Eindhoven University of Technology

MASTER

InScan : specification and implementation of a scan chain inserter

van de Voort, T.A.F.M.

Award date:
1993

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

EB481
1997

InScan:
Specification and Implementation of a scan chain inserter

by

T.A.F.M. van de Voort

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

December 1993

© Philips Electronics N.V. 1993
All rights reserved. Reproduction in whole or in part is prohibited without written consent of the copyright owner.

InScan:

Specification and Implementation of a scan chain inserter

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 design are replaced by scannable variants, 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.
This report describes the specification and implementation of InScan. InScan is a software tool that prepares a design for scan test. InScan has been set up in such a way that the user can interact with the program, to steer the actions performed by it. We will show how InScan modifies the design so that it can be tested with full scan, and we will also discuss how InScan is and can be extended with partial scan algorithms.

Keywords

scan design, full scan, partial scan, scan path, design-for-testability, testability, InScan

© 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 April 1993 up to December 1993.

Acknowledgements

First of all I would like to thank my mentor Paul Merkus for his excellent and valuable support during my work at Philips. Furthermore, I would like to thank Hans Bouwmeester for his useful comments regarding the InScan User Manual. Also, I would like to thank Krijn Kuiper, Rudi Stans, Steven Oostdijk, and the other members of the group for their support and guidance. A special word of thanks is due to my fellow students Martijn van Balen and Ruud van der Meer, for the many fruitful discussions concerning our projects, and their pleasant company during the months I spent at Philips. Finally, I would like to thank my Supervisor Prof. Ir. Rene Segers for giving me the opportunity to carry out this project in his group.

Philips Research Laboratories,

Eindhoven, December 1993

© Philips Electronics N.V.

Contents

1 **Introduction** 5

2 **IC design and testing** 7

2.1 IC design 7

2.2 IC test 9

2.3 Structure test 10

3 **Scan test** 13

3.1 Design for testability 13

3.2 Introduction to scan test 14

3.3 Advantages and disadvantages of scan test 15

4 **A formal model of hierarchical circuits** 17

4.1 Ports 17

4.2 Instances of a set of cells 18

4.3 Port references of a set of cells 18

4.4 Nets of a set of cells 19

4.5 Cells 19

4.6 Descendence 20

4.7 Descendent graph 21

4.8 Functionality of cells 24

4.9 Relation between the model and reality 27

4.9.1 Leaf cells 27

4.9.2 Non-leaf cells 27

4.9.3 Instances 27

4.9.4 Ports 28

4.9.5 Nets 28

4.9.6 Naming conventions 28

5 **InScan** 29

5.1 PrepScan and ScanIt 29

5.2 The interface between PrepScan and ScanIt 30

5.2.1 The routing plan language 30

5.2.2 The match rule format 30

5.2.3 Schematic overview 30

6 **The routing plan language** 31

6.1 Hierarchy: The macro 31

6.2 Clock domains . . . 33
6.3 Chains . . . 34
6.4 Chain elements . . . 35
6.5 Formal model of scan chains . . . 37

7 PrepScan 43

7.1 Introduction . . . 43
7.1.1 Implementation matters . . . 43
7.2 The first four modules . . . 45
7.2.1 Reading the library . . . 45
7.2.2 Checking the library . . . 45
7.2.3 Reading the design . . . 45
7.2.4 Creation of the restart file . . . 46
7.3 Creation of the routing plan . . . 46
7.3.1 Clock domain analysis . . . 46
7.3.1.1 Logic in the clock line . . . 46
7.3.1.2 Tracing the clock lines . . . 47
7.3.2 Implementation . . . 49
7.3.3 An example of routing plan generation . . . 51
7.4 Plugging in partial scan algorithms . . . 55
7.5 Creation of the chain ports . . . 56
7.5.1 Sharing ports . . . 56
7.5.2 Leaf macro port creation . . . 57
7.5.3 Naming conventions . . . 58
7.6 Creation of the match rule file . . . 58
7.6.1 Format of the match rule file . . . 58
7.6.2 Creation of the match rule file . . . 60
7.6.2.1 Checking an existing match rule . . . 60
7.6.2.2 Creating a match rule with direct substitution . . 61
7.6.2.3 Creating a match rule with a multiplexer . . . 64
7.6.3 Extensions for future use . . . 66

8 ScanIt 69

8.1 Introduction . . . 69
8.2 The first three modules . . . 69
8.2.1 Reading the binary restart file . . . 69

 © Philips Electronics N.V.

8.2.2 Reading the match rule file 70
8.2.3 Reading the routing plan 70
8.3 Checking the plans 71
8.4 Executing the routing plan 80

9 Optimisation techniques **83**
9.1 Introduction 83
9.2 Shift register recognition 84
9.2.1 Theory 84
9.2.2 Implementation 85
9.2.3 Conclusion 90
9.3 Pipeline recognition 91
9.3.1 Theory 91
9.3.2 Special pipeline cases 93
9.3.3 Implementation 94
9.4 Timing driven scan chain routing 94
9.5 Layout driven scan chain routing 96

10 Conclusions **97**

A Mathematical notation **99**
A.1 Abbreviations 99
A.2 Sets and tuples 99
A.3 Predefined sets 99
A.4 Operators 100

B Routing plan syntax **101**
B.1 Introduction 101
B.2 Syntax 101
B.3 Semantics 102
B.4 Semantic rules 103
B.4.1 Rules regarding uniqueness 103
B.4.2 Rules regarding order of definition 103
B.4.3 Rule regarding length of chains 103
B.4.4 Rules related to the design 104
B.4.5 Other rules 104

Bibliography **105**

© Philips Electronics N.V.

1 Introduction

Testing of Integrated Circuits (ICs) has become a major issue in research and development of Very Large Scale Integration (VLSI). ICs have been growing continuously in number of transistors and complexity. This has caused an increase in the probability of both design errors and manufacturing faults. Manufacturing faults result in defects on the IC, while design errors result in an undesired functionality of the chip. To prevent design errors, the design must be checked at several stages of the design. It is not possible to prevent manufacturing faults, in fact, the yield is typically between 40 and 80 percent. Because the market asks for reliable, zero defect ICs, every single IC produced must pass several tests. One of these tests is the *structure test*. The idea of structure test is that all structures, created on the silicon surface, must be tested for correctness. The problem of structure testing of ICs is complicated by the enormous amount of possible faults, and the limited accessibility of parts of the ICs via the IC pins.

The requirements for a structure test are strict. The test should be fast because it is performed on every individual IC, and it must still have a high fault coverage since customers do not accept faulty ICs. Furthermore, it is important that we are able to generate this test in a short time in order to prevent lengthy design times. It is generally believed that these requirements can only be met if already during the design phase the IC testability is taken into account. This is referred to as *design for testability* (DFT).

There are many DFT techniques. In general, they operate by improving the controllability and observability of the circuit. This means that test patterns can be transported more easily to isolated parts of the IC and the responses can be more easily captured. Generating test patterns becomes less complex and the fault coverage increases. The DFT technique that we are concerned with in this report is *scan test*.

Scan test is a DFT method that improves the testability of a design drastically by creating an extra 'test' operation 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 do this, all memory elements must be replaced by versions that are equipped with such a test mode (we will call such a version the scannable variant). A scan path is a shift register that is only active during test mode. Through this scan path test patterns can be applied to the rest of the circuit, that now only contains combinatorial logic. After applying a test pattern during normal mode, the responses can be captured at the primary outputs and the inputs of the memory elements. The captured responses in the memory elements can then be shifted out during test mode and gathered by the tester. In this way, test pattern generation has only to be done for the combinatorial part of the circuit, thus easing the process of structure testing very much.

Scan test, as explained in the previous paragraph, is also referred to as *full scan*, which means that all memory elements are replaced by a scannable variant. This variant has a few extra pins and is somewhat larger that the original memory element, due to the implementation of the test mode (typically by putting a multiplexer in front of the memory element). When 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 multiplexer 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 implementation of the scan paths. In general, this will however influence the benefits of scan test, e.g., the testability may decrease, which results in longer test pattern generation times.

This report deals with the specification and implementation of InScan. InScan is a software tool that prepares a design for scan test. It substitutes the memory elements by variants with a test mode, and routes one or more scan paths. A specific language, called the 'Routing Plan Language', has been designed that describes the scan paths in a design by means of a 'routing plan'. Based on this language, a toolkit is created that makes it possible to perform certain operations on a routing plan. The routing plan language makes it very easy to implement or change the scan paths. With its aid, implementing partial scan has become a whole lot easier. InScan has been set up in such a way that partial scan algorithms can be 'plugged in'. A partial scan algorithm is presented that makes an existing shift register part of a scan path without replacing all the memory elements it consists of by scannable variants. The algorithm will modify the routing plan without influencing the testability of the design.

This report comprises the following activities:
Chapter 2 gives a brief introduction to the testing of ICs.

Chapter 3 is addressing the subject of scan test, which is important for this report.

Chapter 4 presents a formal model for the description of hierarchical circuits. This model will be used in subsequent chapters.

Chapter 5 gives an overview of the program this report is concerned with, namely InScan. It discusses the modular construction, based on PrepScan and ScanIt, and introduces the routing plan and match rule file.

Chapter 6 gives an extensive treatment of the routing plan. Also, a formal model for the description of scan chains is given.

Chapter 7 discusses the first part InScan consists of, namely PrepScan. This chapter goes into great detail discussing the operation of PrepScan, especially regarding the creation of the routing plan and the match rule file.

Chapter 8 discusses the second part InScan consists of, namely ScanIt. Especially the checking of the routing plan and match rule file is extensively treated.

Chapter 9 introduces several methods to optimise the routing plan. Their operation and implementation is discussed, as well as their (dis)advantages.

© Philips Electronics N.V.

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 1.1.

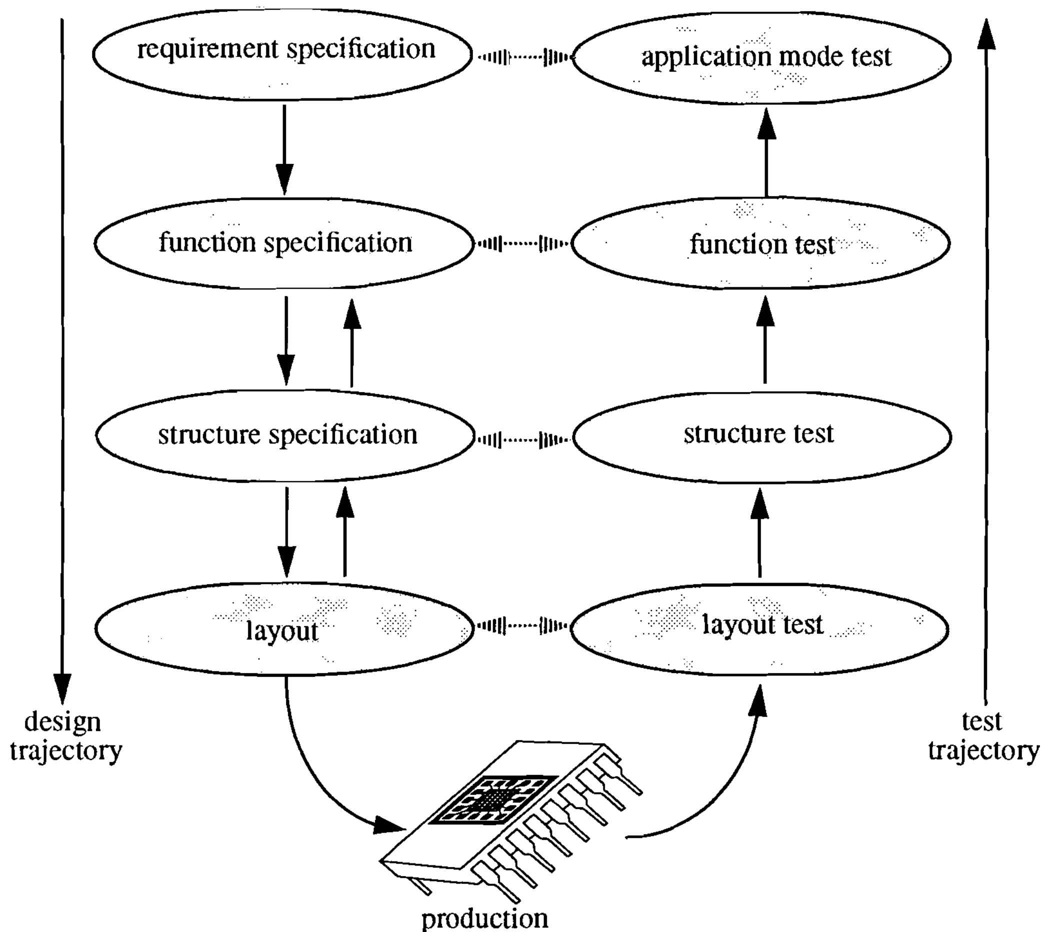


Figure 1.1: The IC design and test trajectory

Below, we describe the design phases that are mentioned in figure 1.1.

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 1.1. 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 advan-

© Philips Electronics N.V.

tage 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 defect. 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 1.1, 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 $2^N.2^P$ different states. If there are only 50 flipflops 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 1.1. 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 sight of figure 1.1:

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 mean-

ing 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.

 © Philips Electronics N.V.

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 [Bennets 84]:

"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're 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 chapter 1 the difference between a non-scan design and a scan design is given. The

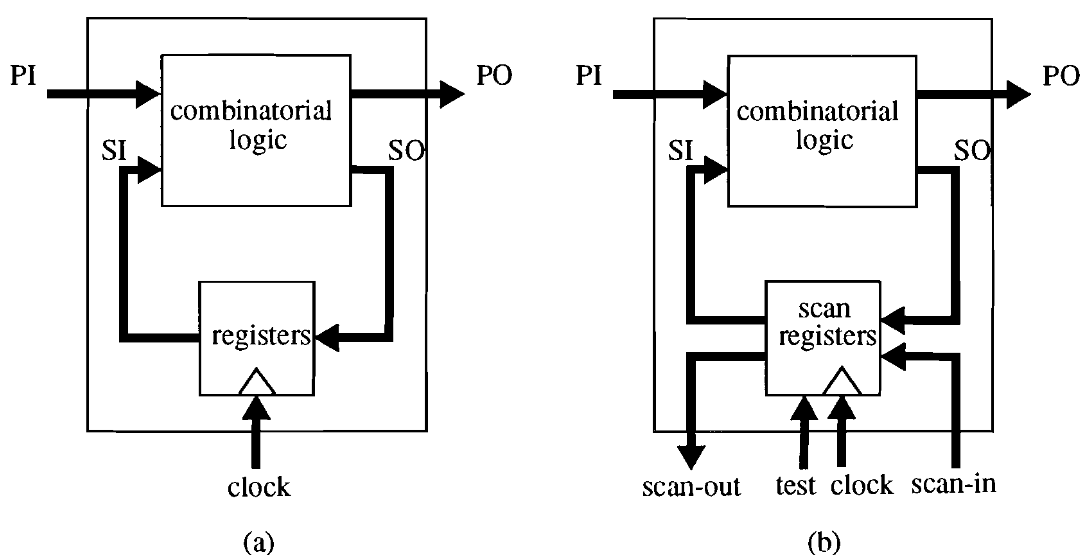


Figure 2.1: (a) Normal sequential circuit
(b) Scan version of sequential circuit

classical Huffman model of a sequential circuit is shown in figure 2.1(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.

In figure 2.1(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 2.1(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 reg-

© Philips Electronics N.V.

ister. This means that the registers now form a shift register when put in test mode. In figure 2.2 this is illustrated. Testing has now become much more easier. Since all memory elements

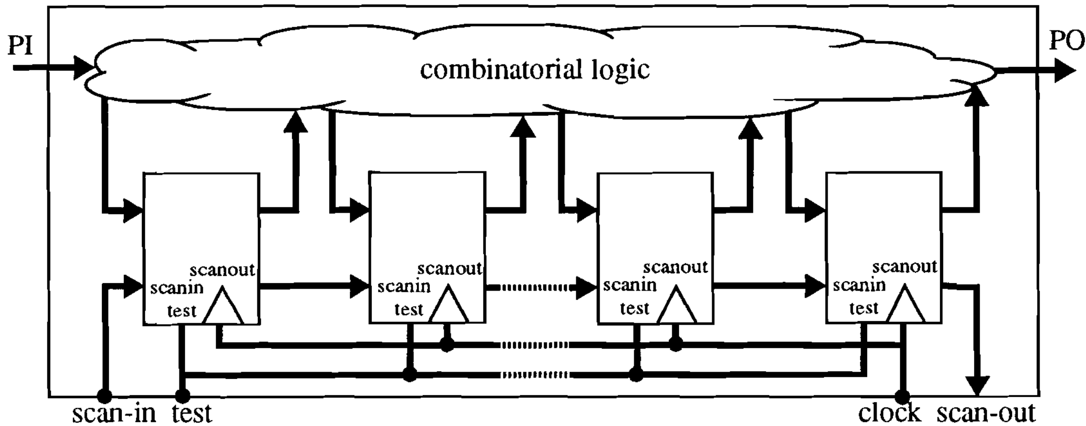


Figure 2.2: Scan register concatenation into a shift register

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.

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 9 (Optimisation techniques) 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 [Bennets 93] 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 [Fujiwara 83].

- 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 [Bennets 93].

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 9, is a way of reducing the costs that come with scan test, and may therefore be of interest to the designer.

© Philips Electronics N.V.

4 A formal model of hierarchical circuits

In this chapter we will define a model for describing multiple instance, hierarchical circuits. The names used for several objects of the model stem from the EDIF [EIA 87] terminology as much as possible. The reader who is familiar with EDIF, may want to use his intuitive interpretation of these terms.

The EDIF format [EIA 87] is used as the netlist format that is used to model designs. Also another format, NDL (Netlist Description Language), a Philips proprietary, is used for this purpose. The program that is described in this report, InScan, is built with the aid of the NDS/LDS toolkit (Netlist Data Structure/Library Data Structure) [NDS]. Unfortunately, the naming conventions that are used in EDIF, NDL and NDS/LDS are not the same. Therefore, an overview of these naming conventions is given at the end of this chapter.

To give the reader a clear understanding of the notions used in the formal model, a more informal explanation with examples will also be given at the end of this chapter. Furthermore, in appendix A the mathematical notation that we use is explained.

4.1 Ports

Definition 4.1: **[set of ports]**

The *set of ports* (denoted by *PORT*) is defined as the set which contains all *ports*. A port is a basic entity which will be used as a base for all further definitions.

□

Definition 4.2: **[set of directions]**

The *set of directions* (denoted by *DIR*) is defined by

$$DIR = \{input, output, inout, undirected\}$$

□

Definition 4.3: **[direction of a port]**

The *direction of a port* (denoted by *dir*) is a relation on *PORT*, defined by

$$dir : PORT \rightarrow DIR$$

□

Definition 4.4: **[set of input ports]**

The *set of input ports* (denoted by *IPORT*) is defined by

$$IPORT = \{p \in PORT \mid dir(p) \in \{input, inout\}\}$$

□

Definition 4.5: **[set of output ports]**

The *set of output ports* (denoted by *OPORT*) is defined by

$$OPORT = \{p \in PORT \mid dir(p) \in \{output, inout\}\}$$

□

4.2 Instances of a set of cells

The definition of a *set of instances* is relative to a set of cells. The definition of a set of cells will be given later on, since it depends on several definitions which are yet to come.

Definition 4.6: **[set of instances]**

Let *CELL* be a set of cells. The *set of instances* (denoted by *INST*) is a relation on *CELL*. $INST_{CELL}$ is the set which contains all *instances* of CELL. An instance is a basic entity which will be used as a base for further definitions.

□

4.3 Port references of a set of cells

Definition 4.7: **[set of port references]**

Let *CELL* be a set of cells. The *set of port references* (denotes by $PORTREF_{CELL}$) is a relation on $INST_{CELL}$ defined by

$$PORTREF_{CELL} = INST_{CELL} \times PORT$$

with tuple element names (*I*, *P*).

□

Definition 4.8: **[port of a port reference]**

Let *CELL* be a set of cells. The *port of a port reference* (denoted by *port*) is a relation on $PORTREF_{CELL}$ defined by

$port : PORTREF_{CELL} \rightarrow PORT$

□

© Philips Electronics N.V.

4.4 Nets of a set of cells

Definition 4.9: **[set of nets]**

Let *CELL* be a set of cells. The *set of nets* (denoted by NET_{CELL}) is a relation on *CELL* defined by

$$NET = \mathcal{P}(PORT \times PORTREF),$$

with tuple element names (*P*, *PR*), where:

$$\forall n \in NET : |n.P| + |n.PR| \in \mathbb{N}^{+}$$

□

Definition 4.10: **[net]**

n is a ***net*** iff $n \in NET$.

□

A net n is a tuple containing a finite set of ports and a finite set of port references. At least one of these sets is non-empty.

4.5 Cells

In this section we will define the *declaration cell of an instance*, a *set of cells*, and the *descendent* relation(s) on a set of cells. Since these definitions are mutually dependent, we have to use forward references, i.e., references to objects which haven't been defined yet.

Definition 4.11: **[declaration cell of an instance]**

Let *CELL* be a set of cells. The *declaration cell of an instance* (denoted *cell*) is a relation on $INST_{CELL}$ defined by

$$cell : INST_{CELL} \rightarrow CELL,$$

with *CELL* the set of cells, which will be defined next.

Definition 4.12: **[set of cells]**

A *set of cells CELL* is defined by

$$CELL = \mathcal{P}(PORT) \times \mathcal{P}(INST_{CELL}) \times \mathcal{P}(NET_{CELL}),$$

with tuple element names *(P, I, N)*, where:

$\forall C \in CELL : (\forall p \in C.P : (\exists ^{1} n \in C.N : p \in n.P))$

All ports of a cell of *CELL* must be a member of exactly one net of that cell.

- $\forall C \in CELL : (\forall i \in C.I : (\forall p \in cell(i).P : (\exists ^{1} n \in C.N : (i, p) \in n.PR)))$

All port references of a cell of *CELL* must be a member of exactly one net of that cell.

- $\forall C \in CELL : (\forall p \in (C.N).P : p \in C.P)$
  All ports used in a net of a cell of *CELL* must be ports of that cell.
- $\forall C \in CELL : (\forall (i, p) \in (C.N).PR : p \in cell(i).P)$
  Each port reference used in a cell of *CELL* contains a port and an instance. This port must be a port of the declaration cell of this instance.
- $\forall C \in CELL : (\forall (i, p) \in (C.N).PR : i \in C.I)$
  All port references used in a net of a cell of *CELL* must refer to instances of that cell.
- $\forall C, C' \in CELL : C \neq C' \Rightarrow C.P \cap C'.P = \varnothing$
  Ports may only be part of one cell.
- $\forall C, C' \in CELL : C \neq C' \Rightarrow C.I \cap C'.I = \varnothing$
  Instances may only be part of one cell.
- $\forall C \in CELL : C \overset{+}{\not\sqsubseteq} C$
  A cell may not (indirectly) use an instance of itself. This is defined using the *descendence* relation, which will be defined in section 4.6.

□

Definition 4.13: **[cell]**

C is a *cell* iff there exists a set of cells *CELL* such that $C \in CELL$.

□

4.6 Descendence

Definition 4.14: **[k^{th} descendent, $k \geq 0$]**

Let *CELL* be a set of cells. k^{th} *descendent* (denoted by $\overset{k}{\sqsubseteq}$) is a relation on *CELL* defined by

- $\overset{k}{\sqsubseteq} : CELL \times CELL \to \mathbb{B}$,

where for $C, C'' \in CELL$:

- $C'' \overset{0}{\sqsubseteq} C \equiv C'' = C$, and
- $C'' \overset{k+1}{\sqsubseteq} C \equiv \exists C' \in CELL : (\exists i \in C'.I : C'' = cell(i)) \wedge C' \overset{k}{\sqsubseteq} C$

Let $C, C' \in CELL$. If $C' \overset{1}{\sqsubseteq} C$ holds we say an instance of C' is used in cell C. If $C' \overset{n}{\sqsubseteq} C$ holds for some $n > 1$ then there exist $n - 1$ cells

$$C_1, C_2, \ldots, C_{n-1} \in CELL$$

such that

$$C' \overset{1}{\sqsubseteq} C_1 \overset{1}{\sqsubseteq} C_2 \overset{1}{\sqsubseteq} \ldots \overset{1}{\sqsubseteq} C_{n-1} \overset{1}{\sqsubseteq} C$$

 © Philips Electronics N.V.

□

Definition 4.15: **[descendent]**

Let *CELL* be a set of cells. The *descendent* relation (denoted by $\overset{*}{\sqsubseteq}$) is a relation on *CELL* defined by

$$\overset{*}{\sqsubseteq} : CELL \times CELL \rightarrow \mathbb{B} ,$$

where for $C, C' \in CELL$:

$$C' \overset{*}{\sqsubseteq} C \quad \equiv \quad \exists k \in \mathbb{N} : C' \overset{k}{\sqsubseteq} C$$

The descendent relation is thus the reflexive and transitive closure of the kth descendent relation.

□

Definition 4.16: **[true descendent]**

Let *CELL* be a set of cells. The *true descendent* relation (denoted by $\overset{+}{\sqsubseteq}$) is a relation on *CELL* defined by

$$\overset{k}{\sqsubseteq} : CELL \times CELL \rightarrow \mathbb{B} ,$$

where for $C, C' \in CELL$:

$$C' \overset{+}{\sqsubseteq} C \quad \equiv \quad \exists k \in \mathbb{N}^{+} : C' \overset{k}{\sqsubseteq} C$$

The true descendent relation is thus the transitive closure of the kth descendent relation.

4.7 Descendent graph

Definition 4.17: **[descendent graph of a set cells]**

The *descendent graph* of a set of cells *CELL* (denoted G_{CELL}) is defined by

$$G_{CELL} = \mathcal{P}(CELL) \times \mathcal{P}(CELL \times CELL) ,$$

with tuple element names *(V, E)*, where:

$$V = CELL \wedge \forall C, C' \in CELL : \ (C, C') \in E \equiv C \overset{1}{\sqsubseteq} C$$

□

Theorem 4.1: *Let CELL be a set of cells. The descendence graph G_{CELL} is a directed acyclic graph (DAG).*

Proof: *Suppose G_{CELL} contains a cycle, i.e., there is a $n \in \mathbb{N}$, such that*

$$(C, C_1, C_2, ..., C_n) \in G_{CELL}.V,$$

and

$$\{ (C, C_1), (C_1, C_2), ..., (C_{n-1}, C_n), (C_n, C) \} \subseteq G_{CELL}.E$$

Hence

$$C \overset{+}{\sqsubseteq} C$$

But $C \overset{+}{\not\sqsubseteq} C$ *for each* $C \in CELL$, *so we can conclude that* G_{CELL} *is acyclic.*

□

Definition 4.18: **[port set of a cell]**

Let *CELL* be a set of cells. The *port set of a cell* (denoted by *P*) is a relation on *CELL* defined by

$$P : CELL \rightarrow \mathcal{P}(PORT),$$

where:

$$\forall C \in CELL : P(C) = C.P$$

□

This relation can be used to retrieve the set of ports of a cell.

Definition 4.19: **[input port set of a cell]**

Let *CELL* be a set of cells. The *input port set of a cell* (denoted by *IP*) is a relation on *CELL* defined by

$$IP : CELL \rightarrow \mathcal{P}(PORT),$$

where:

$$\forall C \in CELL : IP(C) = P(C) \cap IPORT$$

□

This relation can be used to retrieve the set of input ports of a cell.

Definition 4.20: **[output port set of a cell]**

Let *CELL* be a set of cells. The *output port set of a cell* (denoted by *OP*) is a relation on *CELL* defined by

$$OP : CELL \rightarrow \mathcal{P}(PORT),$$

where:

© Philips Electronics N.V.

$$\forall C \in CELL : OP(C) = P(C) \cap OPORT$$

□

This relation can be used to retrieve the set of output ports of a cell.

Definition 4.21: **[port reference set of a cell]**

Let *CELL* be a set of cells. The *port reference set of a cell* (denoted by *PR*) is a relation on *CELL* defined by

$$PR : CELL \rightarrow \mathcal{P}(PORTREF),$$

where:

$$\forall C \in CELL : PR(C) = \{ (i, p) \in PORTREF | i \in C.I \wedge p \in P(cell(i)) \}$$

□

This relation can be used to retrieve the set of port references of a cell.

Definition 4.22: **[input port reference set of a cell]**

Let *CELL* be a set of cells. The *input port reference set of a cell* (denoted by *IPR*) is a relation on *CELL* defined by

$$IPR : CELL \rightarrow \mathcal{P}(PORTREF),$$

where:

$$\forall C \in CELL : IPR(C) = PR(C) \cap \{ (i, p) \in PORTREF | p \in IPORT \}$$

□

This relation can be used to retrieve the set of input port references of a cell.

Definition 4.23: **[output port reference set of a cell]**

Let *CELL* be a set of cells. The *output port reference set of a cell* (denoted by *OPR*) is a relation on *CELL* defined by

$$OPR : CELL \rightarrow \mathcal{P}(PORTREF),$$

where:

$$\forall C \in CELL : OPR(C) = PR(C) \cap \{ (i, p) \in PORTREF | p \in OPORT \}$$

□

This relation can be used to retrieve the set of output port references of a cell.

Definition 4.24:[**leaf cell**]

Let *CELL* be a set of cells. *Leaf cell* (denoted by *leaf*) is a relation on *CELL* defined by

$$leaf: CELL \to \mathbb{B},$$

where:

$$\forall C \in CELL : leaf(C) \equiv C.I = \emptyset$$

A cell for which this relation holds, is called a *leaf cell*. If this relation doesn't hold for a cell its called a *non-leaf cell*.

□

Definition 4.25:[**connection**]

Let *CELL* be a set of cells, and $C \in CELL$. *Connection* (denoted by γ_C) is a relation on *P(C)* defined by

$$\gamma_C : P(C) \times P(C) \to \mathbb{B},$$

where for $p, p' \in P(C)$

$$\gamma_C(p, p') \equiv \exists n \in C.N : p \in n \wedge p' \in n$$

□

Definition 4.26:[**net of a port**]

Let CELL be a set of cells, and $C \in CELL$. *Net of a port* (denoted by net_C) is a relation on *P(C)* defined by

$$net_C : P(C) \to NET,$$

where for $p \in P(C)$:

$$net_C(p) = \{p' \in P(C) \mid \gamma_C(p, p')\}$$

□

4.8 Functionality of cells

We will associate *values* with ports, nets, and port references:

Definition 4.27:[**set of values**]

The *set of values* (denotes by *VAL*) is defined by

$$VAL = \{ 0, 1 \}$$

□

© Philips Electronics N.V.

Definition 4.28: **[value]**

v is a *value* iff $v \in VAL$.

□

We only use the values '0' and '1'. This prohibits the modelling of buses, where ports must be able to write the value 'Z' (high-impedant), in order to avoid interference with other ports connected to the same bus. Also, wired-or and wired-and constructions cannot be described. Since we do not intend to describe buses or wired-or/wired-and constructions, we don't need the value 'Z'. Also the values 'X' (don't care) and 'U' (unknown) are not used, since we do not need them for our purpose.

Definition 4.29: **[value on a port]**

Let $p \in PORT$. The value that port p carries at some point in time t (denoted by $\nu(p, t)$) is defined by

$$\nu : PORT \times \mathbb{N} \rightarrow VAL$$

□

Definition 4.30: **[value on a port reference]**

Let $CELL$ be a set of cells, and $p \in PORTREF_{CELL}$. The value that port reference p carries at some point in time t (denoted by $\nu(p, t)$) is defined by

$$\nu : PORTREF_{CELL} \times \mathbb{N} \rightarrow VAL$$

□

Definition 4.31: **[value on a net]**

Let $CELL$ be a set of cells, and $n \in NET_{CELL}$. The value that net n carries at some point in time t (denoted by $\nu(n, t)$) is defined by

$$\nu : NET_{CELL} \times \mathbb{N} \rightarrow VAL$$

□

Time is modelled by an integer, i.e., time is divided into *time slots*. This suffices for synchronous logic circuits. When describing combinatorial logic, an output port value depends on the input port values, so we can omit time t, and denote the value on a port by $\nu(op)$.

Since there will be a finite set of memory elements, a cell C will have a finite set of states in which it can be. We can therefore associate each state with a unique natural number.

Definition 4.32: **[(internal) state of a cell]**

Let $CELL$ be a set of cells, and $C \in CELL$. The *(internal) state of* C (denoted *state*) is a relation on C and the time t defined by

$$state : CELL \times \mathbb{N} \rightarrow \mathbb{N}$$

□

Let $CELL$ be a set of cells, $C \in CELL$, and denote

$$IP(C) = \{ip_1, ip_2, \ldots, ip_{|IP(C)|}\}$$

then:

$$state(C, t+1) = f_C(state(C,t), \nu(ip_1, t), \nu(ip_2, t), \ldots, \nu(ip_{|IP(C)|}, t))$$

In general, the value of an output port in time slot $t+1$ is a function of the input port values in time slot t, and the internal state in time slot t. Let $CELL$ be a set of cells, $C \in CELL$, and again denote

$$IP(C) = \{ip_1, ip_2, \ldots, ip_{|IP(C)|}\}$$

then:

$$\nu(op, t+1) = f_{op}(state(C,t), \nu(ip_1, t), \nu(ip_2, t), \ldots, \nu(ip_{|IP(C)|}, t))$$

Definition 4.33: **[combinatorial output port]**

Let $CELL$ be a set of cells, and $C \in CELL$. *Combinatorial output port* (denoted by *combop*) is a relation on $OP(C)$, defined by

$$combop : OP(C) \rightarrow \mathbb{B},$$

where for $op \in OP(C)$:

$$combop(op) \equiv \forall st, st' \in STATE(C) : f_{op}(st, \ldots) = f_{op}(st', \ldots)$$

□

This means that the output port value of a combinatorial output does *not* depend on the internal state of its cell, only on the input port values of its cell. This implies that the response to input value changes is *not* time-dependent.

Definition 4.34: **[combinatorial cell]**

Let $CELL$ be a set of cells, and $C \in CELL$. *Combinatorial cell* (denoted by *comb*) is a relation on C, defined by

$$comb : C \rightarrow \mathbb{B}$$

where:

$$\forall op \in OP(C) : comb(op) = true$$

□

© Philips Electronics N.V.

Definition 4.35:[sequential cell]

Let *CELL* be a set of cells, and $C \in CELL$. *sequential cell* (denoted by *seq*) is a relation on *C*, defined by

$$seq : C \rightarrow \mathbb{B}$$

where :

$$seq(C) \equiv \neg comb(C)$$

□

4.9 Relation between the model and reality

4.9.1 Leaf cells

A leaf cell is a cell which is not described in terms of lower-level cells, but contains functionality "by definition". This functionality cannot be described using the model, because it was only intended to describe the hierarchical relation between two or more cells of a design.

4.9.2 Non-leaf cells

A non-leaf cell is a cell that contains a number of instances, which are also called *children*, with possible interconnections between them. The non-leaf cell is also called the *parent* of its children. The functionality of the parent can be determined from the functionality and the interconnection scheme of its children. The concept of parent and children introduces a hierarchy in the design structure.

4.9.3 Instances

A (leaf or non-leaf) cell (also called declaration cell) defines the contents of an entity, and hence is a declaration. Instances (also called instantiation cells) are used to provide multiple instantiations of such a declaration cell. There can be only one declaration of a certain cell, while there can be as many instances of that cell as one may need.
An instance is merely a reference to the declaration cell it belongs to. It has no contents of its own, besides its ports, which we will call portrefs, to distinguish between ports of cells and ports of instances. Again, a portref is only a reference to the corresponding port on the declaration cell. The function of an instance is determined by the function of the corresponding declaration cell.

4.9.4 Ports

Ports of a (leaf or non-leaf) cell represent the interface to and from the external world. They are electrical connectors which can be connected to other ports, or to some external world source/destination. Using the direction of information flow, we distinguish four types of ports:

input ports: Information flow is always directed from a higher-level cell to lower level cell(s).

output ports: Information flow is always directed from a lower-level cell to higher level cell(s).

inout ports: Information flow can be directed from a higher-level cell to lower level cell(s), or vice versa.

undirected ports: The information flow direction is not specified, and must be derived by examination of the structural specification.

4.9.5 Nets

Nets are used to model interconnections between ports. All ports and port references of such a net are electrically connected, i.e., they form a "galvanic unity". At an arbitrary point in time, these ports and port references carry the same value.

4.9.6 Naming conventions

As said before, the naming conventions between EDIF, NDL and NDS/LDS are not the same. This is particularly confusing since EDIF, NDL and NDS/LDS sometimes use the same terms with a different meaning. Therefore, below a table is given which clarifies the relationship between the different terms used by the formats mentioned before. It can be kept in mind that the explanation of the EDIF terms is given above.

EDIF	NDL	NDS	LDS
port	implicit	pin	port
net	N/A	net	(net)
cell	macro	declaration block	cell
instance	implicit	instantiation block	(instantiation block)
library	(the file)	design	library

Table 4.1: Naming conventions

© Philips Electronics N.V.

5 InScan

This report deals with the specification and implementation of InScan. InScan is a program that adds scan paths to a design. This design, that we then call 'scannable', i.e. prepared for scan test (full scan), can be processed by a test pattern generator that recognises the scan path functionality and creates appropriate test patterns for it. InScan must be set up in such a way that it can be extended with partial scan algorithms. Partial scan is a topic that will be addressed in chapter 9. It is concerned with reducing the overhead of full scan by not putting all memory elements in the scan paths. For an elaborate discussion the reader is referred to chapter 9.

5.1 PrepScan and ScanIt

Basically, InScan performs two actions:

1. Replacement of all memory elements (flip-flops) by 'scannable' variants.
2. Creation of the scan paths.

These steps need to be refined somewhat, to allow partial scan algorithms to be 'plugged in' easily. Therefore, a more refined schematic is given below:

1. Propose a default way of creating the scan paths.
2. Allow (partial scan) algorithms to 'optimise' this proposal, i.e., alter it so that it conforms to a certain (partial scan) 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 sort of steps can be distinguished. Steps 1-3 only perform analysis which results in certain proposals. Steps 4-5 only execute these propositions. The proposals that are created in steps 1-3 determine the operation of the entire program. It would be desirable that the user could interact with the program in order to check these propositions and possibly alter them. This would greatly improve the flexibility of the program. Therefore, it has been decided that InScan should be constructed of two separate programs: PrepScan, which performs the steps 1-3, and ScanIt, which performs steps 4-5.

5.2 The interface between PrepScan and ScanIt

The propositions that are created by PrepScan must be stored in some format, so that ScanIt can read them in and execute them. Because we would also like to give the user the possibility of editing these propositions, they should be in some format that is easy to read, to understand, and to modify.

5.2.1 The routing plan language

In the previous section, the steps 1 and 2 create a scan path proposal. This proposal defines how memory elements are assembled to constitute scan paths. To describe this proposal, the *routing plan language* has been designed. We will call a proposal in terms of this language a *routing plan*. The next chapter will give an elaborate discussion of the routing plan format. In appendix B the syntax of the routing plan language is given.

5.2.2 The match rule format

Step 3 in the previous section is concerned with proposing replacements for the memory elements that are to be entered in a scan path. To describe this proposal, the *match rule format* is used. This format will be discussed in chapter 7. We will call a file that contains a proposition in terms of this format a *match rule file*.

5.2.3 Schematic overview

In figure 5.1 a schematic overview is given of InScan. This figure shows how the design is processed, and how the user can interact with the program. Note that the information flow in the 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.

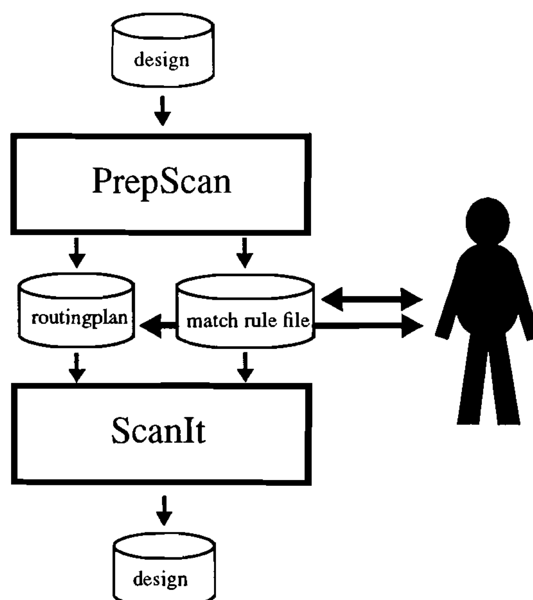


Figure 5.1: Schematic overview of InScan

© Philips Electronics N.V.

6 The routing plan language

The routing plan language has been designed specifically to describe the scan paths in a design in a readable way. This makes it easy for the user to read a routing plan and make changes to it. Below, the routing plan language and some of its concepts will be explained. Furthermore, a formal model will be given with which scan paths can be described. In appendix B the routing plan syntax is described, together with a number of semantic rules to which a routing plan should adhere. This syntax will be explained thoroughly throughout this chapter.
The routing plan language is useful for applications that have to deal with scan paths. PrepScan is an example of an application that describes scan paths that are yet to be created. Other applications may however just as well use this language for describing already existing scan paths.
The routing plan language has been implemented as a toolkit, which defined the data structure on which the routing plan is based, and the operations that can be performed on it. This toolkit is called SDS (Scan Data Structure). More information about it can be found in the SDS user manual [SDS].

6.1 Hierarchy: The macro

The designs that are considered are hierarchical of nature. They can be either in the EDIF 2 0 0 netlist format [EIA 87] or the Philips proprietary NDL (Netlist Description Language) format. In EDIF a design is composed of (declaration) *cells*, the basic entities. Each cell can contain other (instantiation) cells, which results in a hierarchy of cells. Unfortunately, naming conventions aren't very well standardised, since what EDIF calls cells, NDL calls *macros*. Furthermore NDS (Network Data Structure) [NDS], the toolkit that is used to implement InScan, uses the term *block* for this. To avoid confusion, we will use the EDIF terms. In table 4.1 these terms are mentioned, together with the names that NDL and NDS/LDS use.

Since netlists are hierarchical, the scan paths will traverse through a number of levels of hierarchy. The routing plan must reflect this hierarchy in order to be able to describe the scan paths. Therefore the routing plan will be constructed of entities that correspond with the declaration cells in the design. These entities will be called *macros*, not to be confused with the macros that are used in NDL. The syntax of a routing plan and macro is as follows (refer to appendix B for remarks about notational matters):

routing_plan	::=	**[+ macro +]**
macro	::=	**'MACRO' name**
		(['DERIVEDFROM' name] | ['REPLACEDFOR' name])
		[+ clock_domain +]
		('ENDMACRO' | 'END') [';']

As can be seen, macros exist of a number of clock domains. These entities will be discussed in the next section. A macro is used to describe how one or more scan paths traverse the declaration cell that corresponds with the macro. There are basically two types of macros, namely the 'ReplacedFor' and the 'DerivedFrom' macro.

The 'ReplacedFor' macro corresponds with the scannable variant of a leaf declaration cell that is a memory element. For example:

```
Macro dnn10tad_sff ReplacedFor dnn10tad
...
EndMacro
```

denotes that an instance of cell dnn10tad (which is a memory element) can be replaced by an instance of dnn10tad_sff, which should be a scannable variant of dnn10tad. Thus, this macro describes how the scan path(s) traverse(s) the scannable variant of cell dnn10tad. Since the macro corresponds with a variant of a cell, the macro name and the 'ReplacedFor' macro name must be different.
Note that, when creating the scannable variants, PrepScan conforms to the naming convention for these elements, which means that the names of scannable variants are composed of the name of the original elements, followed by the postfix "_sff".

The 'DerivedFrom' macro corresponds with any other declaration cell (leaf or non-leaf) and describes how the scan path(s) traverse(s) this cell. Normally, it is sufficient to give one such description for a certain cell. However, it might sometimes be convenient to be able to give several descriptions of how scan paths can traverse a cell. This can be done with the 'DerivedFrom' construct. For example:

```
Macro A1 DerivedFrom A
...
EndMacro

Macro A2 DerivedFrom A
...
EndMacro
```

denotes that for cell A two scan path descriptions are given. Now certain instances of declaration cell A can be described with macro A1 and others can be described with macro A2. If there is only one description of scan paths through a cell, the 'DerivedFrom' keyword can be omitted from the macro. So Macro A DerivedFrom A is equal to simply Macro A.

There is no need to provide a macro for each declaration cell a design is composed of. Only cells that contain a part of a scan path need to be described with a macro. In general, combinatorial cells will not require a corresponding macro, since the combinatorial logic will generally not be part of any scan path. This is however not always true for all combinatorial logic, since it is possible to use simple elements like a buffer, an inverter, or a multiplexer in a scan path.

 © Philips Electronics N.V.

6.2 Clock domains

When constructing scan paths, it has to be made sure that all memory elements a scan path is composed of are clocked on the same edge(s) of the same clock port(s) (i.e. simultaneously), in order to be able to shift a value through the shift register from the input of the scan path to the output in test mode. To distinguish between elements that are not clocked simultaneously, we introduce the concept of *clock domain.*

A *clock domain* is a part of the design that is clocked simultaneously. Since our designs are divided into cells, we define the clock domain of a cell as being that part of a cell that is clocked simultaneously. A clock domain can now be seen as the union of the cell clock domains that are clocked simultaneously.

Each macro will be constructed with at least one clock domain, the syntax of which is:

clock_domain::='CLOCKDOMAIN' [name]
(['DRIVENBY' clock_list] | ['NOTDRIVEN'])
[+ chain +]
('ENDCLOCKDOMAIN' | 'END') [';']

As can be seen, a clock domain is further subdivided in a number of chains. These entities will be discussed in the next section. Each clock domain of a macro has a name, which must be unique within the macro. Furthermore, the clock domain can be clocked by a number of clock ports. If there are no clock ports (e.g., in the case of a macro that corresponds with a combinatorial cell) this can be indicated with the 'NOTDRIVEN' keyword. For example:

```
Macro inv
        ClockDomain none NotDriven
        ...
        EndClockDomain
EndMacro
```

could be the macro of an inverter which of course is not driven by any clock pin.

If there is at least one clock port, this can be indicated with the 'DRIVENBY' keyword. Each clock port consists of a name and possibly a '-' symbol. This symbol indicates that the memory elements of which the clock domain consists, are clocked on the falling edge of the clock port. Absence of such a symbol indicates that they are clocked on the rising edge of the clock port. For example:

```
Macro X
        ClockDomain cd DrivenBy ( -clk )
        ...
        EndClockDomain
EndMacro
```

indicates that all memory elements of clock domain `cd` are clocked on the falling edge of clock port `clk`. It is possible to enter more than one clock port in the 'DRIVENBY' list. When this is done, it is assumed that all these ports are connected somewhere higher in the

hierarchy of the design, possibly even outside the chip, on the PCB. This is comparable with the "MustJoin" construct in EDIF. Multi-phase clocking is not yet supported by the routing plan language.

6.3 Chains

A chain, or scan chain, is an important entity in the routing plan language. It models a (part of a) scan path. A chain consists of two parts, namely a 'chain port' definition and an enumeration of the elements it consists of. Below, the syntax is given.

```
chain          ::=  'CHAIN' [ {chain}name ]
                    [ length ]
                    [ 'SCANIN' name ]
                    [ 'SCANOUT' port_list ]
                    [ 'SCANENABLE' port_list ]
                    [ 'NORMENABLE' port_list ]
                    [ 'HOLDENABLE' port_list ]
                    [ 'ORDER'
                    order_list
                    ( 'ENDORDER' | 'END' ) [ ';' ] ]
                    ( 'ENDCHAIN' | 'END' ) [ ';' ]
```

The chain name must be unique within the clock domain. Each chain has a length. This length is the sum of lengths of the elements the chain is composed of. The length of an element is the number of clock cycles that are needed to shift a value from the input of the chain to the output. Hence, the length of a memory element is considered to be 1 (one), and the length of a combinatorial element is considered to be 0 (zero).

Each chain has a number of special ports associated with it, which we will call 'chain ports'. These ports are the extra ones that are needed to implement the scan functionality. As can be seen from the syntax, there are five possible ports, of which the last two ('NORMENABLE' and 'HOLDENABLE') are only implemented for future extensions and will not be used in the current version of InScan, as described by this report.
The first three ports indicate how the chain operates. The 'SCANIN' port is the port at which a value can be shifted into the chain, The 'SCANOUT' port is the port at which a value will be shifted out of the chain, and the 'SCANENABLE' port is the port that puts the chain into test mode. In figure 2.2 a chain is depicted with these three ports. A chain can only have one 'SCANIN' port, but more than one 'SCANOUT' and 'SCANENABLE' ports in the routing plan language. However, InScan will only allow one 'SCANOUT' port and one 'SCANENABLE' port per chain.
The 'SCANOUT' and 'SCANENABLE' ports can appear with a '-' symbol. A '-' symbol before the 'SCANOUT' port means that the chain has an inversion incorporated in it. Shifting a value through the entire chain will result in the inverted value at the 'SCANOUT' port. A '-' symbol before the 'SCANENABLE' port means that the chain will enter test mode when a

© Philips Electronics N.V.

logical zero is presented to this 'SCANENABLE' port. If no '-' symbol appears before the 'SCANENABLE' port, a logical one will put the chain in test mode. Below, an example is given of a macro with one chain:

```
Macro X
        ClockDomain cd DrivenBy ( clk )
                Chain chn
                        Lenght          5
                        ScanIn          si
                        Scanout         so
                        ScanEnable      -se
                        ...
                EndChain
        EndClockDomain
EndMacro
```

As can be seen, the chain starts at port si, and ends at port so. It will be put in test mode by applying a logical zero to port se.

The definition of the chain ports can be followed by a list of 'chain elements' of which the chain is constructed. This is explained in the next section.

6.4 Chain elements

Since the designs that we consider are hierarchical of nature, this hierarchy is also reflected in the definition of chains. A chain for a cell will be defined in terms of the chains of the instances that cell consists of. We will explain this with an example. In figure 6.1 a design is shown that consists of a cell named X. X contains two instances of cell Y. We have left out the "normal" connections between the cells for clarity.

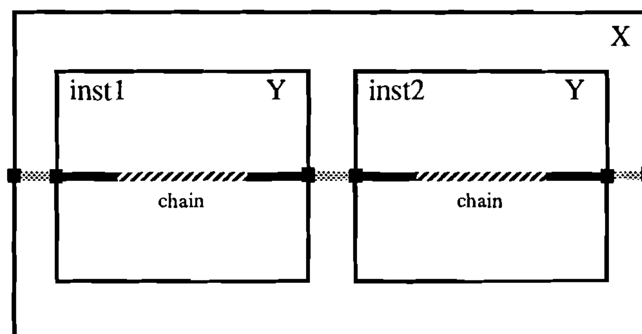


Figure 6.1: Example design with chain

Below the macro for cell Y is given. As can be seen, one chain has been defined for cell Y. This chain is also depicted in the instances of cell Y in figure 6.1.

```
Macro Y
        ClockDomain yclk DrivenBy ( clk )
                Chain ychn
                        Length          8
                        ScanIn          ysi
```

```
                    ScanOut         yso
                    ScanEnable      yse
                    Order
                    ...
                    EndOrder
            EndChain
        EndClockDomain
EndMacro
```

For now, we do not specify the elements of which the chain is composed of. Suppose that both instances of Y are clocked simultaneously. We want to construct a chain for cell X. This chain will be constructed of the chains that are present with the instances used in X. In figure 6.1 the chain is depicted. The following macro can be created:

```
Macro X
        ClockDomain xcl DrivenBy ( cl )
                Chain xchn
                        Length          16
                        ScanIn          xsi
                        ScanOut         xso
                        ScanEnable      xse
                        Order
                                inst1 OfType Y Length 8
                                inst2 OfType Y Length 8
                        EndOrder
                EndChain
        EndClockDomain
EndMacro
```

Chain xchn consists of two 'chain elements', which are mentioned in the order list. Note that these chain elements can appear in the same chain because they are clocked simultaneously. If they were not, macro X would have got two clock domains, both with one chain that consisted of one of the chain elements. The syntax of the order list is as follows:

order_list	**::=**	**element [* element *]**
element	**::=**	**name 'OFTYPE' name**
		[['CLOCKDOMAIN' name]
		'CHAIN' name] [length]

An order list consists of a number of chain elements. The first item a chain element is composed of is its name. This name is supposed to be the name of an instance that is used in the cell that corresponds with the macro. So in the example above, `inst1` and `inst2` are instances that are used in cell X. The second item is the declaration cell type of this instance. In the previous example, the type of `inst1` and `inst2` is Y. The next item is a clock domain name. This item is only necessary if the chain name is not unique within the macro (remember that a chain name had only to be unique within a clock domain). Since macro Y only has one chain, the clock domain is not mandatory in the previous example. The next item is the chain name. This name is only necessary if the referred macro contains more than one chain. That is why it is not mandatory in the previous example. The order list in the previous example could for example also be written in the following way:

```
Order
```


© Philips Electronics N.V.

```
        inst1 OfType Y Chain ychn Length 8
        inst2 OfType Y ClockDomain yclk Chain ychn Length 8
EndOrder
```

The last item is the length field, which indicates the length of the chain element.

The chain elements will be routed in the order as they appear in the order list. So `inst1` will become the first part of chain xchn in the previous example. The user can easily change the order in which the chain elements are routed, just by modifying the order of the chain elements in the order list. We will see that this is also done by specific algorithms that optimise the routing plan (see chapter 9).

Leaf cells do not contain any instances by definition. Therefore, the order list of a macro corresponding with a leaf cell (a leaf macro) is non-existing. The entire order list will be left out in this case. Below, an example is given for the macro of a memory element:

```
Macro dnn10tad_sff ReplacedFor dnn10tad
        ClockDomain clk DrivenBy ( clk )
                Chain chn
                        Length          1
                        ScanIn          dt
                        ScanOut         q
                        ScanEnable      te
                EndChain
        EndClockDomain
EndMacro
```

Because cells that correspond with leaf macros are supposed to have the scan functionality incorporated in them, the chain ports of chains of these macros should be existing ports on the cells. So, in the example above, ports `dt`, `q`, and `te` are supposed to be existing ports on cell `dnn10tad_sff`. The chain ports of non-leaf cells do not have to exist. They will be created by InScan when necessary.

6.5 Formal model of scan chains

In this section we will give a formal definition of scan chains. This definition will closely resemble the routing plan, so that the reader will not find it hard to understand. All definitions are related to a set of cells. To prevent the use of several "Let *CELL* be a set of cells", we will assume from now on that a set of cells is chosen, and we will denote this set of cells by *CELL*.

definition 6.1: **[set of enable ports]**

The *set of enable ports* (denoted by *ENABLE*) is defined by

$$ENABLE = IPORT \times \mathbb{B} ,$$

with tuple element names (*P*, *VAL*).

□

definition 6.2: **[enable port]**

e is an enable port iff $e \in ENABLE$

□

definition 6.3: **[set of clock ports]**

The *set of clock ports* (denoted by *CLOCK*) is defined by

$$CLOCK = IPORT \times \mathbb{B}$$

with tuple element names (*P*, *POL*).

□

definition 6.4: **[clock port]**

c is a clock port iff $c \in CLOCK$

□

definition 6.5: **[set of chain elements]**

The *set of chain elements* (denoted by *CHAINEL*) is defined as the set which contains all *chain elements*. A chain element is a basic entity which will be used as a base for further definitions.

□

definition 6.6: **[position of a chain element]**

The *position of a chain element* (denoted by *pos*) is a relation on *CHAINEL* defined by

$$pos : CHAINEL \rightarrow \mathbb{N} ,$$

where *pos* denotes the position of the chain element in the chain

□

definition 6.7: **[instance of a chain element]**

The *instance of a chain element* (denoted by *inst*) is a relation on *CHAINEL* defined by

$$inst : CHAINEL \rightarrow INST_{CELL},$$

where:

$$\forall el \in CHAINEL : inst(el) \in CELL.I$$

□

 © Philips Electronics N.V.

definition 6.8: **[chain of a chain element]**

The *chain of a chain element* (denoted by *chain*) is a relation on *CHAINEL* defined by

$$chain : CHAINEL \rightarrow CHAIN$$

where:

$$\forall el \in CHAINEL : chain(el) \in CHAIN_{cell(inst(el))}$$

in this definition we have used $CHAIN_C$ (with $C \in CELL$). $CHAIN_C$ will be defined later on. The restriction states that a chain element must refer to a chain, which is part of the cell to which the instance of that chain element refers.

□

definition 6.9: **[set of scan chains]**

Let $C \in CELL$. A *set of scan chains* of *C* (denoted by $CHAIN_C$) is defined by

$$CHAIN_C = IP(C) \times OP(C) \times \mathbb{B} \times \mathcal{P}(ENABLE) \times \mathcal{P}(CLOCK) \times \mathcal{P}(CHAINEL) \times \mathbb{N}$$

with tuple element names *(I, O, INV, E, C, EL, n)*.

A scan chain *ch* of cell *C* hence consists of the following parts:

- *ch.I*: the scan input port
- *ch.O*: the scan output port
- *ch.INV*: a boolean, *false* indicates a non-inverting chain, *true* indicates an inverting chain
- *ch.E*: the set of enable ports, with their enable value
- *ch.C*: the set of clock ports, with their relative polarity
- *ch.EL*: the set of chain elements, which concatenated form the chain ch.
- *ch.n*: the length of this chain, specifying the number of clock cycles needed to shift data from the scan input to the scan output.

For all $CH \in CHAIN_C$ the following must hold:

- $(CH.E).P \subseteq IP(C) \setminus \{CH.I\}$
  The enable ports of *CH* are supposed to be input ports of *C*, but may not be the scan-in port of *CH*.
- $(CH.C).P \subseteq IP(C) \setminus \{CH.I\}$
  The clock ports of *CH* are supposed to be input ports of *C*, but may not be the scan-in port of *CH*.
- $(CH.E).P \cap (CH.C).P = \emptyset$
  An enable port of *CH* cannot also be a clock port of *CH*, and vice versa.
- $\forall e, e' \in CH.E : e.P = e'.P \Rightarrow e.VAL = e'.VAL$
  It is forbidden to define a port twice, but with different values, in the scan-enable set of *CH*.

- $\forall c, c' \in CH.C : c.P = c'.P \Rightarrow c.POL = c'.POL$
  It is forbidden to define a port twice, but with different polarities, in the clock port set of *CH*.
- $\forall 0 \le i < |CH.EL| : (\exists^1 el \in CH.EL : pos(el) = i)$
  Each chain element of *CH* has a unique position in the chain.
- Let $el \in CH.EL$. We define:

$$SI(el) = (inst(el), chain(el).I)$$
$$SO(el) = (inst(el), chain(el).O)$$
$$el_i = \text{the element } el' \in CH.EL, \text{for which } pos(el') = i$$

In case $CH.EL \neq \varnothing$, the following must hold:

- $\gamma_C(CH.I, SI(el_0))$
  The scan-in port of *CH* must be connected to the scan-in port of its first chain element.
- $\gamma_C(CH.O, SO(el_{|CH.EL|-1}))$
  The scan-out port of *CH* must be connected to the scan-out port of its last chain element.
- $\forall (0 < i < |CH.EL| - 1 : \gamma_C(SO(el_{i-1}), SI(el_i)))$
  The scan-out port of a chain element must be connected to the scan-in port of the next chain element (except for the last chain element).
- $CH.INV = \oplus 0 \le i < |CH.EL| : chain(el_i).INV$
  Chain *CH* inverts if the number of inverting chain elements of *CH* is odd .
- $CH.n = \sum 0 \le i < |CH.EL| : chain(el_i).n$
  The length of *CH* is equal to the sum of lengths of its chain elements.
- $(\forall cl \in CH.C : v(net_C(cl.P)) = val(cl.POL)) \Rightarrow$

  $(\forall cn \in CLOCKNET_{CH} : v(cn.N) = val(cn.POL))$

- $(\forall cl \in CH.C : v(net_C(cl.P)) = \overline{val(cl.POL)}) \Rightarrow$

  $(\forall cn \in CLOCKNET_{CH} : v(cn.N) = \overline{val(cn.POL)})$

  If a clock port of a cell carries its specified polarity, then all clock ports of the children, that are driven by that port, must also carry their specified polarity, and vice versa.

- $(\forall en \in CH.E : v(net_C(en.P)) = en.VAL) \Rightarrow$

  $(\forall en \in ENABLENET_{CH} : v(en.N) = en.VAL)$

  If an enable port of a cell carries its specified enable value, then all enable ports of the children, that are driven by that port, must also carry their specified enable values.

with

$$CLOCKNET_{CH} = \{ (n, p) \in C.N \times \mathbb{B} : \exists el \in CH.EL : \exists cl \in chain(el).C : (inst(el), CH.C) \in n.PR \wedge cl.POL = p \}$$

© Philips Electronics N.V.

$$ENABLENET_{CH} = \{ (n, v) \in E.N \times \mathbb{B} : \exists el \in CH.EL : \exists en \in chain(el).E : (inst(el), CH.C) \in n.PR \wedge en.VAL = v \}$$

□

definition 6.10: **[scan chain]**

ch is a *scan chain* iff there is a $C \in CELL$ for which $ch \in CHAIN_C$

□

definition 6.11: **[leaf scan chain]**

Let $C \in CELL$. *Leaf scan chain* (denoted by $leafchain_C$) is a relation on $CHAIN(C)$ defined by

$$leafchain_C : CHAIN_C \rightarrow \mathbb{B}$$

where:

$$\forall ch \in CHAIN_C : leafchain_C(ch) \equiv ch.EL = \emptyset$$

□

definition 6.12: **[set of clock domains]**

Let $C \in CELL$. A *set of clock domains* of C (denoted by $CLOCKDOM_C$) is defined by

$$CLOCKDOM_C = \mathcal{P}(CHAIN_C)$$

with tuple element name (CH).

For all $CD \in CLOCKDOM_C$ the following must hold:

- $\forall CH1, CH2 \in CD : CH1.C = CH2.C$

□

definition 6.13: **[clock domain]**

cd is a *clock domain* iff there is a $C \in CELL$ for which $cd \in CLOCKDOM_C$

□

definition 6.14: **[set of macros]**

Let $C \in CELL$. A *set of macros* (denoted by $MACRO_C$) is defined by

$$MACRO_C = \mathcal{P}(CLOCKDOM_C)$$

with tuple element name (CD)

□

definition 6.15: **[macro]**

mc is a *macro* iff there is a $C \in CELL$ for which $mc \in MACRO_C$

□

definition 6.16: **[leaf macro]**

Let $C \in CELL$. *Leaf macro* (denoted by $leafmacro_C$) is a relation on $MACRO_C$ defined by

$$leafmacro_C : MACRO_C \rightarrow \mathbb{B}$$

where:

$$\forall macro \in MACRO_C \ \forall cd \in macro \ \forall ch \in cd : leafchain(ch) = true$$

□

© Philips Electronics N.V.

7 PrepScan

As was stated in chapter 5, InScan is divided into two parts, to improve the flexibility of the program. The first part, which is called PrepScan, is only concerned with the creation of the plans that describe how the scan chains should be created for a certain design. This chapter will describe PrepScan in greater detail, discussing how the routing plan and the match rule file will be generated. It will describe which actions PrepScan performs, why it does so, and how.

7.1 Introduction

In chapter 5, operation of InScan was divided into five actions, of which the first three should be executed by PrepScan. They are repeated here:

1. Propose a default way of creating the scan chains. This step produces a routing plan.
2. Allow (partial scan) algorithms to 'optimise' this routing plan, i.e., alter it so that it conforms to a certain (partial scan) methodology. This step is optional.
3. Propose a 'scannable' variant for each type of memory element (flip-flop) that is to be entered in a scan path. This step produces a match rule file.

These three steps are the major steps PrepScan will perform. In figure 7.1 a modular description of PrepScan is given, together with the inputs and outputs of the program. This figure describes the most important actions that PrepScan undertakes in order to perform the abovementioned steps. The modules will be discussed in the following sections.

7.1.1 Implementation matters

As was said earlier, InScan is implemented using the NDS/LDS toolkit [NDS]. This toolkit was specially designed to provide a program developer with a general means for the storage of network data and a set of functions that operate on that data. In table 4.1 a summary was given of several terms that are used by NDS/LDS, and their corresponding names in EDIF and NDL. One concept is however not yet thoroughly explained, namely that of 'design' versus 'library'.

A design can be seen as a collection of cells, or, stated in terms of our formal model:

Definition 7.1: **[design]**

A design is a set of cells *CELL*.

□

This description is however not sufficient to model a real design, because we haven't specified what leaf cells are. Every cell in a design is built up of other cells, right down to the level of leaf cells, which do not contain other cells anymore. The properties of a cell are construct-

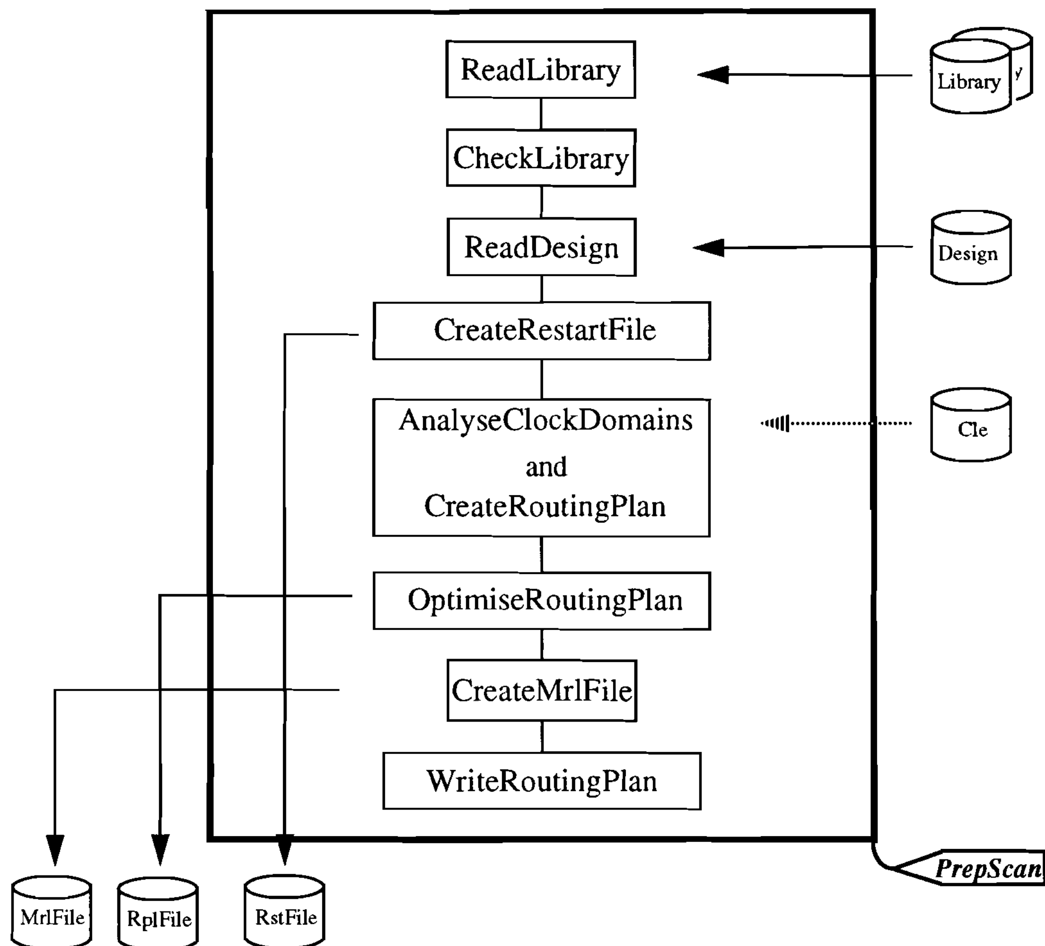


Figure 7.1: Modular architecture of PrepScan

ed from the properties of all the cells it consists of. Leaf cells do not contain other cells anymore, and should thus have a property description contained in them. This is where the library comes in. A library is a collection of leaf cells, with every leaf cell and its ports containing a property description.

Definition 7.2: **[library]**

A library is a set of cells *CELL*, with

$$\{c \in CELL | \ (leaf(c) = true) \wedge c \text{ and all its ports contain property descriptions}\}$$

□

The contents of the property descriptions is left unspecified. Whenever we require some information being specified in this description, we will clearly state so. The libraries we will work with are supposed to be EDT libraries (libraries in the EDT library format [EDT Library format]) that adhere to the ED&T library norm [ED&T Library norm]. The designs we will work with can be either in the EDIF 2 0 0 or NDL format.

 © Philips Electronics N.V.

7.2 The first four modules

7.2.1 Reading the library

The first step that is performed by PrepScan is reading the "system" library. The library should be in the
EDT format (as explained in the previous section), either in ASCII or compiled in a binary file. Optionally, a second "user" library can be specified, in order to supplement or correct the main library. The idea is that the design may contain cells that do not exist in the "system" library, but are only valid for this particular design. Instead of editing the system library, these cells can be described in a separate file, the "user" library. If a cell occurs in both the system and the user library, the port counts and directions of both cells are compared with each other. If they match, all properties of the cell in the system library are copied to this cell. Then the cell in the system library is deleted. If they do not match, the cell in the system library is also deleted. In this way, cells in the system library can be overruled. Hence, the user library has the highest priority.
The advantage of this approach is that the system library does not have to be changed for each design. It remains stable, while it can implicitly be updated via the user library. This greatly improves the surveyability of library files.

7.2.2 Checking the library

The library that is read in must satisfy a few rules, provided InScan can work with it. The library must of course be valid according to the library format [EDT Library format]. This can be checked for with 'LibCheck', the library checker. If 'LibCheck' cannot validate the library, then the library is not considered suitable for processing by InScan.
First it is checked if the library contains at least one D-flip-flop (DFF, this is the kind of memory element that InScan will substitute). If this is not the case, then PrepScan will be stopped because PrepScan cannot perform any actions if no DFFs are recognised. Also, buffers, inverters and Scan-flip-flops (SFFs) are searched for in the library. Not finding any of these elements results in a warning. Furthermore, all ports of DFFs and SFFs must have a *class* property attached to them (a *type* property might also be available, but is not obligatory, since there is a default value defined for it). For more information on these properties refer to [ED&T Library Norm].

7.2.3 Reading the design

Now, a design description can be read in. This description can either be an EDIF 2 0 0 netlist or a Philips NDL design netlist (either in ASCII or compiled in a binary file). The language will be recognised automatically. After reading the design, it must be merged with the library. This means that all references to leaf cells in the design are redirected to leaf cells of the library. Of course, this is only possible if for all leaf cells of the design a matching cell can be found in the library. This is checked for.

7.2.4 Creation of the restart file

After reading the library and the design, a binary restart file can be written. This file contains a binary description of the library and the design, and is to be read in by ScanIt. It serves two purposes: First, it ensures that the modifications, proposed by PrepScan, will be executed on the same netlist/library combination as they were generated for, because the user is not able to modify the binary restart file (doing so will always be detected, since the binary restart file is encrypted). Second, it provides a much faster way of reloading the netlist and library, because the binary file is a direct representation of the netlist data structure, which means that the netlist and library do not have to be parsed again. They can directly be put in a netlist data structure. Also, checking will not have to be performed anymore.

7.3 Creation of the routing plan

Now, we are ready to create a routing plan for the design just read in. A routing plan for a design can be created in many ways. In this section we will describe how this is done by PrepScan. It will be shown that the entire operation is performed during the clock domain analysis.

7.3.1 Clock domain analysis

In chapter 6 we defined a clock domain as the part of a design that is clocked simultaneously, i.e., on the same edge(s) of the same clock port(s). We need this concept to make sure that all memory elements a scan chain is constructed of, are clocked simultaneously. Only then, the scan chain will be able to shift a value from its scan-in port to its scan-out port in test mode.

The clock domain in which a memory element (which is contained in the library) is contained, is determined by two factors, namely the *polarity* property of the clock port of the memory element, and the logic in the clock line of this memory element.
The polarity property defines whether the element clocks on the rising or falling edge of the clock signal. If it is not available, it is assumed that the element clocks on the rising edge of the clock signal.
The logic in a clock line can invert the clock signal, thus influencing the clock domains. This means that this logic has to be recognised and interpreted. Only then are we able to recognise the clock domains. This process we call clock domain analysis.

7.3.1.1 Logic in the clock line

Clock lines can contain logic. Obvious examples are buffers or inverters that are included in the clock line. A designer might however have a good reason for allowing the clock line to pass through other elements than buffers and inverters. There should be a way to tell PrepScan through which elements a clock line is allowed to pass. This can be done with the *clock line elements* file (cle file). This file is denoted with 'Cle' in figure 7.1.

© Philips Electronics N.V.

The cle file conforms to the routing plan language and is thus in fact a routing plan. This is however NOT the way it should be interpreted. The routing plan format was only chosen for the cle file because it already existed and was able of describing paths through cells. So a chain should in this case be interpreted as a *clock chain.*

The cle file consists of macros, each macro corresponding with a cell in the design. Note that these cells do not have to be leaf cells. A macro has at least one clock chain. Each chain denotes that a clock line can pass through the cell that corresponds with the macro, from the scan-in input to the scan-out output. The chains are not supposed to have chain elements, so that all macros are leaf macros. Note that it is hence possible to use a leaf macro for describing the clock chains of a non-leaf cell. A macro can have more chains, indicating more paths through which a clock line can pass. Below, an example is given for an inverter 'YIN1' that can be used in the clock line:

```
Macro YIN1
        ClockDomain none NotDriven
                Chain chn
                        Length 0
                        ScanIn in1
                        ScanOut -out1
                EndChain
        EndClockDomain
EndMacro
```

This macro states that a clock line can pass through any instance of type 'YIN1', from input 'in1' to output 'out1'. Furthermore, the '-' sign before the scan-out output indicates that the clock line will be inverted if it passes through this element via the path denoted by this chain.

Since buffers and inverters are common elements in the clock line, it is not necessary to provide them via a cle file. PrepScan will internally generate a cle data structure with these elements. If, however, the clock should be able to pass through any other element, this element must be provided to PrepScan via a cle file. The cle data structure will then be completed with the buffers and inverters that are not yet contained in it. This updated cle data structure will also be written to disk again.

7.3.1.2 Tracing the clock lines

Now that we know through which elements a clock line can pass, we are able to perform a clock domain analysis. Since clock lines are used in all the hierarchy levels of a design, this analysis involves tracing the clock lines through the entire design.

Tracing of a clock line starts at the lowest level in the hierarchy, with the leaf cells. If a leaf cell would be a valid chain element, i.e., it is a D-flip-flop, then the (master) clock port of this element is taken as the starting point. These ports are the only way through which we can recognise a clock line, since only these ports are marked with a property that identifies them as clock ports. Via this port we trace upwards in the design hierarchy via the net that is connected to it. We will explain this with an example.

In figure 7.2 a design is given with four D-flip-flops. The clock ports of these memory ele-

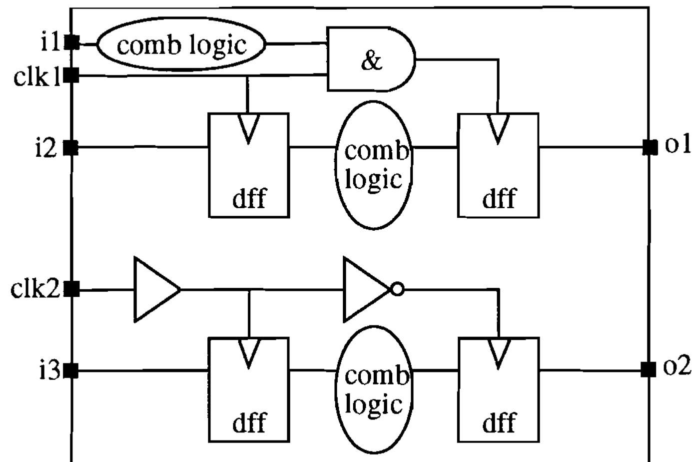


Figure 7.2: Example design for clock domain analysis

ments are connected in various ways. The upper left DFF has its clock port directly connected to clk1 via a net. This means that tracing this element will result in a clock domain, driven by port clk1. We call this clock domain 'clk1_pos'.
The upper right DFF has its clock port connected to the output of an and-gate. Since we did not provide a cle file with this element in it, the clock line cannot pass through this gate. This means that this flip-flop cannot be entered in the routing plan, since we do not know how to get a clock signal to this element. A warning message will be displayed by PrepScan if such a situation occurs.
The lower left flip-flop has its clock port connected to the output of a buffer. Since this element can be traced through by default, tracing will continue at the input of the buffer. From there, the clock line traces to port clk2, which results in our second clock domain, called 'clk2_pos' and driven by port clk2.
The lower right flip-flop has its clock port connected to the input port of an inverter. This element can also be traced through by default, so that tracing will continue at the input of the inverter. Since the tracer passed an inverter, it must be kept in mind that an inversion in the clock line occurred. Now a special case occurs, since we arrive at a point that already has been traced. This is not a problem, tracing can just go on, regardless of the inversion we just encountered. Again, we arrive at port clk2, this time however with an inversion noted in the clock line. Because of this inversion, this flip-flop cannot be entered in clock domain 'clk2_-pos'. Therefore, we create a new clock domain 'clk2_neg'. This clock domain clocks its elements on the falling edge of port clk2.
So, as a result of our clock domain analysis, we have encountered two clock ports which resulted in three clock domains.

As can be seen from the previous example, it is possible that a single clock port clocks elements on both its rising and falling edge. This results in two clock domains driven by the same clock port, however with a different *polarity*. We model a rising edge by a positive polarity, and a falling edge by a negative polarity, just the way as it is modelled in the library norm [ED&T Library norm]. We do not consider pulse triggered memory elements.

 © Philips Electronics N.V.

InScan currently does not support multi-phase clocking. If such clocking schemes are used, InScan will only analyse the master clock ports of the memory elements. Only these ports will show up in the routing plan. The slave clock ports will not be checked, and will not be mentioned in the routing plan.

7.3.2 Implementation

In the previous section we have shown how clock domain analysis takes place. Each time a clock port was encountered at the edge of a cell, a clock domain was created for this cell. These clock domains must group the elements that are clocked simultaneously.
Clock domain analysis must be performed when creating a routing plan. We have seen that the clock domains are already created by it. Since we are in fact tracing the memory elements, we could just as well create a chain for each clock domain, and put each memory element in the chain of the correct clock domain, as a chain element. In this way, the routing plan is build up during the clock domain analysis. The advantage of doing this at the same time is that it results in better programming code. Below, an outline is given of how the algorithm is implemented.

```
DetermineClockDomainsAndCreateRoutingPlan( cell )
{
    forall inst ∈ cell.I : DetermineClockDomainsAndCreateRoutingPlan( cell( inst ) )
    if ( IsDFF(cell) )
    {
        if (MACRO_cell = ∅ )
        {
            CreateSffMacro( cell )
        }
    }
    else
    {
        forall inst ∈ cell.I : if ∃ mc ∈ MACRO_cell(inst)
        {
            forall cd ∈ mc
            {
                cp = GetFirstClockPort( cd )
                ∃ pr ∈ PORTREF_cell : port(pr) = cp.P
                dp = GetDrivingPort( pr, inv )
                ∃ c ∈ CELL : dp ∈ c.P
                chn = GetOrCreateChain( c, cp, inv )
                if ( IsDFF( cell(inst) ) )
                {
                    ∃ chain ∈ cd : chain.INV = false
                    ∃ chel ∈ CHAINEL : chain(chel) = chain
                    chn.EL = chn.EL ∪ chel
                }
```

```
            else
            {
                forall chain ∈ cd
                {
                    ∃ chel ∈ CHAINEL : chain(chel) = chain
                    chn.EL = chn.EL ∪ chel
                }
            }
        }
    }
  }
}
```

IsDFF(cell) returns a boolean value that denotes whether *cell* is a D-type flip-flop. This will be true if cell is a leaf cell and the properties attached to it identify it as such a flip-flop.

CreateSffMacro(cell) creates macros for D-type flip-flops. Such a macro is special because it indicates how a scan chain passes through the *scannable variant* of the D-type flip-flop, not the D-type flip-flop itself. This is of course necessary because the scannable variant is going to replace the D-type flip-flops. The macro will get the name of the DFF followed by "_sff". The macro will have one clock domain, which is driven by the clock port of the scannable variant. The clock domain has at least one chain, with scan-in port and scan-enable port being ports of the scannable variant. The primary output port of the scannable variant is taken as scan-out port of this chain. If the scannable variant has another output port, yet another chain is created, which is exactly the same as the previous chain, with the exception that the scan-out port of this chain will become the second output port of the scannable variant. This port must be the inverse of the primary output port. This makes it possible to specify the scan chain to go from the scan-in port of the scannable variant to the inverted output. This might be advantageous in certain cases (see chapter 9 for such a case). Note that when one of these chains will nowhere be used in the routing plan, it will be deleted from it (this is in fact what PrepScan will do if no optimisation of the routing plan is performed). Below, an example is given of a macro that could be created by this routine:

```
Macro dff_sff ReplacedFor dff
        ClockDomain clk_pos DrivenBy ( clk )
                Chain chn_q
                        Length 1
                        ScanIn scanin
                        ScanOut q
                        ScanEnable scanenable
                EndChain
                Chain chn_qn
                        Length 1
                        ScanIn scanin
                        ScanOut qn
                        ScanEnable scanenable
                EndChain
        EndClockDomain
EndMacro
```


© Philips Electronics N.V.

GetFirstClockPort(cd) returns the first clock port that drives clock domain *cd*. Since the algorithm does not create clock domains driven by more than one clock port, the first clock port will always be the only one possible.

GetDrivingPort(pr, inv) traces the portref *pr* to a port of the containing cell. If the path from the portref to this port contains an inversion, *inv* will become *true*. Elements that are mentioned in the cle file can be traced through. By default, inverters and buffers can be traced through.

GetOrCreateChain(c, cp, inv) returns the chain that belongs to the clock domain which is driven by *cp*, or *-cp* if *inv* is *true*. The clock domain should belong to the macro that corresponds with cell *c*. It is possible that such a macro, clock domain and/or chain does not exist yet. In this case they are created.

7.3.3 An example of routing plan generation

To clarify the algorithm previously presented, we will now give an elaborate example of how a routing plan is constructed. In figure 7.3 a design is shown that we will use in our example.

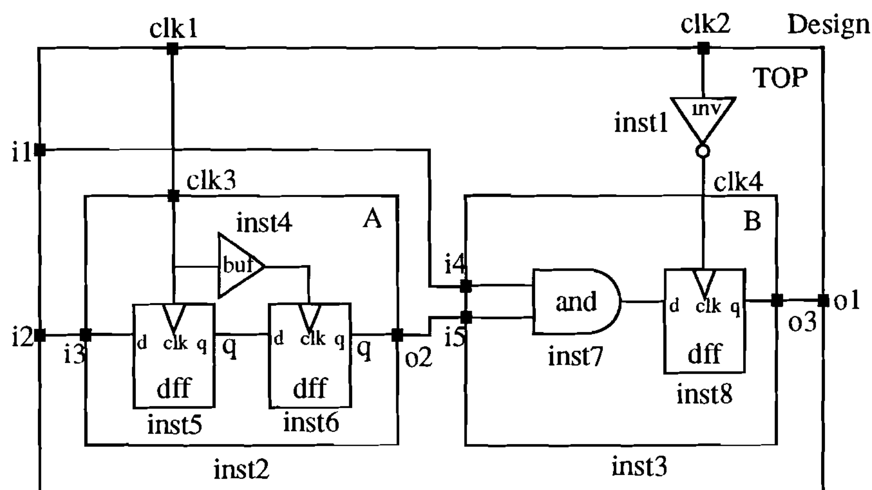


Figure 7.3: Example design for routing plan generation

This design is hierarchical. It consists of a cell **TOP**, which in turn consists of 3 cells, an inverter, a cell **A** and a cell **B**. We will apply the algorithm on this example. The algorithm must be called with TOP as argument:

(1) *DetermineClockDomainsAndCreateRoutingPlan(***TOP***)*

The algorithm will start by recursively calling itself on the instances of **TOP**. This recursion will go on until the deepest level in the hierarchy is reached. This results in the following calls:

(2) *DetermineClockDomainsAndCreateRoutingPlan(***A***)*
(3) *DetermineClockDomainsAndCreateRoutingPlan(***buf***)*
(4) *DetermineClockDomainsAndCreateRoutingPlan(***dff***)*

© Philips Electronics N.V.

(5) *DetermineClockDomainsAndCreateRoutingPlan(***dff***)*
(6) *DetermineClockDomainsAndCreateRoutingPlan(***B***)*
(7) *DetermineClockDomainsAndCreateRoutingPlan(***and***)*
(8) *DetermineClockDomainsAndCreateRoutingPlan(***dff***)*
(9) *DetermineClockDomainsAndCreateRoutingPlan(***inv***)*

When come to the deepest level of the hierarchy, the recursion stops and the algorithm will start analysing the cell under consideration.

In the above example, step (3) is the first step where the recursion stops. The first action the algorithm performs is checking whether the cell under consideration (cell **buf** in this case) is a D-type flip-flop for which no macro exists yet. Since **buf** is a buffer and not a flip-flop, the algorithm is done processing this cell, which means that step (3) is now completed. No action is undertaken, which was exactly what was desired, since we do not want to put any combinatorial logic in the routing plan by default.

Now, step (4) will be performed. Since **dff** is indeed a D-type flip-flop, and no macro does exist for it yet, the routing CreateSffMacro(**dff**) is called. This results in the following macro being created:

```
Macro dff_sff ReplacedFor dff
        ClockDomain clk_pos DrivenBy ( clk )
                Chain chn_q
                        Length 1
                        ScanIn scanin
                        ScanOut q
                        ScanEnable scanenable
                EndChain
        EndClockDomain
EndMacro
```

Or, denoted in our formal model:

$MACRO_{DFF}$ = { dff_sff }, with dff_sff = { clk_pos }
$CLOCKDOM_{DFF}$ = { clk_pos }, with clk_pos = { chn_q }
$CHAIN_{DFF}$ = { chn_q },
with chn_q = { scanin, scanout, *false*, {scanenable}, {clk}, ∅, 1 }

This macro has only one chain because **dff** has only one output, namely 'q'. If dff would also have had an inverted output 'qn', than the above macro would have got an extra chain, equal to the one we have now, except for the scan-out port, which would become 'qn'. The choice of chain pins will be explained later. This macro defines a chain for all elements of type **dff**, so it only has to be created once, and not for each instance of **dff**.

Step (4) is now completed, so the process is proceeded with step (5). Step (5) again involves an element of type **dff**. Since we have just created the macro for this type of element, it needs not to be done anymore. Hence, this step does not result in any actions.

© Philips Electronics N.V.

Since we have now processed all instances of cell **A**, we are able to process this cell now. This is step (2). The algorithm will now take a look at all instances of **A** for which a macro exists. In our case, both instances inst5 and inst6 of type **dff** are represented by a macro (macro `dff_sff`). Instance inst4 is a buffer for which no macro is created, as was explained in step (3).
First inst5 will be processed. Macro `dff_sff` has one clock domain, named `clk_pos`, which is driven by a clock port 'clk'. This clock port corresponds with the clock port of element **dff**. The corresponding port of inst5 is also called 'clk' in the design. This port is traced to a port of cell **A**. As can be seen in figure 7.3, this only involves following the net that is connected to 'clk' to port 'clk3', so no inversion is encountered. Now it is time to create a part of the routing plan. It is checked if a macro for **A** already exists. Since this is not yet the case, one is created. Then, it is checked if the macro corresponding with A already has a clock domain that is driven by port 'clk3'. Since this is also not the case, one is created. At the same time with creating the clock domain, a chain is created in this clock domain. Now, the algorithm performs its most important task: It adds a new chain element to the just created chain. This chain element refers to the chain that was present for inst5. So, we get the following macro:

```
Macro A
        ClockDomain clk3_pos DrivenBy ( CLK3 )
                Chain chn_clk3_pos
                        Length 1
                        ...
                        Order
                                inst5 OfType dff_sff Length 1
                        EndOrder
                EndChain
        EndClockDomain
EndMacro
```

Or, denoted in our formal model:

$MACRO_A$ = { A }, with A = { clk3_pos }
$CLOCKDOM_A$ = { clk3_pos }, with clk3_pos = { chn_clk3_pos }
$CHAIN_A$ = { chn_clk3_pos }, with chn_clk3_pos = { ∅, ∅, *false*, {∅}, {clk3}, {inst5}, 1 }

Note that the chain pins are not present yet. They are not created until the routing plan is complete, because some optimisations are possible. This will be discussed later.
Instance inst5 is now processed, so we can proceed with instance inst6. The same actions are performed. Note that during tracing of port 'clk' of inst6, inst4 will be encountered. Since this element is a buffer, which can by default be traced through, tracing will continue at the input port of inst4, and end up at port 'clk1'. This port is the same one that we encountered during tracing of inst5. Since no inversion took place during tracing, just as was the case with tracing inst5, we can conclude that inst5 and inst6 are clocked simultaneously and should thus be put in the same clock domain. Therefore, we add the chain element, referring to the chain of inst6, in the chain of the existing clock domain of inst5 in macro **A**. Since this was the last instance of cell **A** that we had to consider, macro **A** now is complete. It is shown below:

```
Macro A
        ClockDomain clk3_pos DrivenBy ( clk3 )
                Chain chn_clk3_pos
                        Length 2
                        ...
                        Order
                                inst5 OfType dff_sff Length 1
                                inst6 OfType dff_sff Length 1
                        EndOrder
                EndChain
        EndClockDomain
EndMacro
```

Or, denoted in our formal model:

$MACRO_A$ = { A }, with A = { clk3_pos }
$CLOCKDOM_A$ = { clk3_pos }, with clk3_pos = { chn_clk3_pos }
$CHAIN_A$ = { chn_clk3_pos },
with chn_clk3_pos = {∅, ∅, *false*, {∅}, {clk3}, {inst5, inst6}, 1 }

Cell **B** will be processed in the same way. We therefore will not discuss steps (6), (7) and (8) in detail. Below, the macro for cell **B** is shown:

```
Macro B
        ClockDomain clk4_pos DrivenBy ( clk4 )
                Chain chn_clk4_pos
                        Length 1
                        Order
                                inst8 OfType dff_sff Length 1
                        EndOrder
                EndChain
        EndClockDomain
EndMacro
```

Or, denoted in our formal model:

$MACRO_B$ = { B }, with B = { clk4_pos }
$CLOCKDOM_B$ = { clk4_pos }, with clk4_pos = { chn_clk4_pos }
$CHAIN_B$ = { chn_clk4_pos },
with chn_clk4_pos = { ∅, ∅, *false*, {∅}, {clk4}, {inst8}, 1 }

Step (9) involves processing a combinatorial leaf cell, and will have the same effect as step (3), namely doing nothing.

Now, all instances of **TOP** are processed, and **TOP** itself can be processed. This is done in the same way as in steps (2) and (6). The algorithm will only consider the instances that have corresponding macros. These are inst2 (of type **A**) and inst3 (of type **B**).

Let's start with inst2. The algorithm will take the first clock port of each clock domain of the corresponding macro **A**. In our case, with only one clock domain, this is clock port 'clk3'. This port is found in the design, and traced to a port of **TOP**. This will result in port 'clk1'.

© Philips Electronics N.V.

No inversion took place during tracing, so we will create a macro TOP with a clock domain driven by port 'clk1', and a chain in this clock domain, containing a chain element that refers to the one chain of macro A.

Now inst3 can be processed. Tracing encounters an inverter. This element can be traced through by default, but this results in an inversion in the clock line. Tracing will stop at port 'clk2', so that a new clock domain is created which is driven by port '-clk2' (Note the '-' symbol to denote that the elements in this clock domain are clocked on the falling edge of 'clk2'). The clock domain will get a chain with one chain element referring to the chain of macro **B**. This concludes the algorithm. The macro for cell **TOP** is shown below:

```
Macro TOP
        ClockDomain clk1_pos DrivenBy ( clk1 )
                Chain chn_clk1_pos
                        Length 2
                        Order
                                inst2 OfType A Length 2
                        EndOrder
                EndChain
        EndClockDomain

        ClockDomain clk2_neg DrivenBy ( -clk2 )
                Chain chn_clk2_neg
                        Length 1
                        Order
                                inst3 OfType B Length 1
                        EndOrder
                EndChain
        EndClockDomain
EndMacro
```

Or, denoted in our formal model:

$MACRO_{TOP}$ = { TOP }, with TOP = { clk1_pos, clk2_neg }
$CLOCKDOM_{TOP}$ = { clk1_pos, clk2_neg },
with clk1_pos = { chn_clk1_pos } and clk2_neg = { chn_clk2_neg }
$CHAIN_{TOP}$ = { chn_clk1_pos, chn_clk2_neg },
with chn_clk1_pos = { ∅, ∅, *false*, {∅}, {clk1}, {inst2}, 2 }
and chn_clk2_neg = { ∅, ∅, *true*, {∅}, {clk2}, {inst3}, 1 }

So, two scan chains have been found. These scan chains cannot be combined into one chain because they reside in different clock domains.

7.4 Plugging in partial scan algorithms

At this point, PrepScan has proposed a routing plan (without chain pins). All D-type flip-flops used in the design are entered in this routing plan, so we have used the full scan method. PrepScan is implemented in such a way, that it is possible to use a partial scan algorithm to create an 'optimised' version of the routing plan. In this optimised version, not all flip-flops

used in the design will be entered in the routing plan. The algorithm decides which will become scannable and which will not. In chapter 9 this will be discussed in further detail, and an algorithm will be presented that performs a certain optimisoptimisation.

PrepScan has been set up in such a way, that it is very easy for a programmer to add partial scan algorithms to PrepScan. This should be done at this place, right after creation of the default routing plan. The routing plan is internally stored in a 'Scan Data Structure' (SDS). This structure has been specially designed to perform operations on a routing plan, so that it is possible to change the routing plan that PrepScan proposes. The operations that SDS can perform are described in the SDS user manual [SDS].

The programmer that supplies an algorithm has the complete freedom to use all routines, provided by SDS, on the routing plan. He might however alter the routing plan in such way, that it does not constitute a valid routing plan for InScan anymore. This is possible because the routing plan syntax is more comprehensive than InScan supports. Therefore, only a subset of the possible routing plans is valid for use by InScan. To be part of this subset, a routing plan must obey to a number of rules. These rules are further explained in chapter 8, the ScanIt chapter, because ScanIt will check whether the routing plan it reads satisfies the rules. It is undesirable that an algorithm produces a routing plan that cannot be validated by ScanIt. Therefore, the programmer has to make sure that a routing plan, created by an optimising algorithm, obeys to the rules given in chapter 8. This can be easily checked by applying ScanIt to this routing plan.

Of course, to be able to do this, PrepScan must terminate correctly. In order to do so, the routing plan must be written. This means that the routing plan should at least be writable: It should conform to the rules given in appendix B. Furthermore, some rules apply to the creation of the chain pins of new chains. When and when not to create them will be explained in the next section.

7.5 Creation of the chain ports

7.5.1 Sharing ports

During the creation of the routing plan, most chain ports are not created yet. This is done because some optimisoptimisation is possible with this creation. Consider the following example:

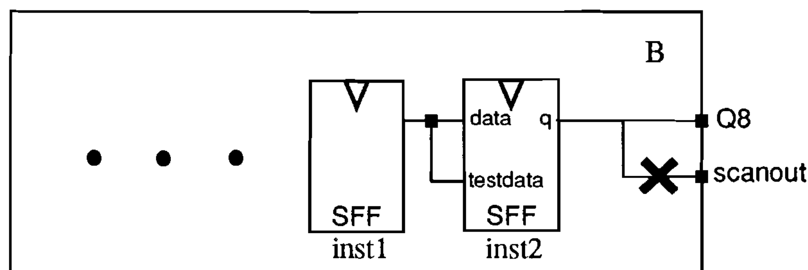


Figure 7.4: Example of existing output pins

In figure 7.4 an example is given where the output of a SFF is directly connected to the border of its containing cell B. Now consider the situation where this SFF is entered in a chain as the last chain element:

```
Macro B
        ClockDomain clk DrivenBy ( clk )
                Chain chn
                        ...
                        ScanOut scanout
                        Order
                                ...
                                inst2 OfType dff_sff Length 1
                        EndOrder
                EndChain
        EndClockDomain
EndMacro
```

This would mean that the scan-out port of this chain is in fact the scan-out port of SFF (its last chain element). If in the routing plan just a default port name is used for the scan-out port of this chain, e.g. 'scanout', as is done in the sample of the routing plan above, then we would get the situation as depicted in figure 7.4. The scan-out port 'scanout' is created, but this was not really necessary. There already existed a port 'Q8' that just as well could have been used for this purpose, since it is connected properly. This kind of optimisation can only be done if the routing plan is stable, since only then is it known which chain elements are at the end of a chain. Therefore, creation of the ports is done after the optimisation of the routing plan, since this optimisation can change the order in which chain elements occur in chains, and can delete certain elements from chains.

Since this kind of optimisation is easily acquired, PrepScan will perform it automatically. Also, ScanIt will check if existing ports can be used, because the designer may have modified the routing plan. If ScanIt detects the situation described above, it will report it and automatically use the existing port. However, a switch is provided with InScan to disable this property of ScanIt, so that it is possible for a designer to override the choice of ports PrepScan proposes, and let ScanIt use the ports he indicated, regardless of occurrence of the situation described above. This makes the ports to come out more predictable.

7.5.2 Leaf macro port creation

Chain ports will always be created for non-leaf macros. For leaf macros some special cases occur:
If a leaf macro has its name equal its 'DerivedFrom' name (or has no 'DerivedFrom' name, which is the same), the chain ports are supposed to exist already. This type of macro will not be created by PrepScan by default, but an optimising algorithm can create it. This algorithm should then also create the chain ports. The reason for this is that these kind of leaf macros do not refer to scannable elements by nature. Therefore, it is not simply possible for the port creation algorithm to easily decide which ports are chain ports, since no properties are attached to them that identify them as having a certain scan functionality. The optimising algorithm that creates this macro should know what this macro is used for, and thus should also know

which ports should be the chain ports.
If a leaf macro has its name not equal its 'DerivedFrom' name, the scan-out ports are supposed to exist already. The other chain ports do not have to exist. They will be created. The reason that the scan-out ports must already exist, is that, when the leaf macro has two chains, e.g. it refers to a scannable flip-flop with two scan-out ports (one the inverse of the other), it is not possible to determine which scan-out port belongs to which chain, since the only thing that these chains will normally differ in is in fact their scan-out port. Since an optimising algorithm might want to distinguish between these chains, the scan-out ports should be present.

7.5.3 Naming conventions

The naming convention for ports created by PrepScan is the following:
The port names of chains of non-leaf macros by default consist of a prefix, followed by an underscore, the clock domain name, another underscore, and the chain name. The prefix can be either 'scanin', 'scanout', or 'scanenable'. For example, the scan-out port of a chain 'x' of clock domain 'y' would be called 'scanout_y_x'. This naming convention assures that each port name is unique within the macro. It can however produce rather long names, and may therefore be improved somewhat in the future.

The port names of chains of leaf macros are the names of the corresponding ports on the scannable variant of the leaf cell. They should always exist on the scannable variant.

7.6 Creation of the match rule file

Now that the routing plan is created, we are able to create the *match rule file*. The match rule file contains descriptions of how the D-type flip-flops must be matched by scannable variants. We will call every single description a *match rule*. First, we will give an overview of the format of the match rule file and then discuss how the file will be generated.

7.6.1 Format of the match rule file

The match rule file format describes for every D-type flip-flop in the design that is to be made scannable, with what it should be substituted to create a scannable variant. For this purpose, a netlist description language is suitable. Therefore, the match rule file will be created in the NDL format.

We have to create an entry in the match rule file for every leaf macro in the routing plan that has its name not equal to its 'ReplacedFor' name. We will call these macros 'ReplacedFor' macros, since they indicate a replacement of the original element with a scannable variant. These macros, which are created by the CreateSffMacro(**cell**) routine that was explained in the previous section, describe how the chains pass through the scannable variants of the D-type flip-flop, but do not specify how these scannable variants are constructed. That's exactly what is specified in the match rule file. Consider the following macro:

```
Macro dnn10tad_sff ReplacedFor dnn10tad
        ClockDomain clk_pos DrivenBy ( clk )
```

 © Philips Electronics N.V.

```
                Chain chn_q
                        Length 1
                        ScanIn si
                        ScanOut q
                        ScanEnable se
                EndChain
                Chain chn_qn
                        Length 1
                        ScanIn si
                        ScanOut qn
                        ScanEnable se
                EndChain
        EndClockDomain
EndMacro
```

This macro is a typical example of how a 'ReplacedFor' macro would look like. It states that the element 'dnn10tad' will be replaced by element 'dnn10tad_sff', which must be seen as the scannable variant of 'dnn10tad'. 'dnn10tad_sff' is the element that must be created in the match rule file. It is a cell that contains the functionality to represent the scannable variant of 'dnn10tad'. The most obvious example of this functionality is of course a scan flip-flop (SFF), but other implementations can be thought of also. They will be described further on. Below, an example is given of a possible entry in the match rule file, corresponding with the macro above.

```
        MACRO
MACRO dnn10tad_sff
# I( d, clk, si {scanin}, se {scanenable} )
# O( q {scanout}, qn {scanout} )
sff sfs32tad I( d, clk, se, si ) O( q, qn )
        MEND
```

Please keep in mind that the example is in the NDL format. As can be seen in table 4.1, the NDL concept of 'macro' is in fact equal to the EDIF concept of 'cell'. So, an NDL macro is something different than a routing plan macro.

The example states that the scannable variant of 'dnn10tad', called 'dnn10tad_sff', can be composed of a scannable flip-flop of type 'sfs32tad'. The connections between 'dnn10tad_sff' and its instance 'sff' of type 'sfs32tad' are given, to be able to substitute an instance of type 'dnn10tad' with this element correctly. The chain ports of 'dnn10tad_sff' contain properties that describe the type of chain port. We will call these properties *chain port properties* in the remainder of this report. Possible properties are {scanin}, {scanout} and {scanenable}. In the example above, port 'si' is the scan-in port of 'dnn10tad_sff', port 'se' is the scan-enable port, and port 'q' and 'qn' are the scan-out ports. The properties will be created by PrepScan to ease further processing. They are also convenient for the user when inspecting the match rule file.

7.6.2 Creation of the match rule file

Now that we understand the format of the match rule file, let us proceed with the creation of the file. As was said before, we have to create an entry in the match rule file for every 'ReplacedFor' macro that can be found in the routing plan. It would however be convenient for the user if it was possible to use an existing match rule file when one is present. Therefore, the following approach is taken:

```
CreateMatchRuleFile( routing plan )
{
    forall 'ReplacedFor' macros mc in the routing plan
    {
        if ( an entry corresponding with mc already exists in the original match rule file )
            CheckMatchRule( mc )
        else
            CreateMatchRule( mc )
    }
}
```

This approach allows (possibly incomplete) match rule files to be read and updated. The two routines CheckMatchRule(*mc*) and CreateMatchRule(*mc*) will be explained below.

7.6.2.1 Checking an existing match rule

If a match rule already existed in the original match rule file, the match rule should be checked, to be sure that it is indeed the correct match rule. This checking is done by CheckMatchRule(*mc*). If a check fails, PrepScan will generate an error, and will exit. The following checks are performed:

- The various port counts of a match rule cell must be correct. This means that the number of input ports of the match rule cell must be equal to the number of input ports of the original D-type flip-flop (DFF) plus 2, since the match rule cell must have a scan-in port and scan-enable port extra. The number of inout ports of the match rule cell must be equal to the number of inout ports of the original DFF, and the number of output ports of the match rule cell must at least be equal to the number of output ports of the DFF (Scan-out ports are supposed to be existing output ports of the original DFF and thus do not introduce new output ports on match rule cells, but it is no objection to let the scannable variant have more outputs (for example an extra inverting output) than the DFF).

- All ports that are present on the original DFF should be present on the match rule cell, before the declaration of the chain ports.

- The chain ports should be labelled with chain port properties. If this is not the case, the first chain port is assumed to be the scan-in port, and the second chain port is assumed to be the scan-enable port. Note that this are assumptions. It cannot be checked easily

 © Philips Electronics N.V.

whether a port is the scan-in or the scan-enable port, since that would require us taking a look inside the match rule cell to see how the port is connected internally. This can be very complex, since it could mean tracing through unknown cells, and hence is not done.

The above checks should reassure that the existing match rule is correct. Of course, these checks do not cover the contents of a match rule cell, since it is in general virtually impossible to determine if the contents of a match rule cell has indeed the same functionality as the original DFF, plus scan functionality.

7.6.2.2 Creating a match rule with direct substitution

When no match rule does exist yet for a certain DFF type, one must be created. This is done by the routine CreateMatchRule(*mc*). Creation of a match rule consists of finding a scannable variant for the original DFF, and putting this variant in the match rule file. The scannable variant does not have to be a single element, it can also be constructed of several cells. PrepScan will consider two possibilities. The first one is trying to find a single cell in the library that corresponds with the original DFF type, but also has scan functionality. The second one is trying to find or build a 2-input multiplexer, and then build the scannable variant of the original DFF type with that DFF and the multiplexer. We will first discuss the first possibility. The second one will be explained in the next section.

The easiest and most direct way to create a match rule, is trying to find a cell in the library that has the same functionality as the original DFF type for which we create this match rule, plus scan functionality. Below, we present the algorithm that is used to accomplish this task.

FindScannableVariant(dff)
{
    forall $c \in library$
    {
        if (*IsSFF(c)*)
        {
            if (
                $(|IP(c)| = |IP(dff)| + 2) \wedge$
                $(|OP(c)| \geq |OP(dff)|) \wedge$
                $(|IP(c) \cap OP(c)| = |IP(dff) \cap OP(dff)|) \wedge$
                AreCellsPortEquivalent(dff, c)
)
                return (*c*)
        }
    }
    return (*NoMatch*)
}

We have found a scannable variant for *dff* if the variant is a scan-type flip-flop (SFF), the port count of this variant is correct, and the two cells are equivalent regarding port types. The call *IsSFF(cell)* returns true if *cell* is a scan-type flip-flop. The call *AreCellsPortEquivalent(cell1, cell2)* is explained below.

```
AreCellsPortEquivalent( dff, substitute )
{
    forall ip ∈ IP (dff)
    {
        mp = FindMatchingPort( substitute, ip )
        if ( MatchingPortFound )
            MarkPortAsMatched( mp )
        else
            return ( false )
    }
    Let ScanEnableFound = false
    Let ScanInFound = false
    forall ip ∈ IP (substitute)
    {
        if (¬PortAlreadyMatched (ip))
        {
            if ( PortIsScanEnable( ip ) )
            {
                if ( ScanEnableFound )
                    return ( false )
                ScanEnableFound = true
            }
            else
            if ( PortIsScanIn( ip ) )
            {
                if ( ScanInFound )
                    return ( false )
                ScanInFound = true
            }
            else
                return ( false )
        }
    }
    forall op ∈ OP (dff)
    {
        mp = FindMatchingPort( substitute, op )
        if ( MatchingPortFound )
            MarkPortAsMatched( mp )
        else
            return ( false )
```

 © Philips Electronics N.V.

```
    }
    return ( true )
}
```

Note that the algorithm uses some routines that have the following function:
FindMatchingPort(cell, port) returns the first non-marked (and thus non-matched) port of *cell* that is equivalent to *port*. Equivalent here means that both ports have the same *type*, *class* and *order* property (this are specific LDS properties). These properties are sufficient to specify the function of a port. If a matching port can be found, *MatchingPortFound* will become *true*, else it will become *false*.

MarkPortAsMatched(port) marks a port, to indicate that this port already has been matched and thus is not available for another match.

PortAlreadyMatched(port) returns *true* if this port was marked previously with *MarkPortAsMatched(port)*, or *false* otherwise.

PortIsScanEnable(port) returns true if *port* can be identified as a scan-enable port via its properties, or *false* otherwise.

PortIsScanIn(port) returns true if *port* can be identified as a scan-in port via its properties, or *false* otherwise.

When a direct substitute is found, the match rule can be created, according to the format explained before. Creating the match rule in the NDL format is just a matter of creating the network, consisting of cells, nets and ports, in NDS, and then writing it to disk with a "WriteNdl" call. In figure 7.5 the example, that was given in the section about the match rule file format, is repeated.

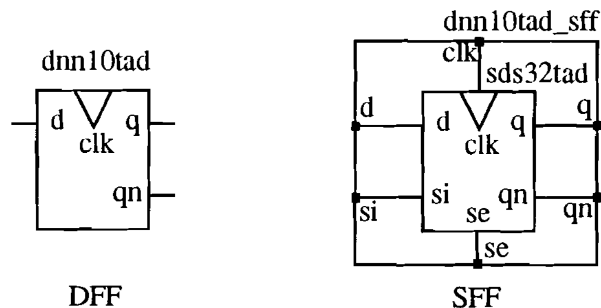


Figure 7.5: Example of direct substitution

The match rule for this example is given below.

```
        MACRO
MACRO dnn10tad_sff
# I( d, clk, si {scanin}, se {scanenable} )
# O( q {scanout}, qn {scanout} )
sff sfs32tad I( d, clk, se, si ) O( q, qn )
        MEND
```

The correspondence between figure 7.5 and this match rule is clear. The match rule is just the netlist description of the network drawn on the right in figure 7.5.

When no substitute can be found with the above algorithm, another approach is taken. This approach is explained in the following section.

7.6.2.3 Creating a match rule with a multiplexer

It might occur that a scannable variant for a certain DFF is not available. In that case, PrepScan will try to create one. In figure 7.6 it is shown how this can be done.

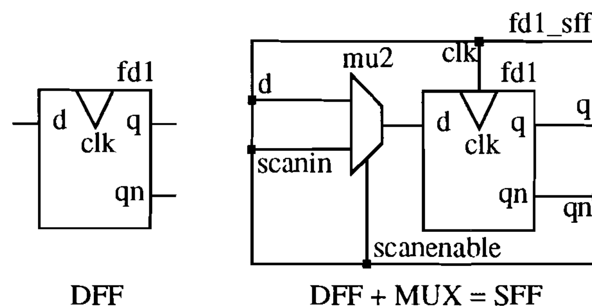


Figure 7.6: Creation of the scannable variant of a DFF

On the left hand side a certain DFF, named 'fd1', is drawn. Now suppose that no scannable variant can be found for this DFF. On the right hand side of figure 7.6 it is then shown how a scannable variant can be constructed with this DFF and a 2-input multiplexer, named 'mu2'. This scannable variant will be put in the match rule file, and will look like this:

```
        MACRO
MACRO fd1_sff
# I( d, clk, scanin {scanin}, scanenable {scanenable} )
# O( q {scanout}, qn {scanout} )
mux mu2 I( d, scanin, scanenable ) O( net1 )
dff fd1 I( net1, clk ) O( q, qn )
        MEND
```

The only difficulty that occurs with this method is that we have to identify a 2-input multiplexer from the library. This is not as trivial as it may seem, because the function of a library elements is only expressed with a *function* property at its output port(s), that describes the function in terms of an expression. E.g., a 2-input and-gate could be described with the following function property connected to its output: (and a1 a2), where 'a1' and 'a2' are the two input ports of the and-gate. The expression must be described in primitive cell types, such as 'and', 'or', 'inv', 'xor' and 'xnor'. In NDS/LDS, there are only routines available that can identify these primitive cells.

However, a 2-input multiplexer is not a primitive cell. It can be described in several ways with an expression. The most obvious way to describe the function of the output port is the following one: (or (and a1 s) (and a0 (inv s))), where 's' is the selection input, 'a0' is the input that is active when 's' is 0, and 'a1' is the input that is active when 's' is 1. Another way could be found by applying DeMorgans rule to this expression, resulting in (inv (and (or (inv s) (inv a1)) (or (s) (inv a0)))). It is impossible to recognise all possible imple-

© Philips Electronics N.V.

mentations of a 2-input multiplexer, let alone multiplexers with more than 2 inputs. Therefore, we are satisfied with only being able to recognise the most obvious description of a 2-input multiplexer, just as it was described by the first rule above.

In order to recognise these multiplexers, we have to expand the library. Expanding the library means that all (leaf) cells are given a contents according to the function property. In figure 7.7 it is shown how this affects the 2-input multiplexer that we want to recognise.

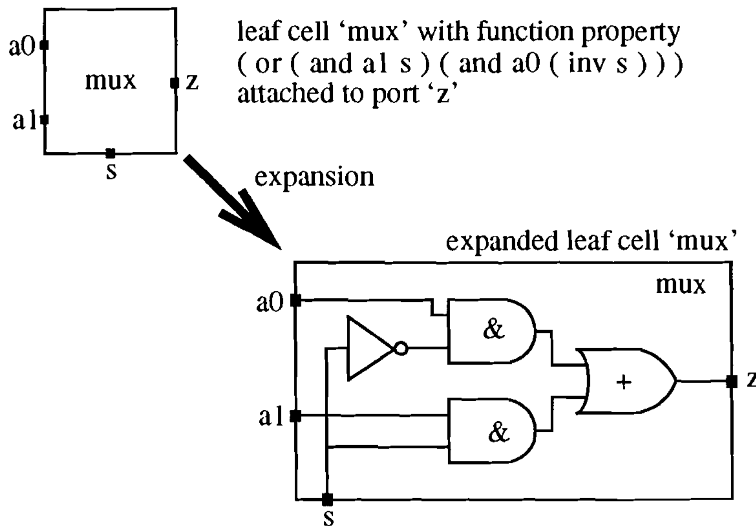


Figure 7.7: Expanding a 2-input multiplexer

The only way to recognise such a multiplexer, is to check whether all elements, ports, and nets, that are depicted in figure 7.7, do exist in a certain library cell in the way as is depicted in figure 7.7, and nothing more does exist in it. If this is the case, then we can conclude that this cell is a 2-input multiplexer. The algorithm that performs this recognition will not be presented here, since its actions, described above, are rather trivial.

It still is possible that no 2-input multiplexer can be found in the library. In this case, PrepScan will try to build such a multiplexer itself. This will be done in the way as is depicted in figure 7.7. This does however require that an inverter, a 2-input and-gate, and a 2-input or-gate are present in the library. Since this are primitive cells, there are routines available in NDS/LDS with which these cells can be found. If one of these type of cells can not be found, there is no other way for PrepScan to generate a scannable variant for the DFF under consideration, and PrepScan will exit without delivering the routing plan and match rule file. If the cells can be found, the multiplexer will be built, and the match rule will look like this:

```
        MACRO
MACRO __mux2
# I( in_0, in_1, sel )
# O( out )
and1 andx I( in_1, sel ) O( netname1 )
and2 andx I( netname0, in_0 ) O( netname2 )
inv1 invx I( sel ) O( netname0 )
or2 orx I( netname1, netname2 ) O( out )
        MEND
```

```
        MACRO
MACRO fd1_sff
# I( data, clk, scanin {scanin}, scanenable {scanenable} )
# O( q {scanout}, qn {scanout} )
dff fd1 I( net1, clk } O( q, qn )
scanmux __mux2 I( data, scanin, scanenable ) O( net1 )
        MEND
```

In this match rule, 'andx' is recognised as a 2-input and-gate, 'invx' is recognised as an inverter, and 'orx' is recognised as a 2-input or-gate. With these gates, the multiplexer '__mux2' is built, which is then used to build the scannable variant 'fd1_sff' of fd1. Note that in this case, the match rule file becomes hierarchical due to the definition of the multiplexer. This is perfectly valid.

7.6.3 Extensions for future use

The match rule file generation, as described above, is sufficient to satisfy all the requirements currently set for InScan. However, InScan will be supplied with partial scan algorithms. One of the algorithms that is most likely to be implemented in the near future is the pipelining recognition algorithm. This algorithm must have the ability to substitute DFFs with SFFs with a hold mode (SFFH). For a discussion of this technique the reader is referred to chapter 9.

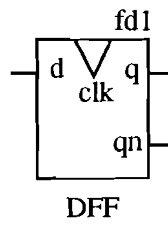
Although NDS/LDS does not support the notion of hold ports in a library yet, PrepScan already is prepared for the support of SFFHs. In particular, this means that the routing plan language has the ability to describe scan chains with a hold-enable port. Also, the match rule file generation algorithm will be able to perform a DFF to SFFH substitution.

There are several ways to perform such a substitution. Depending on the availability of a SFFH, SFF, DFFH or DFF, the following substitutions can be performed by PrepScan:

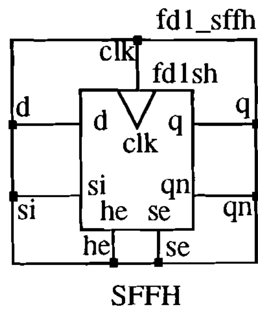
- Direct substitution of DFF with a SFFH
- Substitution of DFF with a SFF + multiplexer, where the multiplexer implements the hold functionality
- Substitution of DFF with a DFFH + multiplexer, where the multiplexer implements the scan functionality
- Substitution of DFF with a DFF + 2 multiplexers, where the multiplexers implement both the scan and hold functionality

In figure 7.8, these substitutions are depicted for a certain type of DFF. Note again that these substitutions cannot be performed yet, because NDS/LDS does not support the notion of hold ports on a library cell. Therefore, hold ports cannot be recognised, and substitutions cannot take place.

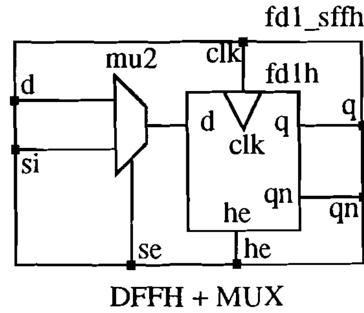
In figure 7.8, 'si' is the scan-in port, 'se' the scan-enable port, and 'he' the hold-enable port.



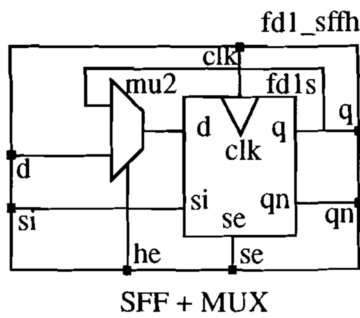
DFF



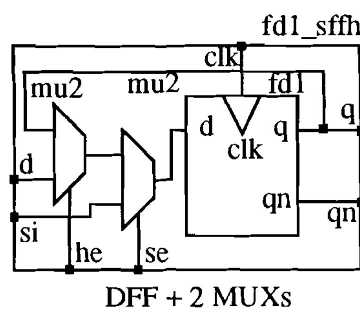
SFFH



DFFH + MUX



SFF + MUX



DFF + 2 MUXs

Figure 7.8: Substitutions for DFF to SFFH

8 ScanIt

The second part InScan consists of is called ScanIt. ScanIt is concerned with the execution of the routing plan and match rule file. In this chapter we will discuss which steps ScanIt performs, and how they are executed.

8.1 Introduction

In Chapter 5, operation of InScan was divided into five actions, of which the last two should be executed be ScanIt. They are repeated here:

1. Replace all memory elements, that are to be entered in a scan path, by their 'scannable' variant, which was proposed in the previous step.

2. Create the scan paths

These steps thus execute the match rule file and the routing plan. InScan was divided into two parts to give the user the opportunity to make changes to the routing plan and the match rule file, hence increasing the flexibility of the program. However, the user could make mistakes while editing these plans, resulting in an erroneous routing plan or match rule file. When these mistakes are syntactic, they will be found by the parsers that read these plans. When they are not, they will not be detected automatically, and ScanIt would not be able to perform its actions correctly. Therefore, a number of checks needs to be performed, to assure that the routing plan and match rule file are correct. This will have to be the first step after reading the routing plan and match rule file, before these plans are executed.

In figure 8.1 a modular description of ScanIt is given, together with the inputs and outputs of the program. This figure describes the most important actions that ScanIt undertakes to perform the steps mentioned above. The modules will be discussed in the following sections.

8.2 The first three modules

The first three modules have to do with reading the various files, needed to operate, namely the restart file, the match rule file, and the routing plan.

8.2.1 Reading the binary restart file

During this step, the binary restart file that PrepScan created is read. This file contains the library and the design on which ScanIt must perform its modifications. If this file does not exist, ScanIt will read the library and design separately, in the same way as PrepScan did. This will however consume more time, since these files are larger than their binary description, and they must also be parsed and checked again.

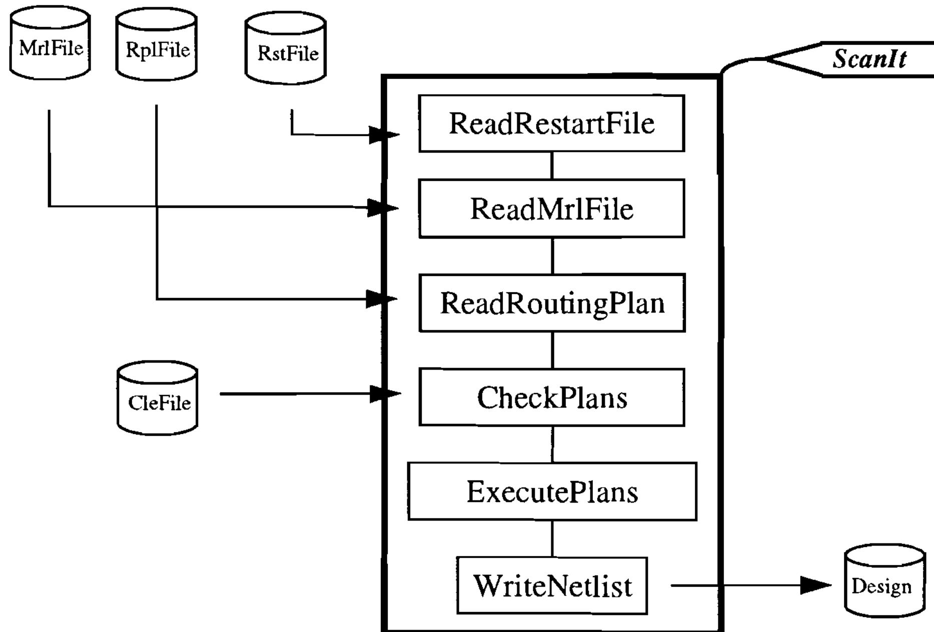


Figure 8.1: Modular architecture of PrepScan

8.2.2 Reading the match rule file

After reading the library and design, the match rule file is read. Since the match rule file is given in the NDL format, a netlist description format that is supported by NDS/LDS, this can be easily done by ScanIt. The NDL parser will report syntax errors and warnings.

8.2.3 Reading the routing plan

Now, the routing plan is read. The routing plan parser will report syntax errors and warnings. Also, some semantic errors and warnings will be reported.
One semantic check that is performed by the parser is worth mentioning in particular, namely the one that checks the existence of enable ports. If a certain chain in the routing plan has some type of enable port (scan-enable, hold-enable or norm-enable, currently InScan only uses the first type), then it is checked if all scan chains that use this chain as a chain element do also have an enable port of the same type. This is required since an enable port of a chain element must be connected to the corresponding enable port of the containing chain. Failing this check results in an error.
The other way around it is also checked that if a chain has some type of enable port, at least one of its chain elements does have a corresponding enable port. Failing this check results in a warning, since it is not critical when this is not the case, it will only result in nets and ports being created used are not used.

 © Philips Electronics N.V.

8.3 Checking the plans

When all necessary file are read, it is time to check the correctness of the files generated by PrepScan, i.e. the routing plan and the match rule file. This checking does not have to be performed with respect to the syntax, since the syntax of both files is already checked by the parsers. Checking only concerns the semantics of these files. Below, we give an extensive summary of the checks that are performed by ScanIt. The order of the checks is the order in which they are performed by ScanIt. This order is imposed by the implementation, since most checks will rely on other checks being passed.

- **Checking the macro validity**

  The first check that is to be performed, is to see if all routing plan macros have a corresponding cell in the design, library, or match rule file. This is of course necessary because a macro describes how scan chains are routed through a (part of the) design. Therefore, there must be some relation with the design. The following procedure is used to check the validity of a macro. Besides checking the existence of a corresponding cell, a few things more are checked, as can be seen in the routine:

```
CheckMacro( mc )
{
    if ( a cell in the design with the 'DerivedFrom' name of mc does exist )
        return
    if ( a cell c1 in the library with the 'DerivedFrom' name of mc does exist )
    {
        if ( leafmacro_c1( mc ) = false )
            exit error
        if ( seq( c1 )
            forall cd ∈ mc forall ch ∈ cd : if ( ch.n = 0 ) exit error
        else
            forall cd ∈ mc forall ch ∈ cd : if ( ch.n ≠ 0 ) exit error
        if ( the 'DerivedFrom' name of mc is different from its 'ReplacedFor' name )
        {
            if ( a cell c2 in the match rule file with the name of mc does not exist )
                exit error
            else
                CheckPortCount( c2, mc, c1)
        }
    }
    else
        exit error
}
```

CheckPortCount(c2, mc, c1) checks that the correct number of input, output and inout ports is available on *c2*, by comparing these figures with the expected port counts which can be calculated from *c1* and *mc*.

- **Checking the internal connections of ports of a match rule**

  One of the more usual errors a designer could make during editing of the match rule file is typing in incorrect port names. Since the match rule file is in the NDL format, and the NDL format does not explicitly model nets between ports (but represents nets with a port name), an incorrect port name would result in the port not being connected to other ports, hence creating a major error. Therefore, it is checked if this situation occurs. All ports of all match rules are checked to see if they are internally connected to other ports. If not, it is assumed that there is something wrong with the match rule, and an error is generated.

- **Checking the ports of a match rule**

  Since we have to substitute the DFFs in our design by their scannable variants, defined in the match rule file, we must have a way to match the ports of a DFF with the ports of its scannable variant. This matching is done by name. Therefore we demand that the original ports of a DFF are present on its scannable variant, and that the chain ports, defined with the chains of the macro that corresponds with the scannable variant, are also present on the match rule. For example, if we have a DFF with the following interface:

  ```
  dnn10tad I( d, clk ) O( q, qn )
  ```

  and the following macro for this DFF:

  ```
  Macro dnn10tad_sff ReplacedFor dnn10tad
          ClockDomain clk DrivenBy ( clk )
                  Chain chn
                          Length 1
                          ScanIn si
                          ScanOut q
                          ScanEnable se
                  EndChain
          EndClockDomain
  EndMacro
  ```

  then the interface of the match rule element should be something like this:

  ```
  dnn10tad_sff I( d, clk, si, se ) O( q, qn )
  ```

  All ports of `dnn10tad` are present (ports `d`, `clk`, `q`, and `qn`), plus the scan-in and scan-enable ports (`si` and `se`) that are mentioned in the macro. The scan-out port is already an existing port of `dnn10tad`. With these naming conventions we are always able to uniquely match the ports.
  We demand that the ports of the original cell are defined before the definition of the chain ports, to give PrepScan the ability to check an existing match rule in which no chain port properties are present. In this case it is possible to identify the chain ports by their position in the port list definition.

  This check makes sure that the correct ports are present on the match rule in the correct order (original ports before chain ports). Note that the order between original ports is not important.

- **Checking the chain port names**

© Philips Electronics N.V.

This check makes sure that the scan-in port, the scan-out port, and the scan-enable port of a single chain are different. It is merely present to prevent errors of the user when editing the routing plan.

- **Matching the chain element with instances in the design**

  A macro describes the scan chains that go through a declaration cell. Such a scan chain is constituted of smaller parts, which are called chain elements. These chain elements are defined through the instances that are contained in the declaration cell. Hence, a chain element corresponds with an instance in the design. This is realised by giving the chain element the name of the instance. We have to check if the instance, that corresponds with a certain chain element in the routing plan , does really exist in the design. This process is called *matching*.

  There is one exception that we would like to make, namely chain elements that correspond with combinatorial elements, such as buffers or inverters. It is convenient to allow the use of this type of element in the routing plan without the element actually having to exist in the design. Therefore, chain elements referring to combinatorial instances do not necessarily have to be matched with an existing instance in the design.

  There are two factors with which a match can be made, namely the name of the instance (which should be equal to the name of the chain element), and the type of the instance (which should be equal to the 'derived' type of the chain element). If an instance with the correct name and type can be found, we will say that this is a *perfect match.* We will first try to find all perfect matches.

  It might however be possible that we're not able to make a perfect match. In this case, we could sometimes find a so-called *type match.* A type match is a match by type, so the names of the chain element and instance are not equal. The type match is introduced to cover certain errors that are introduced by users who manually changed the routing plan. A type match can only be made if there is exactly one possibility. So, if a chain element of type 'X' is found in the routing plan, and there exists exactly one non-matched instance of type 'X' in the correct cell, they can be matched. However, if there would be two non-matched instances of type 'X' in the cell, it is not possible to make a unique match, and an error is produced.

  Type matching is not done for combinatorial elements. This results from the fact that these elements do not necessarily have to exist in the design. Therefore, it is not required that a match exists for such an element. Trying to match it anyway might produce a wrong match. Therefore, this is not done.

  Below, the algorithm is given that performs the abovementioned matching.

FindMatches(**routing plan** *)*

{

  forall macros *mc* in the **routing plan**

  {

    forall $cd \in mc$ **forall** $ch \in cd$

```
            FindPerfectMatch( mc, ch )
        forall cd ∈ mc forall ch ∈ cd
            FindTypeMatch( mc, ch )
    }
}

FindPerfectMatch( mc, ch )
{
    forall ce ∈ ch
    {
        find the instance in (in the cell that corresponds with mc)
            that corresponds with chain element ce
        if ( ce not registered with in )
        {
            register ce with in   *
            MarkChainElementAsMatched( ce, in )
        }
    }
}
```

MarkChainElementAsMatched(ce, in) creates a reference between the chain element *ce* and the instance *in* with which it is matched.

* For each type of instance that appears as a chain element in the routing plan, a macro exists that defines one or more chains through this type of cell. For each instance, all these chains may appear in the routing plan as chain elements. Therefore, these chain elements are registered when they are encountered in the routing plan. If a certain chain element is already registered with an instance, this chain element cannot be matched with this instance anymore. This is the reason why we need the registration.

```
FindTypeMatch( mc, ch )
{
    forall ce ∈ ch
    {
        if ( ¬ MarkedAsMatched( ce ) ∧ chain(ce).n ≠ 0 )
        {
            Let PossibleMatchesCount = 0
            Let PossibleMatch = NoMatch
            forall instances in in the cell corresponding with mc
            {
                if ( in corresponds with ce and ce is not registered with in )
                {
                    PossibleMatchesCount = PossibleMatchesCount + 1
                    PossibleMatch = in
                }
            }
```


© Philips Electronics N.V.

```
            if ( PossibleMatchesCount = 1 )
            {
                register ce with PossibleMatch
                MarkChainElementAsMatched( ce, PossibleMatch )
            }
            else
                exit error
        }
    }
}
```

MarkedAsMatched(ce) returns *true* if chain element *ce* is marked by the routine *MarkChainElementAsMatched.*

As can be seen, elements of length 0 (combinatorial) are not matched by type. The type match is only created if there is exactly one possibility.

- **Checking the clock ports**

  We have to check if the ports that are mentioned in the 'DrivenBy' lists of all the clock domains in the routing plan are indeed clock ports in the design. This is done in two phases. The first phase is checking if these ports do exist on the cells in the design. This step is not difficult, since it only involves searching for a certain port on the cell corresponding with the macro.
  The second phase is to check if these ports are indeed clock ports. This step is also easy, since, prior to this check, the entire design will be traced, in order to identify all the clock ports in the entire design. The clock ports, encountered by the tracing process, will receive a property with which they can be identified as being a clock port. The second phase hence only involves looking for such a property on the port under consideration. If it exists, it is a clock port, if not, it is no clock port, and an error is issued.

- **Checking shared scan-in and scan-out ports of chain elements**

  So far, we have not imposed any restrictions on the chain ports of a chain in the routing plan. Hence, it is possible to specify one and the same port as scan-in or scan-out port of several chains. Consider the following example:

```
Macro X
        ClockDomain clk DrivenBy ( clk )
                Chain chn1
                        ...
                        ScanOut so
                        Order
                                ...
                                insty OfType A Length 4
                        EndOrder
                EndChain
                Chain chn2
                        ...
                        ScanOut so
                        Order
```

```
                                ...
                                instz OfType B Length 6
                        EndOrder
                EndChain
        EndClockDomain
EndMacro
```

Regardless of the contents of the chains, chn1 and chn2 share scan-out port so. In figure 8.2 this situation is depicted.

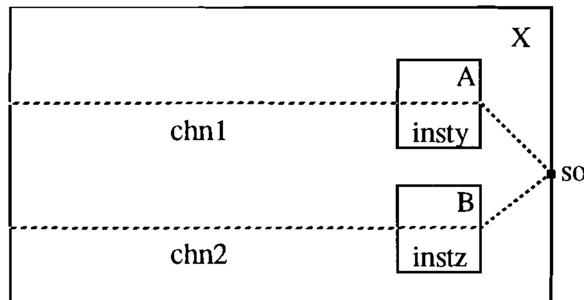


Figure 8.2: Example of incorrect port sharing

This should not be allowed, since port so is now driven by two other ports, which can produce conflicts. Sharing of scan-in or scan-out ports is however not always forbidden. Consider for example a 2-input multiplexer. Below a possible macro of such an element is given.

```
Macro mux
        ClockDomain clk DrivenBy ( clk )
                Chain viaA
                        Lenght 0
                        ScanIn a
                        ScanOut z
                        ScanEnable s
                EndChain
                Chain viaB
                        Lenght 0
                        ScanIn b
                        ScanOut z
                        ScanEnable s
                EndChain
        EndClockDomain
EndMacro
```

In this case, sharing the scan-out port z is allowed, since of course only one of the chains is active at a time. Now a special case occurs when such an element is used as last chain element in two chains. This is pictured in figure 8.3. Both chains chn1 and chn2 share scan-out port so, but since their last chain elements do really share the scan-out port, this situation is valid.

ScanIt will check if scan-in/scan-out ports may be shared between two chains. This is only allowed if the first/last chain elements of both chains refer to the same instance, and render the same scan-in/scan-out port up to the leaf macro level. If in a leaf macro two chains share such a port, this is allowed, since we have no means of checking whether this is correct or not. In this case, it is assumed that the leaf macro has some sort of selection

© Philips Electronics N.V.

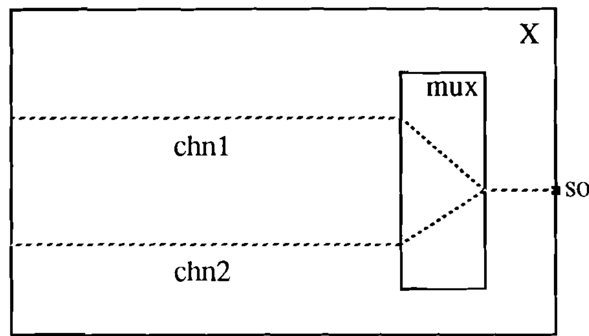


Figure 8.3: Example of correct port sharing

mechanism (like a multiplexer) to choose between the chains. Only one can be active at a time.

- **Checking for resource conflicts**

  As was shown in the previous check, it sometimes is allowed to share certain scan-in or scan-out ports between chains. However, this leads to another problem: If two chains share a scan-in or scan-out port, they should not be used together in another chain, since only one of the port-sharing chains can be active at a time. This would mean that the scan chain would never be able to shift a value from its scan-in port to its scan-out port in test mode. Hence, the chain is not valid, which cannot be the intention. We call such a situation a *resource conflict.*
  It must be checked that this case does not occur. If it does, the routing plan is illegal, and an error is issued.

- **Checking the order of certain chain elements**

  In principle, the order in which chain elements appear within a chain is free. There are however a few exceptions. These exceptions regard elements of which a scan-in port is an already existing port. There are two categories of elements that fall under these exceptions:

  The first category is all combinatorial elements. A combinatorial element will never get a new scan-in port. The specified chain ports must always be existing ports of the element. For example:

```
Macro inv
        ClockDomain none NotDriven
                Chain chn
                        Length 0
                        ScanIn in1
                        ScanOut -out1
                EndChain
        EndClockDomain
EndMacro
```

  is the macro of an inverter. Ports `in1` and `out1` must exist on the element that corresponds with this macro.

The second category is the DFFs that are used in the routing plan. Note that we are talking about DFFs, not SFFs. DFFs can be entered in the routing plan in certain cases. For a detailed description of why and how this is done, the reader is referred to Chapter 9. Below, an example is given of how the macro of a DFF would look like:

```
Macro dff
        ClockDomain clk DrivenBy ( clk )
                Chain chn
                        Length 1
                        ScanIn d
                        ScanOut q
                EndChain
        EndClockDomain
EndMacro
```

The chain of this macro does not have scan-enable port, and the ports d and q are existing ports on the element that corresponds with this macro.

All these elements have existing scan-in ports for the chain ports, and these ports are in use in the design. This means that we cannot just connect these ports to other ports, since they might already be driven. If we want to enter one of these elements in the routing plan, the element must therefore already be connected correctly, i.e., the scan-in port of the element must be connected to the scan-out port of the previous chain element.
Stated in other words this means that these chain elements must be mentioned in the correct order. They have a fixed position relative to their surrounding elements. In figure 8.4 this is explained.

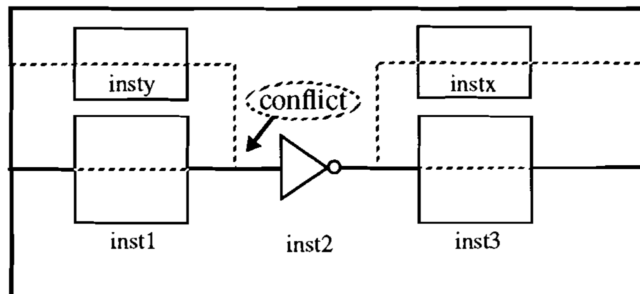


Figure 8.4: Example of fixed chain element order

In this figure, the solid line indicates a scan chain, going through the elements inst1, inst2 and inst3, via existing nets. Chain element inst2 is a combinatorial element (an inverter), and must thus be specified directly after inst1 in the definition of the chain. It is not possible to specify a scan chain through insty, inst2 and inst3, because this would introduce a conflict (two net writers at the same net) at the scan-in port of the inverter.
Note that, although the scan-out port of inst2 is also an existing port, it is not mandatory to specify inst3 directly after inst2 in the definition of this chain, because we can always use the value at the output of inst2 as input to another element (this is shown with instx). This does not introduce any conflict (two net readers at the same net is allowed).

- **Checking the usage of chain elements**

 © Philips Electronics N.V.

It might happen that, when the user changed the routing plan manually, some chain elements within an instance are not used, while others are. This can be done intentionally, but the user could also have made a mistake. Therefore, it is checked if this situation occurs. If it does, a warning is issued. The program will however continue.

- **Checking the clock domains**

  One of the important things we have to do is the clock domain analysis. The user could have easily put elements in the wrong clock domain if he edited the routing plan. This must be detected, because in this case the scan chain is invalid. Therefore, we have to perform a clock domain analysis. This analysis is slightly easier than the one performed by PrepScan, since now we do not have to start at the lowest level in the hierarchy with a recursive algorithm. We can do the analysis on a per chain element basis. Below, the algorithm is given that performs this analysis. *ce* Is the chain element that is currently under consideration, *ch* is the chain it is contained in.

```
CheckClockDomain( ce, ch )
{
    forall cp ∈ chain(ce).C
    {
        Let polarity = cp.POL
        find the port reference pr on inst( ce ) that corresponds with cp
        dp = GetDrivingPort( pr, inv )  *
        if ( inv = true )
            polarity = ¬ polarity
        if ( ¬∃ p ∈ (ch.C) : (dp = p.P ∧ p.POL = polarity) )
            exit error
    }
}
```

* *GetDrivingPort(pr, inv)* has already been explained in section 7.3.2.

- **Testing the number of chain ports**

  InScan sets certain restrictions on the number of chain ports allowed. There should be exactly one scan-in and one scan-out port, and at most one scan-enable port per chain. These restrictions must be checked, since the routing plan language does not impose them.

- **Checking the chain port consistency**

  Chain ports can be shared between chains. We already have seen how this was possible for scan-in and scan-out ports, but of course also scan-enable ports will generally be shared between chains. When certain chains in a macro share chain ports, we have to check that, when these chains are used as chain elements in other chains, these other chains do not introduce conflicts.

  For example, if two chains 'ci1' and 'ci2' of a macro share the scan-enable port, and these two chains are instantiated as chain elements in two other chains 'c1' and 'c2', then these two other chains should both share their scan-enable port. This is necessary because the

instantiated chains already share their scan-enable ports, which means that the scan-enable ports of 'c1' and 'c2' will be connected (because the scan-enable port of a chain will be connected with all the scan-enable ports of its chain elements). If these scan-enable ports are not one and the same, than a conflict occurs when the values on these ports are not equal.

The same reasoning applies to chains with shared scan-in ports, with the exception that these ports only have to be shared when the sharing chains are instantiated as first chain elements in other chains. Chains with shared scan-out ports are only observed if they are instantiated as last chain elements in other chains. Although shared scan-out ports do not introduce conflicts, they can produce unnecessary ports, and therefore are also observed.

8.4 Executing the routing plan

Now that we can be sure the plans are correct, we can execute them. Execution of the routing plan is rather straightforward. Below, the actions that are undertaken, are given:

1. Create the new declaration cells.

   For every macro that has its name not equal to its 'DerivedFrom'/'ReplacedFor' name, a new declaration cell is created with the name of the macro. The contents of this new cell is copied from the original cell (the cell that corresponds with the 'DerivedFrom' name).

2. Redirect certain references.

   The newly created (declaration) cells are still not referenced by any instances (instantiation cells). Therefore, the references of some of the instances must be redirected to these new cells. Which instances must have their references redirected can be found in the routing plan. If a chain element refers to the chain of a macro that corresponds with a just created cell, the instance that corresponds with this chain element must have its references redirected to this new cell.

3. Create the chain ports on all cells.

   When all cells, corresponding with macros, do exist, we can continue with creating the chain ports on these cells. Each chain that is defined for a cell will need some chain ports. It is possible that certain chain ports already exist on cells. In this case we must make sure that it is allowed to use these existing ports. They could be chosen by PrepScan, in which case they are valid, but they could also have been edited into the routing plan by the user. Therefore, we have to check existing ports.

   When the chain port does not exist on the correct cell, it can safely be created. When it does exist, we have to check the port.
   When the existing port is a scan-in or scan-enable port (an input port), it is checked if this port is already driven from the outside of the cell. If this is not the case, it is allowed to use the port. If the port is already driven it cannot be used. In this case a port with another name will be created.
   When the existing port is a scan-out port (an output port), it has to be checked that this

© Philips Electronics N.V.

port is not already driven from inside the cell. If this is not the case, the port can be used. If it is already driven, a port with a different name will be created.

As was explained in section 7.5, PrepScan is able to recognise existing ports that can be used as chain ports. These ports will be put in the routing plan and used by ScanIt when the routing plan is executed. ScanIt also has the functionality to detect if existing ports can be used as chain ports. If the routing plan proposes a new chain port while ScanIt detects an existing port which can be used for this purpose, Scanit will warn the user and automatically use the existing port. This might not always be desired by the user. Therefore, a switch is provided to disable this option. When disabled, ScanIt will not try to be smarter than the routing plan, and will just execute it.

4. Move the match rule file to the design.

Although we have created all cells that we'll need, and have updated the references of the instances to the declaration cells, we still have not incorporated the scan functionality in our design. We already have created the cells for the scannable variants in step 1, but the contents of these cells is just the contents of the original cells. These contents must be replaced by the contents of the corresponding cells in the match rule file.

Therefore, the entire contents of the match rule file is copied to the design. This is possible since the match rule file is in fact a design. During this copying, the newly created scannable variants (with a wrong contents) are replaced by their corresponding cells in the match rule file, which have the correct contents. When this step is done, all instances of type DFF that were to be replaced by SFFs according to the routing plan, now refer to a declaration cell of type SFF.

5. Create the not yet existing combinatorial cells.

As has been mentioned before, ScanIt allows combinatorial elements to be entered in a scan chain, even if these elements do not yet exist in the design. Therefore, these elements must be created during execution of the routing plan. That is what is done by this step. The element will not be connected yet.

6. Route the chains

At this point, the entire design is complete, except for the connections that define the scan chains. This routing process has become very easy, since all ports now exist.
For each chain, the scan-enable port must be connected to all scan-enable ports of its chain elements. This means that in the design the corresponding scan-enable port must be found on the declaration cell, and connected to the scan-enable ports on the instances that correspond with the chain elements.
The scan-in port of each chain must be connected to the scan-in port of its first chain element, and the scan-out port of each chain must be connected to the scan-out port of its last chain element. Any other scan-out port of a chain element must be connected to the scan-in port of its successor.

7. Flatten the substituted cells.

As the reader may have noted, the substitution of DFFs to SFFs has not been done directly. An intermediate level was introduced, which we called the *scannable variant*. The scannable variant could consist of just a single SFF, but also of a DFF with a multiplexer for example. It might happen that the user does not appreciate the new level in the design. This is especially the case when the design that we are processing is already flattened, i.e., has no hierarchy anymore. Therefore, ScanIt has the possibility to flatten this level, which results in removal of this level. By default, ScanIt will only flatten this level if the substitution of DFF to SFF is one to one, i.e., each DFF will be substituted by an SFF existing of a single element (a 'real' SFF, not a DFF with multiplexer). If this is not the case, no flattening will occur. Switches are provided to (de-)activate flattening. It has to be kept in mind that flattening does change the instance names used in the design. If the instance names must be retained, this method of flattening should not be used.

© Philips Electronics N.V.

9 Optimisation techniques

9.1 Introduction

So far, we have discussed the implementation of InScan that performs *full scan*, which means that all DFFs will be replaced by SFFs. In Chapter 3 we discussed the advantages and disadvantages of full scan. 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. Below, a summary is given of this impact [Bennets 93].

disadvantages of partial scan, compared to full scan, are:

- The benefit of guaranteed automatic and algorithmic test-pattern generation is reduced. Now, a pattern-generator for complex sequential circuits is needed, which is both expensive to buy and to run. Also, a way of selecting the memory elements to be replaced, is needed. This is not a trivial problem.
- Fault simulation costs will increase, because the fault simulator must be run on the full sequential circuit. This requires a more sophisticated fault simulator and will cost more in run time.
- The ability to use the scan path for design debug is reduced. There is less scan path access, which results in lower controllability and observability inside the circuit.
- The design environment becomes less manageable. The designer gets more freedom (to introduce timing problems).
- The reduced partitioning through the scan path will reduce the ability to locate faults in order to fix products or processes.

The advantages of partial scan, compared to full scan, are:

- The reduction in replacing DFFs with SFFs results in a reduction of the silicon area needed, and possibly in less IC-pins. This reduces the costs involved with full scan.
- The impact on timing, possibly resulting in an impact on performance, with full scan, can be avoided by basing partial scan on excluding the critical elements from the scan path, thus restoring the original timing behaviour.

As can be seen, partial scan has an impact on all the arguments for and against full scan, as mentioned in Chapter 3. The designer must decide if it is advantageous to use partial scan instead of full scan. In general, when silicon area is critical, partial scan will be useful for the designer.

© Philips Electronics N.V.

Although we now know what partial scan is concerned with, we still have not discussed how it can be implemented. There are a lot of ways to perform partial scan. 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. Each approach has a different impact on the arguments for and against partial scan. Therefore, it is important for the designer to know which arguments are the most important to him, and which partial scan methods do result in the best performance regarding these arguments.

With InScan, we are particularly interested in partial scan methods with which sequential pattern generation is not necessary. Since this is probably the most severe drawback of partial scan, not requesting it is a major advantage over other methods. The other arguments will also be influenced positively with these methods. Below, we will discuss some partial scan methods that are, or will be, implemented in InScan. We will indicate how they work, and discuss their impact on the arguments given above.

Another subject that we have not touched yet, is the order in which chain elements are routed. This order can be of considerable importance. We will see that the partial scan algorithms that are discussed below are based on the positions of the DFFs in the design. There are however also other kind of optimisations possible that optimise the scan chain routing with respect to certain targets. We will discuss two of them at the end of this chapter.

9.2 Shift register recognition

A scan chain is composed of a number of SFFs that form a shift register, when put in test mode. When a design is processed with full scan, all DFFs will be replaced by SFFs, regardless of the functionality of the design. It is however possible to make an easy profit by recognising functional shift registers in the design.

9.2.1 Theory

When full scan is performed on a shift register, all the DFFs the shift register is composed of will be replaced by SFFs, which is however not necessary! The replacement is only done to ensure that the flip-flops form a shift register in test mode, but since an existing shift register already IS a shift register, this replacement serves no purpose anymore. Therefore, this replacement does not have to be done, which results in less silicon area. The entire shift register can be included in a scan path as a part of it. To be able to shift a value into this part, only the first element of the existing shift register must be made scannable. The other elements do not have to be replaced by SFFs. In figure 9.1 this is depicted. For convenience, only the scan-in and scan-out nets are drawn.

The example design consists of a number of DFFs and some combinatorial logic. One shift register already exists in the design. In the figure is shown how the existing shift register is integrated in a scan path. The first element must become scannable and is thus replaced by an SFF. The other elements of the shift register are not replaced.

© Philips Electronics N.V.

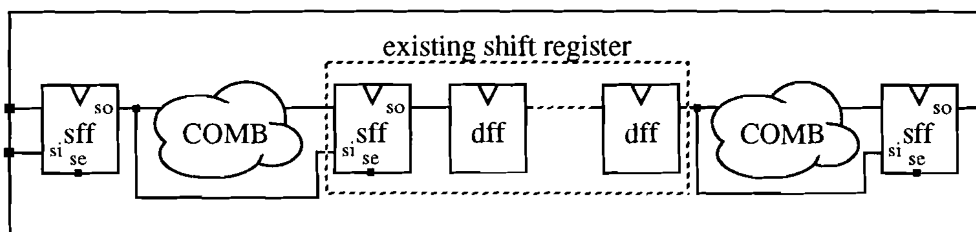


Figure 9.1: Example of integrating an existing shift register in a scan path

Let there be S shift registers in a design, with s_i being the ith shift register of the design, let n_i be the length of shift register s_i, let N be the total number of flip-flops used in the design, let N_s be the total number of flip-flops used in shift registers, and let A be the amount of silicon needed for a DFF:

Of course, $N_s = \sum_i^S n_i$, and $N \geq N_s$

Each SFF results in a certain area overhead α (we will assume that only one type of DFF to SFF replacement is needed), such that the amount of silicon needed for a SFF equals $(1+\alpha)A$. Therefore, the silicon area needed for applying full scan to this design totals $(1+\alpha)A.N$.

If we apply the shift register recognition technique, the silicon area needed totals $(1+\alpha)A(N-N_S+S)+A(N_S-S)$, which can also be written as $A(N(1+\alpha)-\alpha(N_S-S))$. If we divide these two figures by each other, we get the following result:

$$1-\frac{\alpha(N_S-S)}{N(1+\alpha)},$$

in which the factor $\frac{\alpha(N_S-S)}{N(1+\alpha)}$ denotes the gain that shift register recognition yields regarding silicon area of flip-flops. Note that this factor does not take into account the silicon area used by the combinatorial logic. For example, consider a design with $\alpha = 0.2$, $N = 1000$, $N_S = 200$, and $S = 10$, then the gain would be 3.2%. This means that the flip-flops would require 3.2% less silicon area when shift register recognition is used with this design.

9.2.2 Implementation

Below, the algorithm is given that performs shift register recognition.

```
RecogShiftRegs( routingplan )
{
        forall non-leaf macros mc in the routingplan
        {
                forall cd ∈ mc
                {
                        forall ch ∈ cd
                        {
```

```
                    let NoShiftRegFound = false
                    while ( NoShiftRegFound = false )
                    {
                        ce = FindShiftRegElement( ch )
                        if ( ce  =  ∅ )
                            NoShiftRegFound = true
                        else
                        {
                            let regch = CreateChainWithLink( cd, ch )
                            MoveElementToChainAsFirst( ce, regch )
                            ExtendShiftRegAtBegin( ce, regch )
                            DeleteFirstElementsIfComb( regch )
                        }
                    }
                }
                forall chain ∈ cd
                {
                    if ( ChainContainsShiftReg( chain ) )
                        TransformShiftRegElements( chain, cd, routingplan )
                }
            }
            MoveShiftRegsToOriginalChain( )
        }
    }
```

Below, the sub-routines used by this procedure are explained.

DeleteFirstElementsIfComb(chain) deletes all combinatorial chain elements *chain* starts with. These elements serve no purpose in the chain, but were created to ease implementation. Therefore they can now be deleted.

ChainContainsShiftReg(chain) returns *true* if *chain* is created by this routine and thus contains a shift register.

MoveShiftRegsToOriginalChain( ) moves the contents of all the chains that were just created, and thus contained scan chains consisting of a shift register, to the chains they were originally contained in. The shift registers will be appended at the end of the chain element list of those chains.

```
FindShiftRegElement( chain )
// Returns an element that is part of a shift register. This element is found by determining
// if it is of type DFF, and is connected to another DFF via one of its data input ports. For this
// other DFF a chain element must exist.
{
    forall ce ∈ chain
        let inst = inst( ce )
```


© Philips Electronics N.V.

```
        if ( IsDFF( cell( inst ) ) )
        {
            let si = {pr ∈ PORTREF_cell(inst) : pr.I = inst ∧ IsDataInput(pr.P) }
            forall siel ∈ si
            {
                let opr = GetDrivingPortRefThroughBufAndInv( siel )
                if ( IsDFF( cell( opr.I ) ) ∧ (opr.I ≠ inst) ∧
                    ( ∃ chel ∈ chain.EL : inst(chel) = opr.I
                      ∨
                    ( ∃ ch : HasLink(ch, chain) ∧
                        ∃ chel ∈ ch.EL : inst(chel) = opr.I ∧
                        IsLastElement( chel ) )
                    )
                )
                    return ( ce )
            }
        }
    }
    return ( NoElement )
}
```

IsDFF(cell) returns a boolean value that denotes whether cell is a D-type flip-flop. This will be true if *cell* is a leaf cell and the properties attached to it identify it as such a flip-flop.

IsDataInput(port) returns a boolean value that denotes whether *port* is a data input port of the cell it belongs to. This can be determined with the "class" property.

GetDrivingPortRefThroughBufAndInv(portref) traces *portref.P* (which is an input port), via the external net connected to it, through any buffers and/or inverters to the output port of another instance. The portref corresponding with this "driving" port will be returned. Note that there can only be one driving port.

CreateChainWithLink(clockdomain, chain) creates a new chain in *clockdomain* and creates a link to *chain*. The new chain is returned.

HasLink(linkchain, chain) returns true if *linkchain* has a link to *chain*.

MoveElementToChainAsFirst(chainelement, shiftregisterchain) moves *chainelement* from the chain it is contained in to *shiftregisterchain*, where *chainelement* becomes the first chain element.

```
ExtendShiftRegAtBegin( chainelement, shiftregchain )
// chainelement is an arbitrary element of a shift register. This routine extends this
// shift register at its beginning.
{
    let inst = inst( chainelement )
    let si = {pr ∈ PORTREF_cell(inst) : pr.I = inst ∧ IsDataInput(pr.P) }
```

```
        let siel ∈ si
        let opr = GetDrivingPortRef( siel )
        if ( IsDFF( cell( opr.I ) ) )
        {
            if ( ∃ chel ∈ chain.EL : inst(chel) = opr.I )
            {
                MoveElementToChainAsFirst( chel, shiftregchain )
                ExtendShiftRegAtBegin( chel, shiftregchain )
            }
            else
            if ( ∃ ch : HasLink(ch, chain)  ∧
                ∃ chel ∈ ch.EL : inst(chel) = opr.I  ∧
                IsLastElement( chel ) )
            {
                MoveChainToChain( shiftregchain, ch )
            }
        }
        else
        if ( IsBuf( cell( opr.I ) )  ∨  IsInv( cell( opr.I ) ) )
        {
            CreateCombMacro( cell( opr.I ) )
            chel = CreateCombElement( shiftregchain, opr.I )
            ExtendShiftRegAtBegin( chel, shiftregchain )
        }
}
```

GetDrivingPortRef(portref) traces *portref.P* (which is an input port), via the external net connected to it, to the output port of another instance. The portref corresponding with this "driving" port will be returned. Note that there can only be one driving port.

IsLastElement(chainelement) returns *true* if *chainelement* is the last chain element of the chain it is contained in.

MoveChainToChain(fromchain, tochain) moves *fromchain* to *tochain*, and deletes *fromchain* afterwards.

CreateCombMacro(cell) creates a macro corresponding with *cell*. *cell* must be a combinatorial cell.

CreateCombElement(chain, inst) creates a chain element in *chain*, that corresponds with *inst*. *inst* must be a combinatorial element. The chain element is returned.

IsBuf(cell) returns true if *cell* is of type buffer.

IsInv(cell) returns true if *cell* is of type inverter.

 © Philips Electronics N.V.

```
TransformShiftRegElements( chain, clockdomain, routingplan )
// Chain contains all chain elements that constitute a shift register. The chain elements
// corresponding with memory elements are however still referring to macros of SFFs. This
// routine transforms this chain so that the chain elements refer to macros of DFFs, except for
// the first element, which must remain scannable.
{
    tempch = CreateChain( clockdomain )
    let ce = GetLastElement( chain )
    let prevcell = ∅
    while ( ce ≠ ∅ )
    {
        let dbh = cell( inst( ce ) )
        if ( IsBuf( cell )  ∨  IsInv( cell ) )
            MoveElementToChainAsFirst( ce, tempch )
        else
        {
            if ( ce = GetFirstElement( chain ) )
                macro = FindMacro( routingplan, GetName( GetMacro( ce ) ) )
            else
                macro =
                FindMacro( routingplan, GetDerivedFromName( GetMacro( ce ) ) )
            if ( macro = ∅ )
                macro = CreateDffMacro( ce )
            let refChain = ∅
            if ( prevcell = ∅ )
                refChain = GetFirstChain( macro )
            else
                refChain = GetRefChain( prevcell, ce, macro )
            CreateElement( tempch, ce, refChain )
            DeleteElement( ce )
        }
        prevcell = cell( inst( ce ) )
        ce = GetLastElement( chain )
    }
    MoveChainToChain( tempch, chain )
}
```

CreateChain(clockdomain) creates an empty chain in *clockdomain.*

FindMacro(routingplan, name) returns the macro with name *name* in *routingplan.*

GetFirstChain(macro) returns the first chain in *macro.*

GetRefChain(cell, chainelement, macro) returns the chain of *macro,* of which the scan-out port of *inst(chainelement)* can be traced, starting from the data input port of *cell.*

CreateElement(chain, chainelement, refChain) creates a new chain element at the beginning of *chain*, referring to the instance *inst(chainelement)*, and derived from the chain *refChain.*

DeleteElement(chainelement) deletes *chainelement* and removes it from the chain it is contained in.

GetLastElement(chain) returns the last chain element of *chain.*

GetFirstElement(chain) returns the first chain element of *chain.*

CreateDffMacro(chainelement) creates a macro for the cell corresponding with *inst(chainelement).* This macro is special because it indicates how a scan chain can pass through an element of type DFF, instead of the scannable variant of such an element. The chains of this macro do not have any enable ports, and the scan-in and scan-out ports must be existing ports on the DFF. Below, an example is given of such a macro. Refer to section 7.3.2 to compare this macro with the one created for the scannable variant of the DFF.

```
Macro dff
        ClockDomain clk_pos DrivenBy ( clk )
                Chain chn_q
                        Length 1
                        ScanIn d
                        ScanOut q
                EndChain
                Chain chn_qn
                        Length 1
                        ScanIn d
                        ScanOut qn
                EndChain
        EndClockDomain
EndMacro
```

9.2.3 Conclusion

The method discussed above can be seen as an intermediate form between full scan and partial scan. Although all flip-flop are still used in the scan paths, they are not all scannable. Therefore, this method benefits from both the advantages of full scan and partial scan.
Since all flip-flops are still used in the scan paths, there is no need for a sequential test pattern generator. Also, the controllability and observability remain untouched because of the same reason. The advantage is the lessened need for silicon area, and possibly less impact on timing.
Clearly, the disadvantages of using partial scan are not present with this method, while the advantages are. Therefore, there is absolutely no reason not to use this method, since it will cost nothing.

9.3 Pipeline recognition

A second method we would like to discuss is the pipeline recognition method. It will appear that not all DFFs in a pipeline will have to be replaced by SFFs. These flip-flops will also not have to be put in the scan chains, which means that this method can be seen as a partial scan method. On the other hand however, we will see that the we are able to avoid some of the problems that other partial scan methods introduce, which makes this method clearly distinct from them.

9.3.1 Theory

We have seen how shift registers did not have to become entirely scannable. This was possible because of the inherent scannable nature of a shift register. Pipelines also have a specific architecture, that makes them candidates for scan path optimisation. In figure 9.2, an example of a pipeline is given.

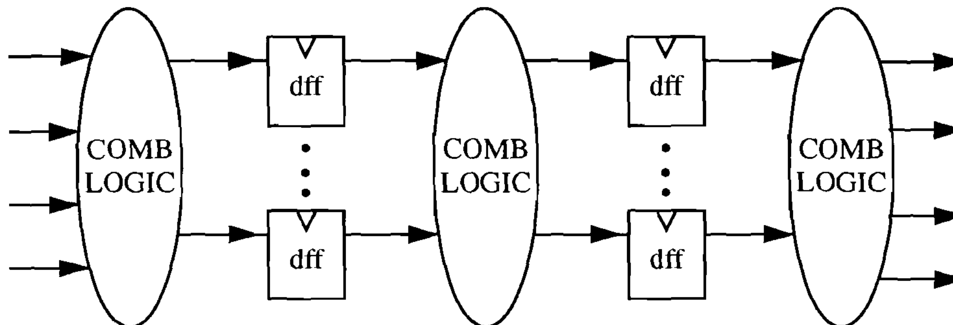


Figure 9.2: Example of a pipeline

None of the DFFs 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 9.2 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 9.2. Because of the nature of the pipeline, these values will remain stable as long as the input values are stable.

The DFFs a pipeline is constituted of, do not have to be entered in a scan chain. This is the main difference with the shift register recognition algorithm, where the DFFs did have to be entered in a scan chain. As a result of this, the logic to be tested between the scan chains is not fully combinatorial anymore, which makes test pattern generation more difficult. This disadvantage is not that great, since the logic does not contain feedback, so that advanced test pattern generation algorithms will not have a hard time creating test patterns for it.

Also, testing is now not trivial anymore due to the sequential nature of the logic to be tested. In general, more than one clock cycle must be applied to the logic 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 SFF. This means that a *test protocol* is needed, in which is described how many clock cycles are needed for signal propagation. Such a test protocol must be created by a test pattern generator or by another tool, such as InScan, when this method is implemented in it.

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.

The extra delay, introduced by using SFFs instead of DFFs, can now be avoided in the pipeline. This is especially useful, since pipelines are often used to speed up the design. 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.

Since less flip-flops have to be replaced by their scannable variants, this method reduces the amount of silicon area needed, compared to full scan. Below, we will give a derivation of the gain that this method yields:

Let N be the total number of flip-flops used in the design, let N_p be the total number of flip-flops used in pipelines, and let A be the amount of silicon needed for a DFF:

Of course, $N \geq N_P$

Each SFF results in a certain area overhead α (we will assume that only one type of DFF to SFF replacement is needed), such that the amount of silicon needed for a SFF equals $(1+\alpha)A$. Therefore, the silicon area needed for applying full scan to this design totals $(1+\alpha)A.N$.

If we apply the pipeline recognition technique, the silicon area needed totals $(1+\alpha)A(N-N_P)+A(N_P)$, which can also be written as $A(N(1+\alpha)-\alpha N_P)$. If we divide these two figures by each other, we get the following result:

$$1-\frac{\alpha N_P}{N(1+\alpha)},$$

in which the factor $\frac{\alpha N_P}{N(1+\alpha)}$ denotes the gain that pipeline recognition yields regarding

silicon area of flip-flops. Note that this factor does not take into account the silicon area used by the combinatorial logic. For example, consider a design with $\alpha = 0.2$, $N = 1000$, and $N_P = 200$, then the gain would be 3.3%. This means that the flip-flops would require 3.3% less silicon area when pipeline recognition is used with this design.

© Philips Electronics N.V.

The gain that this method yields is independent of the gain that shift register recognition yields. Therefore, these methods can both be applied to a design, and their gains can be summed.

Test and simulation times will generally be smaller when using this method. This results from the fact that we are able to test multiple combinatorial blocks at the same time. When we look at figure 9.2 again, we see that full scan would result in three combinatorial blocks to be tested separately (in serial). When applying the pipeline recognition method, the entire pipeline is considered a singe block, which means that all three combinatorial parts are tested in parallel. Although there might be more test patterns needed to test this large block, generally less shift actions will be needed to gather the test results, since the scan paths are shorter due to the pipeline flip-flops which are not used in the scan paths.

9.3.2 Special pipeline cases

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 9.3.

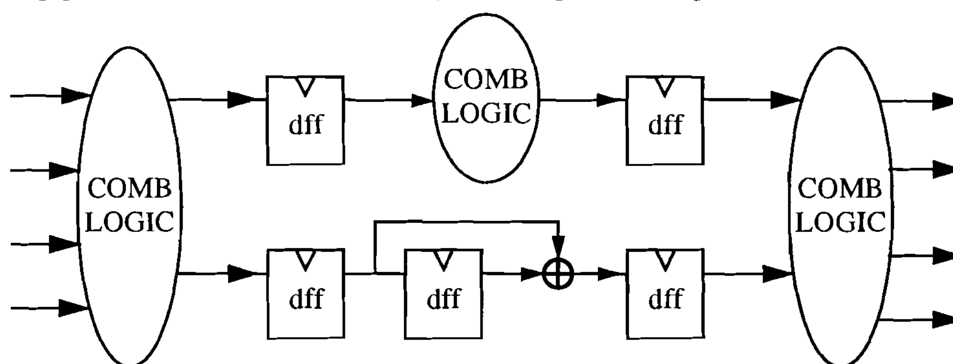


Figure 9.3: Example of a non-symmetrical pipeline

The middle stage of the pipeline contains a DFF 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 DFF. In order to be able to put the DFF 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 DFF must be replaced by an SFF.

another case is depicted in figure 9.4. In this figure, the pipeline again is disturbed by a sequential element of type DFF in one of its stages. This time however, the output of the sequential element is fed back to the sequential circuit. Now, this stage has become a pure sequential circuit, which cannot be easily tested without special care. Therefore, the DFF 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 DFF in a known state and then apply a number of clock cycles, the circuit remains sequential of nature, and we would have

© Philips Electronics N.V.

gained nothing. The state of the DFF 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 DFF does not change states, it can be seen as a "constant generator", thus being a combinatorial block. Therefore, the DFF must be replaced by a scannable hold flip-flop. The hold functionality can then be used to freeze the state of the DFF.

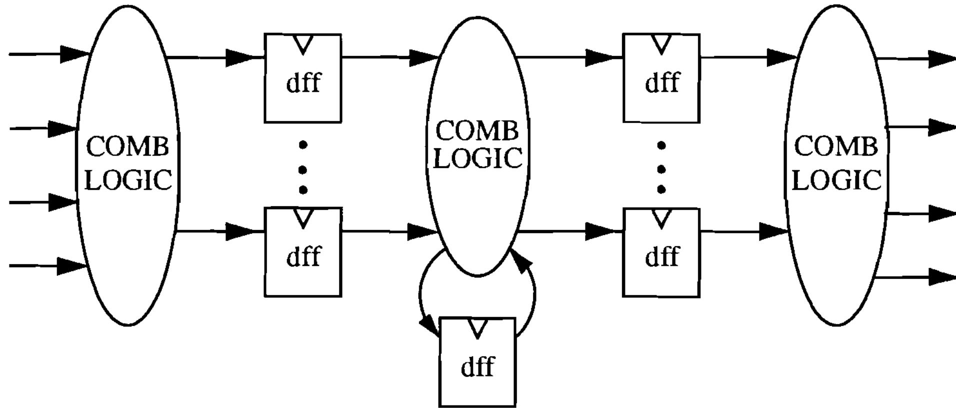


Figure 9.4: Example of a pipeline with sequential loop

9.3.3 Implementation

The pipeline recognition algorithm has not been implemented in InScan yet. It is supposed to be implemented in the near future, and as soon as it has been implemented, the new version of InScan will be named SmartScan. This name already implies that the algorithm does not perform an entirely trivial task. The main problem with implementing the pipeline recognition algorithm is how to recognise a pipeline.

For this problem, it is intended to use the retimer toolkit. The retimer toolkit contains algorithms to modify the temporal behaviour of clocked digital circuits, whilst leaving the functional behaviour unchanged. Although the algorithms are not designed for this purpose, it is possible to use them in such a way that they detect which memory elements form a pipeline.

How these retimer algorithms must be integrated in InScan is still a subject that requires investigation. One of the problems that should be addressed is the question how to recognise the special cases discussed before.

InScan already has been prepared for the implementation of the pipeline recognition method. The method requires scannable hold flip-flops to become part of the scan chain, which InScan will be able to create and recognise. This is explained in more detail in section 7.6.3.

9.4 Timing driven scan chain routing

The partial scan algorithms treated before are based on minimising the number of DFFs that must be replaced by SFFs, in order to minimise the silicon area needed and as a side effect minimise the delays introduced by scan test. We can also approach the routing plan optimisation from other angles, of which we will discuss two in this and the next section.

 © Philips Electronics N.V.

We have seen that PrepScan created the chain elements in a rather arbitrary order. No optimisations were done in this process. The shift register recognition algorithm changed this order when it found a shift register, so that it could perform its optimisations. There are however more ways to change the order of chain elements, in order to get an optimised scan chain routing. This type of optimisation does not result in less scan chain elements, but in other advantages, as we will see. One of these type of optimisation methods is *timing driven scan chain routing.*

An important action during scan test is the transport of test patterns via the scan chains to the blocks that must be tested, and the transport of the responses back to the boundary. If a block demands a lot of test patterns, we could speed up the testing process if the test patterns could be transported via short chains to and from this block. In figure 9.5 this is shown with an example. The design shown consists of a RAM and a block with combinatorial logic. A RAM is known to demand a huge amount of test vectors. Suppose that the RAM needs ten times as much test vectors as the combinatorial block. Now consider the following two situations:

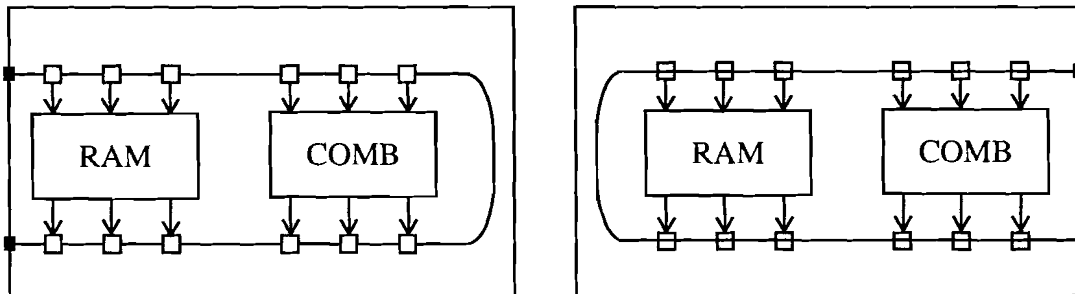


Figure 9.5: Example of scan chain routing with respect to test time

The situations only differ in the routing of the scan chain. On the left hand side, the RAM is closest to the input and output port of the scan chain. On the right hand side, the combinatorial logic is closest to these ports. It is clear that test patterns are faster transported to the block that is closer to the scan-in port. Also, the responses on the test patterns of a block are faster transported to the boundary of the design when the block is closer to the scan-out port of the chain. Therefore, the block that demands the most test patterns should be as close to these ports as possible, in order to optimise the test time. Hence, the situation depicted on the left hand side of figure 9.5 will result in the shortest test time.

This type of optimisation can be done by changing the order of chain elements in the routing plan. The only information needed for it is the amount of test patterns needed for each of the blocks used in the design.

9.5 Layout driven scan chain routing

Another way to optimise the scan chain routing is on the basis of the layout. When a rudimentary cell-placement has been performed before routing, the routing process could use this placement information to find an optimal routing. Of course, it then is a prerequisite that the final layout maintains the original placing as much as possible. We will denote methods, based on layout information, with the term *layout driven scan chain routing*.

It is clear that if routing is performed on the basis of distance between element to be routed, we are able to minimise the wiring needed for the scan chain. This results in less silicon area needed and simplifies the routing process of the entire chip. Also, we are able to decide if buffering is needed in the scan chains. Of course, the shorter the wires between scan chains are, the less quicker buffering is necessary.

Another decision that must be made during the routing is the choice of scan-out ports. As we have seen, if a DFF has two output ports (one the inverse of the other), both ports are valid scan-out ports. Which one to choose when routing the scan chain through this element, can for example be based on the load, present on both outputs. If one of the ports already is heavily loaded, it might be wise to take the other port, to prevent driving problems and avoid the need for buffering.

 © Philips Electronics N.V.

10 Conclusions

In this report we described the specification and implementation of InScan. InScan is a software tool with which an IC design can be prepared for scan test. During operation of InScan, two plans are created that determine the actions InScan should perform. These plans are called the routing plan and the match rule file. To make the program as flexible as possible, InScan is set up in such a way that the user can inspect these plans and make changes to them, if he should find this necessary. This is implemented by breaking InScan into two parts, of which the first is called PrepScan, and the second is called ScanIt.

PrepScan is only concerned with generating the routing plan and the match rule file. For this purpose, it performs an extensive analysis of the circuit. It is shown that the creation of the routing plan mainly involves an analysis of the clock domains. During the clock domain analysis, the routing plan can be built up. The initial routing plan, created by PrepScan, prepares the design for full scan. After creation of this routing plan, it can be processed by dedicated algorithms to optimise certain factors such as silicon area needed, delays, test time, etc.
When the final routing plan is available, the match rule file can be created. A smart algorithm will try to find a scannable variant in the library for each DFF that must be made scannable. If it cannot find one, it will try to create one by putting a multiplexer in front of the DFF.
When the plans are created, they are written out in a readable format, enabling the user to edit them. The match rule file is described in the NDL format, which is a netlist description language. The routing plan is described with the routing plan language, which is especially designed to describe scan chains for a design.

ScanIt is concerned with execution of the routing plan and the match rule file. Since the user may have edited these files, an extensive number of checks is performed to be sure the plans are correct. When they are, ScanIt will make the changes to the design, just as is indicated in the plans.

Finally, this report discusses some algorithms that are or can be inserted in PrepScan, to optimise the routing plan. Especially the shift register recognition algorithm and the pipeline recognition algorithm are treated. These algorithms are interesting since they result in interesting gains regarding the silicon area needed and delays, while the costs that are introduced by using them are minimal.
The shift register recognition algorithm is treated in detail. It has been implemented in the current version of InScan. It results in a gain in the silicon area needed, since not all DFFs in a shift register have to be replaced by SFFs. On the other hand, no extra costs are introduced by using this method.
The pipeline recognition algorithm has not been implemented yet. This report describes what it should do, and how it can be done. The method results in a gain in the silicon area needed, while also extra delays, due to the replacement of DFFs by SFFs, are avoided in the pipelines, since the DFFs in a pipeline do not have to be replaced by SFFs. Also, test and simulation times will generally be smaller using this method. A small disadvantage is that the test pattern generation has become more difficult. Also, the test protocol is not trivial anymore, since with this method not all DFFs are used in the scan paths anymore.

A Mathematical notation

This appendix describes the mathematical notations used in this report. Many readers will already be familiar with most of these notations, but some notations are defined here.

A.1 Abbreviations

iff: *iff* stands for "if and only if".

A.2 Sets and tuples

Set: A *set* S is a collection of zero or more entities, all of the same type. For example:

$$S = \{1, 2, 4, 100, 355\}$$

is a set of natural numbers.

Powerset: The *powerset* of a set S (denoted by $\mathcal{P}(S)$) is the set that contains all subsets of S, including $\varnothing$ and S itself:

$$\mathcal{P}(S) = \{T | T \subseteq S\}$$

Tuple: A *tuple* is a finite, non-empty, ordered set. It is denoted by

$$(el_1, el_2, el_3, \ldots, el_n)$$

A.3 Predefined sets

Set of natural numbers: $\mathbb{N}$ denotes the *set of natural numbers*:

$$\mathbb{N} = \{0, 1, 2, 3, \ldots\}$$

Set of positive natural numbers: $\mathbb{N}^+$ denotes the set of *positive natural numbers*:

$$\mathbb{N}^+ = \{1, 2, 3, \ldots\}$$

Set of booleans: $\mathbb{B}$ denotes the *set of booleans*:

$$\mathbb{B} = \{\text{true}, \text{false}\}$$

Set of values: VAL denotes the *set of values*:

$$VAL = \{ 0, 1 \}$$

A.4 Operators

For all: $\forall$ denotes the *for all* operator. It is used to denote the fact that a certain expression holds for all elements of a certain set. For example:

$$\forall n \in \mathbb{N}^+ : n > 0$$

Exists: $\exists$ denotes the *exist* operator. It is used to denote the fact that a certain expression holds for at least one element of a certain set. For example:

$$\exists n \in \mathbb{N}^+ : n < 3$$

Exists n ($n \in \mathbb{N}$): $\exists^n$ ($n \in \mathbb{N}$) denotes the *exists precisely n* operator. It is used to denote the fact that a certain expression holds for precisely n elements of a certain set. For example:

$$\exists^1 n \in \mathbb{N}^+ : 4 < n < 6$$

Cardinality: $|S|$ denotes the cardinality operator. It denotes the number of elements the set S consists of.

Exor: $\oplus$ denotes the exor operator. It performs the logical exor operation on its arguments.

The dot operator, used on a tuple: To refer to an *element of a tuple*, the *dot notation* is used. For example, if T is a tuple of type

$$(a, b, c, d, e)$$

the fourth element of it will be denoted by

$$T.d$$

This means that, if we want to use the dot notation for a tuple, we have to give a (unique) name to all elements of that tuple. We define these names along with the definition of the tuple itself.

The dot operator, used on a set of tuples: We will also define the dot operator when used on a set of tuples. Let T be a tuple type, d an element name of T, and $ST \subseteq \mathcal{P}(T)$ be a set of such tuples. The dot operator used on such a set is defined as

$$ST.d = \bigcup_{t \in ST} t.d$$

The inversion operator, used on the set of values ***VAL*****:** The inversion operator, denoted by a bar above an element of *VAL*, is defined by

$$\bar{0} = 1 \quad , \quad \bar{1} = 0$$

The inversion operator, used on the set of booleans $\mathbb{B}$: The inversion operator, denoted by a bar above an element of $\mathbb{B}$, is defined by

$$\overline{true} = false \quad , \quad \overline{false} = true$$

© Philips Electronics N.V.

B Routing plan syntax

B.1 Introduction

A scan chain routing plan is a description of a possible routing of scan chains in a design. The initial description of the plan is generated by PrepScan, the designer may modify this plan or even replace it by another plan. The plan can be complete but incomplete definitions are also possible.

Note that the routing plan language is restricted to tree-constructs. This means that the description of the scan chain connections is given from the top level design down to library cell level. Per hierarchical level the routing on that particular level is indicated. Hence, it is not possible to directly specify a connection between two scan cells at different levels of hierarchy. If such a connection is required the scanin and scanout terminals have to be propagated through the hierarchy levels such that proper scanin and scanout terminals exist at the various levels of hierarchy.

B.2 Syntax

```
routing_plan  ::=  [+ macro +]
macro         ::=  'MACRO' {macro}name
                   ( [ 'DERIVEDFROM' {derivedmacro}name ] |
                   [ 'REPLACEDFOR' {replacedmacro}name ] )
                   [+ clock_domain +]
                   ( 'ENDMACRO' | 'END' ) [ ';' ]
clock_domain  ::=  'CLOCKDOMAIN' [ {domain}name ]
                   ( [ 'DRIVENBY' {clocks}clock_list ] |
                   [ 'NOTDRIVEN' ] )
                   [+ chain +]
                   ( 'ENDCLOCKDOMAIN' | 'END' ) [ ';' ]
chain         ::=  'CHAIN' [ {chain}name ]
                   [ length ]
                   [ 'SCANIN' {scanin}name ]
                   [ 'SCANOUT' {scanouts}port_list ]
                   [ 'SCANENABLE' {scanenables}port_list ]
                   [ 'NORMENABLE' {normenables}port_list ]
                   [ 'HOLDENABLE' {holdenables}port_list ]
                   [ 'ORDER'
                   order_list
                   ( 'ENDORDER' | 'END' ) [ ';' ] ]
                   ( 'ENDCHAIN' | 'END' ) [ ';' ]
length        ::=  'LENGTH' {length}int
clock_list    ::=  '(' [ - ] {clock_port}name [* [ , ] [ - ] {clock_port}name *] ')'
port_list     ::=  [ - ] {chain_port}name |
                   '(' [ - ] {chain_port}name [* [ , ] [ - ] {chain_port}name *] ')'
order_list    ::=  element [* element *]
```

```
element    ::= {instance}name 'OFTYPE' {decl}name
               [ [ 'CLOCKDOMAIN' {domain}name ]
               'CHAIN' {chain}name ] [ length ]
name       ::= '"' [* anychar *] '"' |
               char [* char | int *]
char       ::= 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' |
               'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' |
               'v' | 'w' | 'x' | 'y' | 'z' | '$' | '_' | '-' | '/' | '\'
anychar    ::= any non-quoted character
int        ::= [0-9]*
```

Comma's, spaces, tabs and newlines are separators.
'*', '!' and '#' till the end of a line is comment.
[X] means that X can occur here at most once.
[* X *] means that X can occur here any number of times.
[+ X +] means that X should occur here at least once.
'TOKEN' means that TOKEN (a string) should appear here.
Although all tokens are printed in upper case, it is allowed to use any case for tokens.
{tag} is a tag that is used in the next section for referencing to the routing plan syntax. No attention should be paid to these right now.

B.3 Semantics

The semantic tags "{...}" have the following meaning:

- {macro} has no semantic meaning, unless it equals {derivedmacro}/{replacedmacro} or there is no {derivedmacro}/{replacedmacro} (which means the same), in which case it refers to the declaration name of the cell in the netlist.
- {derivedmacro} and {replacedmacro} refer to the original ("template") macro for hierarchy expansion.
- {domain} refers to the clock domain. It has only a semantic meaning when used in the rule for 'instance', in which case it must be the name of a clock domain of macro {decl}.
- {scanin} refers to the name of a scan-in port of the cell {macro} in the netlist.
- {clocks} refers to a list of clock port names of the cell {macro} in the netlist.
- {scanouts}, {scanenables}, {normenables} and {holdenables} refer to lists of port names of the respective port types of the cell {macro} in the netlist.
- {clock_port} refers to the name of a clock port.
- {chain_port} refers to the name of a scan-in, scan-out, scan-enable, norm-enable or hold-enable port.
- {length} refers to the length of the scan chain.

© Philips Electronics N.V.

- {chain} has only a semantic meaning when used in the rule for 'instance', in which case it must be the name of a chain of clock domain {domain}.
- {instance} refers to the instance name in the {macro}.
- {decl} refers to the declaration name of the mentioned {instance}.
  Hierarchy is implicitly described by using some {macro} as {instance}.

LENGTH, SCANIN, SCANOUT, SCANENABLE, NORMENABLE and HOLDENABLE may be defined in any order.

B.4 Semantic rules

Below, rules are given to which the routing plan must conform in order to represent a valid routing plan, i.e. a routing plan that corresponds with the design. These rules are an addition to the syntax, in which it is not possible to render such rules. All rules will be checked by ScanIt after reading the routing plan.

B.4.1 Rules regarding uniqueness

- All macros must be unique within the routing plan.
- All clock domains must be unique within each macro.
- All chains must be unique within each clock domain.
- All clock ports must be unique within a clock port list.

B.4.2 Rules regarding order of definition

In general, it can be stated that each entity to which is referred should be defined before this reference, so the rule 'define before use' applies. In particular, the following rules should be adhered to:

- Referred instances must be (pre) declared in the routing plan.
- Referred clock domains must be (pre) declared on the macro in the routing plan.
- Referred chains must be (pre) declared on the macro in the routing plan.

B.4.3 Rule regarding length of chains

- In principle, the {length} of a scan chain as is specified in the routing plan should be the sum of the lengths of its separate chain elements. However, incorrect lengths are only flagged as a warning, and correct values are calculated.

B.4.4 Rules related to the design

The following rules mention the relationships between the routing plan and the design.

- if {macro} equals {derivedmacro} or there is no {derivedmacro} then {macro} should be the name of a declaration cell in the design (or library).
- {derivedmacro} should be the name of a declaration cell in the design (or library).
- The ports that are mentioned in {clocks} must be clock ports in the design.

{instance} would normally be the instance name of a cell in the design that is contained in cell {macro}. This is however not always necessary. E.g. one could define buffers in a scan chain of the routing plan which do not yet exist in the design.

B.4.5 Other rules

- {derivedmacro} and {replacedmacro} refer to the original ("template") macro for hierarchy expansion. The 'REPLACEDFOR' construct should be used when {macro} is a leaf macro (i.e. a macro of which the chains do not have chain elements) which refers to a substitution, e.g. the line 'MACRO dff_sff REPLACEDFOR dff' states that the declaration of flip-flop 'dff' is to be replaced by 'dff_sff'.
  In all other cases, the 'DERIVEDFROM' construct must be chosen, unless {macro} equals {derivedmacro}, in which case the 'DERIVEDFROM' construct is optional. This construct defines a different scan chain constellation.
- A clock domain that has no clock ports must be indicated with the 'NOTDRIVEN' construct. This construct will only be used for defining a scan chain through a combinatorial element which clearly has no clock ports. Multiple clock ports are allowed with the 'DRIVENBY' construct. This means that all the ports that are mentioned in the clock port list {clocks} should be connected at a higher level of the hierarchy (compare the "MustJoin" in Edif). Please note that this construct does not refer to multi-phase clocking.
- The scan ports that are mentioned in each chain are all optional, with the exception that each chain should have at least one scan-in and one scan-out port. There can be only one scan-in port, but the number of other ports is not restricted.
- For each {instance} its type is given. If this type information is adequate for determining the right chain (because this type of macro has only one chain) then the 'CLOCKDOMAIN' and 'CHAIN' constructs are not necessary. If there is only one clock domain but this clock domain contains more than one chain, then only the 'CHAIN' construct is necessary. If there is more than one clock domain, then both 'CLOCKDOMAIN' and 'CHAIN' are necessary.

 © Philips Electronics N.V.

Bibliography

[Beenker 90] Beenker, F.P.M., M. Van Der Star, R.W.C. Dekker, R.J.J. Stans,
"Implementing Macro Test in Silicon Compiler Design",
IEEE Design & Test of Computers, April 1990,
pp. 41-51.

[Bennetts 93] Bennets, R.G., F.P.M. Beenker,
"Partial Scan: what Problem does it Solve?",
Proceedings European Test Conference, Rotterdam, The Netherlands, April 19-22 1993,
pp. 99-106.

[Bennetts 84] Bennets, R.G.,
"Design of testable logic circuits",
Addison-Wesley, 1984.

[Claasen 89] Claasen, T.A.C.M., F.P.M. Beenker, J. Jamieson, R.G. Bennets,
"New Directions in Electronic Test Philosophy, Strategy and Tools",
Proceedings of the 1st European Test Conference , Paris, 1989,
pp. 5-13.

[ED&T Library Format] "EDT Library Format",
EDTH-GEN-UM-004,
Philips Electronic Design & Tools,
P.O. BOX 32,
Hilversum, The Netherlands.

[ED&T Library Norm] "ED&T Library Norm v1.1",
EDTH-GEN-UG-001,
Philips Electronic Design & Tools,
P.O. BOX 32,
Hilversum, The Netherlands.

[EIA 87] EIA,
"Electronic Design Interchange Format version 2 0 0",
Electronic Industries Association,
Engineering Department,
2001 Eye Street,
N.W. Washington, D.C. 20006
1987.

[Fujiwara 85] Fujiwara, H.,
"Logic testing and design for testability",
The MIT Press, 1985.

[Fujiwara 83] Fujiwara, H., T. Shimono,
"On the Acceleration of Test Generation Algorithms",
IEEE Transactions on Computers, Vol. C-32, No. 12, December 1983
pp. 1137-1144.

[Goel 81] Goel, P., B.C. Rosales,
"PODEM-X: An Automatic Test Generation System for VLSI Logic Structures",
Proceedings 18th Design Automation Conference, June 1981,
pp. 260-268.

[NDS] "NDS User Manual",
EDTH-NDS-UM-000,
Philips Electronic Design & Tools,
P.O. BOX 32,
Hilversum, The Netherlands.

[SDS] "SDS User Manual",
EDTE-SDS,
Philips Electronic Design & Tools,
P.O. BOX 80000
Eindhoven, The Netherlands.

[Woudsma 90] Woudsma, R., F.P.M. Beenker, J. van Meerbergen, C. Niessen,
"PIRAMID: and Architecture-Driven Silicon Compiler for Complex
DSP Applications",
Proceedings ISCAS Conference, New Orleans, May 1990.

 © Philips Electronics N.V.

Author: T.A.F.M. van de Voort

Title: **InScan: Specification and implementation of a scan chain inserter**

Copy to:

M.T.M. Segers	WAY-3 Nat.Lab
P.W.M. Merkus	WAY-3 Nat.Lab
J.M.C. Jonkheid	WAY-3 Nat.Lab
K. Kuiper	WAY-3 Nat.Lab
R.J.J. Stans	WAY-3 Nat.Lab
R.J.G. Morren	WAY-3 Nat.Lab
H.A. Bouwmeester	WAY-3 Nat.Lab
S.R. Oostdijk	WAY-3 Nat.Lab
A. van der Veen	WAY-3 Nat.Lab
J. Bakker	WAY-3 Nat.Lab
R.M.C. Lenarts	WAY-3 Nat.Lab
L.A.R. Eerenstein	WAY-3 Nat.Lab
F.G.M. de Jong	WAY-3 Nat.Lab
F. van der Heyden	WAY-3 Nat.Lab
A.S. Biewenga	WAY-3 Nat.Lab
M.N.M. Muris	WAY-3 Nat.Lab
F.G.M. Bouwman	WAY-4 Nat.Lab
A.P. Kostelijk	WAY-4 Nat.Lab
E.J. Marinissen	WAY-4 Nat.Lab
C.R. Wouters	WAY-4 Nat.Lab
F. Hapke	Semiconductors Hamburg