a network the variation of all output signals of the network, if a variation in the input signals occurs, is delayed (advanced) by one clock cycle compared to the original circuit.

In this thesis a network is called 'pipelined' when it contains one or more sets of memory elements as defined above.

5.1.2 Circuits with only pipeline flip-flops

Pipelines have a specific architecture, that makes them candidates for scan path optimization. In figure 5-1, an example of a pipelined circuit is given.

![Figure 5-1: Example of a symmetrical pipelined Circuit](image-url)
None of the D-type flip-flops drawn in this figure have to become scannable variants because of the nature of the pipeline: The pipeline does not contain any other sequential elements and there is no feedback of the sequential elements in the pipeline, as can be seen in the figure. Values, applied to the combinatorial logic at the left side of figure 5-1 can be propagated through the pipeline in two clock cycles. After two clock cycles, the resulting values will be present at the outputs of the combinatorial logic at the right side of figure 5-1. Because of the nature of the pipeline, these values will remain stable as long as the input values are stable. The D flip-flops a pipeline is constituted of, do not have to be embedded in a scan chain. As a result, the logic to be tested between the scan chains is not fully combinatorial anymore. However, we can still generate patterns for this logic with a combinational ATPG tool. This can be done by modifying the logic in such a way that the ATPG tool does not recognise that logic in between the two scannable registers in not purely combinational ("ATPG fooling"). The non-scannable flip-flops are replaced by buffers. Apart from clock lines and their timing behaviour, buffers and D flip-flops have equal characteristics with respect to testing. Both have one input and one output and copy the data on the input to the output. Compared to the stuck-at fault model, a buffer as well as a D flip-flop can have stuck-at faults at input or output. A flip-flop can be considered "a buffer that waits for the clock".

5.1.3 Circuits with pipeline and non-pipeline flip-flops

Sometimes, a pipeline might be disturbed by the presence of a sequential element that is not part of the pipeline. Consider the following case, depicted in figure 5-2.

![Figure 5-2: Example of a non-symmetrical pipeline in a circuit](image)

The middle stage of the pipeline contains a D flip-flop besides some combinatorial logic. This flip-flop makes the pipeline asymmetrical, i.e., the results of the middle stage are not all available at the next clock cycle. The lower part of the pipeline needs an extra clock cycle to present its result to the next stage. The result after the first clock cycle is depending on the state of the D flip-flop. In order to be able to put the D flip-flop in the middle stage in a known state, hence being able to predict the output of the stage after one clock cycle, we must make it scannable. Therefore, this D flip-flop must be replaced by a scan flip-flop.
Another case is depicted in figure 5-3. In this figure, the pipeline again is disturbed by a sequential element of type D flip-flop in one of its stages. This time however, the output of the sequential element is fed back into the sequential circuit. Now, this stage has become a purely sequential circuit, which cannot be easily tested without special care. Therefore, the D flip-flop must become scannable to be able to perform scan test on the circuit. This is however not enough, because of the fact that we are applying multiple clock cycles to the pipeline in order to shift the test patterns through it. If we would only put the D-type flip-flop in a known state and then apply a number of clock cycles, the circuit remains sequential of nature, and we would have gained nothing. The state of the D-type flip-flop must be frozen during the application of the clock cycles, in order to assure that the circuit can be seen as a combinatorial one. When the D-type flip-flop does not change states, it can be seen as a

![Figure 5-3: Example of a pipelined circuit with sequential loop](image)

"constant generator", independent of the clock, like a combinatorial element. Therefore, the D flip-flop must be replaced by a scannable hold flip-flop. The hold functionality can then be used to freeze the state of the D flip-flop.

The need for a holdable scannable flip-flop can also be explained in another way. In figure 5-4 a part of a circuit, on which the smart scan approach has been applied, is shown. The fat vertical bars represent the flip-flops that are scannable. The thin vertical bars represent flip-flops which are not scannable. In this circuit representation the logic has been left out for clarity. Looking at the figure it can be seen that there are two normal flip-flops between the scan flip-flop at the left and the top right scan flip-flop. Between the left and the bottom right scan flip-flop three normal flip-flops are present. This means that the paths from the left scan flip-flop to its following scan flip-flops are of different length with respect to the number of D flip-flops on it. In the figure the path lengths are denoted in the number of clock cycles need-
ed to propagate data in a scan flip-flop to the following flip-flop. With respect to these differences it has to be made sure that test signals can travel along even the longest of these paths. A solution for this problem is to use holdable scannable flip-flops.

![Figure 5-4: circuit on which SmartScan approach has been applied](image)

When the left scan flip-flop in figure 5-4 is made holdable scannable the differing path lengths can be accounted for. When the data in this flip-flop is held for three clock cycles and after this a normal clock cycle is applied (without holding) both scan flip-flops on the right contain the propagation of the same data that was in the left scan flip-flop. In figure 5-5 the three flip-flop types that can exist in a SmartScan circuit are represented. In this figure it can be seen that the extra area overhead introduced by a holdable scannable flip-flop is roughly twice the overhead produced by a scannable flip-flop.

![Figure 5-5: The different kinds of flip-flops in a SmartScan circuit](image)

Normally however, scan flip-flops are often available as a hard macro. This means that the area \( A_{sff} \) needed by a scan flip-flops is less than the area needed by a D-type flip-flop \( A_{dff} \) plus the area needed by a multiplexer \( A_{mux} \). A holdable scan flip-flop is seldom available as a hard macro. When not available, such a flip-flop must be constructed from a scan flip-flop and a multiplexer. The area needed for a holdable scan flip-flop \( A_{hsff} \) becomes thus the area needed for a scan flip-flop \( A_{sff} \) plus the area needed for a multiplexer \( A_{mux} \). The area overhead needed to make a D-type flip-flop holdable scannable is thus normally more than twice the area overhead needed to obtain a scan flip-flop.
\[
A_{sff} < A_{dff} + A_{mux} \\
A_{hsff} = A_{sff} + A_{mux} > (A_{sff} - A_{dff}) \times 2 + A_{dff}
\]

With respect to test pattern generation for the above two examples, it is possible to use a combinatorial ATPG tool. Here also the non-scannable (pipeline) flip-flops are replaced by buffers, for the same reasons as mentioned above. Furthermore, the D-type flip-flops that need to become scan flip-flop or holdable scannable flip-flop are routed as scan flip-flops in a scan chain. This way a combinatorial ATPG tool can be used.

5.2 Test Pattern Generation

Looking at the depth of the combinatorial logic with which the ATPG tool has to deal it can be seen in figure 5-6 that this is larger than the depth of the logic blocks in the original circuit. Every flip-flop that needn’t be made scannable, when the SmartScan method is used, is replaced by a buffer for test pattern generation purposes. Due to the fact that the execution times of combinatorial ATPG tools depend on the depth of the logic, it can be expected that the time needed to generate appropriate test patterns for this circuit is larger than for its Full Scan counterpart.

![Diagram showing increase in logic depth due to replacement of non-scannable registers by buffers](image)

Figure 5-6: Increase in logic depth due to replacement of non-scannable registers by buffers

The controllability and observability of the circuit do not change, compared to full scan. Also, the fault coverage remains 100% of all detectable faults targeted by the pattern generator. Thus, the testability of the circuit does not deteriorate when using this method over full scan.

In chapter 9 the generation of test patterns on Full Scan and SmartScan circuits is compared. The fault coverage reached by the combinatorial pattern generator is sometimes less for the SmartScan circuit. This drop in fault coverage is caused by a certain type of redundancy in the circuit. This phenomenon will be explained in chapter 9.
5.3 Test vector application

Testing is now not trivial anymore. Due to the replacement of the sequential elements, which need not be made scannable, by buffers, the circuit on which the patterns are generated and the circuit that will be tested aren’t equal. In general, more than one clock cycle must be applied to the circuit to propagate signals entirely through it, in contrast with the situation with full scan, where only one clock cycle was needed for a signal to propagate through the combinatorial logic and get captured in a scan flip-flop. This means that a non-default test protocol is needed, in which is described how many clock cycles are needed for signal propagation. A tool must be developed which generates such a test protocol.

To account for the varying path lengths, as discussed in the previous section, all flip-flops that need to be scannable also need to be made holdable. The distance between the scan flip-flops in a circuit is expressed in the number of clockcycles needed for data to travel from one scan flip-flop to another. From all these distances the maximum, called MaxDist, can be determined. With this MaxDist the protocol, according to which the circuit will be tested, can be set up. Both the Full Scan and the SmartScan test protocol are presented in figure 5-7.

![Figure 5-7: Test protocols for both Full Scan and SmartScan](image)

In both test protocols first the test stimuli are shifted into the scan chain. This shifting takes as many clock cycles as the chain contains flip-flops. After this the test stimuli are applied to the inputs of the circuit after which the actual test takes place. In the case of Full Scan this takes only one clock cycle in which the circuit is put in its normal operating mode. In the case of SmartScan the normal mode is preceded by MaxDist -1 clockcycles in which the scan chain is held in its hold mode and during which the stimuli at the inputs of the circuit aren’t changed. MaxDist denotes the maximum distance in number of clock cycles between two scan flip-flops in the circuit, so MaxDist-1 cycles are needed to compensate for the differences in the various path lengths. To capture the test responses, the scan chain is set for exactly one clock cycle in its normal mode. After the normal mode in both methods, the responses are shifted out in the scan mode.
5.4 Smart scan compared to other partial scan methods and full scan

First of all, the fault coverage that can be achieved with the *smart scan* method will be discussed. In most *partial scan* methods the fault coverage achieved is less than with *full scan*, because less flip-flops are made scannable. Making less flip-flops scannable means that the controllability and observability within the circuit drops and so the fault coverage drops. Also, some *partial scan* methods make use of sequential test pattern generators. These pattern generators only work well for small examples but aren't suitable to generate patterns for large circuits. They will almost always achieve a fault coverage which is less than that which can be achieved with a combinatorial pattern generator.

When the *smart scan* method is used a combinatorial pattern generator can be used, resulting in a high fault coverage. This coverage will be as equally high as in the *full scan* case if the circuit doesn't contain a certain type of redundancy. What sort of redundancy this is will be explained in chapter 9. It can be concluded that when a circuit doesn't contain redundancy the fault coverage with the *smart scan* and the *full scan* method are equal, so in contrast to most *partial scan* methods *smart scan* doesn't trade fault coverage for less scan.

When *smart scan* is used the time needed for the test will be almost always less than with *full scan*. Looking at one test cycle (see figure 5-7) the following can be seen. The duration of a test cycle (D\text{testcycle}) of *full scan* in number of clock cycles is equal to the length of the scan chain (D\text{scanchain}) (while response data is scanned out of the chain new stimuli are clocked in at the same time) plus one for the normal mode. In the *smart scan* case the number of holdcycles needed must be added to this figure. If, with smart scan, the drop in length of the scan chain (D\text{drop}) is larger than the number of hold cycles (D\text{holdcycles}) needed then the testtime for *smart scan* is less than for *full scan*. When a circuit contains one pipeline which consists of two flip-flops a drop in scan chain length of 2 flip-flops is achieved (only one scan chain is used). Because the number of holdcycles can be at most one a drop in testtime is already achieved compared to *full scan*.

\text{full scan: } D_t = D_{\text{scanchain}} + 1
\text{smart scan: } D_t = D_{\text{scanchain}} + 1 + D_{\text{holdcycles}} - D_{\text{drop}}
\text{if } D_{\text{drop}} > D_{\text{holdcycles}} \text{ then a drop in testtime is achieved.}

Because of the fact that with the experimental version of SmartScan the distances that each scan flip-flop has to its subsequent scan flip-flop(s) can’t be determined, all flip-flops which need to be made scannable also need to be made holdable because of the varying distance problem. From section 5.1.3 it can be concluded that a holdable scannable flip-flop introduces at least twice as much area overhead than a scannable flip-flop. In order to make experimental SmartScan, compared to *full scan*, profitable with respect to silicon area, less than half of the flip-flops should be made scannable. To be able to replace holdable scannable flip-flops by scannable variants the distances, as described above, must be calculated. All scan flip-flops which send out their data to one or more scan flip-flops, all at the same distance, need only to be made scannable because they don’t have to account for a difference in distance.

The extra delay, introduced by using scan flip-flops instead of D flip-flops, can now be avoided in the pipeline. This is especially useful, since pipelines are often used to speed up the de-
sign. The longest delay in a stage of the pipeline determines the maximum clock frequency. It is therefore desirable that the scan test functionality does not have an impact on the delays in the pipeline, since this would influence the maximum clock frequency at which the circuit can be run negatively.

A judgement on if and how well the *smart scan* theory works can only be made when this concept is applied to real-life examples. To be able to do this first a experimental SmartScan tool will be built. The heart of this experimental version will consist of a tool called ScanLab which divides the memory elements in the circuit in two groups. One of which must be made scannable for test purposes. This tool will be described in chapter 7. After this discussion the building of the experimental SmartScan with this tool in it will be further discussed.
6 SmartScan

This report deals with the specification and implementation of SmartScan. SmartScan is a program that adds, according to the *smart scan* method, scan information to a design description. Based on this information the scan path insertion tool InScan must add scan paths to a design. This design will then be called 'smart scannable', i.e. prepared for *smart scan*.

This design can be processed in such a way that a combinatorial pattern generator, which recognises the scan path functionality, can create appropriate test patterns for it. Based on the scan information and information on the routing of the scan chains in the circuit a special test protocol can be generated. According to this test protocol the generated test patterns can be applied to test the 'smart scannable' circuit.

6.1 SmartScan and Inscan

Basically, SmartScan is thus a preprocessing tool for InScan. It adds information to a design description so that InScan is able to make the design 'smart scannable'. In figure 6-1 a schematic overview is given of how the tool SmartScan is related to InScan. This figure shows how the design is processed. It is also shown that the user can interact with the tool InScan. The information flow in this figure is not complete. There are more files associated with program execution, but they are not mentioned here to avoid confusion of the reader at this point.

![Figure 6-1: Relation between SmartScan and InScan](image)

© Philips Electronics N.V.
6.2 The development track of SmartScan

In this section an overview is presented on how the SmartScan tool has been developed. Because the rest of this thesis closely follows the development track of SmartScan this section also serves as an overview of the subjects that will be treated in this thesis.

1. Description and test of the tool ScanLab

At Philips Research a software tool which selects, according to the *smart scan* method, the memory elements in a circuit that need to be made scannable was already developed. This tool is called ScanLab and was developed by Ir. Albert van der Werf. A description of this tool will follow in chapter 7.

To be able to give judgement on how well the *smart scan* method performs on real life circuits a test flow was built up around ScanLab. This test flow is described in chapter 8 and is denoted as the experimental SmartScan tool. In this chapter the development of a test protocol generator is also treated. Based on the scan information and information on the routing of the scan chains in the circuit a special test protocol can be generated.

In chapter 9 the results achieved with the *smart scan* method, implemented in ScanLab, are compared to testing with the *full scan* method.

2. Specification and implementation of the tool SmartScan

In chapter 10 it is discussed how the ScanLab tool is implemented on a new data structure. In parallel with this action the name of the tool ScanLab is changed into SmartScan.

The specification and implementation of the first optimisation on the tool SmartScan is described in chapter 11.

In chapter 12 results achieved with the optimised SmartScan tool are presented.

3. Future optimisations

In chapter 13 future optimisations that must be performed on SmartScan and on the tools that are directly related to SmartScan are discussed.
Chapter 7: ScanLab

As mentioned in the previous section, this report deals with the specification and implementation of SmartScan. SmartScan is a tool that prepares a design for a special form of partial scan called smart scan. By adding information to the design description SmartScan selects a subset of memory elements in the design that should be made controllable and observable. The flip-flops that need not to be made scannable are the flip-flops in the circuit that are there for pipeline purposes. All other flip-flops, with the exception of those in a shift register, are made scannable. A retiming algorithm is used to decide whether a flip-flop is a pipeline flip-flop and thus needn’t be made scannable. Due to the fact that retiming can’t change the number of flip-flops in feedback loops, each one of these loops will contain at least one flip-flop and so it is possible to use a combinational test pattern test generator. Based on the existing retiming tool RetLab, which was developed by Ir. Albert van der Werf, the tool ScanLab has been developed (also by Ir. Albert van der Werf). This tool divides the memory elements in a circuit into two groups for purposes of smart scan. The memory elements in one group must all get a testmode the memory elements in the other group stay unchanged.

7.1 The basics of retiming synthesis

7.1.1 Introduction

Because retiming synthesis is a rather innovative and unknown technique, the basics of retiming synthesis will be explained in this section. Two major features are achieved by retiming synthesis. The first feature is the modification of the temporal behaviour of a clocked digital circuit, whilst leaving the functional behaviour unchanged. The second feature of retiming synthesis is the reduction of area which implies the reduction of power dissipation. Also pure combinatorial circuits, like datapaths, can be retimed, because paths with a long delay will be divided by flip-flops into sections with a shorter delay. So retiming synthesis will always yield a synchronous circuit.

Retiming synthesis is achieved by adding, deleting or repositioning flip-flops in a circuit until the desired specification is met. The goal for retiming is to increase the clock frequency of the circuit and/or to reduce the area and dissipation by reducing the number of flip-flops.

7.1.2 A basic rule of retiming synthesis

As described in the introduction of this chapter retiming synthesis will not change the functional behaviour of the circuit, it is allowed to change the timing relationship of signals, what results in postponement or advancement of one or more clock periods.

A basic rule for retiming is that flip-flops may freely be shifted along nodes with the guarantee that the relative timing relationship between branches remains unaltered. If a fork is encountered this may result in an increase of the number of flip-flops, if a junction is encountered this may result in a reduction of the number of flip-flops. The retiming principle
explained here is illustrated in figure 7-1. From left to right we see a flip-flop which is connected to two branches. Shifting this flip-flop over the fork and the combinational logic means that in both branches a flip-flop should be placed.

From right to left we see in figure 7-1 how flip-flops can be shifted over a junction of two branches.

7.1.3 A simple example.

In figure 7-2a we have an example of a FIR filter performing the operation:

\[ o = i + i_{-1} + i_{-2} \]
The input \( i \) is delayed by two flip-flops in order to obtain \( i_{-1} \) and \( i_{-2} \). The path with the longest delay, also called the critical path of this circuit is from the output of \( ff2 \) via \( add1 \) via \( add2 \) to output \( o \), limits the maximal clock frequency. The basic rule for retiming (see section 7.1.2 on page 37) allows us to shift flip-flops across nodes in the circuit, provided that we maintain the same timing relationship between the branches. In figure 7-2b this rule has been used to replace flip-flop \( ff1 \) by two flip-flops \( ff1a \) and \( ff1b \) (splitting). Flip-flop \( ff2 \) has been shifted along the node to a new position. The function of the circuit has not been changed. The next step is pushing flip-flops \( ff1a \) and \( ff1b \) to a position between the adders. Because these 2 flip-flops are referred to in the same time frame they are “joined” back to a single flip-flop (\( ff1 \) in figure 7-2c). This results in the circuit of figure 7-2d, which now only has 1 adder in the critical path. This circuit can operate at a much higher frequency than the original circuit (the maximum clock frequency at which the circuit can operate has increased by a factor 2).

![Timing diagram of the original FIR filter](image)

**Figure 7-3: Timing diagram of the original FIR filter**

In figure 7-3 the original circuit has been illustrated by a timing diagram. From this timing diagram we see that the critical path is twice the delay of the adder. The delay of both adders is just within the clock period, which means that this circuit is running on the highest feasible clock period. If we would have a faster circuit, we can either choose for faster cells, or we can use retiming to speed up the circuit. Since the first proposal is restricted to the speed of the cells we will use retiming. As we saw in figure 7-2 the circuit can be easily retimed using the basic rules of retiming. In the timing diagram of the retimed circuit (see figure 7-4), the critical path is limited by the delay of only one adder. Because the delay in the retimed circuit is

© Philips Electronics N.V. 39
halved, we can use a clock period which is twice as small as the clock period of the original circuit. Note that in this case, the improvement in performance has been achieved without any increase in area.

Figure 7-4: Timing diagram of the retimed FIR filter

7.1.4 The delta time shape

In the previous circuit each change on the input caused a response of the output after a certain combinational delay. This can be represented by a “time shape” diagram of the circuit (see figure 7-5a). If a flip-flop is inserted, the time shape will be like figure 7-5b. If we speak of a delta time shape, this would be 1 when circuit figure 7-5b is created from figure 7-5a. The diagrams can be explained as follows:

A: If a variation in the input signal occurs, an immediate variation in the output signal occurs within one clock tick, but after a certain combinational delay.

B: If a variation in the input signal occurs, a variation in the output signal occurs after one clock tick.
The time shape in figure 7-5 is not the only possible shape. This will be illustrated with the next example which is an arbitrary feedback circuit with a carry line between two adders. If the specification of the design allows the outputs o and u to be valid in a different clock peri-
od as in the original circuit, which is illustrated in figure 7-6b, retiming can change the time shape of this circuit as illustrated in figure 7-6d. The critical path of figure 7-6a is determined by two adders. The retiming synthesizer will insert a flip-flop in the carry line as illustrated in figure 7-6c, which will cause a decrease of the critical path delay. The extra flip-flop in the carry line causes output u to change one clock tick later then output o.)If we take the original time shape (see figure 7-6b as the reference point then the deviation from this point is called the Delta Time Shape (DTS). It is not very difficult to conclude that the DTS of output o equals zero because the time shape after retiming has not been changed in time. The DTS of output u equals one because the time shape has changed one clock tick (see figure 7-6d). If the designer specifically insists on synchronous activation of the outputs, a retiming tool will have to leave the time shape of the circuit untampered.

Generally a circuit with only feedback loops (see figure 7-7) is not retimeable because insertion or removal of flip-flops in this case will cause a different functional behaviour (feedback values would either arrive too soon or too late).

![Figure 7-7: A non-retimeable circuit](image)

### 7.2 Retiming-based scan flip-flop selection

Normally retiming is used in the design of circuits that should operate at a certain (high) clock frequency. When a retiming tool is available the designer can concentrate on the functionality of the circuit, without taking timing constraints into account. After the designer has finished the design it can be retimed by shifting just enough flip-flops into the circuit so that it can operate at the required clock frequency.

When we provide the retiming tool with a very low clock frequency it will shift out the pipeline flip-flops that are incorporated in the design. All the pipeline flip-flops will be shifted out if the specified clock-frequency is low enough. The only flip-flops remaining in the circuit after this retiming action are non-pipeline flip-flops. This selection into two groups was used as a basis for ScanLab.

The problem that rises when retiming is used to partition the flip-flops into two groups, as mentioned above, is that the pipeline flip-flops are shifted out of the circuit and the non-pipeline flip-flops aren't always at the same place in the retimed circuit as they were in the original circuit. With flip-flop selection for partial scan purposes however the placement of flip-flops in the circuit and their number may not be changed. What is needed is a tool that divides
the memory elements of a circuit into two groups but leaves the placement of the flip-flops and their number in the circuit unchanged. At the Philips Research Laboratories such a tool has been developed out of the retiming tool RetLab and is called ScanLab. Both these tools are developed by Ir. Albert van der werf. (The name RetLab is an anagram of Albert!)

ScanLab divides the memory-elements in a design into two groups. The division is made by using two different instantiation names for the flip-flops in the circuit description.

7.3 ScanLab based on Retlab

7.3.1 Introduction

The retiming algorithm is based on expressing the retiming problem as a set of constraints on the delay of paths through the network and on the retiming cells. Because of the fact that for SmartScan purposes the retiming algorithm will be used without timing constraints, only the remaining constraints, called causality constraints, that exist in this situation will be discussed. After the discussion of the retiming constraint, the mapping of the constrained network on the graph is briefly discussed. This has to precede the discussion on the constraints that have to be added on behalf of ScanLab because one of the ScanLab constrains exists due to this graph construction.

7.3.2 The causality constraint

When the retiming problem is stated without timing constraints, then only the causality constraint must be guaranteed by retiming. This means that no negative number of flip-flops may occur on a net.

These causality constraints will result in a network with the amount of flip-flops minimised. This constraint is expressed by the following causality constraint for every situation in the retiming network as shown in figure 7-8.

\[ r(\text{output}_i) - r(\text{input}_j) \leq W(e1) \]

Figure 7-8: The causality constraints

For every input to output relation a constraint has to be generated as shown below, where \( W(e1) \) is the number of flip-flops between \( \text{output}_i \) and \( \text{input}_j \) and retiming value \( r(i) \) is defined as the number of flip-flops which are moved from behind the cell and replaced in front of input \( i \).
Proof: \( W_r(e) = W(e) + r(\text{input}_j) - r(\text{output}_i) \geq 0 \)

\( r(\text{input}_j) - r(\text{output}_i) \geq -W(e) \)

\( r(\text{output}_i) - r(\text{input}_j) \leq W(e) \)

This constraint means that no negative number of flip-flops may occur on any output to input connection after retiming. The number of flip-flops on an output to input connection is also denoted as the weight of this output to input connection. The abovementioned constraint thus guarantees that no negative weight occurs on any output to input connection after retiming.

The total retiming problem without timing constraints is now decomposed in three steps: the first is generating the constraints, the second step is finding a solution to these constraints and the third step is minimising the following cost function.

\[
C = \sum_{e \in \text{Edges}} W_r(e)
\]

The set of constraints can be directly mapped on a graph.

### 7.3.3 Mapping the constrained network on a graph

A graph consists of vertices connected by edges ([Gibb89]). But how to map a network built of cells, nets and ports on a graph? We solved this problem by assuming that every cell has a vertex in the graph. There is a close relation between an edge and a constraint, actually an edge is a representation of a constraint. An edge is a constraint with a field containing the weight value. When all the edges are created between the vertices, the vertices and the edges are given to the retiming algorithm. This algorithm will perform some operations on the graph and will end with a new set of edges and updated fields.

### 7.3.4 The ScanLab constraints

The generation of the ScanLab constraints can be expressed in the same way as the causality constraints. When a circuit must be scanned, the number of scan flip flops on a input to output connection (denoted as the scanweight \( W_r \)) may not be higher than the original number of D flip flops on that connection

\[ r(\text{input}_j) - r(\text{output}_i) \leq 0 \]

Proof: \( W_r(e) = W(e) + r(\text{input}_j) - r(\text{output}_i) \leq W(e) \)

Before the second ScanLab constraint is explained first it must be explained why this constraint exists. In figure 7-9 two example circuits are represented. The fat vertical lines represent flip flops. It can be seen that the number of flip-flops between pins a and b in circuit a is the same as in circuit b. This also holds for a and c in both circuits. Despite the fact that the two circuits aren’t equal, they will be treated as such in the graph because the distances between the various in- and output pins are equal.
Looking at the circuit in figure 7-9b the following can be seen. When one or more flip-flops in this circuit are replaced by scan flip-flops, there will always be equal or more scan flip-flops between a and c then there will be between a and b. For figure 7-9a this doesn’t have to be true. Here a flip-flop between a and b can be made scannable without making a flip-flop between a and c scannable. Because the graphical representation of these two circuits is equal, information on how the flip-flops were actually placed in the circuit is lost. Due to this loss of information the constraint that exists on a layout as depicted in figure 7-9b will also be generated for the layout from figure 7-9a. The constraint is formulated as follows: “If on a single input to multiple output connection an input to output connection a has more flip-flops (a larger weight) than an other input to output connection b, with the same input but another output, then the number of scan flip-flops on connection a will be equal or higher than on connection b.”

\[
\text{if } W(e_1) > W(e_2) \text{ then } r(\text{input}_k) - r(\text{input}_j) \leq W(e_1) - W(e_2)
\]

Proof: \(W(e_1) > W(e_2) \Rightarrow W_r(e_1) \geq W_r(e_2)\)

\[
W_r(e_1) = W(e_1) + r(\text{input}_j) - r(\text{output}_i) \geq W(e_2) + r(\text{input}_k) - r(\text{output}_i) = W_r(e_2)
\]

\[
r(\text{input}_j) - r(\text{input}_k) \geq W(e_2) - W(e_1)
\]

\[
r(\text{input}_k) - r(\text{input}_j) \leq W(e_1) - W(e_2)
\]
7.4 Mapping the graph back to a network

After the retiming algorithm the retiming values of the vertices are calculated. With these retiming values the Scanweight \( W_r \) of the inputs (number of scan flip-flops before this input) are calculated as follows. Also the original weights \( W \) in the graph before the retiming algorithm was performed on it are stored

\[
W_r(input_j) = W(input_j) + (r(input_j) - r(output))
\]

\( W_r(input_j) \) and \( W(input_j) \) are respectively the scanweight and the weight of \( input_j \).

As long as a consuming pin of a net has a weight \( W \) larger than zero a flip-flop is placed before this net. If one of the consuming pins of this net has a scanweight \( W_r \) larger than zero and a weight equal to 1 the flip-flop is labelled as scan flip-flop. The driving pin of this net is rewired to the D-input of the flip-flop. If there is a consuming pin the Q-output is wired to this net. If another consuming pin of this net has a weight equal to zero, this pin is rewired to the D-input of the added flip-flop. If the weight of this consuming pin is larger than zero then the weight is decremented with one. If a flip flop, which is labelled as scan flip-flop, is placed and the scanweight \( W_r \) is larger than zero then the scanweight is decremented with one. A further explanation for this placement algorithm will be given in section 10.6.2 on page 85.

All the flip flops in the netlist labelled as scan flip-flops have to be made holdable scannable for test purposes. The flip flops not labelled as scan flip-flops are left unchanged.
8 Experimental SmartScan based on ScanLab

8.1 Introduction

In the last chapter ScanLab was described. ScanLab is a tool that divides the memory elements in a circuit into two groups for *smart scan* purposes. ScanLab makes the division visible by changing the instantiation names of the memory elements. The flip-flops that must be equipped with a hold- and testmode get instantiation names that begin with 'fx' followed by the flip-flop number. The flip-flops that remain unchanged have instantiation names beginning with 'ff' followed by its flip-flop number.

Based on ScanLab it is possible to directly start developing the SmartScan tool. However, since developing a tool costs a lot of time and money one wants to know if the effort really pays off. To get to know this a theoretical calculation could be made on the results coming out of ScanLab, but this still wouldn’t be a 100% guarantee that *partial scan* based on ScanLab works. Also, this method would take a lot of calculation time per example. Instead, if, with acceptable effort, an experimental *partial scan* tool based on ScanLab could be developed, real life experiments would be possible. Based on these experiments a decision, as to whether or not the SmartScan tool based on ScanLab must be developed, can be taken.

Because a lot of tools are available within ED&T and taking the above into account it was decided to build an experimental SmartScan, based on a flow of tools with ScanLab as basis. The flow contains also a simulation step so that the results coming out of the experimental tool can be tested by their correctness.

8.2 The flow in abstracted form

In figure 8-1 on page 48 an abstraction of the developed flow is represented. First, this flow will be described by giving a short explanation of the function of the different blocks of which this flow consists. Later, the building blocks will be described in detail.

To prevent the occurrence of problems in the flow due to the use of too long names in the circuit description first all these names are abbreviated to a length which can be handle by all tools in the flow.

Second block in the flow is the tool ScanLab which divides the memory elements in the circuit into two groups by changing their instantiation names. The first group is the group of memory elements in the circuit that are there for pipeline purposes and thus needn’t be made scannable for testing. The other group, are the non-pipeline memory elements in the circuit which need to be made holdable scannable for testing.
In figure 8-1 it can be seen that the flow is divided into three subflows after the ScanLab tool has modified the circuit description. These subflows will be discussed from left to right.

The left building block accounts for the test pattern generation. To be able to use a combinatorial ATPG it is necessary to replace all DFFs that need not be scannable by buffers. After this replacement all the remaining DFFs are replaced by their scannable variants and put in a scan chain. The resulting circuit is called 'fooling' circuit because the pattern generator is fooled with this circuit to be able to generate patterns.

With results coming from ScanLab and the pattern generation block a Test Protocol is generated in the middle block. Because the test protocol for SmartScanned circuits is different from that for Full Scan circuits a test protocol generator has been built.

In the right block the netlist is built, according to the smart scan method, with the aid of the instantiation names for the memory elements produced by ScanLab. The non-pipeline memory elements in the circuit are replaced by their holdable scannable variants and placed in a
scan chain.

In the next block of the flow the test patterns created for the 'fooling' circuit and the test protocol are combined to produce test vectors for the SmartScanned circuit.

In the last block of the flow a simulation from the test is performed. The test vectors are applied on the circuit and the test results are compared with the predicted results. When these results match for all test vectors testing the circuit with the *smart scan* method was successful.

### 8.3 Building blocks of the flow in detail

After a brief discussion of the abstracted flow in the previous the building blocks of the flow will be discussed in detail now. Because ScanLab has been discussed in section 8.2, this block won't be discussed here anymore. The abbreviation of the names in the netlist also will not be discussed anymore.

#### 8.3.1 Pattern generation on 'fooling' circuit

In figure 8-2 the block in which the patterns are generated on the 'fooling' circuit is shown in detail.

![Pattern generation on 'fooling' circuit](image)

**Figure 8-2: Pattern generation on ‘fooling’ circuit**

Due to the fact that a combinatorial test pattern generator is used, the memory elements that won't be changed for purposes of *smart scan* have to be substituted by non-sequential elements in order to make the test pattern generator work ("ATPG fooling"). This replacement
must be such that the same stuck-at pattern is used by the test generator and that the same
function is performed by the combinatorial substitution circuit, without the sequential part to
it. For instance, a DFF with only a q output is replaced by a buffer which input is connected
to the same net as the d input of the DFF was. The output is connected to the same net as the q
output of the DFF was. The function of both elements is the same when we don’t look at se­
quentiality. The test pattern generator won’t introduce extra stuck at faults because it sees the
buffer as a normal wire. Naturally, the modified timing behaviour of the circuit on which the
test patterns are generated must be accounted for.

The first tool in the left sub flow is LapUp. The inputs for this tool are the circuit description
coming from ScanLab, the library file and a matchrule file. This tool replaces all instantia­
tions of the memory elements with instantiation names beginning with FF (which needn’t be
changed for partial scan) by the circuit given in the matchrule file. This action is performed
for reasons mentioned above. The output of this tool is thus a netlist description of the input
circuit where this action is performed on. In figure 8-3 the substitution of a DFF by a buffer is
represented:

![Figure 8-3: Substitution of a D flip flop by a buffer](image)

InScan, the next tool in the flow, reads in the output of LapUp and the library. InScan is a pro­
gram for the insertion of scan chains in a digital, (block-)synchronous IC design. Because
LapUp was run before InScan, InScan will only find the memory elements in the circuit with
instantiation names beginning with ‘fx’ and make only these scannable. Now only the flip­
flops that should be scannable according to ScanLab are made scannable. InScan writes out
the design description with the inserted scan chain. Also a Routing plan file (Rpl), which
shows the routing of the scan chain through the circuit, and a Matchrule file (Mrl), which
shows the replacement of the DFF by a SFF chosen by InScan, are written out.

After InScan the program ToAmsal is run. ToAmsal expands the library cells to Amsal primi­
tive cells. This tool reads in the circuit description coming out of InScan and the library. Out­
put of this tool consists of the circuit description which contains also the description of all
Amsal primitives. ToAmsal also creates a Control file for Amsal, which is used to steer the
actions Amsal preforms.

The output coming out of ToAmsal is read in by Amsal. Amsal stands for Automatic Multi re­
startable Scan test pattern generation And Localization of faults. Amsal generates test pat­
terns for combinational circuits. Because the FFs that shouldn’t be made scannable,
according to ScanLab, are substituted by Buffers and all the remaining FFs are made scanna­
ble and routed in a scan chain by InScan, this combinational ATPG tool can be used to create
test patterns for this circuit. The output of Amsal is a Pattern File with stimuli and predicted
responses.
8.3.2 Building the SmartScanned netlist

In figure 8-4 the block in which the SmartScanned netlist is built is shown in detail.

In the right subflow block LapUp is also used. In this case the tool is used to replace all the flip-flops that need not be made holdable scannable by a combinatorial element so InScan won’t “see” these memory elements. InScan won’t place these elements in the Routing Plan file. After InScan has inserted the scan chain these combinatorial elements have to be replaced by the original flip-flops. To be sure that LapUp only replaces the sequential elements with instantiation names beginning with FF and leave the rest of the circuit unchanged the memory elements need to be replaced by a combinatorial element with equal number of input and output pins. For example if a DFF has two inputs (clk and d) and one output (q) it can be...
replaced by a 2 input AND port. Input for LapUp is thus the library and a Matchrule file which gives a replacement without changing the number of in- or output ports. The output of LapUp is a circuit description.

After the circuit has been updated by LapUp InScan is run on it. The steps that InScan takes can be described as follows:

1. Propose a default way of creating the scan paths.
2. Allow algorithms to 'optimise' this proposal i.e., alter it so that it conforms to a certain methodology. This step is optional.
3. Propose a 'scannable' variant for each type of memory element that is to be entered in a scan path.
4. Replace all memory elements that are to be entered in a scan path by their 'scannable' variant, which was proposed in the previous step.
5. Create the scan paths.

In this schematic, basically two sorts of steps can be distinguished. Steps 1-3 only perform analysis which results in certain proposals. Steps 4-5 only execute these proposals. To provide flexibility, InScan is constructed of two separate programs: PrepScan, which preforms the steps 1-3, and ScanIt, which performs steps 4-5. In this flow use is made of this flexibility that is built into InScan.

First PrepScan is run on the circuit description coming out of LapUp. Besides this description PrepScan needs the library. PrepScan generates two files. The first file is the Routing plan file in which the routing of the scan chain in the circuit is given. To obtain a routing were also a holdline is routed this file has to be updated. AddHold performs this updating by inserting a holdsignal into this Rpl file. The second file PrepScan generates is the Matchrule file. In this file the replacement for the memory elements by a scannable variant is given. Because of the fact that our testmethod needs holdable scannable variants this Mrl file is replaced by a user specified Mrl file. Within this file the replacement by a holdable scannable memory element is given.

After the results, coming from PrepScan, have been updated, the scan chain can be inserted by ScanIt. For this action ScanIt reads in the Rpl file updated by AddHold and the user specified Mrl file. Output of ScanIt is the circuit description with the scan chain, including hold enable pins and wires, inserted in it.

Now the tool LapUp is used to perform the reverse of the action it did before on the circuit description. Now all the flip-flops that it replaced by a combinatorial element, with the same number of in- and output pins are placed back in it. For reasons that will be explained later the circuit must now be placed in a top block. This action is performed by Push. Push inserts a top block macro in the circuit description. Within this description the original circuit is instantiated so that all its pins are wired to the boundary of the top block.

Now the circuit description is translated from the Ndl format into the Edif format. This trans-
lation is done by the tool Oma2Edif which needs the library to know the pin names. This action is performed because the tool Tplg can only handle circuit descriptions in Edif format.

8.3.3 Test protocol generation

In figure 8-5 the block in which the test protocol is generated is shown in detail.

![Diagram](image)

**Figure 8-5: Test Protocol generation**

The SmartScanned circuit, of which the creation was described in the last subsection, is the circuit that has to be put under test. However, the test patterns are generated on the 'fooling netlist', which is not equal to the SmartScanned netlist. The difference between those circuits is that in the fooling circuit the pipeline flip-flops from the SmartScanned circuit are replaced by buffers. Furthermore the SFFs in the fooling circuit are HSFFs in the SmartScanned circuit which is necessary to be able to execute the required test protocol. When the test patterns are applied to the SmartScanned circuit with a normal Full Scan test protocol, the outputs coming out of the circuit won't match the test results generated by the combinatorial ATPG. To overcome this problem the Full Scan test protocol must be enhanced with a number of hold cycles in each test cycle. This test protocol has been explained in section 5.3 on page 25. For each example on which we want to perform experimental SmartScan, a test protocol file has to be generated. For this reason the test protocol file generator GenTpr has been created.

8.3.3.1 Example of a test protocol file

In figure 8-6 on page 54 an example of a user defined test protocol file is presented. After `$cell` the cell is identified by name. The pattern file name behind `$patterns` describes the pattern file containing the test patterns corresponding with the initial test protocol. The user defined test protocol describes test control signals to be applied at the cell input ports, stimuli to be applied at the cell input ports and responses to be observed at the cell output ports. The respective descriptions have a similar form. They consist of a sequence of `a=b` expressions. Test Control signal are described as a sequence of `input signals = condition values;` expressions.
sions as can be seen in figure 8-6. The input signals are described by means of a port at a clock cycle, e.g. “p<0>” denotes port p at clock cycle 0. The condition values are described between square brackets. A value is “0” or “1” they stand for the logical zero and logical one.

```
$cell cell1;
$protocol p;
$patternfile "cell1.pat";

$condition
  TC<4:-1>=[1]^;  
  TC<0:1>=[0]^;  
  TC<2:5>=[1]^;  
  TH<4:-1>=[0]^;  
  TH<0>=[1]^;      
  TH<1:5>=[0]^;   

$apply
  TD<4:-1>=TD[1:4]  
  uvbin<0:1>=uvbin, uvbin;  
  loaduv<0:1>=loaduv, loaduv;  

$observe
  uuvv_1<1>=uuvv_1;  
  uuvv_0<1>=uuvv_0;  
  TQ<2:5>=TQ[1:4];  
```

Figure 8-6: Example of a user defined test protocol file

Figure 8-7: Graphical representation of the test protocol example
The number of signals and the number of values must be equal. For example "p<0:1>=[01]" denotes that \( p<0>=0 \) and \( p<1>=1 \). "p<0:1>=[1]^" denotes that \( p=1 \) at both clock cycle 0 and 1.

After `$apply` the stimuli are described. Stimuli are described as a sequence of 'input signals = pattern values;' expressions as can be seen in figure 8-6. The input signals are described by means of a port at a clock cycle, as the test control signals. The pattern values are identifiers referring to stimulus bits of a pattern in the pattern file. The number of input signals and the number of pattern values must be equal. Each input signal and corresponding pattern value describe a stimulus to be applied.

After `$observe` the responses are described. Responses are described as a sequence of 'output signals = pattern values;' expressions as can be seen in figure 8-6. The responses and stimuli forms are similar.

In the example test protocol file of figure 8-6 ‘TC’ is the test control port from the flip flops and ‘TH’ is the hold control. It can be seen that data is scanned in from clock cycle -4 till clock cycle -1, thus ‘cell1’ has a scan chain of length 4. After the data is scanned in one hold cycle is applied, which means that one pipeline has to be crossed. After the normal cycle data can be observed. During the hold and normal cycle the stimuli are applied to the input ports. In figure 8-7 a graphical representation of the example test protocol is depicted.

### 8.3.3.2 Creation of the test protocol file

Now that the format of the test protocol file is understood, the creation of such a file can be discussed. The test protocol file generator uses a routing plan and a netlist of a circuit. The formats of these files are explained in [Voo93]. The creation of the test protocol file is divided into three parts, creation of the control part, creation of the stimuli part and creation of the response part. This division will also be used to discuss the creation of such a file.

First an outline is given of how the control block generation algorithm is implemented.

```plaintext
CreateControlBlock( netlist, routing plan )
{
   ll = CreateList( )
   forall topmacros mt in the routingplan
   {
      forall clockdomains cd \in\ mt forall chains ch \in\ cd
      {
         Let HoldTime = GetHoldFactor( cd )
         forall scanenablepins sep \in\ ch
         {
            if( NewPin( sep, ll) )
            {
               Let ChainLength = 0
               ChainLength = LargestChainScanEnable( sep, cd )
               Let sepName = GetName( sep )
               if ( GetPolarisation( sep ) = POSITIVE )
               {

```
/generate control signal as follows:
sepName<-ChainLength-HoldTime:-1-HoldTime>=[1]^ 
sepName<-HoldTime:0>=[0]^ 
sepName<1:ChainLength>=[1]^ 
else
/*generate control signal as follows*/
sepName<-ChainLength-HoldTime:-1-HoldTime>=[0]^ 
sepName<-HoldTime:0>=[1]^ 
sepName<1:ChainLength>=[0]^ 
}
}
forall holdenablepins hep in ch
{
  if ( NewPin( hep, II ) )
  {
    Let LargestLength = 0
    ChainLength = LargestChainHoldEnable( hep, cd )
    Let hepName = GetName( sep )
    if ( GetPolarisation( hep ) = POSITIVE )
    {
      /*generate control signal as follows:*/
      hepName<-ChainLength-HoldTime:-1-HoldTime>=[0]^ 
      hepName<-HoldTime:-1>=[1]^ 
      hepName<0:ChainLength>=[0]^ 
    }
    else
    {
      /*generate control signal as follows:*/
      hepName<-ChainLength-HoldTime:-1-HoldTime>=[1]^ 
      hepName<-HoldTime:-1>=[0]^ 
      hepName<0:ChainLength>=[1]^ 
    }
  }
  }
NewPin( pin, II ) returns a boolean value that denotes whether the chainpin pin is already described in the control block (is already present in the linklist II). NewPin will return true if pin isn’t described yet. This is to prevent the input signal from being described more than once in the control block. Due to this the readability of the test protocol file will increase. Also NewPin will add pin to II if it wasn’t described yet.

NewPin( chp, II )
{
    Let NewElement = true
    forall chainpins cp ∈ II
    {
        if ( GetName(cp) = GetName(chp) )
            NewElement = false
    }
    if( NewElement )
        Add chp to II
    return NewElement
}

LargestChainScanEnable( pin, cd ) returns an integer value that denotes the length of the largest chain, within the clock domain cd, of which ‘pin’ is a scanenablepin. This guarantees that the chainpin ‘sep’ is defined over the largest period it has to be active.

LargestChainScanEnable( pin, cd )
{
    Let LargestChain = 0
    forall chains ch ∈ cd
    {
        Let Chainlength = GetLength( ch )
        forall scanenable pins sep ∈ ch
        {
            if ( GetName( sep ) = GetName( pin ) ∧ ChainLength > LargestChain )
                LargestChain = Chainlength
        }
    }
    return LargestChain
}

The function LargestChainHoldEnable( pin, cd ) returns the length of the largest chain, within clockdomain cd, of which pin is a holdenable pin. The implementation of this function is almost equivalent to the implementation of the LargestChainScanEnable function so it will not be discussed here.
After the generation of the control block the stimuli block from the test protocol file must be generated. Below an outline is given of how the stimuli block generation algorithm is implemented.

CreateStimuliBlock( netlist, routing plan )
{
    Let MaxHoldTime = 0
    MaxHoldTime = GetMaxDist( netlist )
    for topblock tb in the netlist
    {
        forall input pins ip ∈ tb
        {
            if( not IsClockPin( ip ) )
            {
                if ( ip has a property SmartDist )
                {
                    Let Hold = false
                    Let HoldTime = 0
                    if ( SmartDist contains only one distance )
                        HoldTime = GetNumber( SmartDist ) - 1
                    else
                        HoldTime = MaxHoldTime
                    Hold = true
                }
                ipname = GetName( ip )
                if ( not( Hold ) ∨ HoldTime = 0 )
                {
                    generate stimulus signal as follows
                    ipname<-HoldTime> = ipname;
                }
                else
                {
                    /*generate stimulus signal as follows:*/
                    ipname<-HoldTime:0> =
                    for HoldTime times do
                        write(" ipname;")
                        write("ipname;")
                }
            }
        }
    }
    for topmacro mt in the routing plan
    {
}
forall clockdomains cd ∈ mt
{
    Let HoldTime = GetHoldFactor(cd)
forall chains ch ∈ cd
{
    Let ChainLength = 0
    ChainLength = GetChainLength(ch)
sip = GetScanInPin(ch)
sipname = GetName(sip)
generate stimuli as follows
    sipname<-ChainLength-HoldTime:-1-HoldTime>
    = sipname[1:Chainlength];
}
}

The function GetMaxDist(netlist) returns the maximum distance, in D flip-flops, between two scan flip-flops in the netlist. The value of the property MaxDist, which is a property of the topmacro in the netlist, denotes this maximum distance in clock cycles. The last block to be generated for the test protocol file is the response block. Below an outline is given of how the response block generation algorithm is implemented.

CreateResponseBlock(netlist, routingplan)
{
    Let MaxHoldTime = 0
    MaxHoldTime = GetMaxDist(netlist)
    for topblock tb in the netlist
    {
        forall output pins op ∈ tb
        {
            opname = GetName(op)
generate observation as follows
            opname<0> = opname;
        }
    }
    for topmacro mt in the routingplan
    {
        forall clockdomains cd ∈ mt forall chains ch ∈ cd
        {
            Let ChainLength = 0
            ChainLength = GetChainLength(ch)
            forall scanoutpins sop ∈ ch
            {
                sopname = GetName(sop)
            }
        }
    }
}
8.3.4 Test vector generation

In figure 8-8 the block in which the test vectors for the simulation of the circuit test are created is shown in detail.

After the discussion of the three parallel flow blocks in figure 8-1 on page 48 now the creation of the test vectors with results coming out of these flows can be discussed. At this point in the flow test patterns have been generated on the 'fooling' circuit. Also a test protocol file, which describes how to use the patterns in the patternfile to perform the test on the SmartScanned circuit, has been created. With these two files test vectors have to be made which can be used to test the SmartScanned circuit. A tool which can perform such an action is TPLG. TPLG provides tools that can expand test protocols for a testable block at a lower level in the device to device level. An expanded test protocol specifies how a testable block is to be accessed for testing from the pins of the device. TPLG can also assist in creating test vectors from test patterns created at the block level and a expanded test protocol. This is done by the tool TASS. Because the test patterns and the test protocol are already generated TPLG TASS can thus be used to create suitable test vectors for the SmartScanned circuit with these two files as input.

However there is still a problem that has to be overcome. TPLG can only expand test protocols to a higher level than for which they are created. This means that test protocols can’t be expanded to the level of the block under test itself. This problem is already accounted for in the block which builds the smart scan netlist, described in section 8.3.2. Herein the block that has to be tested is placed within a top block called 'top'. This doesn’t change the function of the circuit because the pins on the top block and the testblock have a one on one rela-
tion and are directly connected.
As a result of the actions performed by TPLG and TASS a test vector file 'top.atf' is created which contains the testvectors that can be used to test the SmartScanned circuit.

8.3.5 Simulation of circuit test

In figure 8-9 the block in which the simulation of the circuit test is performed is shown in detail.

![Diagram of circuit test simulation]

**Figure 8-9: Circuit test simulation**

In the final block of the flow the circuit test will be simulated to be able to conclude if the smart scan theory works. This simulation will be performed by the tool LeapFrog. LeapFrog needs three input files to perform this action. First it needs a VHDL description of the SmartScanned circuit. This description can be created by the tool Oma2Vhdl, which translates Ndl circuit descriptions to VHDL.

Next LeapFrog needs the test vector file top.atf, which is produced by TPLG and TASS in the previous block of the flow. The third file LeapFrog needs is the test bench file. This test bench reads out the stimuli in the test vector file and applies them to the circuit under test. Meanwhile it compares the output values coming from the test circuit with the responses described in the test vector file. When a response doesn't match an output value, an error message is created. Such a test bench file is created by the tool Atf2Vhdl. This tool uses the test vector file as input.

8.4 The total experimental SmartScan flow in detail

In figure 8-10 the total experimental SmartScan flow, built up from the building blocks as shown in figure 8-1 on page 48, is represented in detail.
Figure 8-10: The total experimental SmartScan flow in detail
9 Experimental SmartScan compared to Full Scan

In this chapter the experimental SmartScan will be compared with making a circuit testable by means of Full Scan. To be able to make this comparison, the experimental SmartScan flow described in the last chapter was run on every example to see if the circuit test with experimental SmartScan succeeded. After this, both a *smart scan* and a *full scan* version of the circuit were build, for which test patterns were generated with the combinatorial ATPG tool Amsal. This way the two test methods could be compared.

First the results for the ISCAS'89 Benchmark circuits will be discussed. After this the comparison will be made for subblocks of a circuit for high speed video called the Melzonic. Also results on a audiochip design, the Fadic123, will be discussed.

9.1 Results on the ISCAS'89 benchmark circuits

In table 9-1 on page 66 the results for both experimental SmartScan and Full Scan are listed. First the meaning of the results in each column will be explained.

In the column SFF the percentage of the FFs in the circuit made scannable by experimental SmartScan is denoted. Because the distance(s) from a scan flip-flop to its follower(s) or the outputs of the circuit isn't known, all scan flip-flops must also be made holdable. Thus a gain in area for experimental Smartscan compared to Full Scan (in which all FFs are made scannable) is only achieved when the percentage in the column SFF is below 50%.

In the second column, the number of holdcycles within each *smart scan* testcycle is given. These cycles must be introduced in the testcycle to clock the signals through the remaining D flip-flops in the circuit which aren't accounted for by the combinatorial ATPG tool.

The third and fourth column contain respectively the fault coverage, achieved by the combinatorial ATPG tool Amsal, for the *full- and smart scan* case.

In the column ‘Patterns’ the percentage gain in number of generated test patterns for *smart scan* method compared to *full scan* is given. A negative percentage denotes a drop in the number of patterns.

The last two columns contain the CPU times used by Amsal to generate the test patterns for both the *full- and the smart scan* case.

9.1.1 Sequential redundancy

Looking at the column ‘SFF’ it can be seen that the percentage of the flip-flops made holdable scannable for *smart scan* purposes is in all but one case (s953 benchmark) above 50%. In more than half of the cases all flip-flops are made scannable which means that there is no difference between the two test methods. It can be seen that the percentage of pipeline flip-flops in these circuits is very low or even zero. The difference in Stuck-At fault coverage on the *full scan* circuits and the *smart scan* ‘fooling’ circuits, achieved by the combinatorial ATPG tool Amsal, is very small. The largest drop in fault coverage is 0.09% for the s13207 benchmark. This drop in fault coverage is caused by redundancy in the example circuits. This is the redundancy which exists between two logic blocks which are connected by flip-flops. In the Full Scan case this redundancy is hidden because all the flip-flops between the logic blocks
are controllable and so the redundancy is broken up. With the smart scan method however not all flip-flops have to be made scannable and so redundancy existing over flip-flop borders between two logic blocks becomes visible. In figure 9-1 an example of this kind of hidden redundancy is given.

![Figure 9-1: Example of hidden redundancy in Full Scan case](image)

This example circuit contains a pipeline of three flip-flops. When the smart scan method is used to test this circuit none of these flip-flops will be made scannable. This way the Stuck-At fault $A_{sa1}$ can't be detected due to redundancy in the circuit. When $i_1$ and $i_2$ are both on "1" and $i_3$ is made "0" the output $o$ of the circuit is always "1" after one clock cycle, even if $A$ is $sa1$. When Full Scan is used, all flip-flops are made scannable. This means that the output of NAND-port n1 is observable, so $A_{sa1}$ is detectable.

### 9.1.2 Drop in number of test patterns for smart scan compared to full scan

Looking at the results for the s953 Benchmark it can be seen that only 21% of the flip-flops need to be made scannable. On this Benchmark the combinatorial ATPG achieved for both scan methods, full scan and smart scan, an equal fault coverage of 100%. However, looking at the number of patterns generated a drop of 1% in patterns for experimental SmartScan compared to Full Scan is achieved. This drop is caused by the fact that the circuit, used with the smart scan method, on which patterns are generated, contains less stuck-at faults to be covered than the Full Scan circuit. This only holds if, with experimental SmartScan, less flip-flops are made scannable than with the full scan method. In figure 9-2 a comparison is made between the replacement of a scan flip-flop for combinatorial ATPG and the replacement of a D flip-flop in a SmartScanned circuit for combinatorial ATPG. In this figure it can be seen that for every flip-flop that does not have to be made scannable, when experimental SmartScan is used, a reduction of 5 Stuck-At faults compared to the Full Scan circuit is achieved. This means that for every flip-flop in the circuit that is not made scannable with experimental SmartScan, the combinatorial ATPG tool has to cover 5 Stuck-At faults less, which can lead to a drop in the number of patterns.
9.1.3 Further discussion of the results on the ISCAS’89 benchmarks

For the s832 benchmark the experimental SmartScan Stuck-At fault coverage is 0.01% higher than the coverage for Full Scan. This gain is probably a rounding inaccuracy within Amsal. When experimental SmartScan is applied to a circuit all flip-flops not made scannable are replaced by buffers in the circuit on which the patterns are generated. Due to this substitution the logic depth of the logic within the circuit increases. This can cause an increase in the CPU time needed by the pattern generator. Looking at the last two columns it can be seen that, for these examples, the growth in CPU time used by Amsal to generate patterns for SmartScanned circuits isn’t very large. An explanation for this is that for these benchmarks only a small percentage of the flip-flops isn’t made scannable, thus the increase in logic depth will also be small in most cases. Sometimes the used CPU time even drops a little.

Looking at the overall result achieved by experimental SmartScan on the ISCAS’89 benchmarks the following can be concluded:
Due to the fact that these benchmarks contain almost no pipeline flip-flops, experimental SmartScan must make a large percentage of the flip-flops scannable. These flip-flops all have to be holdable scannable because of the multiple distances which can exist. So for circuits which contain a relatively small number of pipeline flip-flops experimental SmartScan is less cost effective than testing with the full scan method.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>SFF (%)</th>
<th>Hold Cycles</th>
<th>SA fault coverage</th>
<th>Patterns (Δ%)</th>
<th>TPG Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Full</td>
<td>Smart</td>
<td></td>
</tr>
<tr>
<td>s27</td>
<td>100</td>
<td>0</td>
<td>100.00</td>
<td>100.00</td>
<td>0</td>
</tr>
<tr>
<td>s208</td>
<td>100</td>
<td>0</td>
<td>100.00</td>
<td>100.00</td>
<td>0</td>
</tr>
<tr>
<td>s208.1</td>
<td>100</td>
<td>0</td>
<td>100.00</td>
<td>100.00</td>
<td>+3</td>
</tr>
<tr>
<td>s298</td>
<td>100</td>
<td>0</td>
<td>100.00</td>
<td>100.00</td>
<td>+4</td>
</tr>
<tr>
<td>s344</td>
<td>100</td>
<td>0</td>
<td>100.00</td>
<td>100.00</td>
<td>0</td>
</tr>
<tr>
<td>s349</td>
<td>100</td>
<td>0</td>
<td>99.59</td>
<td>99.59</td>
<td>+7</td>
</tr>
<tr>
<td>s382</td>
<td>71</td>
<td>1</td>
<td>100.00</td>
<td>100.00</td>
<td>0</td>
</tr>
<tr>
<td>s386</td>
<td>100</td>
<td>0</td>
<td>100.00</td>
<td>100.00</td>
<td>0</td>
</tr>
<tr>
<td>s400</td>
<td>71</td>
<td>1</td>
<td>98.90</td>
<td>98.86</td>
<td>+7</td>
</tr>
<tr>
<td>s420</td>
<td>100</td>
<td>0</td>
<td>100.00</td>
<td>100.00</td>
<td>0</td>
</tr>
<tr>
<td>s444</td>
<td>71</td>
<td>1</td>
<td>98.31</td>
<td>98.25</td>
<td>+4</td>
</tr>
<tr>
<td>s510</td>
<td>100</td>
<td>0</td>
<td>100.00</td>
<td>100.00</td>
<td>+4</td>
</tr>
<tr>
<td>s526</td>
<td>100</td>
<td>0</td>
<td>99.90</td>
<td>99.90</td>
<td>+2</td>
</tr>
<tr>
<td>s526n</td>
<td>100</td>
<td>0</td>
<td>100.00</td>
<td>100.00</td>
<td>+9</td>
</tr>
<tr>
<td>s641</td>
<td>100</td>
<td>0</td>
<td>100.00</td>
<td>100.00</td>
<td>+3</td>
</tr>
<tr>
<td>s713</td>
<td>100</td>
<td>0</td>
<td>96.62</td>
<td>96.62</td>
<td>+3</td>
</tr>
<tr>
<td>s820</td>
<td>100</td>
<td>0</td>
<td>100.00</td>
<td>100.00</td>
<td>-1</td>
</tr>
<tr>
<td>s832</td>
<td>100</td>
<td>0</td>
<td>98.94</td>
<td>98.95</td>
<td>+3</td>
</tr>
<tr>
<td>s838</td>
<td>100</td>
<td>0</td>
<td>100.00</td>
<td>100.00</td>
<td>+1</td>
</tr>
<tr>
<td>s953</td>
<td>21</td>
<td>1</td>
<td>100.00</td>
<td>100.00</td>
<td>-1</td>
</tr>
<tr>
<td>s1196</td>
<td>89</td>
<td>1</td>
<td>100.00</td>
<td>100.00</td>
<td>-2</td>
</tr>
<tr>
<td>s1238</td>
<td>89</td>
<td>1</td>
<td>96.55</td>
<td>96.53</td>
<td>+1</td>
</tr>
<tr>
<td>s1423</td>
<td>97</td>
<td>1</td>
<td>99.29</td>
<td>99.29</td>
<td>+6</td>
</tr>
<tr>
<td>s1488</td>
<td>100</td>
<td>0</td>
<td>100.00</td>
<td>100.00</td>
<td>+3</td>
</tr>
<tr>
<td>s1494</td>
<td>100</td>
<td>0</td>
<td>99.49</td>
<td>99.49</td>
<td>+2</td>
</tr>
<tr>
<td>s5378</td>
<td>92</td>
<td>0</td>
<td>98.82</td>
<td>98.78</td>
<td>+1</td>
</tr>
<tr>
<td>s9234</td>
<td>96</td>
<td>3</td>
<td>94.12</td>
<td>94.10</td>
<td>-2</td>
</tr>
<tr>
<td>s13207</td>
<td>82</td>
<td>4</td>
<td>99.07</td>
<td>98.98</td>
<td>+1</td>
</tr>
<tr>
<td>s15850</td>
<td>95</td>
<td>5</td>
<td>98.05</td>
<td>98.02</td>
<td>+1</td>
</tr>
<tr>
<td>s35932</td>
<td>100</td>
<td>0</td>
<td>91.32</td>
<td>91.32</td>
<td>0</td>
</tr>
<tr>
<td>s38417</td>
<td>90</td>
<td>2</td>
<td>99.68</td>
<td>99.67</td>
<td>-9</td>
</tr>
<tr>
<td>s38584</td>
<td>100</td>
<td>1</td>
<td>94.80</td>
<td>94.78</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 9-1: Experimental SmartScan results for ISCAS’89 Benchmarks
9.2 Results on the Melzonic benchmarks

In table 9-2 the results coming from both experimental SmartScan and Full Scan performed on building blocks of the Melzonic are listed.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>SFF (%)</th>
<th># Hold Cycles</th>
<th>SA fault coverage</th>
<th>Patterns (Δ %)</th>
<th>TPG Time</th>
<th># Test Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Full</td>
<td>Smart</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>formatteruv</td>
<td>29</td>
<td>3</td>
<td>100.00</td>
<td>100.00</td>
<td>0:06 m</td>
<td>405</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0:06 m</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>activity_fallback</td>
<td>57</td>
<td>3</td>
<td>99.64</td>
<td>96.36</td>
<td>0:17 m</td>
<td>5795</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>96.36</td>
<td></td>
<td></td>
<td>3213</td>
</tr>
<tr>
<td>aver_xy_med</td>
<td>0</td>
<td>4</td>
<td>99.39</td>
<td>98.62</td>
<td>0:13 m</td>
<td>455</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>98.62</td>
<td></td>
<td></td>
<td>54</td>
</tr>
<tr>
<td>div_off</td>
<td>0</td>
<td>2</td>
<td>98.75</td>
<td>98.58</td>
<td>0:06 m</td>
<td>383</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>98.58</td>
<td></td>
<td></td>
<td>195</td>
</tr>
<tr>
<td>formatterxy</td>
<td>48</td>
<td>1</td>
<td>100.00</td>
<td>100.00</td>
<td>0:04 m</td>
<td>1034</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>100.00</td>
<td></td>
<td></td>
<td>63</td>
</tr>
<tr>
<td>limit</td>
<td>0</td>
<td>2</td>
<td>99.59</td>
<td>99.49</td>
<td>0:08 m</td>
<td>53127</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>99.49</td>
<td></td>
<td></td>
<td>24544</td>
</tr>
<tr>
<td>median_xy</td>
<td>0</td>
<td>2</td>
<td>98.05</td>
<td>96.30</td>
<td>0:10 m</td>
<td>2813</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>96.30</td>
<td></td>
<td></td>
<td>120</td>
</tr>
<tr>
<td>minerr_sb</td>
<td>62</td>
<td>2</td>
<td>99.81</td>
<td>99.79</td>
<td>0:31 m</td>
<td>53127</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>99.79</td>
<td></td>
<td></td>
<td>24544</td>
</tr>
<tr>
<td>noric</td>
<td>43</td>
<td>11</td>
<td>99.29</td>
<td>99.18</td>
<td>1:07 m</td>
<td>53127</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>99.18</td>
<td></td>
<td></td>
<td>24544</td>
</tr>
<tr>
<td>reformatteruv</td>
<td>78</td>
<td>1</td>
<td>100.00</td>
<td>100.00</td>
<td>0:05 m</td>
<td>755</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>100.00</td>
<td></td>
<td></td>
<td>612</td>
</tr>
<tr>
<td>reformatterxy</td>
<td>64</td>
<td>3</td>
<td>100.00</td>
<td>100.00</td>
<td>0:03 m</td>
<td>275</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>100.00</td>
<td></td>
<td></td>
<td>180</td>
</tr>
<tr>
<td>upc_devide</td>
<td>0</td>
<td>2</td>
<td>96.97</td>
<td>96.53</td>
<td>0:11 m</td>
<td>1719</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>96.53</td>
<td></td>
<td></td>
<td>105</td>
</tr>
<tr>
<td>upc_mix</td>
<td>0</td>
<td>4</td>
<td>100.00</td>
<td>99.67</td>
<td>0:20 m</td>
<td>8909</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>99.67</td>
<td></td>
<td></td>
<td>868</td>
</tr>
<tr>
<td>uv_upconv_zoom</td>
<td>0</td>
<td>6</td>
<td>99.17</td>
<td>98.35</td>
<td>0:21 m</td>
<td>8909</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>98.35</td>
<td></td>
<td></td>
<td>868</td>
</tr>
</tbody>
</table>

Table 9-2: Experimental results for Melzonic Benchmarks

This table contains two more columns which are added to the right. In these two columns the test times, in total number of clock cycles, for both the full scan and the smart scan method are listed.

The Melzonic is a high speed video circuit. To be able to work at a high clock frequency this circuit is strongly pipelined. In the column ‘SFF’ this pipelined character of the building blocks can be seen. For almost all circuits the percentage of the flip-flops that needs to be made holdable-scannable is beneath 50%. Five circuits contain nothing but pipeline flip-flops, none of them needs to be made holdable-scannable.

The number of hold cycles that must be introduced in each test cycle never grows to unacceptable numbers. The largest number of hold cycles needed in each test cycle is 11 for the ‘noric’ benchmark. Taking into account that the noric contains 457 flip-flops of which 196 are made holdable-scannable, it can be seen that the largest part of the test cycle is needed for shifting in and out scan data.

The largest drop in Stuck-At fault coverage is 3.28% for the ‘activity-fallback’ benchmark. As explained in section 9.1 this drop can be ascribed to the redundancy existing in the circuit which is invisible for the combinatorial ATPG tool when Full Scan is used. In test mode all flip-flops are controllable and observable in the Full Scan method, redundancy in logic blocks...
of the circuit divided by flip-flops can, as a result of this, not be seen by the pattern generator.
With experimental SmartScan the redundancy at the flip-flops which are not made scannable is visible through a lower Stuck-At fault coverage.

With respect to the CPU time needed by the combinatorial ATPG tool Amsal to generate patterns the following can be seen. The benchmarks for which only a small part or even no flip-flops have to be made holdable-scannable, when experimental SmartScan is used, an increase upto 30 times the CPU time for the Full Scan case is needed. This increase can, as explained in the previous section, be ascribed to the growth in logical depth of the circuit on which Amsal has to generate patterns. Each flip-flop in the circuit which isn’t made holdable-scannable, when experimental SmartScan is used, is replaced by a buffer for pattern generation purposes. This can enhance the logical depth of the circuit considerably. In the extreme case that none of the flip-flops is made holdable-scannable the whole circuit becomes one logical block of which the logical depth can be very large. The longest used CPU time for pattern generation on these benchmarks is 6 hours and 31 minutes, which is still considered to be acceptable.

With the results listed in the last two columns the total duration of the tests for both methods can be compared. As can be seen the test time for these examples always drops when experimental SmartScan is used. The large drop in test time, upto 16 times, is achieved by the enormous reduction in scan chain length when the smart scan method is used. The hold cycles that have to be introduced in each test cycle are far less than the number of flip-flops that need to be made scannable, so overall the test time is strongly reduced.

The overall result for experimental SmartScan with respect to the Melzonic benchmarks is very positive. In almost all cases a reduction in area overhead compared to Full Scan is achieved. In some cases the area overhead for test purposes is even reduced to 0%. The CPU time needed by the combinatorial ATPG tool Amsal, in some cases, increases with a factor 20 or more but never becomes unacceptable high. In all cases the total test time is reduced compared to Full Scan. In some cases a reduction with a factor 10 or more is achieved.

9.3 Results on the Fadic123 benchmark

In table 9-3 the results coming from both experimental SmartScan and Full Scan performed on the Fadic123 benchmark are listed.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>SFF (%</th>
<th># Hold Cycles</th>
<th>SA fault coverage</th>
<th>Patterns Δ (%)</th>
<th>TPG Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Full</td>
<td>Smart</td>
<td></td>
</tr>
<tr>
<td>fadic_logic</td>
<td>58</td>
<td>2</td>
<td>98.88</td>
<td>98.08</td>
<td>-5</td>
</tr>
</tbody>
</table>

Table 9-3: Experimental SmartScan results for Fadic123 Benchmark

The Fadic123 is a chip designed for audio purposes. This means that it is not as strongly pipelined as the Melzonic chip which is used for high speed video. With this example a check can be performed if the smart scan method also works on less pipelined real life circuits.
In the table it can be seen that 58% of the flip-flops needs to be made holdable scannable (234 out of 405 DFFs). This percentage is larger than 50% so the area overhead for experimental Smartscan is larger than for Full Scan in this example. The number of hold cycles in each smart scan test cycle is 2 which is very small compared to the scan chain length of 234 flip-flops. The Stuck-At fault coverage drops with 0.8% compared to Full Scan. This drop is caused by the redundancy existing in the circuit as described in section 9.1 on page 63.

9.4 Conclusions on results with the experimental SmartScan Tool

Based on the test results described in the previous sections a decision had to be made with respect to the further development of a SmartScan tool.

The area overhead caused by use of the smart scan method for test depends on the number of flip-flops which are in the circuit for pipeline purposes. Because pipeline flip-flops aren't made scannable when smart scan is used, circuits which contain a high percentage of these flip-flops are very suitable to be tested with the smart scan method. At this point the area overhead caused by experimental SmartScan is less than or equal to that for Full Scan if the percentage of pipeline flip-flops in the circuit is greater or equal to 50%. In the test results it can be seen that there are many example circuits which contain more than 50% of pipeline flip-flops so with respect to area it can be concluded that for highly pipelined circuits testing with the experimental SmartScan method decreases the area overhead compared to Full Scan.

With respect to the Stuck-At fault coverage achieved by the combinatorial ATPG tool the following can be said. The Stuck-At fault coverage for the smart scan method compared to full scan only decreases due to redundancy existing in the design. This redundancy is hidden when Full Scan is used because then all flip-flops are made observable and controllable (see section 9.1 on page 63 for explanation). When no redundancy exist in the design (responsibility of the designer) then the Stuck-At fault coverage for both methods will be equal.

The CPU-time used by the combinatorial ATPG tool increases when the logical depth of the circuit increases. When experimental SmartScan is used, all flip-flops which aren't made holdable-scannable are replaced by buffers in the circuit on which the test patterns are generated, the logical depth of this circuit is thus larger than the Full Scan circuit. This causes the combinatorial ATPG tool to use more CPU-time for pattern generation, but in the examples it never became unacceptable.

Based on the results, described in the last few sections, it can be concluded that testing with SmartScan in its experimental form produced already promising results. These results are the justification for further development and optimization of this tool.
10 Implementation of ScanLab on the new datastructure

In this chapter the process of implementing ScanLab on the new datastructure is described. In parallel with this change of data structure the name ScanLab is changed into SmartScan. From now on, when the implementation of ScanLab on the new data structure is denoted, the name SmartScan is used. Because the retiming algorithm, which is used by ScanLab, is already implemented on a new datastructure, much of this code can be reused to implement SmartScan. First the reuse of the retiming code from the RetLab tool will be discussed. After this a description follows of the extra code needed for SmartScan purposes.

The development of the retiming tool RetLab on the new datastructure is described in [Hoge93]. This development was performed by C.D.S. van der Blij and A. Hogenhuis. The reader who is interested in a more detailed description of the implementation of SmartScan is advised to read chapters 12 and 13 of [Hoge93].

10.1 The RetLab class structure

To be able to describe the Retimer class structure first the icons for class relationships must be defined. In figure 2-1 the three icons for inheritance, using and instantiation relationships are depicted.

Figure 10-1: Icons for class relationships

Now that the icons for class relationships are defined the retimer class structure can be described. In figure 10-2 on page 72 the class structure of the Retimer is given. The RDesign, the Library and the Graph are related to the Retimer as uses relationships which can be seen as a 'part of' relation. The User Interface is of the relationship instantiation, because the Retimer and the user Interface are associated to each other, one User Interface is needed in a Retimer.

The class structure Graph is needed because the retiming algorithm needs a graph, it needs a list of vertices and edges on which it can work.

The class structure RDesign has evolved from the fact that NDS is used as the basic structure of RDesign. At the time of the development of this Retimer class structure it was already known that the tool 'SmartScan', at that time denoted as a 'scan chain router', was going to be developed. To prepare the data structure for this future development, which would have functions in common with the Retimer, an extra layer between NDS and the RDesign was developed. This layer was called GDesign.

GDesign was designed so that both the future 'scan chain router' and the retimer use a graph
structure in which the retiming algorithm is embedded. Where the ‘scan chain router’ will use a graph structure for the recognition of pipelines, the retimer will use it to minimize the amount of flip-flops. So GDesign contains all the methods which are related to both the ‘scan chain router’ and the retimer. The retimer can access these methods from the RDesign because RDesign is inherited from GDesign and therefore can use the public and protected methods of GDesign. The methods which have some relation with retiming, are embedded in RDesign. This implementation provides RDesign all the functionality of NDS and GDesign.

![Figure 10-2: The class structure of the retimer](image)

**10.2 The SmartScan class structure**

In [Hoge93] the assumption was made that GDesign *alone* would be used by the developers of what was then called the ‘scan chain router’. Assuming that the division of functionality over the different classes in the class structure was made with this idea in mind, the first attempt to implement ScanLab on the new datastructure was conform this idea.

In figure 10-3 on page 73 the class structure of SmartScan and the retimer are shown both. Because these structures have some layers in common the best way to make clear how the class structures are woven is to show them both in one figure. In this figure it can be seen that the class structure of the Retimer itself isn’t changed compared to figure 10-2.

Beside the Retimer structure the SmartScan structure is drawn. This structure also uses NDS and GDesign. The SmartScan class structure, doesn’t make use of RDesign, because in RDesign only the methods which have some relation with retiming are embedded. Figure 10-3
also shows that in this SmartScan class structure no counterpart of RDesign exists. SmartScan only uses GDesign which must be sufficient to develop SmartScan.

In the way described above and shown in figure 10-3 the SmartScan class structure was set up. In GDesign the extra code needed to develop SmartScan was added. Soon it became clear that a lot of functionality needed by SmartScan was implemented in RDesign, which is strictly for retiming methods.

Figure 10-3: First version of the class structure of SmartScan and the Retimer

This implied that the GDesign didn't contain all the methods which are related to both 'scan chain router' and retimer.

To preserve the class structure for SmartScan as shown in figure 10-3 a lot of methods had to be moved from RDesign to GDesign. This action wouldn't change the class structure of the Retimer and SmartScan but would lead to a lot of changes in the code to make the retimer work again. Also adding a lot of methods to GDesign leads to a large and difficult to understand layer in the class structure.

Because holding on to the original idea of the class structure of SmartScan had a high priority this movement of methods from RDesign to GDesign was performed. This action wasn't just a matter of copying code. A lot of problems were encountered such as the fact that a GNet (inherited from Net from NDS) doesn't exist in GDesign but does in RDesign (RNet).

After a time period of trying to make SmartScan work with the initial class structure it became clear that this method of implementation wasn't suitable. A new class structure for SmartScan had to be made.
The idea behind the new class structure for SmartScan is the following. Like the Retimer, SmartScan must have an extra layer on the level of RDesign which, like RDesign for the Retimer, contains all the methods which have a relation with SmartScan. This extra layer is called SSDesign (SmartScan Design). In figure 10-4 on page 74 the new and definitive class structure for SmartScan is drawn.

Figure 10-4: Definitive version of the class structure of SmartScan and the Retimer

SSDesign inherits, just like RDesign, from GDesign. SSDesign is used by SmartScan. Because SSDesign and RDesign are strongly related the first setup of SSDesign was made by copying RDesign. Because in SmartScan the retiming algorithm is used without the timing constraints, all code, coming from RDesign, which was used for these constraints could be deleted in SSDesign.

In the next section the functionality that had to be added to the Retimer to obtain SmartScan will be discussed.

10.3 The SmartScan program flow

The basis of the implementation of ScanLab on the new datastructure is formed by RetLab. To be able to decide which parts of the RetLab functionality can be left out and which functionality has to be added for SmartScan purposes a look is taken at the RetLab program flow.
In figure 10-5 on page 75 the RetLab program flow is shown. The function of each of the blocks in the flow will be briefly discussed. Also, for each block, it will be discussed if it is necessary in the SmartScan program flow and what changes need to be performed on it.

- In the first block of the flow the input files are read. Because this functionality is also required in SmartScan this part is left unchanged.

- The second step in the flow consists of the removal of buffers and flip-flops. In this step, when removing the flip-flops, weights are assigned to the consuming pins of the nets. These weights indicate the number of flip-flops that were between this consuming pin and the driving pin of this net in the original circuit. These weights are needed to build up the graph for the retiming algorithm. Because these weights are still needed
for SmartScan this action must be preserved. The removal of the buffers in the netlist has to be deleted however. This action is performed for reasons of fanout and thus isn’t needed in SmartScan

• The third step is adapting net loads. After the nets where a cell or flip-flop can’t drive the connected cells, buffers are placed. Because SmartScan may not change the original netlist structure this action has to be omitted.

• The fourth step, in the RetLab timing flow, is the timing analysis. This analysis is performed to get an estimate of the timing behaviour of the cells. Obviously this step also isn’t needed in the SmartScan flow.

• The generation of retiming constraints is performed according to the values calculated during the timing analysis step. Also the constraints given by the user are generated. Here only the causality constraints need to be generated for SmartScan purposes. After these are generated the extra constraints needed for SmartScan must also be generated.

• The retiming step is performed by an algorithm developed by Philips Research which is considered confidential. This step is also needed in the SmartScan flow.

• After the retiming step the network should be rebuilt according to the results of the algorithm. This means that flip-flops should be placed on the right position. The insertion of flip-flops in SmartScan is different than for Retlab. SmartScan must place all the flip-flops, that were in the original netlist, back at their original places. By means of different instantiation names it is made clear is a flip-flop must be made scannable or not.

• After inserting the flip-flops the buffers which are not necessary anymore are removed. This step also is performed for timing reasons, so it is obsolete in SmartScan.

• The last step will be the writing of the output files. Trivially, this step must be kept in SmartScan.

In figure 10-6 on page 77 the SmartScan program flow is depicted. The step in which the scan minimisation constraints are generated didn’t exist in RetLab, so this step had to be implemented. The insert flip-flops step had to be changed as described above. Before the implementation of the scan constraints mapping algorithm can be discussed, first an explanation must be given on how the constraints are mapped on the graph. This explanation will be given in the next section.
10.4 Mapping the constraints on the graph

After the removal of the flip-flops in the network a constrained network without flip-flops exists. Every consuming pin of the nets in this network has a weight \( W \) denoting the number of flip-flops that existed between the driving pin of this net and this consuming pin. As mentioned in section 7.3 on page 43 a graph is build from the constrained network. This is done by assuming that every cell has a vertex in the graph. The constraints are represented by the edges in the graph. An edge is a constraint with a field containing the weight value. When the graph is totally build up, it is given to the retiming algorithm.

In figure 10-7 on page 78 a general example of a NDS network is given. This example will be used to explain how a graph is built up from a constrained network. Each consuming pin \( p \) of a net in this constrained network has an attribute \( W(p) \), called weight, which denotes the number of flip-flops on this net from the driving pin of this net to the consuming pin \( p \). First all the vertices will be put in the graph. As mentioned above, each cell in the network is represented by a vertex in the graph. Beside these vertices, the inputs and outputs of the network also get a vertex in the graph. The first constraints that will be mapped on the graph are the causality constraints as described in section 7.3 on page 43.
For every input to output relation a constraint has to be generated. Looking at figure 10-7 the constraint for the input to output relation $i_0$ to $i_1$ is shown below, where $W(i_1)$ is the number of flip-flops between $i_0$ and $i_1$ and retiming value $r(i)$ is defined as the number of flip-flops which are moved from behind the cell and replaced in front of input $i$.

$$r(i_0) - r(i_1) \leq W(i_1)$$

In the graph this constraint will be represented by a directed edge from the vertex of cell $c_1$ to the vertex of input $i_0$ with a weight $W(i_1)$. The resultant graph, from the example network, after all the causality constraints are mapped on it, is shown in figure 10-8.
Next the SmartScan constraints will be mapped on the graph. These constraints are also described in section 7.3 on page 43. The first constraint makes sure that the number of scan flip-flops on an input to output connection isn't higher than the original number of D flip-flops on that connection. If the input to output connection $i0$ to $i1$ is again taken as an example, the constraint is as follows:

$$r(i1) - r(i0) \leq 0$$

It can be seen that in this equation the places of $i0$ and $i1$ are interchanged. This means that the edge must point in the opposite direction of the causality constraint edge for this connection. The edgeweight is 0. In figure 10-9 the graph is shown after the mapping of these constraints on it.

The second type of SmartScan constraints deal with input to output connections which have the same input but different outputs. Let $a$ and $b$ be connections with the same input. If the original number of flip-flops (weight $W$) on connection $a$ is larger than the weight of connection $b$ then the number of scan flip-flops on connection $a$ (weight $W_r$) will be equal or larger than on connection $b$. Looking at the input to output connections $o1$ to $i2$ and $o1$ to $i3$ and assuming that $W(i2)$ is larger than $W(i3)$ then the following constraint must be made.

$$r(i3) - r(i2) \leq W(i2) - W(i3)$$

For this constraint an edge, from $c2$ to $c3$ with weight $(W(i2) - W(i3))$, must be added to the graph. The resulting graph is shown in figure 10-9 on page 79. This graph is given to the retiming algorithm.
In the next section the implementation of the algorithm with which the scan minimisation constraints are added to the graph will be discussed. The mapping of the causality constraints on the graph was already implemented because they are also needed by RetLab.

### 10.5 Implementing the mapping of the SmartScan constraints on the graph

Now that it is known under which conditions and how the constraints need to be mapped on the graph it can be explained how this mapping is implemented.

Because the mapping of the causality constraints and the other constraints, only used by the Retimer, is implemented in GDS the mapping of the extra constraints for the Smartscanner will also be implemented in this data structure. The generation of the Smartscanner constraints is as follows.

```c
GenerateScanMinimizationConstraints( )
{
    forall input pins ip of netlist
    {
        inh = GetInternNet( ip )
        if ( inh ∧ GetPinCount( inh )>1 )
        {
            forall consuming pins cp of inh
            {
                vip = GetMyVertex( ip )
                GenerateNetScanConstraints(inh, vip, cp)
            }
        }
    }
}
```
forall childblocks cb in the netlist
{
    forall outputpins op of cb
    {
        enh = GetExtemNet(op)
        if (enh)
        {
            forall consuming pins cp of enh
            {
                vcb = GetMyVertex(cb)
                GenerateNetScanConstraints(enh, vcb, cp)
            }
        }
    }
}

GenerateNetScanConstraints(net, vertex, pin)
{
    if (GetDeclarationPin(pin))
    {
        vpin = GetMyVertex(GetOwnerBlock(pin))
        Generate an edge from vertex to vpin with weight 0
        forall consuming pins pin1 of net
        {
            if (GetDeclarationPin(pin1))
            {
                if ((GetWeight(pin) > GetWeight(pin1)) \ (pin ≠ pin1))
                {
                    vpin1 = GetMyVertex(GetOwnerBlock(pin1))
                    Generate an edge from vpin to vpin1 with weight
                    GetWeight(pin) - GetWeight(pin1).
                }
            }
            else
            {
                if ((GetWeight(pin) > GetWeight(pin1)) \ (pin ≠ pin1))
                {
                    vpin1 = GetMyVertex(pin1)
                    Generate an edge from vpin to vpin1 with weight
                    GetWeight(pin) - GetWeight(pin1).
                }
            }
        }
    }
}
10.6 Updating the flip-flop insertion algorithm

In this section first the flip-flop insertion algorithm of RetLab will be discussed. With this algorithm as a basis the SmartScan flip-flop insertion algorithm will be explained. After this the implementation of the insertion algorithm for SmartScan is discussed.

10.6.1 The RetLab flip-flop insertion algorithm

After the retiming algorithm is performed on the graph, the constraints on the network are updated. This is done by updating the weights of the consuming pins of the nets. They are changed into the resulting weights $W_r$ which are calculated by the retiming algorithm. Now the flip-flops have to be placed back in the circuit.

In figure 10-11 a net of the constrained network after retiming is shown.
This net has one driving and three consuming pins. Each consuming pin has a weight $W_r$ which denotes the number of flip-flops that have to be placed back in the network between the driving pin and this consuming pin of the net. The placement of the flip-flops on this net will be explained step by step.

Looking at the weights it can be seen that every input to output connection needs at least one flip-flop, so the first flip-flop can be placed on the net. After this the weights of all consuming pins are decremented with one because on the resulting net, to which they are connected, one flip-flop less is needed from the driving pin to each of the consuming pins. In figure 10-12 the result of this action is shown.

After the placement of the first flip-flop a problem rises. The weight of the top consuming pin is now 0 while the weights of the other consuming pins are still larger than zero. This problem can be solved by connecting this consuming pin to the net that is connected to the input of the next flip-flop that will be placed. When placing the next flip-flop only the weights of the consuming pins connected to the net after this flip-flop will be decremented with one. In figure 10-13 the resulting network after the placement of the second flip-flop is shown.
The placement of the third flip-flop causes no problems. Both consuming pins of the net before which the flip-flop is placed have a weight higher than zero. In figure 10-14 the situation after the placement of the third flip-flops is shown.

When placing the last flip-flop the same problem as with the second flip-flop is encountered. This problem is solved in the same way. The resulting network after all the flip-flops are placed is shown. Now all weights are zero so no more flip-flops have to be placed.

Figure 10-13: Network after placing the second flip-flop

Figure 10-14: Network after placing the third flip-flop

Figure 10-15: Resulting network after all flip-flops are placed
10.6.2 The SmartScan flip-flop insertion algorithm

A big difference between the flip-flop placement algorithm for RetLab and SmartScan is that SmartScan has to place the original number of flip-flops back on the original places in the network. Thus for SmartScan purposes the original weights, at the consuming pins, have to be preserved in the constrained network. Also the weights coming from the graph, on which the retiming algorithm is performed, must be visible at the consuming pins for the flip-flop placement algorithm. With these weights, after retiming, it can be decided which flip-flops have to be made scannable.

To be able to preserve the original weights at the consuming pins, an extra attribute has to be added to the pins in the constrained network. In this attribute the original weight can be stored.

In figure 10-11 a net of the constrained network after retiming algorithm has been run is shown.

![Figure 10-16: Net of the constrained network after the retiming algorithm](image)

Each consuming pin has a weight $W$ and a weight $W_r$. The weights $W$ denote the original number of flip-flops between the driving pin of the net and the consuming pin. The weights $W_r$ denote the number of flip-flops out of the original number of flip-flops which has to be made scannable for test purposes, this is the weight that comes out of the graph after retiming. Due to the constraints that were made, $W$ is always greater than or equal to $W_r$.

In the placement algorithm of SmartScan a shift register recognition is built in. This means that the number of scan flip-flops on an input to output connection, denoted by $W_r$, can be optimised. How this optimisation is achieved will be explained on the example of figure 10-11. Looking at the example it can be seen that all weights $W$ are larger than 0, which means that on all input to output connections a flip-flop has to be placed. The question is, does it have to be a scan flip-flop or not? In the example both weights of $o1$ are 2. This means that on the connection $i$ to $o1$ both flip-flops have to be scannable, i.e two flip-flops directly following each other without a tap on the net between them. To be able to directly control $o1$ it is sufficient to make the last of these two flip-flops scannable without loss of controllability or observability. The first flip-flop to be placed thus can be a normal D flip-flop. In the network is shown after the placement of the first flip-flop. After this action the weights $W$ of all the consuming pins are decremented with 1. The weights $W_r$ are left unchanged because no scan flip-flop is placed.
Figure 10-17: Network after placing the first flip-flop

Now weight \( W \) of 01 is 1, meaning that between the already placed flip-flop and 01 only 1 flip-flop has to be placed. weight \( W_r \) of 01 is larger than 0 denoting that 01 has to be directly controllable. The next flip-flop to be placed thus has to be a scan flip-flop otherwise this requirement can't be met. Consuming pin 03 has a \( W \) equal to 2 but these two flip-flops needn't be scannable because \( W_r = 0 \). The question can be asked if making one or more of these flip-flops scannable is a problem. The answer is no, the constraints given to the retiming algorithm aren't violated when on an input to output connection more flip-flops are made scannable than necessary. The network after placement of the second flip-flop is depicted in figure 10-18. Because a flip-flop is placed all the weights \( W \) are decremented with 1 and because this flip-flop has to become a scan flip-flop for test also all the weights \( W_r \), which aren't zero, are decremented with 1.

Figure 10-18: Network after placement of the second flip-flop

Weight \( W \) of 01 is now 0, which means that 01 has to be connected to the net preceding the next flip-flop that will be placed. The weight \( W_r \) of this consuming pin is still 1 but for reasons described above this needn't be a problem. Looking at the other two consuming pins it can be seen that the next flip-flop to be placed can be a D flip-flop (there is no consuming pin with \( W=1 \) and \( W_r > 0 \)).
In figure 10-19 the network is shown after placement of the third flip-flop. Only the weights \( W \) of the consuming pins, connected to the net after the last placed flip-flop are decremented with 1.

![Diagram of network after placement of the third flip-flop](image)

**Figure 10-19: Network after placement of the third flip-flop**

Only before \( o2 \) still two flip-flops need to be placed. Because \( W_r \) of \( o2 \) is larger than zero the last one of these two flip-flops needs to become scannable for test purposes (see the placement of the first two flip-flops). \( o3 \) is connected to the net preceding the first of those two flip-flops because \( o3 \) doesn't need flip-flops placed in front of him anymore. In figure 10-19 the network is shown in which all flip-flops are placed.

![Diagram of network after placement of all flip-flops](image)

**Figure 10-20: Network after placement of all flip-flops**

### 10.6.3 Implementation of the SmartScan flip-flop insertion algorithm

Now that it is known under how the flip-flops are placed back in the network it can be explained how this placement is implemented in SmartScan. Because the flip-flop placement algorithm used in SmartScan is different from RetLab the...
flip-flop placement must be placed in the data structure that is specific for SmartScan, SSDS. The implementation of the flip-flop placement algorithm as explained in section 10.6.2 is as follows.

InsertFlipflops( library, netlist, domaincontrol )
{
    Let flipflopnumber = 0
    create a null pointer clock_nh
    create an empty list ll
    forall nets nh in netlist
    {
        if ( nh is not a clock net )
        {
            add nh to linklist ll
        }
        else
        {
            clock_nh = nh
        }
    }
    if ( !clock_nh )
    {
        Add a clock pin and a net to the netlist
    }
    targetflipflop = GetTargetFlipflop( domaincontrol )
    forall nets nh in ll
    {
        InsertNetFlipflops(netlist, clock_nh, library, targetflipflop, flipflopnumber )
    }
}

InsertNetFlipflops(netlist, clock_nh, library, targetflipflop, flipflopnumber )
{
    Let insertscanreg = false
    forall consuming pins cons_ph of nh
    {
        while ( weight W of cons_ph > 0 )
        {
            forall consuming pins cons_ph1 of nh
            {
                if ( weight Wr of cons_ph1 > 0 ∧ weight W of of cons_ph1 =1)
                {
                    insertscanreg = true
                }
            }
        }
    }
}
flipflopnumber++
if ( insertscanreg )
    create flipflop instantiation name "fx"<flipflopnumber>
else
    create flipflop instantiation name "ff"<flipflopnumber>
make new net new_nh and connect it to the D pin of the new flip-flop
forall driving pins dp of nh
{
    disconnect dp from net nh
    connect dp to net new_nh
}
connect the Q pin of the new flip-flop to net nh
connect the clock pin of the new flip-flop to net clock_nh
UpdateWeights( nh, new_nh, insertscanreg )
}

UpdateWeights( nh, new_nh, insertscanreg )
{
forall consuming pins cons_ph of nh
{
    if ( weight W of cons_ph = 0 )
    {
        disconnect pin cons_ph from net nh
        connect pin cons_ph to net new_nh
    }
    else
    {
        decrement weight W of cons_ph with 1
        if ( insertscanreg ➗ weight W_r of cons_ph > 0 )
            decrement weight W_r of cons_ph with 1
    }
}
11 Optimising SmartScan

11.1 Introduction

So far, the program which divided the memory elements into two groups was called ScanLab. In the previous chapter the implementation of this selection program on a new data structure was discussed. From now on this selection algorithm will be called SmartScan.

Until now the SmartScan algorithm that is implemented divides the memory elements in a circuit into two groups by means of altering the instantiation names. The memory elements in the first group, denoted by instantiation names beginning with ‘ff’ in the netlist, needn’t be changed for test purposes. The memory elements in the second group, denoted by instantiation names beginning with ‘fx’, need to be controllable and observable for test purposes. In figure 11-1 a part of a circuit, on which this division is performed by SmartScan, is shown. The fat vertical bars are the memory elements that have to become controllable and observable. The thin vertical bars are the memory elements that needn’t be changed for test purposes.

Looking at the above figure it can be seen that the distance, in number of clock cycles, from the left scan flip-flop to its following scan flip-flops is different for both paths. For the top path three clock cycles are needed to propagate data from the left scan flip-flop to the top right scan flip-flop. For the bottom right scan flip-flop four clock cycles are needed. This difference in distance forms a problem when the circuit is under test.

When the circuit is under test the responses, that are clocked into the elements of the scan chain during the normal mode, must all be available at one point in time. This way the data that is scanned out of the scan chain can be compared with the response data generated by the combinatorial pattern generator. In the above example data from the left scan flip-flop can be propagated to the top right scan flip-flop in three clock cycles. For the bottom right scan flip-flop this takes four clock cycles so these two responses aren’t available at the same point in time for both scan flip-flops at the right. In section 5.1.2 on page 22 a solution to this problem is presented. By making the left flip-flop holdable scannable and holding the same stimulus data for three clock cycles in this flip-flop the two right scan-flip-flops will, after four clock cy-
cles, contain response data that belongs to the same stimulus. Until now, determining which scan flip-flops send out on more than one path with varying distances was impossible with SmartScan. This means that all flip-flops, selected to become controllable and observable, have to be made holdable scannable. Also, the maximum distance between the scan flip-flops, in number of clock cycles, cannot be calculated by SmartScan. To get a first indication of this distance the number of pipelines that needn’t be made controllable and observable is taken.

Looking at the status of SmartScan until now the following shortcomings can be listed.

- SmartScan cannot determine if a memory element, that must be made scannable, sends out on more paths with varying distances. For this reason all these memory elements have to be made holdable scannable. This introduces an area overhead which is two times the area overhead caused by a scan flip-flop. When the SmartScan algorithm selects more than half of the flip-flops to become controllable and observable the area overhead is more than with Full Scan.

- Second, with SmartScan, the exact distances between the memory elements that need to become holdable scannable cannot be determined. Only a slight indication can be achieved by counting the number of pipelines in the circuit.

In the next section a Partial Scan method called “BALLAST” is described. In this section a summary is given of [Gupt90]. This summary still describes the BALLAST method in detail because later a comparison of this method with SmartScan will be made. According to the results coming from this comparison it can be decided if the scan path implementation methods used in this method can also be used for SmartScan.

11.2 The BALLAST methodology for structured Partial Scan design

BALLAST (BALAnced structure Scan Test) is a partial scan technique which selects scan path storage elements (SPSE’s) such that the remainder of the circuit has certain desirable testability properties. A complete test set is obtained using combinatorial ATPG. The number of SPSE’s that needs to be provided with a hold mode is minimized by ordering the registers in the scan path and formatting the test patterns appropriately. The BALLAST methodology is described in [Gupt90]

11.2.1 B-structures

In the BALLAST method scan flip-flops are selected in such a way that the kernel (the portion of the circuit excluding the scan path) belongs to the class of easily testable structures called B(alanced)-structures.

The topology of the circuit is modelled by a directed topology graph. In this graph each vertex represents a maximal cloud of combinatorial logic in the circuit such that its inputs are either PI’s or outputs of flip-flops and its outputs are either PO’s or inputs of flip-flops. Each
edge in the graph represents a connection between two clouds through a register. When S is a sequential circuit that is modelled by a graph G then S is said to be a balanced sequential structure (B-structure) if

- G is acyclic;
- \( \forall v_1, v_2 \in \text{Vertices} \), all directed paths (if any) from \( v_1 \) to \( v_2 \) are of equal length (this actually includes condition 1); and
- \( \forall e \in \text{Edges} \), if e represents a hold register and e is removed from G, the resulting graph is disconnected.

Given such a balanced sequential structure \( S^B \) then its combinational equivalent \( C^B \) is the combinational circuit formed by replacing each FF in every register in \( S^B \) by a wire (if the output of the register uses the \( Q \) output of the FF) or an inverter (if it uses the \( \bar{Q} \) output of the FF) or both. The depth \( d \) of \( S^B \) is the longest directed path in its topology graph. If an input pattern I is applied to \( S^B \) then the single pattern output of \( S^B \) for I is defined as the steady state output of \( S^B \) when I is held constant at the inputs to \( S^B \) and all its registers are operated in the LOAD mode for at least \( d \) clock cycles. If \( S^B \) contains a fault f and the single-pattern outputs for I of the good and the faulty circuits are different, then I is a single-pattern test for f.

The two interesting properties of B-structures which allow them to be used as kernels in the BALLAST partial scan design are:

- they are single pattern testable; and
- a complete single-pattern test set can be derived using combinatorial test generation techniques.

### 11.2.2 Scan design using B-structures

When a register is included in a scan path it serves as a control and observation point for the rest of the circuit. It becomes a PO for the cloud feeding it and a PI for the cloud it drives. Thus, in the circuit model described above, the inclusion of a register in the scan path corresponds to its removal from the topology graph; the reduced topology graph represents the kernel, i.e., the portion of the circuit to be tested using the scan path.

The BALLAST methodology is outlined as follows.

1. Construct topology graph G of the circuit as defined in section 11.2.1.

2. Select a minimum cost set of arcs \( R \) to be removed from G such that the remaining topology graph is balanced. The arcs in \( R \) represent the registers that must be included in the scan path. If \( S^B \) is the B-structure corresponding to the resulting topology graph, which represents the kernel of the circuit.
3. Determine the combinational equivalent \( C^B \) of the kernel \( S^B \). With combinational ATPG, determine a complete test set \( T \) for \( C^B \). Since \( S^B \) is balanced, \( T \) is a complete single-pattern test set for all detectable faults in the kernel \( S^B \).

4. Scan path construction by appropriately ordering registers in \( R \) and connecting them so that they are capable of

1. shifting test patterns in/out,
2. holding patterns constant at the kernel inputs for \( d \) clock cycles, and
3. Loading test results from the kernel.

Requirement 2. can be achieved by providing all scan registers with a HOLD mode. In section 11.2.3 it will be shown that the need for all scan registers to have a HOLD mode can be partially removed.

A test plan for a circuit designed in the manner described above is as follows. \( N \) is the number of test patterns and \( I \) is the number of FF's in the scan path.

1. Operate all scan registers in the SHIFT mode for \( I \) clock cycles. (scan in the first test pattern.)

2. repeat \( N \) times:

1. Place all scan registers in the HOLD mode and all nonscan registers in the LOAD mode for \( d \) clock cycles. (Data can propagate through the kernel.)

2. Operate all scan registers in the Load mode for one clock cycle. (load test result into the scan registers.)

3. Operate all scan registers in the SHIFT mode for \( I \) clock cycles. (Scan out test result and scan in next test pattern.)

11.2.3 Implementation of the scan path

The BALLAST test plan, as described in the previous section, is the same testplan as used until now with the smart scan method. However the authors of the article on BALLAST also introduce another test method. By introducing dummy bits the need for some scan path registers to have a hold mode can be eliminated.

The distance between a driving scan path register and a receiving scan path register, when at least one path exist between them, is the number of registers in any path between the driver and the receiver (excluding themselves) through the kernel. If the distance from a given driver to all receivers to which it has a path (including itself if it is also a receiver) have the same value, define the distance of the driver as this value. Otherwise the distance is undefined. When a driver has a distance \( x \) the test result loaded into the scan path during clock cycle \( k \) depends only on the value present in the driver at clock cycle \( k-1-x \).

The implementation of the scan path in the BALLAST method is as follows.

© Philips Electronics N.V. 94
For all drivers which have an undefined distance a test result due to each test pattern depends on a single pattern being present in the driver for more than one clock cycle.

1. All drivers which distances are undefined must have hold modes

The Scan path is constructed by forming $d+2$ groups of scan path registers, where $d$ is the maximal distance in the B-structure. Group 0 is the group closest to the scan in pin, group $d+1$ is the group closest to the scan out pin. Within each group the ordering of the elements may be based on routing considerations. The scan path registers are assigned to groups according to the three rules below:

2. To minimize the number of shifts needed to shift a new pattern into the scan path all pure receivers are placed in group $d+1$ because they never have to be supplied with test data. Drivers with a defined distance need not have a hold mode, provided the arrival time of the desired test data is synchronized with the appropriate arrival times of test data in the other drivers.

To achieve this drivers with a large distance are placed further along the scan path. Test patterns for all are concatenated into a single pattern and scanned in. Between the test data destined for different groups of drivers dummy bits are inserted into the pattern, so that test data for drivers with smaller distances lag behind those destined for drivers with larger distances. The third rule is as follows.

3. All drivers not having hold modes and having distance $i$ must be placed in group $i$

Drivers with hold modes are placed in group $d$ so that they are among the first drivers to receive their correct pattern, which they then hold for the duration of the test.

4. All drivers with HOLD modes for test and all drivers with HOLD modes already present are placed in group $d$.

To ensure that the pattern for each group of drivers lags behind the pattern for its neighbouring group by one clock cycle the following rule is presented.

5. Each single-pattern test is modified by introducing one dummy bit between each pair of consecutive groups $i$ and $i+1$, $0 \leq i \leq d-1$.

If a group contains no scan path elements, still a dummy bit for this group is introduced. This way between the single-patterns for neighbouring groups with a difference in distance of $n$, $n$ dummy bits are introduced.

If the scan path is implemented as described in this section, the formal test procedure given in section 11.2.2 needs to be modified by generalizing step 2.1 as follows.

2.1. Place all scan registers having a HOLD mode in the HOLD mode, all other scan registers in the SHIFT mode, and all nonscan registers in the LOAD mode for $d$ clock cycles.

In figure 11-2 an example of a scan path implemented in the way described above is shown.
Example of a test pattern that is shifted into the scan chain:

$1X0X10XX1X010$

Figure 11-2: Example of a BALLAST scan path implementation

The figure is a simplified representation of a circuit with a scan path. The scan path registers are denoted by rectangles. The memory elements that are not contained in the scan path are denoted by fat horizontal bars. Above the top scan path elements the distance as previously defined is given. U means that the distance for this element is undefined. In this example the scan path contains no pure receivers. The two memory elements, in the scan chain, with undefined distances are made holdable and have been placed at the end of the scan chain. The scan path registers with defined distances are placed in the chain before the hold registers. They are placed in the scan chain in order of increasing distance for reasons described above.

In the figure an example of a test pattern that is shifted into the scan chain is depicted. In this pattern the 'X' bits are stuffing bits. First the test pattern for the hold flip-flops and for the flip-flop with the largest distance (5) are shifted in. The hold flip-flops are placed in the hold mode when they have clocked in their test pattern. In order to ensure that the results from the test patterns for the flip-flops with distance 4 and distance 5 reach the scan chain at the bottom at the same time a stuffing bit must be inserted between these two patterns. This ensures that the test pattern for the flip-flop with distance 4 lags one clock cycle behind the test pattern for the flip-flop with distance 5. The same holds for the flip-flops with distance 2 and the flip-flop with distance 4. Only now two stuffing bits have to be inserted in the test pattern because the difference in distance between the two groups is 2. Performing the same stuffing bit insertion between the test patterns for the other groups results in the test pattern as shown in
the figure.
By shifting the resultant test pattern, with inserted stuffing bits, into the scan chain it is made sure that the test results are all at the same time available for the scan chain elements in the bottom scan chain of the figure. These results can thus all be observed with one shift out action.

11.3 The BALLAST method compared to the SmartScan method
In the BALLAST methodology the flip-flops that need to be entered in the scan chain are selected in such a way that the remainder of the circuit is a balanced structure. For a graph that is build up from this structure, in the way described in the previous section, the following three rules must hold.

1. The Graph must be acyclic
2. for all vertices v1 and v2 in the graph all directed paths in the graph from v1 to v2 must be of equal length and
3. If and Edge in the graph denotes a holdable flip-flop, removing this edge from the graph must result in a disconnected graph

If the same rules hold for the resultant sequential circuit, from the SmartScan flip-flop selection algorithm, can be shown by imaginary building up the BALLAST graph for this circuit. The reader is reminded of the fact that the SmartScan selection algorithm is based on the retiming algorithm.
As explained in chapter 5 there are no timing constraints because the retiming algorithm is given an unendless clock period at which the circuit must operate. What remains is the causality constraint from the retiming algorithm to which two constraints are added for SmartScan purposes. Adding those two constraints can never result in a solution for which the causality constraint, and thus the retiming constraints, don't hold anymore. Also the optimisations performed during the flip-flop placement only concern shift registers which are a pure cascade of memory elements. This algorithm makes sure that only one element in such a register is made scannable. This results in the same observability and controllability. Also all connections in the circuit that must be broken according to the retiming algorithm remain broken because at least one controllable and observable memory element will remain on such a connection after the placement algorithm.
After the BALLAST graph is build from the SmartScan circuit the above mentioned three rules must hold.

1. Because retiming cannot change the number of memory elements in a cycle, according to this algorithm all these elements must be made scannable. After the flip-flop placement, still at least one of these elements is scannable. So when These two algorithms are applied to a circuit all its cycles will be at least broken by one scan element.
2. When a circuit contains a reconvergent path retiming can only remove the same number of memory elements from each path which belongs to this reconvergent path. When flip-
flops remain on a path which is part of a reconvergent path at least one of them is made
scannable after the flip-flop placement. This results in a circuit in which all reconvergent
paths of unequal length are broken up by a scan flip-flop.

3. Because hold registers aren’t supported by the retiming algorithm circuits which contain
these elements can’t be SmartScanned. The third rule can therefore always be applied.

Since all three rules hold for a graph build up from a circuit resulting from SmartScan it can
be concluded that a SmartScanned circuit is a B-structure. The implementation of the scan-
path used in the BALLAST method can thus also be used for SmartScan.

11.4 Optimising SmartScan for use of BALLAST scan path
implementation method

To be able to use the BALLAST scan path implementation method SmartScan must add the
following information to its resultant netlist.

1. For each PI of the circuit and for each driving scan path register (also a primary input)
the distance(s) to the POs of the circuit and to receiving scan path registers (also primary
outputs) must be calculated if there exists at least one path between them. Each Primary
input and each scan path memory element must have a property in which its distance(s)
are listed.

2. Each memory element in the circuit must have a property which defines if this memory
element must be inserted into the scan path. Also, when a memory element must be in-
serted into the scan path the property must make clear if this memory element should
have a hold mode.

3. The netlist itself must have a property which denotes the maximum distance within the
circuit. This distance is needed for the flip-flops in the scan path which have a hold
mode. When they have received their test data they must hold it for a number of hold cy-
cles which equals this distance. Furthermore the netlist must receive a property from
SmartScan which marks this netlist as a SmartScanned netlist.

11.4.1 Determining the distances in the SmartScanned circuit

11.4.1.1 Theory

Before the distances in a SmartScanned circuit can be determined first a definition of these
distances has to be made. In the BALLAST method this distance was defined as the number
of registers between a driving scan path register and a receiving scan path register, when at
least one path exists between them. Let a driver have a distance x, as defined above. The test
result loaded into the receiver in the scan path during clock cycle k depends only on the value
present in the driver at clock cycle k-1-x.
For SmartScan the distance is defined as follows:

The distance between a driving scan path register and a receiving scan path register, when at least one path exists between them, is the number of clock cycles needed to propagate data, present in the driver, through the circuit to the receiver.

When for example two scan path registers are connected by one path which contains two registers (no scan path registers) then the BALLAST distance is 2. The SmartScan distance is 3 because it takes three clock cycles to propagate data from the driver into the receiver. The SmartScan distances are thus always 1 larger than the BALLAST distances on the same paths.

When the distances form a driver to all the receivers to which it has a path aren’t all equal the distance will be undefined for this driver in BALLAST. In SmartScan such a driver will receive a list of all distances to the receivers to which it has a path. This driver thus will have a multiple distance.

Let a driver have a SmartScan distance x. The test result loaded into the receiver in the scan path during clock cycle k depends only on the value present in the driver at clock cycle k-x.

Now that the SmartScan distance is defined it will be explained how these distances are determined in the SmartScanned circuit.

In figure 11-3 a circuit which has been SmartScanned is drawn.

![Figure 11-3: SmartScanned circuit](image)

In this circuit the combinatorial elements are represented by the rectangles filled with 'comb'. The flip-flops that are selected, by SmartScan, to be inserted in the scan chain are represented by rectangles filled with 'spff' (scan path flip-flop). The rectangles filled with 'ff' are the flip-flops in the circuit which needn’t be entered in the scan chain. The clock net isn’t drawn because it isn’t needed to explain the distance calculation algorithm.

First all cells in the circuit must have an attribute which denotes the growth in distance on a path due to this cell. For this purpose one of the weights, denoting the number of flip-flops that have to be placed back in the circuit (see section 10.6.2 on page 85), that are on each pin...

© Philips Electronics N.V.

99
in the circuit, can be reused. Because each path through a cell passes through one of its inputs the distance growth factor (DGF) will be presented in the weight attribute of each input pin of the cell. For combinatorial cells this factor will be 0 because the distance on a path doesn’t change when such a cell is passed. For each sequential cell the factor will be 1 because the distance of the path grows with 1 if it passes such a cell.

Each primary output pin of the circuit also receives a DGF of 1. This can be explained as follows. If between two scan path registers one path exists which contains no sequential elements then the distance for the driver is 1. This means that one clock cycle is needed to propagate the data from the driver to the receiver on this path. When one path exists between a scan path register and a primary output no clock cycles are needed to propagate the data from the driver to this primary output, denoting a distance 0. However, when the circuit is under test the results on the primary outputs are always checked in the clock cycle that precedes the first shift out clock cycle. Test results on the primary outputs are thus always one clock cycle sooner present than test results in the scan chain elements. A scan path driver which drives on both paths, as described above, needn’t be made holdable because the data on the primary output must arrive one clock cycle before the data in the scan path receiver. Both paths have thus equal length, so the primary output will also receive a DGF of 1. Figure 11-3 shows the placement of the DGFs in the circuit.

Figure 11-4: Placement of DGFs in the circuit

Now the distances of the paths in the circuit can be calculated with the DGFs. This is done by propagating these DGFs from the primary outputs back to the primary inputs and the driving scan path registers, for which the distances must be calculated.

The output ports of each cell as well as the input ports of the circuit will receive an attribute which lists the distance(s) from this cell or port to the scan path registers or the primary outputs to which it has a path.

The list of distances (LOD) for an output port of a cell can only be calculated when the following holds. (The same rules hold for each input port of the circuit.)

1. The output port of the cell is directly connected by a net to one or more primary output(s) of the circuit or to one or more scan path register inputs.

© Philips Electronics N.V.
2. For all cells, not being scan path registers, from which one or more input(s) are connected to the net for which this output port is a driving port the following must hold. The LODs of all the output ports of all of these cells are already calculated.

When these two rules hold for an output port o of a cell the LOD can be calculated as follows. (The same holds for an input port of the circuit.)

1. Add the DGF of all primary outputs or inputs of scan path elements, to which the output port o is connected by a net, to the LOD of the output port o if this distance isn't already in it.

2. For each input port i, of a cell c which is not a scan path register, to which the output port o is connected perform the following action.
   For each element e from each LOD from each output port of c add the DFG from i to e. The resulting distance from this addition is added to the LOD of o if this distance isn't already in it.

Looking at the circuit in figure 11-3 the only output ports for which the LOD can be calculated, in this situation, are the ports directly connected to the primary outputs of the circuit or connected to the inputs of scan path registers or both. The result of this calculation is shown in figure 11-3. The LODs of the output ports are listed between parenthesis.

![Figure 11-5: Calculation of the first two LODs](image-url)
Now the LODs of the output ports connected to cells for which the LODs of all output ports are calculated can be determined. The result is depicted in figure 11-3.

![Diagram](image)

**Figure 11-6: Result after the second calculation run**

In figure 11-3 it can be seen that only the LOD of the input port i isn’t calculated yet. According to the calculation rules this LOD can now also be determined. In figure 11-7 the final result of the LOD calculation is depicted.

![Diagram](image)

**Figure 11-7: Final result of the distance calculation**

After the distance calculation algorithm has finished all outputs from every scan path flip-flop and all inputs of the circuit have a list of distances attached to them. In this example the distance lists for the scan path elements in the circuit are complete. However, when a scan path element has two output ports (q and q̄) it has two distance lists attached to it. These lists have to be merged to obtain the list of distances for the scan path element. This merging operation is performed during the placement of the distance properties.
in the netlist which will be discussed in section 10.6.4. In the next section first the implementation of the distance calculation algorithm will be discussed.

11.4.1.2 implementation

The distance calculation algorithm is implemented in SSDS because this code is only used by SmartScan.

CalculateDistances( netlist )
{
    forall output ports op of netlist
        SetWeight(op) = 1
    forall child blocks ch of netlist
    {
        forall input ports ip of ch
            {
                if( GetType( ch ) = Combinatorial )
                    SetWeight(ip) = 0
                else
                    SetWeight(ip) = 1
            }
    }
    forall nets nh of netlist
    {
        setUserflag(nh)=0
    }
    Let again = true
    Let netnotready = false
    while( again )
    {
        again = false
        forall nets nh of netlist
        {
            if ( UserFlag( nh ) = 0 )
            {
                netnotready = CheckIfNetCanBeCalculated(nh)
                if( netnotready )
                    again = true
                else
                    CalculateNetDistances( nh )
            }
        }
    }
CheckIfNetCanBeCalculated(nh)
{
    Let netnotready = false
    if( GetUserFlag(nh) = 0 )
    {
        forall consuming pins cons_ph of nh
        {
            if( GetDeclarationPort(cons_ph) )
            {
                owner_ch = GetOwnerCell(cons_ph)
                if ( not( GetType(owner_ph) = ScanFlipFlop ) )
                {
                    forall output ports out_ph of owner_ch
                    {
                        out_nh = GetExternNet( out_ph )
                        if( out_nh )
                        {
                            if( GetUserFlag( out_nh ) = 0 )
                            {
                                netnotready = true
                                return netnotready
                            }
                        }
                    }
                }
            }
        }
        return netnotready
    }
}

CalculateNetDistances( nh )
{
    SetUserFlag(nh) = 1
    forall consuming ports cons_ph of net nh
    {
        if( GetDeclarationPort(cons_ph) )
        {
            owner_ch = GetOwnerCell( cons_ph )
            if ( not( GetType(owner_ch) = ScanFlipFlop ) )
            {
                forall output ports out_ph of owner_ch
                {
                    if( NumberOfHoldElements( out_ph ) )
                    {
                        forall holdelements hold of out_ph
                        {
                            forall driving ports drv_ph of nh
AddHold( drv_ph,
    (hold-1+GetWeight(cons_ph)) )
}
}
ClearCellLinklist( owner_ch )
}
else
{
    forall driving ports drv_ph of nh
        AddHold( drv_ph, GetWeight(cons_ph) )
    }
}
else
{
    forall driving ports drv_ph of nh
        AddHold( drv_ph, GetWeight(cons_ph) )
    }
    ClearCellLinkList( owner_ch )

    let done = true
    forall input ports ip of owner_ch
    {
        ext_nh = GetExternNet( ip )
        if( ext_nh )
        {
            if( GetUserFlag( ext_nh ) = 0 )
                done = false
        }
    }
    if( done )
    {
        forall output ports op of owner_ch
            ClearHold( op )
    }
}
10.6.4 Adding the SmartScan properties to the netlist

10.6.4.1 Theory

The SmartScan algorithm first selects the memory elements that should be entered in the scan chain. The flip-flops which will not become an element of this chain can be marked as D-flip-flops in the SmartScanned netlist.

After the SmartScan distances have been calculated the distance information can be added to the netlist.

Based on this distance information it can be decided which scan path elements, selected by the SmartScan algorithm, need a hold mode. This hold mode must be introduced for those scan path elements which are drivers for two or more paths with differing distances. These scan path elements must be marked as holdable scannable flip-flops in the SmartScanned netlist. The remaining scan path registers can be marked as scan flip-flops.

From the distance information also the maximum distance can be determined and added as information to the netlist. Finally the netlist will be marked as being a SmartScanned netlist. The information added by SmartScan to the netlist will have the following syntax.

"SmartType = DFF" denotes that this flip-flop must be left unaltered

"SmartType = SFF" denotes that this flip-flop must be entered in the scan chain and must be replaced by its scannable variant.

"SmartType = HSFF" denotes that this flip-flop must become a scan chain element and must be replaced by its holdable scannable variant.

"SmartDist = distancelist" denotes the SmartScan distance(s) of this scan chain element or primary input port.

"MaxDist = distance" denotes the maximum SmartScan distance in the circuit.

"SmartScan" denotes that this netlist is SmartScanned.

In the next section the implementation of the placement of this information in the circuit netlist will be described.

10.6.4.2 Implementation

WriteSmartScanProperties( netlist )
{
  Let MaximumDistance = 0
  forall cells ch in the netlist
  {
    if( ch is a scan chain element )
    {
      Let CellDistanceList = empty
      }
forall output ports out_ph of ch
{
    forall distances d in the PortDistanceList of out_ph
    {
        if( d not element from CellDistanceList )
            Add d to the CellDistanceList
        if( d > MaximumDistance )
            MaximumDistance = d
    }
    if( Number of elements of the CellDistanceList > 1 )
        Add the property “SmartType = HSFF” to ch
    if( Number of elements of the CellDistanceList = 1 )
        Add the property “SmartType = SFF” to ch
        Add the property “SmartDist = CellDistanceList” to ch
    Clear CellDistanceList
}
forall input ports ip of the netlist
{
    if ( ip is not a clock port )
    {
        if ( number of elements in the PortDistanceList > 1 )
            Sort the elements in the PortDistanceList
        if ( number of elements in the PortDistanceList > 0 )
            Add the property “SmartDist = PortDistanceList” to port ip
        else
            !SmartScan warning: input pin ip not connected
    }
    Add the property “SmartScan” to the netlist
    Add the property “MaxDist = MaximumDistance” to the netlist
}
12 SmartScan compared to ScanLab

12.1 Introduction

In chapter 10 it was discussed how the tool ScanLab was implemented on a new data structure. In parallel with this action ScanLab was renamed to SmartScan. The object oriented programming environment, on which SmartScan is implemented, simplifies the implementation of new optimisations a lot.

The first optimisation that has been developed and implemented was discussed in chapter 11. With this optimisation the distance(s) from each scan flip-flop and primary input of the circuit to its subsequent scan flip-flop(s) or primary output(s) can be determined in the number of clock cycles. With this distance information it can be determined which scan flip-flops need to receive a hold mode and which don't. Also the BALLAST scan path implementation method can now be used to rout the scan chains in the circuit.

In this chapter the performance of SmartScan, which has been optimised as described above, will be compared to ScanLab. A comparison between the results from the combinatorial ATPG tool Amsal run on a circuit processed by SmartScan or ScanLab isn't made in this chapter. SmartScan and ScanLab both use the same (retiming) algorithm to determine which flip-flops in a circuit need to become observable and controllable for test purposes. In both cases these flip-flops are replaced by scan flip-flops for test pattern generation purposes. Because the same flip-flops are selected to be made scannable the scan chain routing (BALLAST) can be made identical for both methods. This will results in identical circuits on which test patterns have to be generated. The result of the test pattern generation tool will thus also be equal which makes a comparison useless.

12.2 Results on the ISCAS'89 benchmark circuits

In table 12-1 on page 111 the experimental results for both SmartScan and ScanLab are listed.

In the columns SFF the percentage of D-type flip-flops that must be made scannable is denoted. The columns HSFF list the percentages of D-type flip-flops that needs to become holdable scannable is listed.

For the listing of the ScanLab results the column SFF is left out. Because ScanLab adds no distance information to each flip-flop that needs to be changed for test purposes, all flip-flops need to be made holdable scannable to account for paths with multiple distances (see section 11.1 on page 91). Only when all memory elements in the circuit are selected by ScanLab to receive a test mode each flip-flop must only be made scannable. In this situation the result of ScanLab (which selects memory elements according to the smart scan method) is the same as when full scan method is used. In this situation SmartScan will also make all flip-flops scannable (uses the same selection algorithm) so its performance will be equal to ScanLab.

In the columns 'Extra area' the percentage gain in area compared to the extra area introduced
when testing with the full scan method is given. It is assumed that making a D-type flip-flop holdable scannable produces twice the extra area of making a D-flip-flop scannable. The percentages in these columns are calculated as follows:

When all D-type flip-flops are selected by ScanLab or SmartScan to receive a test mode the test method will be equal to the full scan method. The extra area introduced, compared to the extra area introduced by the full scan method, will thus be 0%. In all other cases the percentage extra area compared to the full scan extra area is calculated as follows:

Extra area (Δ%) = SFF (%) + 2*HSFF (%) - 100%

Looking at the results on the ISCAS’89 benchmarks (see table 12-1 on page 111) the following can be seen. When 100% of the D-type flip-flops receives a test mode, as expected, the difference in extra area compared to the full scan method is 0% for both SmartScan and ScanLab. When less flip-flops are selected to receive a test mode mostly an enormous reduction in extra area is achieved with SmartScan compared to ScanLab. Only for the s953 benchmark the extra area introduced, compared to the extra area of full scan, for both tools is equal. All flip-flops, with test mode, in this benchmark have thus a multiple distance meaning that SmartScan can’t optimise the ScanLab solution. For the rest of the benchmarks the extra area introduced, compared to the extra area of full scan, drops with percentages between 38 and 91% for SmartScan compared to ScanLab. For the s15850 benchmark the extra area, compared to full scan, even drops from 90% for ScanLab to -1% for SmartScan! A reduction from an extra area that is almost twice the extra area of full scan to an extra area that is even less than that of full scan!

In the last column of the table the SmartScan extra area is compared to the full scan extra area. Here it can be seen that even for these benchmarks, which contain almost no pipelines, the extra area introduced by SmartScan, compared to full scan, is never more than 33%. For ScanLab this percentage could rise upto 92%, meaning almost a duplication of the full scan extra area.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>ScanLab</th>
<th>SmartScan</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HSFF (%)</td>
<td>Extra area(Δ%)</td>
</tr>
<tr>
<td>s27</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>s208</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>s208.1</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>s298</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>s344</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>s349</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>s382</td>
<td>71</td>
<td>42</td>
</tr>
<tr>
<td>s386</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>s400</td>
<td>71</td>
<td>42</td>
</tr>
<tr>
<td>s420</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>s444</td>
<td>71</td>
<td>42</td>
</tr>
<tr>
<td>s510</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>s526</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>s526n</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>s641</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>s713</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>s820</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>s832</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>s838</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>s953</td>
<td>21</td>
<td>-58</td>
</tr>
<tr>
<td>s1196</td>
<td>89</td>
<td>78</td>
</tr>
<tr>
<td>s1238</td>
<td>89</td>
<td>78</td>
</tr>
<tr>
<td>s1423</td>
<td>97</td>
<td>94</td>
</tr>
<tr>
<td>s1488</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>s1494</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>s5378</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>s9234</td>
<td>96</td>
<td>92</td>
</tr>
<tr>
<td>s13207</td>
<td>82</td>
<td>64</td>
</tr>
<tr>
<td>s15850</td>
<td>95</td>
<td>90</td>
</tr>
<tr>
<td>s35932</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>s38417</td>
<td>90</td>
<td>80</td>
</tr>
<tr>
<td>s38584</td>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 12-1: Results for the ISCAS'89 Benchmarks

© Philips Electronics N.V. 111
12.3 Results on the Melzonic benchmarks

In table 12-2 on page 113 results coming from both ScanLab and SmartScan performed on building blocks of the Melzonic are listed. In this table a strange phenomenon can be seen. Because ScanLab and SmartScan use the same selection algorithm to determine which flip-flops should receive a hold mode, it can be expected that the same amount of flip-flops is selected by both tools. However, looking at the results on the formatteruv it can be seen that this is not the case for this example. According to ScanLab 29% (HSFF percentage) of the flip-flops should receive a test mode. With SmartScan this percentage is 57% (SFF percentage + HSFF percentage)! This phenomenon also occurs at some other benchmarks.

After research on this phenomenon it became clear that this rise in flip-flops with test mode, for SmartScan, is caused by an equal optimal but different solution of the retiming algorithm compared to ScanLab.

‘Equal optimal’ means that after the retiming algorithm the summation of all the weights $W_r$ on all consuming pins from all nets in the network are equal for both the SmartScan and the ScanLab retiming. The total summation of these weights $W_r$ however doesn’t contain any information on how these weights are spread over the consuming pins and thus over the nets. When these weights are more spread over the nets the situation can occur that the flip-flop insertion algorithm (see section 10.6.2 on page 85) can perform less optimisations on the retiming solution and thus places more flip-flops that need a test mode. An example of this situation is presented in figure 12-1.

Figure 12-1: Example of a less optimal retiming solution for smart scan purposes

In this figure the retiming solution for both situations is equal optimal. For both the total weight $W_r$ is 2. The flip-flop insertion algorithm (explained in section 10.6.2) can however optimise the retiming solution in example A through shift register recognition. Example B can’t be optimised by this algorithm. The retiming algorithm in SmartScan thus presents a solution in which the weights $W_r$ are more spread over the nets so that it can be less optimised by the flip-flop insertion algorithm.
Looking at the results in table 12-2 it can be seen that for all circuits which contain flip-flops with a test mode the SmartScan solution is more optimal than the ScanLab solution, even with a less optimal selection of the test mode flip-flops. For the reformatteruv benchmark the extra area even drops from 56% more extra area than the full scan method for ScanLab to 22% less extra area than the full scan method for SmartScan. When the extra area introduced by both tools is compared to the extra area when full scan is used the following can be seen.

With ScanLab three benchmarks have a larger extra area. This growth in extra area for the benchmark reformatteruv is even 56%. For SmartScan the extra area on all benchmarks is less! For all these benchmarks testing with the smart scan method will thus be more attractive than with full scan due to less extra area, less test time and no drop in fault coverage when the circuit contains no sequential redundancy!

### 12.4 Results on the Fadic123 benchmark

In table 12-3 results from SmartScan and ScanLab, when applied on the Fadic123 benchmark, are listed. As mentioned earlier this chip is designed for audio purposes and is thus not as highly pipelined as the Melzonic. With ScanLab 58% of the flip-flops on this chip needed to be made holdable scannable. This caused an extra area of 16% more compared to the full scan extra area. With SmartScan this extra area drops to 41.8% less extra area than with full scan
scan. Looking also at the results on the Fadic123 in section 9.3 it can be concluded that testing this benchmark with SmartScan is much more cost effective than with the full scan method.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>ScanLab</th>
<th>SmartScan</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HSFF (%)</td>
<td>Extra area(Δ%)</td>
</tr>
<tr>
<td>fadic_logic</td>
<td>58</td>
<td>16%</td>
</tr>
</tbody>
</table>

Table 12-3: Results on the Fadic123 benchmark

12.5 Conclusions on the results achieved with SmartScan

Looking at the results, described in the last few sections and in chapter 9, achieved by SmartScan the following can be concluded. The extra area that was still a major burden for ScanLab, when a circuit was not strongly pipelined, has been removed with the use of SmartScan. Also for less pipelined circuits testing with the smart scan method is now in many cases more attractive than testing with full scan.

For almost all real life circuits which contain one or more pipelines testing with smart scan will be more cost effective than testing with full scan. Smart scan reduces both extra area for test and test time while keeping the same fault coverage as full scan when the circuit contains no ‘sequential redundancy’ (see section 9.1.1)!
13 Future optimisations

In this chapter future optimisations on the SmartScan tool and on tools related to SmartScan will be discussed.

13.1 Future optimisations on SmartScan

13.1.1 Flip-flop placement

During the SmartScan program flow, which was treated in detail in chapter 10, the circuit description is altered twice. First the flip-flops are removed from the circuit. During this removal each consuming pin from every net in the circuit description receives a weight $W$ denoting the number of flip-flops that was present between the driving pin of this net and this consuming pin. The network that results after all memory elements have been removed is called the constrained network.

From this constrained network a graph is built up on which the retiming algorithm is performed. After the retiming algorithm has finished the flip-flops are placed back in the network.

During this placement the weights after retiming are used to determine which flip-flops need to be altered for test purposes. The original weights, from the constrained network before retiming, are used so that on each connection the original number of flip-flops is placed back. However, as explained in section 6.3.4, by removing the flip-flops and inserting the weights in the network exact information on how the flip-flops where originally placed is lost.

For retiming purposes information on how the flip-flops are exactly placed in the original circuit is of no importance. The retiming tool RetLab may add delete or reposition flip-flops in a circuit until the desired specification is met.

SmartScan, however, may not alter the position and number of flip-flops in the circuit. This tool may only add information to the netlist for smart scan purposes. Information on the exact number of flip-flops and their place in the circuit must therefore be preserved. Preserving this information can be done in two ways.

1. Providing the constrained network with more information.

The first way is to add more information to the constrained network during the removal of the flip-flops. This information must be such that the flip-flop placement in the original circuit can be derived from it.

When this method is used more code sharing can be practiced between the retiming tool RetLab and SmartScan because their program flows remain more identical. The extra information added to the constrained network needn't be used to built up the graph. This way the graph, built up from the constrained network isn't changed compared to the old graph.

It can be seen that with this method only minor modifications are needed to add the extra information to the constrained network during flip-flop removal and to substract the extra information from it during flip-flop placement.

However, using this method the following problem rises. Because the graph isn't changed, it

© Philips Electronics N.V. 115
still lacks the exact information on how the flip-flops are placed in the network. For this reason the constraint that exists because of this lack of placement information (see section 6.3.4) must still be mapped on the graph. This can cause the retiming algorithm to come up with a non-optimal solution to our problem.

2. Not removing the memory elements from the network

The second method of preserving the exact placement of the flip-flops is by not removing them. With this method a different program flow must be developed. In order to be able to use the same algorithm to map the constrained network on the graph it must be built up in the following way:

For each flip-flop that is encountered in the network all the consuming pins of the net, that is connected to the output of the flip-flop with its driving pin, receive a weight 1. The consuming pins of all other nets have a weight 0. The flip-flops thus aren’t removed from the network anymore. An example of such a constrained network is shown in figure 13-1.

![Figure 13-1: Example of a constrained network](image)

In this constrained network all consuming pins of one net will always have the same weight so the second smart scan constraint (see section 6.3.4) isn’t needed anymore. When the graph is built up from this network each memory element, like all combinatorial elements, will also be represented by a vertex in the graph. The constraints are mapped in the same way as before on the graph. Because the second constraint isn’t needed anymore the mapping of this constraint can be removed from the program flow. The graph which is built up from this constrained network will contain the exact information on how the flip-flops are placed in the network. For this reason the retiming algorithm will always present the optimal solution. In figure 13-2 the graph that is built up in this way from the example in figure 13-1 is shown. It can be seen that like the combinatorial elements also each flip-flop in the network has a vertex in the Graph.
After retiming the weights $W_r$ are placed on the constrained network (see section 10.6.2). The weight $W_r$ for all consuming pins of net connected to the output of a flip-flop can be 0 or 1 (all consuming pins on one net have the same $W_r$). $W_r=0$ denotes that this flip-flop needn't be made scannable. When $W_r=1$ the flip-flop must be made scannable. The flip-flop insertion algorithm explained in section 10.6.2 must thus be transformed into a flip-flop tagging algorithm. This algorithm tags the flip-flops as follows: When the consuming pins from a net have a weight $W=1$ and a weight $W_r=0$ the flip-flop that drives this net must remain unaltered and thus must be tagged as a D-type flip-flop. When weight $W=1$ and weight $W_r=1$ the flip-flop that drives this net must be tagged as a scan flip-flop. Nets of which the consuming pins have a weight $W=0$ aren't driven by a flip-flop so nothing has to be tagged here. Looking at the flip-flop insertion algorithm in section 10.6.2 it can be seen that this algorithm sometimes optimises the solution of the retiming algorithm. This optimisation is performed when more than one flip-flop has to be made scannable, according to the retiming solution, in a pure shift register (a chain of flip-flops of which only the last one drives a net with more than one consuming pin). In this situation the insertion algorithm will make only the last flip-flop scannable. In order to make the tag algorithm as optimal as the insertion algorithm the constrained network has to be processed before this tag algorithm is performed on it. After this process the consuming pins of only one flip-flop per shift register may have a weight $W_r$ equal to 1. This way only one register per shift register will be tagged as scan register.

### 13.1.2 Recognition of shift registers which contain combinatorial elements

Until now optimisation of the retiming solution by the flip-flop insertion algorithm is only performed on shift registers of which only the last flip-flop sends out on a net with more than one consuming pin and in which no combinatorial elements are present (see figure 13-3a).
However, a chain of flip-flops in which each Q output of a flip-flop is connected to the D input of the following flip-flop by one or more combinatorial elements and only the last flip-flop sends out on a net with more than one consuming pin (see figure 13-3b) can sometimes also be a candidate for optimisation of the retiming solution.

A further optimisation of SmartScan can thus be achieved when these shift registers can also be recognised. This recognition algorithm must adapt the weights $W_r$ in the same way as the adaption of weights for shift registers mentioned in the previous section, with which it can be combined.

### 13.1.3 Optimisation on the SmartScan selection algorithm

In section 12.3 the problem of a less optimal flip-flop selection of SmartScan compared to ScanLab was addressed. Erik Jan Marinissen has come up with a possible solution to this problem, which will be presented here.

The idea behind this solution is to make the retiming algorithm come up with a solution that is more optimal for **smart scan** purposes. As can be seen in section 12.3 a retiming solution is more optimal for **smart scan** when the spreading of the weights $W_r$ over the nets is minimised. When this is the case the flip-flop insertion algorithm can perform the largest optimisation on the retiming solution.

The Retiming algorithm can be forced to produce such a more optimal solution. When a larger number of scan flip-flops on a connection is represented by a relatively smaller weight than a smaller number of flip-flops retiming will search for a solution in which scan flip-flops are more concentrated on a few connections because his way the total of the weights $W_r$ will be less and thus the solution more optimal according to retiming.
13.2 Optimisation of InScan

As described in chapter 6 SmartScan is a preprocessing tool for the scan path insertion tool InScan. SmartScan adds information to the design description which must be used by InScan to make the design ‘SmartScannable’. In figure 13-4 a schematic overview of InScan is given...

![Figure 13-4: Schematic overview of InScan](image)

In this figure it can be seen that within InScan two sorts of tools can be distinguished, PrepScan and ScanIt. In PrepScan an analysis from the design is performed which results in certain proposals which can be edited by the user. ScanIt only executes these propositions. From this figure it can be seen that the design description which is processed by SmartScan must be analysed by PrepScan. PrepScan thus must be changed in such a way that it can use the information, added to the design description by SmartScan, to generate a scan path implementation proposition according to the BALLAST scan path implementation technique. ScanIt must be able to execute this proposition that makes the design ‘SmartScannable’ when it is executed by ScanIt.

13.2.1 One chain per distance

In section 11.2.3 the BALLAST scan path implementation was discussed. When this implementation is used the difference in distance between the neighbouring distance groups in the scan chain has to be accounted for by the test protocol. This means that while the stimuli are shifted in the scan chain the test protocol must add stuffing bits between patterns that belong to different distance groups. When, however, each group of scan path flip-flops with the same distance would be routed in one or more separate scan chain(s) (all flip-flops that need a hold mode are also routed in one or more separate scan chain(s)) no bit stuffing is needed. Because within each scan chain all
flip-flops now have the same distance the stuffing bits, that account for differences in distances, aren't needed anymore.

The Routing plan file (Rpl) that is generated by PrepScan from the SmartScanned netlist, according to this method, must have the following properties:

1. All flip-flops within one chain in the routing plan must have the same distance and must be driven by the same clock. With respect to this all flip-flops with more than one distance are considered as flip-flops with the same distance so they can be placed together in one or more scan chains. These chains must have a holdenable pin because their chain elements must have a hold mode.

2. The routing plan must contain a clockdomain for each group of flip-flops with the same distance. In each clockdomain all chains which contain memory elements from the same distance group, driven by the same clock, are placed.

3. Because each clockdomain contains all the memory elements from one distance group the distance information for each group must only be mentioned once in the routing plan, by adding a distance property to each clockdomain. Due to this a distance group can be routed in more than one chain without the need of adding more distance information to the routing plan.

4. The clockdomain in which the chains are placed which contain the memory elements with multiple distance must have a distance property which is equal to the maximum distance present in the circuit.

5. Besides a macro of a chain which contains one scan flip-flop a macro of a chain with one holdable scan flip-flop must also be present when the design contains flip-flops with multiple distances.

The matchrule file must also contain a matchrule for a holdable scannable flip-flop when flip-flops with multiple distances exist in the SmartScanned circuit.

When the Routing plan file (Rpl) and the Match rule file are constructed by PrepScan as described above ScanIt must execute these propositions. ScanIt must be updated so that it can handle a routing plan which contains a distance property at each clockdomain. The actions that ScanIt performs needn't be changed. Like before, it has to execute the propositions of PrepScan only now it must be able to ignore the distance properties in the routing plan. ScanIt has already been updated to this specification by Paul Merkus.

The test protocol generator that is needed, when this method of scan chain routing is used, has already been developed. The test protocols that it generates will be explained with the aid of an example. A circuit is defined as follows:

It contains two scan chains, ch1 with a distance 2 and a length 4 and ch2 of length 3 with elements with multiple distances. The circuit has two inputs, i1 with a distance 1 and i2 with a
multiple distance, and one output o. Let the MaxDist in the circuit be 3.
The test protocol that must be generated for this circuit with information from the routing plan and the SmartScanned netlist is graphically represented in figure 13-5.

![Graphical representation of the test protocol for the example circuit](image)

**Figure 13-5: Graphical representation of the test protocol for the example circuit**

It can be seen that the distance from a chain is accounted for by earlier scanning in its pattern. The pattern of ch2 is held for MaxDist cycles because its elements have multiple distance. For the input with multiple distance the pattern is applied MaxDist-1 clock cycles earlier. The pattern for i1 is applied at clock cycle 0 because its distance is 1. Due to all this, the responses at all normal outputs of the circuit can be watched at clock cycle 0. The scan out of the responses in all scan chains can start at clock cycle 1, just as with a normal full scan test protocol.

The test protocol generator which generates a test protocol as described above is already implemented. The implementation is discussed in section 8.3.3.2.

When this scan path implementation is used the test protocol that needs to be generator doesn't have to account for differences in distances within one scan chain (no need for bit stuffing). With this method however the variation in length of the scan chains depends on the distance variation in the circuit. This can lead to a non optimal spreading of the flip-flops over the scan chains. Also if the scan path memory elements in the circuit have many different distances equal as many scan chains have to be inserted in the circuit. This can lead to a need for too many in- and output pins.

### 13.2.2 BALLAST scan path implementation

The BALLAST scan path implementation was discussed in section 11.2.3. When this method is used to make a design scannable PrepScan must be optimised to generate a routing plan in the following way:
1. the elements in each scan chain in the routing plan must be placed according to the placement rules defined in section 11.2.3.

2. A property has to be added to each scan chain which defines which distances are represented in the scan chain and how many elements of each distance are present. With this information a test protocol can be generated which adds the correct amount of stuffing bits at the correct places. This information can, for reasons of readability, also be put in a separate file.

In this case ScanIt must be updated in the same way as described in the previous section.

With this scan path implementation method the scan chains can be routed more freely. The order within the scan chain is still strictly defined but it may now contain elements with different distances. This extra freedom in routing can lead to a drop in area overhead compared to the previous method. The overhead, due to routing, of both routing methods will almost always be more than the overhead from routing without restrictions.

The protocol for shifting in the test patterns in the scan chains was also discussed in section 11.2.3. Due to this protocol, which accounts for the differences in distances in the scan chain, the responses will all be present at the same time in the scan chain. The appliance of the stimuli at the inputs of the circuit and controlling the responses are performed in the same way as described in the previous section.
Conclusion

In this report a special partial scan method, called smart scan, was described. Also the specification and implementation of a part of SmartScan was described. SmartScan is a software tool that selects, according to the smart scan method, memory elements that must be replaced by their scannable variant. Based on the selection information the scan path insertion tool InScan must add scan paths to a design. This design will then be called 'smart scannable', i.e. prepared for smart scan.

At Philips Research a software tool which selects the memory elements, as described above, was already developed. This tool is called ScanLab.

To be able to decide whether or not testing with the smart scan method is a promising technique, worth the effort of further developing a tool for, a test flow was built up around ScanLab to test the smart scan theory on real life circuits. This flow was called 'experimental SmartScan'.

The results that were achieved with this flow on real life circuits were very promising. Especially on highly-pipelined circuits, such as high-speed video and audio circuits, testing with the smart scan method always leaded to a drop in test time and in many cases also to a drop in area cost for test compared to testing with the full scan method. The average drop in test time was 50%. Due to the large number of less pipelined examples the extra area for test was averagely 10% more for smart scan compared to full scan. Based on the positive results with respect to test time for all examples and the reduction in extra area for the highly pipelined circuits it was decided to further develop the tool ScanLab.

In order to make the implementation of future optimisations on ScanLab much more easier, the implementation of this tool was first put on a new data structure. In parallel to the implementation on the new data structure the name ‘ScanLab’ was changed into ‘SmartScan’.

The first optimisation that has been developed addresses the problem of extra area for test introduced by ScanLab, which was still averagely larger than with full scan.

To account for the difference in number of flip-flops on paths driven by the scan path flip-flops, all of them needed to be made holdable scannable. The area overhead of smart scan compared to full scan was thus only smaller when less than half of the flip-flops was selected to receive a test mode.

Based on the BALLAST scan path implementation technique and test protocol a first optimisation was developed and implemented on SmartScan to reduce this problem of too much extra area for test. By determining the distance(s) of each selected flip-flop to the subsequent selected flip-flops it could be decided if it needed a hold mode.

Results from experiments with the optimised SmartScan showed an enormous improvement with respect to extra area for test. Now the extra area needed was averagely 15% less compared to testing with full scan!

Looking at the overall results it can be concluded that for almost all circuits, which contain one or more pipeline(s), testing with smart scan will be more cost effective than testing with full scan. Experiments on real life circuits showed an average reduction of 15% on extra area for test compared to testing with full scan! Averagely the test time, compared to full scan,
was even halved. For most examples *smart scan* achieved the same fault coverage as *full scan*. Only for circuits which contain 'sequential redundancy' the fault coverage will be less for *smart scan*.

In the final chapter of this report some optimisations that can be performed on SmartScan, and tools which are related to SmartScan are discussed. In some cases SmartScan altered the number of flip-flops in the circuit. Two methods are presented which could be used to prevent SmartScan from altering the circuit. Also an optimisation in the selection of flip-flops is presented which can lead to a smaller set of flip-flops selected to receive a test mode. For the scan path insertion tool InScan optimisations are discussed, which must lead to the support of the BALLAST scan path implementation method or a scan path implementation derived from this.
Bibliography

The best flip-flops to scan
In: Proceedings International Test Conference 1991, Nashville, TN, USA,

A complete solution to the partial scan problem
In: Proceedings International Test Conference 1987, Washington, DC, USA,
1-3 Sept. 1987, p. 44-51.

A complete solution to the partial scan problem
In: International Test Conference 1987 Proceedings: 'Integration of Test with
Design and Manufacturing.', Washington, DC, USA, 1-3 Sept., p. 44-51.

Designing circuits with partial scan

[Benn84] Bennetts, R.G.
Design of testable logic circuits
Addison-Wesley, 1984

[Benn93] Bennetts, R.G. et al.
Partial Scan: what Problem does it Solve?
Proceedings European Test Conference, Rotterdam, The Netherlands,
April 19-22 1993, pp. 99-106.

Pascant: a partial scan and test generation system
In: Proceedings of the IEEE 1991 Custom Integrated Circuit Conference,

[Brgl89] Brglez, F. et al.
Combinational profiles of sequential benchmark circuits
In: 1989 IEEE International Symposium on Circuits and Systems, Portland, OR,
[Chen89.1] Cheng, K.T. et al.
Concurrent test generation and design for testability

[Chen89.2] Cheng, K.-T. et al.
An economical scan design for sequential logic test generation

[Chen90] Cheng, K.-T. et al.
A partial scan method for sequential circuits with feedback
Murray Hill, NJ, USA: AT&T Bell Lab., 1990.

[Chen92] Chen, P.C. et al.
Gradually-on structure for scan design

[Chic90] Chickermane, V. et al.
An optimization based approach to the partial scan design problem

[Chic91] Chickermane, V. et al.
A fault oriented partial scan design approach

[Clus84] McCluskey, E.J.
A survey of design for testability techniques

[Der89] Dervisoglu, B.I.
Scan-path architecture for pseudorandom testing

[Edt93] EdtKit v0.9 User Manual
[Fuji83]  Fujiwara, H. et al.
On the Acceleration of Test Generation Algorithms
pp. 1137-1144.

[Funa79]  Funatsu, S. et al.
Easily testable design of large digital circuits

[Funa89]  Funatsu, S. et al.
Scan design at NEC

[Gibb89]  Gibbons A.
Algorithmic graph theory

On the selection of a partial scan path with respect to target faults
In: Proceedings of the European Conference on Design Automation, EDAC,

The Ballast methodology for structured partial scan design
Los Angeles, CA, USA: Dept. of Electr. Eng. -Syst., Univ. of Southern
California, 1990.

[Hoge93]  A. Hogenhuis , C.S.D. van der Blij
Development of RetLab: a tool to retime digital circuits

Scan design simplifies IC testing

Timing-driven partial scan
In: 1991 IEEE International Conference on Computer-Aided Design Digest of
Technical Papers, ICCAD '91, Santa Clara, CA, USA, 11-14 Nov. 1991,
p. 408-411.
Partial scan by use of empirical testability

An analytical approach to the partial scan problem
Karlsruhe, West Germany: Design and Fault Tolerance Karlsruhe Univ., 1990

[Kunz92] Kunzmann, A.B.
Bounding the test sequence length of sequential circuits by the partial scan design

On determining scan flip-flops in partial-scan designs

[Lioy91] Lioy, A. et al.
A hierarchical multi-level test generation system

An incomplete scan design approach to test generation for sequential Machines

A methodology for partial scan design
[Tris80] Trischler, E.
Incomplete scan path with an automatic test generation methodology
In: Proceedings International Test Conference 1980, Philadelphia, PA, USA,

[Tris83] Trischler, E. Testability analysis and incomplete scan path
In: Proceedings International Conference on Computer-Aided Design,

[Tris84] Trischler, E.
Design for testability using incomplete scan path and testability analysis

[Werf91] Werf A. van der
RetLab: Optimal design of synchronous digital circuits by Retiming
internal publication of Philips Electronics N.V., RWR-124-AW-91113-aw
REPORT 6581, Eindhoven, 1991

[Zhan91] Zhang, Z. et al.
An approach to optimal DFT using partial parallel scan
In: Proceedings Pacific Rim International Symposium on Fault Tolerant Systems,