High-Level Built-In Self-Testable Synthesis of Digital Systems

by

Laurence Tianruo Yang
B.Eng and B.Sc, Tsinghua University, 1992
Lic.Eng, Linkoping University, 1996

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

High-Level Built-In Self-Testable Synthesis of Digital Systems

by

Laurence Tianruo Yang

Supervisory Committee

Dr. J. C. Muzio, Supervisor, Department of Computer Science

Dr. D. M. Miller, Department of Computer Science

Dr. D. M. German, Department of Computer Science

Dr. K. F. Li, Department of Electrical and Computer Engineering

---

Dr. M. A.Thornton, External Examiner

Supervisory Committee:

Dr. J. C. Muzio, Supervisor, Department of Computer Science

Dr. D. M. Miller, Department of Computer Science

Dr. D. M. German, Department of Computer Science

Dr. K. F. Li, Department of Electrical and Computer Engineering

Dr. M. A.Thornton, External Examiner

ABSTRACT

Driven by the rapid growth of the Internet, communication technologies, pervasive computing, automobiles, airplanes, wireless and portable consumer electronics, Embedded Systems and Systems-on-Chip (SoC) have moved from a craft to an emerging and very promising discipline in today's electronic industry.

Testing of a fabricated chip is a process that applies a sequence of inputs to the chip and analyzes the chip's output sequence to ascertain whether it functions correctly. As the chip density grows to beyond millions of gates, Embedded Systems and Systems-on-Chip testing becomes a formidable task. Vast amounts of time and money have been invested by the industry just to ensure the high testability of products. On the other hand, as design complexity drastically increases, current gate-level design and test methodology alone can no longer satisfy stringent time-to-market re-

quirements. The High-Level Test Synthesis (HLTS) system, which this thesis mainly focuses on, is to develop new systematic techniques to integrate testability consideration, specially with Built-In Self-Test (BIST) techniques into the synthesis process. It makes it possible for an automatic synthesis tool to predict testability of the synthesized embedded systems or chips accurately in the early stage. It also optimizes the designs in terms of test cost as well as performance and hardware area cost.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Driven by the rapid growth of the Internet, communication technologies, pervasive computing, automobiles, airplanes, wireless and portable consumer electronics, the complexity of VLSI technology has reached the point where we are trying to put more than 100 million transistors on a single chip. Table 1.1 shows the ITRS (International Technology Roadmap for Semiconductor [7]) of the Semiconductor Industries Association. It is anticipated that the number of transistors will be about 4-5 times higher, the clock frequency will be 2 times faster, in 5-6 years. These trends have a profound effect on the costs and difficulty of chip testing.

Testing of a fabricated very large scale integrated (VLSI) circuit is a process that applies a sequence of inputs to the circuit and analyzes the circuit's output sequence to ascertain whether it functions correctly. As the chip density grows beyond millions of gates, and the increase of I/O pin count and clock frequency for test is relatively slower than the increase in transistor count, VLSI circuit testing becomes a formidable task. As an example described in [86], in high performance Application Specific Integrated Circuits (ASICs), complexity per I/O pin is shown in Figure 1.1. This figure shows very well that complexity per pin will increase exponentially. Therefore the test data transferred between a chip and a tester will become a major bottleneck, the test application time will seriously increase and the unit price is likely to become more expensive because of depreciating equipment costs. Vast amounts of time and money have been invested by the semiconductor industry just to ensure the high testability

Year	1997-2001	2003-2006	2009-2012
Feature size μm	0.25-0.15	0.13-0.10	0.07-0.05
Millions of transistors/cm^2	4-10	18-39	84-180
Number of wiring layers	6-7	7-8	8-9
Die size, mm^2	50-385	60-520	70-750
Pin count	100-900	160-1475	260-2690
Clock rate MHz	200-730	530-1100	840-1830
Voltage, V	1.2-2.5	0.9-1.5	0.5-0.9
Power W	1.2-61	2-96	2.8-109

Table 1.1. *VLSI Chips, present and future [7]*

of products. Intel has been reported as indicating that the combination of verification testing and manufacturing testing is its major capital cost, and not its 2 billion dollar silicon fabrication lines. Many system companies consider testing to be 50-60% of their equipment manufacturing cost. However, the most important cost can be the loss in time-to-market due to hard-to-detect faults. Recent studies [82] show that a six-month delay in time-to-market can cut profits by 34%. Thus, testing can pose serious problems in VLSI design.

Part of the reason testing costs so much is the traditional separation of design and testing whose general design flow is depicted in Figure 1.2. Testing is often viewed inaccurately as a process that should start only after the design is complete. Due to this separation, the designer usually has little appreciation of testing requirements, whereas the test engineer has little input into the design process. In order to effectively reduce testing cost, methods which take into account testability of the final product are needed and these are usually called **Test Synthesis** [28, 54]. This approach is motivated by the high complexity of current design and related testing costs. The design test related activities, such as test generation and test application, usually

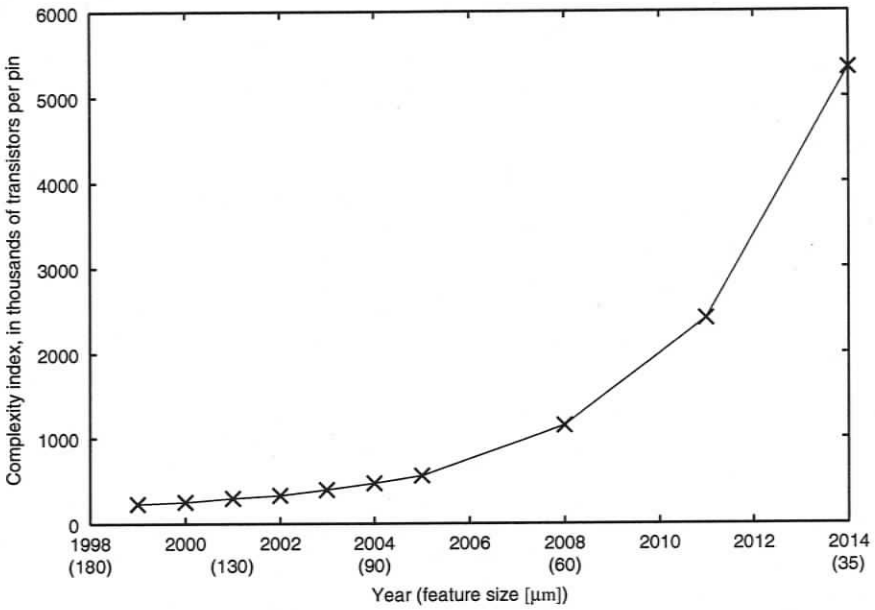


Figure 1.1. *Functional complexity per I/O pin*

have a relatively large share of the total design and test cost. In some cases, this can reach as high as 50% of the total cost. Thus the main idea of **Test Synthesis** is to improve testability of the design during the early stages, and this is expected to reduce the later design testing costs.

On the other hand, as design complexity drastically increases, current gate-level synthesis methodology alone can no longer satisfy the stringent time-to-market requirement. **High-level Synthesis** [15, 30, 57, 59] which takes a behavioral specification of a digital system and a set of design constraints as input and generates a Register-Transfer Level (RTL) hardware implementation is hence considered as a promising technology to boost design quality and shorten the development cycle. The data described in Figure 1.3 [25] demonstrate the advantage of starting the synthesis from the behavioral rather than the register-transfer level. Using a high level synthesis tool, the size of the specification, the gate count, the simulation time, and the number of man-hours can be significantly reduced.

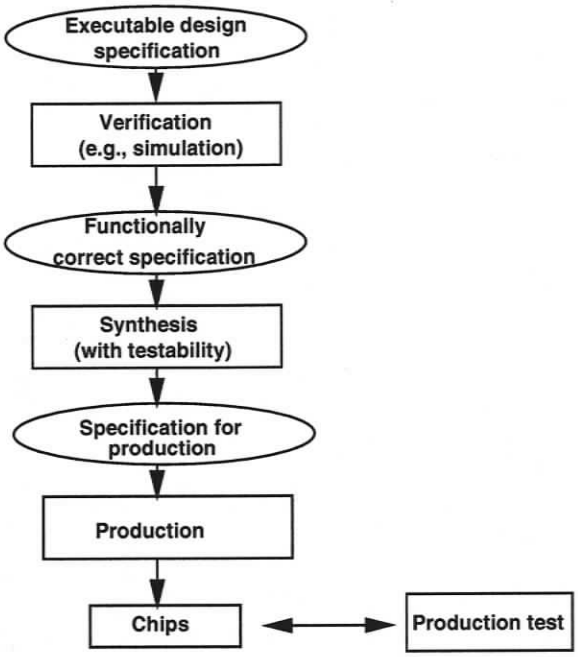


Figure 1.2. *Testing activities in the design process*

In this thesis, our main focus is on **High-level Test Synthesis**, by developing new systematic techniques to integrate testability consideration into the synthesis process. It makes possible for an automatic synthesis tool to predict testability of the synthesized circuits accurately in the early stage and optimize the designs in terms of test cost as well as performance and area cost.

1.1 Thesis Contribution and Organization

Due to the increasing transistors-to-pin ratios shown in Figure 1.1 which significantly limit the feasibility of testing digital circuits externally, this thesis primarily discusses a built-in self-test (BIST) system which allows a circuit to be tested from within the circuit itself and declares itself faulty or fault-free depending upon the outcome of the test. This eliminates the need of deterministic test generation and an external tester, at the expense of some area and delay overhead, and possible low test quality

Metric	Manual RTL approach	Behavioral Compiler approach
Design methodology	RTL VDHL Top-Down	Behavioral Top-Down
Lines of VHDL	23.8 K	4.6 K
Gate count	90 K	~50 K
Throughput	28 cycles/frame	32 cycles/frame
Simulation time	450 mins/frame	19 mins/frame
Man-hours detailed design	3360	1512

Figure 1.3. *Advantage of high level synthesis [25]*

in terms of faults that are covered. The main framework is depicted in Figure 1.4. The thesis is mainly organized based on the flow of such a system. In chapters 1 and 2, we start with the preliminary introduction, and background and related work, respectively.

Design representation and transformations

Our system takes a VHDL behavioral specification of a digital system and a set of design constraints as input and generates a Register-Transfer Level (RTL) hardware implementation which consists of a data path and a controller. The kernel of the system is an intermediate design representation, called Extended Timed Petri Net (ETPN), which can be used both for testability analysis and high-level synthesis [76]. In ETPN, the structural properties of the data path and controller are explicitly captured in order to facilitate accurate analysis of the intermediate design in term of performance, area and testability. We describe such a design representation in chapter 3.

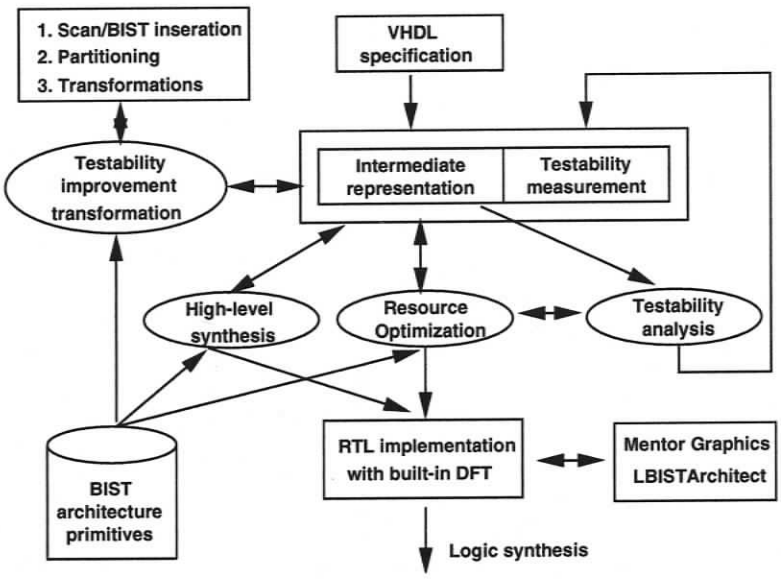


Figure 1.4. *The built-in self-test synthesis system*

Testability analysis

Based on the design representation, we have developed data path testability metrics to evaluate various BIST configurations and make improvement decision [103, 105]. The objective of testability metrics is to analyze and quantify BIST testability for a given register transfer level design. Basically, our BIST testability metrics quantify two important testability aspects, namely controllability and observability. The early decision about testability improvement gives the possibility that designs can be optimized in later synthesis processes. The testability analysis carried out at high-level abstraction also reduces the computational complexity, since the complexity of a design at this level is significantly lower.

Due to the large computational complexity of testability analysis and the need to perform such analysis after each of the synthesis steps, we have applied a similar systematic technique used for Automatic Test Pattern Generation (ATPG) technique for the present BIST technique to approximate accurately the repeated testability calculation and evaluation [103, 105] which is based partially on explicit re-calculation

and partially on gradient techniques for incremental testability to update the test property.

BIST partitioning

Based on the above testability measurements, we develop a new improvement method with BIST techniques. Partitioning for a design can lead to the simplifications of many design procedures such as synthesis and test. Partitioning for testability leads to the simplification of test efforts and the ability to apply different test strategies to different partitions. The proposed partitioning technique described in chapter 5 [103, 105] transforms some hard-to-test registers and/or lines to boundary components. These components act as normal registers and/or lines in the normal mode and serve as partitioning boundaries in test mode or test registers. Therefore, a design is partitioned into several sub-circuits and each of them can be tested and controlled based on BIST test schemes. Therefore, the resulting design is fully self-testable.

Data path allocation

Considering testability issues at high-level synthesis can lead to a more efficient exploration of the design space, thus resulting in a digital circuit that requires minimal BIST area overhead and has high test concurrency while guaranteeing the test quality.

In chapter 6, we describe a high-level data path allocation algorithm [100] which generates highly testable data path designs while maximizing the sharing of modules and test registers. Module allocation is guided by a testability balance principle where *incompletely embedded* modules can be mapped into the same function module that is *completely embedded.* In this way, the *incompletely embedded* module after allocation will be fully testable. The register allocation is mainly based on the sharing degrees of registers which reflect the number of modules for which the register can be configured as Pseudo-Random Pattern Generators (RTPG) and the number of

modules for which it can be configured as a Multiple-Input Signature Register (MISR). Using this measure the register allocation is guided by choosing mergers that result in large increases in the sharing degrees of registers over those resulting in smaller increases. This would result in registers with high sharing degrees, thereby requiring a smaller number of BIST registers globally in the design.

However, the approach in [100] still has some drawbacks, for example, if an *incompletely embedded* module can not find a match to be merged with a *completely embedded* module during the iterative allocation algorithm, it probably will not become fully testable. If there are several such modules un-mapped in the design, the resulting testing quality is not satisfactory. This motivates us to make use of two types of redundant transformations introduced in [68, 71, 73], which add redundancy that improves test resources to be shared in the data path without affecting the scheduling step (latency) and functional resource requirement of the behavior, to improve our data path allocation algorithm and to make all *incompletely embedded* modules become fully testable [107].

Before going to RTL implementation or performing the above proposed high-level data path allocation or synthesis algorithm, a pre-synthesis resource optimization can be applied. It mainly involves modification of the hardware on the chip so that the chip has the capability to test itself. In this allocation algorithm we also describe an optimal modification approach [101] based on Integer Linear Programming formulation [68] to find BIST embeddings in the data path prepared for the allocation or synthesis algorithm or before going to RTL implementation such that the cost of modification is minimal.

Integrated synthesis algorithm

After our system takes a VHDL behavioral specification of a digital system and a set of design constraints as input, the design representation is always unscheduled. Therefore, we need to consider not only operation scheduling but also data path

allocation.

In chapter 7, we describe a high-level test synthesis algorithm for operation scheduling and data path allocation [102, 104, 108, 110, 111]. This generates highly testable data path designs while maximizing the sharing of test registers, which means that only a small number of registers are modified for BIST. The algorithm also produces designs with high test concurrency, thereby decreasing test time. In our approach, module allocation is guided by a testability balance principle where *incompletely embedded* modules can be mapped into the same function module that is *completely embedded.* In this way, the *incompletely embedded* module after allocation will be fully testable. The register allocation is guided by an incremental sharing measurement which chooses merges that result in large increases in the sharing degrees of registers. When two modules are merged, the operations executed on these modules must be scheduled in different control steps so that they can share the same component. Similarly for registers, the variables stored in these registers must be disjoint. We present the rescheduling transformation which is performed by a merge-sort algorithm. These transformations change locally the execution orders of some operations in the current schedule in order to improve the testability and satisfy the scheduling constraints imposed by the merger. Contrary to other works in which the scheduling and allocation tasks are performed independently, our approach integrates scheduling and allocation by performing them simultaneously so that the effects of scheduling and allocation on testability are exploited more effectively.

In [102], we also introduce some concepts and techniques to improve our previous work [104] during the operation scheduling part, specially to determine the execution order of different operations when rescheduling transformations are performed.

However, the above described integrated approaches [102, 104] still have some drawbacks as we described previously, for example, during the module allocation, if an *incompletely embedded* module can not find a match to be merged with a *completely embedded* module during the iterative allocation algorithm. It probably will

not become fully testable. Similarly we make use of two types of redundant transformations introduced in [68, 73], which add redundancy that improves test resources to be shared in the data path without affecting the scheduling step (latency) and functional resource requirement of the behavior, to improve our data path allocation algorithm and to make all *incompletely embedded* modules become fully testable [108, 110, 111]. Also, this can avoid the increase of scheduling steps because one of the operations can be merged with the introduced redundant operations at different scheduling steps.

Testability metrics-based synthesis

In chapter 8, we present a different BIST synthesis methodology, namely a BIST testability metrics-based algorithm for operation scheduling and data path allocation [99, 106, 109, 110, 112]. It is based on the BIST data path testability analysis algorithm at register-transfer level described above. In this approach, module and register allocation are guided by a testability balance technique. In our approach, the selection of nodes to be merged is based on the testability measures generated by the testability analysis algorithm. The main goal is to generate a data path with good controllability and observability for all the nodes and with as few loops as possible. The basic idea is to fold nodes with good controllability and bad observability to nodes with good observability and bad controllability. Note that the controllability of a node is defined as the best controllability of any of its input lines. While the observability of a node is the best observability of any of its output lines. In this way, the new node inherits the good controllability from one of the old nodes and the good observability from the other. The synthesis algorithm introduces scheduling constraints imposed by data path allocation and performs scheduling and allocation simultaneously in an iterative fashion so that their effects on testability are exploited more effectively.

With the help of an incremental BIST testability analysis with its incremental

approach, we mainly make use of some concepts and techniques to improve the above work [99] not only during the data path synthesis part, but also during the operation scheduling part [106].

Similarly redundant transformations have been introduced [109]. We add redundancy that improves test resources to be shared in the data path without affecting the scheduling step (latency) and functional resource requirement of the behavior, to improve our data path allocation algorithm and to make all modules become fully testable. Also this can avoid the increase of scheduling steps because one of the operations can be merged with the introduced redundant operations at different scheduling steps [109, 110, 112].

1.2 Summary

In section 1.1, we highlight the main contribution of this dissertation by addressing testability during high-level synthesis for a built-in self-test (BIST) methodology of testing digital system designs. The background and related work are to be presented in next chapter.

Chapter 2

Background and Related Work

2.1 Testing of Digital Systems

VLSI digital systems are tested by applying a sequence of inputs to the circuit and comparing the output sequence with an expected output sequence. The directly accessible input and output ports are also called *primary inputs/outputs* to distinguish them from component inputs/outputs which are usually not directly accessible for test equipment. Any difference from the expected output sequence is an *error*. The cause of an error is a fault.

2.1.1 Fault model

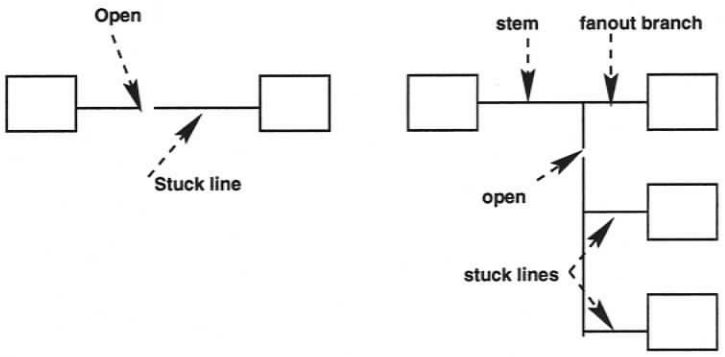


Figure 2.1. *Fault models in hardware testing*

Commonly used models are single stuck-at-fault model, multi struck-at-fault model,

transition fault model, bridging fault model and delay model. The fault model determines how well we can model the real defects in the manufactured products. In general, fault models assume that components are fault-free and only their interconnections are affected. Typically faults affecting interconnections are shorts and opens. In many technologies, the effect of an open on a signal line with only one fanout is to make the input that has become unconnected due to the open assume a constant logic value and hence appear as a stuck-at-fault as shown in Figure 2.1. An open on a signal line with fanout may result in a multiple stuck-at-fault involving a subset of its fanout branches, as illustrated in Figure 2.1. The single stuck-at-fault model is the most commonly used which assumes that when a fault stuck at 1 is represent the line (wire) is always high. On the other hand, when a stuck-at-fault 0 is present the line is directly connected to ground. Others are used as a complement according to design features and requirements.

2.1.2 Test evaluation

An important issue in testing is *test evaluation*, which refers to determining the effectiveness, or quality, of a test.

Test evaluation is done in the context of a fault model, and the quality of a test is measured by the ratio between the number of faults it detects and the total number of faults in the assumed fault universe; this ratio is referred to as *fault coverage*. Test evaluation is carried out via a simulated testing experiment called *fault simulation*, which computes the response of the circuit in the presence of all faults in the test being evaluated. A fault is detected when the response it produces differs from the expected response of the fault-free circuit.

2.1.3 Test generation

Test generation is the process of determining the stimuli necessary to test a digital system. Test generation primarily depends on the testing method employed. On-line testing methods do not require test generation. Little test generation effort is needed when the input patterns are provided by a feedback shift register working as a pseudo-random sequence generator. In contrast, test generation for design verification testing and the development of diagnostic programs involve a large effort that, unfortunately, is still mainly a manual activity. Automatic test generation refers to test generation algorithms that, given a model of a system, can generate tests for it. There are basically two types of test generation, namely *random test generation (RTG)* and *deterministic test generation (DTG)*. The RTG is a simple process that involves only generation of pseudo-random vectors. The DTG produces tests by processing a model of the circuit, which is more difficult, but produces shorter and higher-quality tests. The *automatic test pattern generation* (ATPG) is done by an ATPG tool. It has been proven that the test generation problem is NP-complete even for combinational circuits [3].

In the following, we give a small example to describe how to use test vectors generated to cover the fault. In Figure 2.2(a) we have the gate-level description of the function $z = a$ and b and c. For instance, if we have a fault marked as in Figure 2.2(b) and we want to create a test vector to detect such fault. To detect the fault we must create a test vector that tries to force the line with the fault to 1. That can be achieved by a test vector where $a = 1$ and $b = 1$. To observe the fault we have to propagate it to the output z. In the faulty case where a line is forced to 0 the output will always be 0. In the fault-free case the output can be 1 for a test vector such as $a = 1, b = 1$ and $c = 1$. We have now created a test vector that can distinguish a fault-free design from a faulty.

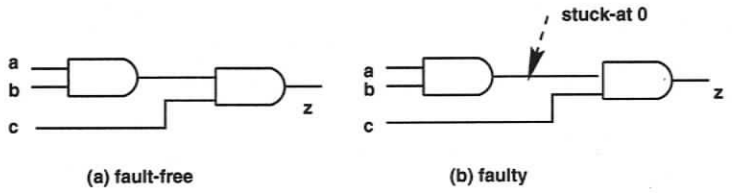


Figure 2.2. *An example of test vector generation*

2.2 Design for Testability

Design-for-testability (DFT) is a collection of techniques, rules and guidelines which are employed to ensure that a circuit is testable. The circuit is testable if the related test vectors can be generated in a reasonable time and the fault coverage is high.

The techniques in design-for-testability can be divided into two categories: ad hoc approaches and structured approaches [3, 4, 5, 84, 92, 98]. The ad hoc techniques are those techniques which can be applied to a given circuit with special properties. They usually do offer relief, and their cost is probably lower than the cost of the structured techniques, but they are not directed at solving general testing problems. The structured approach, on the other hand, aims to solve general testing problems with a systematic design methodology, such that when designs using one of the particular techniques in this category, test generation and fault simulation can be carried out at acceptable costs. The structured approach lends itself more easily to design automation. Within the ad hoc category, there are methods such as adding extra testing points to obtain direct control and observation of some internal points [3, 43, 44], and partitioning design to make test generation and test application easier [37, 43]. Within the structured approaches, there are methods such as multiplexer technique [3], scan design techniques [3, 97] and built-in self-test technique [3, 4, 5, 84].

The basic idea of design-for-testability is to enhance *controllability* and *observability* of selected lines in a design. Controllability indicates the relative difficulty of setting a particular line to a specific value from a primary inputs and observabil-

ity indicates the relative difficulty of propagating a value assigned to a particular line to a primary output. They reflect the two phases of test generation, namely test sensitizing and test propagation. Methods have been developed for estimation of controllability and observability measures at logic level [34] and RTL level [51]. These estimations can be used by test synthesis tools to guide the selection of the best testability enhancement method.

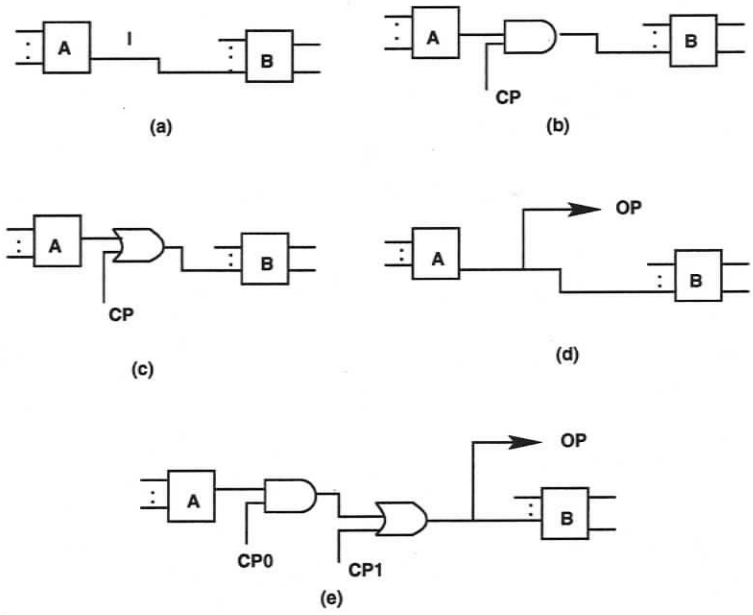


Figure 2.3. *An example of test point techniques*

In the rest of this subsection, several well-known ad hoc and structured methods for testability enhancement are introduced.

2.2.1 Test points

The main idea of the test point techniques is to increase design testability by injecting a test pattern to a line for direct control and/or adding a new observation point on a line for direct observation [3, 21]. Figure 2.3 illustrates the test point technique. Figure 2.3(a) is an original design, where A and B are parts of the design connected

by line l. Figure 2.3(b) illustrates the method of injecting a 0 into line l. The output of the AND gate is connected to the input of B, and one of its inputs is connected to the output of A and the other input, called control point CP, is connected to an input pin. If a 1 is fed into this input, line l works in its normal mode, passing the value in its test mode, that is, a 0 is injected to the line through this input pin. By this means, we can set a value 0 on any line. Similarly, Figure 2.3(c) shows the method of injecting a value of 1 into line l in the test mode. In Figure 2.3(d), we connect the line l to an observation point OP, which is connected to an output pin. By this means, this line becomes directly observable. Figure 2.3(e) describes the method of 0/1 injection and direct observation. $CP0$ and $CP1$ are used as control points. When $CP0 = 1$ and $CP1 = 0$, it works in the normal mode. If $CP0 = 0$ and $CP1 = 0$, a value 0 is injected into the line. If $CP1 = 1$, value 1 is injected into the line. OP is used as an observation point.

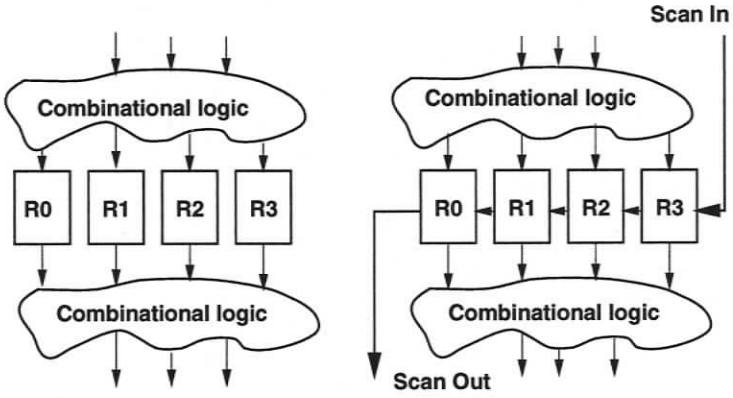


Figure 2.4. *An example of the scan design technique*

2.2.2 Scan design

Scan design is a very popular approach which improves the testability by increasing the controllability and observability of the memory elements of sequential circuits [34, 81, 95, 97]. Motivated by the main problem of test pattern generation for a

sequential logic, setting and observing the state of registers, the scan design technique solves this by reducing the sequential test generation problem to a combinational one. This is done by using scan registers which operate in two basic modes: a normal functional mode and a test mode [98]. In the normal functional mode a scan register acts as a normal register while in the test mode all scan registers are configured into a shift register in order to control as well as observe their states described in Figure 2.4. However, scan design usually introduces a significant area and delay overhead due to the multiplexors in the scan latches and the routing area for the scan chains and the extra inputs. To minimize these overheads, some approaches are based on the concept of *partial scan* where only a few selected latches are scanned [19, 22, 39, 66, 89]. These testability features such as scan are usually incorporated in a post-synthesis step. As the performance and quality requirements of VLSI circuits increase, these DFT methods of improving testability become less attractive due to their unacceptable area and performance overheads.

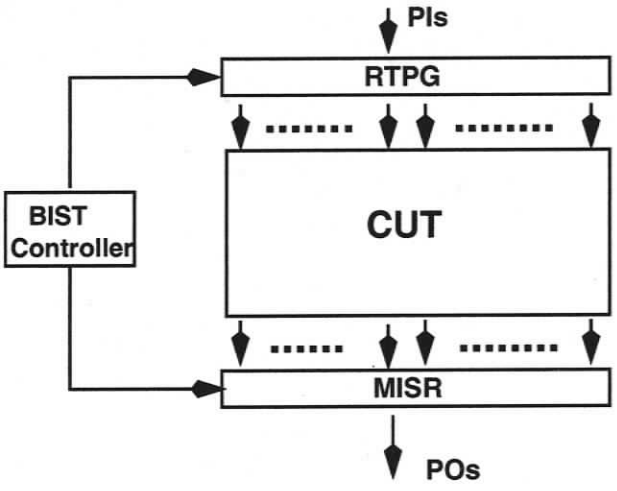


Figure 2.5. *An example of the built-in self-test technique*

2.2.3 Built-in self-test

In a typical BIST implementation described in Figure 2.5, a circuit is tested by an on-chip pseudo-random *test pattern generator* and *signature analyzer* [11]. Additional logic and functionality are added to parts of circuits so that they can be configured as a test pattern generator implemented as a *linear feed-back shift register* (LFSR) and signature analyzer based on a *multiple input signature register* (MISR) during testing. The BIST technique allows a circuit to be tested from within the circuit itself and declare itself faulty or fault-free depending upon the outcome of the test. It eliminates the need for deterministic test generation and an external tester at the expense of some area and delay overhead, and possible low test quality in term of faults that get tested. The signature analyzer compresses the output sequence generated by the circuit into a much smaller sequence, known as a signature, and matches it with the pre-stored signature for the fault-free circuit. This eliminates the need to store the entire expected response of the circuit, at the expense of a small probability of missing an error in the output sequence, thus declaring a faulty circuit as fault-free. Additionally, the fault coverage generated by a BIST technique is comparable with the one generated by external test such as ATPG. Figures 2.6 and 2.7 taken from [86] show fault coverage comparison on BIST and external test for a 16-bit array multiplier with Booth's algorithms and 16-bit Wallace tree multiplier with Booth's algorithms, respectively. Much research has been done to provide numerous effective BIST schemes [47] and to analyze and improve the effectiveness of BIST techniques [3, 4, 5, 84].

2.3 High-Level Synthesis

The state-of-the-art overall VLSI design flow is shown in Figure 2.8 and 2.9. Figure 2.9 also shows that designers are twelve times more productive working at the behavioral level instead of the layout level. However, if anything other than layout design is used,

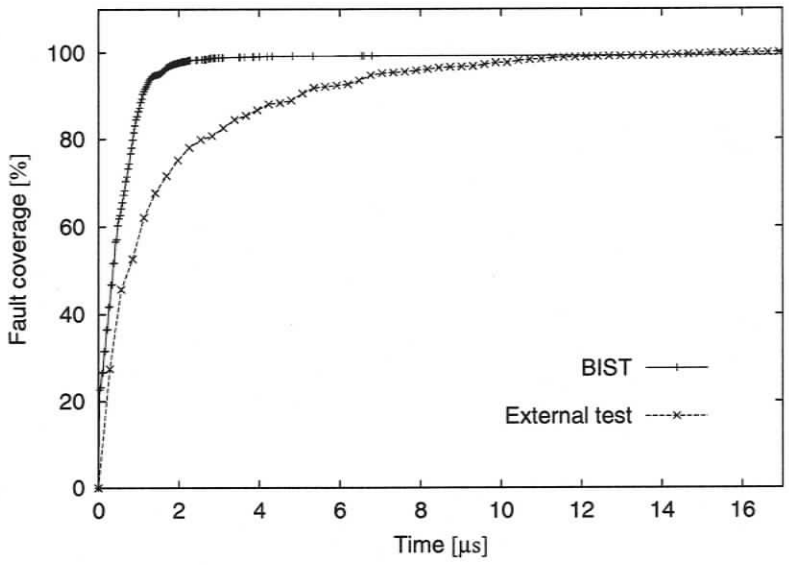


Figure 2.6. *Fault coverage versus time for a 16-bit array multiplier*

the designs have to be transformed into the layout level before they can be physically implemented. This process is referred to as synthesis and is typically performed by automatic or semi-automatic tools. So starting from the behavioral level the synthesis steps are:

- *High-level synthesis* starts with a behavioral description of a digital system and synthesizes a register-transfer level (RTL) macroscopic structure consisting of functional units, registers and multiplexors or buses.
- *Register Transfer Level/logic synthesis* starts with an RTL description and synthesize a logic or gate level netlist using a standard cell library of cells.
- *Physical synthesis* deals with placement, layout and routing of the gate-level netlist.

Each high-level/RTL/logic/physical synthesis task goes through a number of iteration until the design constraints are met. In the traditional synthesis flow, testability is considered after the logic synthesis stage. Test pattern generation, modification of

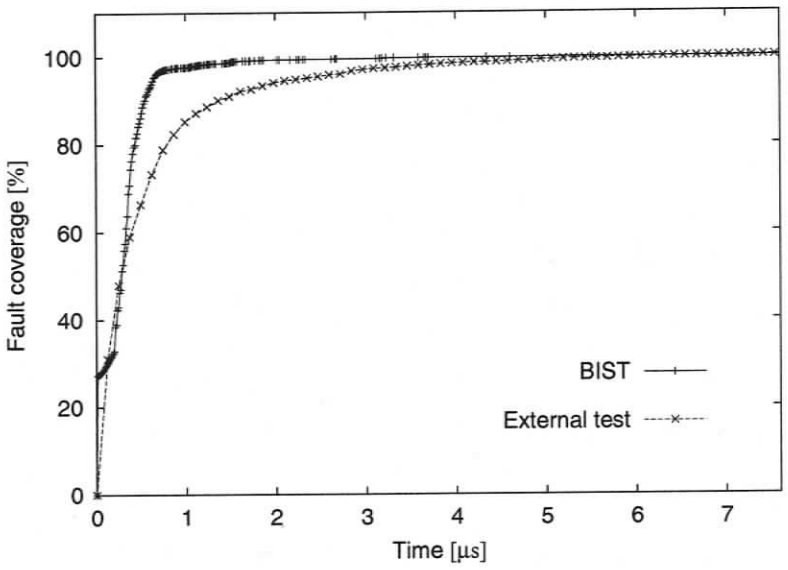


Figure 2.7. *Fault coverage versus time for a 16-bit wallace tree multiplier*

design for scan or BIST is done after the logic level netlist has been synthesized. There is a trend towards incorporating more and more of the testability features in an RTL design before the logic synthesis stage. To ensure that testability insertion at RTL level meets the testability goals of area overhead and test quality, testability consideration needs to be incorporated into the preceding stage in the synthesis flow, namely high-level synthesis. Figure 2.9 [58] illustrates and shows that designers are twelve times more productive working with high level synthesis from behavioral level instead of the logic or layout/physical level.

High-level synthesis consists of the construction of the RTL structure of a digital system that implements a given behavior and satisfies a set of design constraints. Usually the input to the high-level synthesis is given in an algorithmic-level specification, such as behavioral VHDL, which gives the required mapping from sequences of inputs to sequences of outputs. Here the specification should constrain the internal structure of the system to be designed as little as possible. From the input speci-

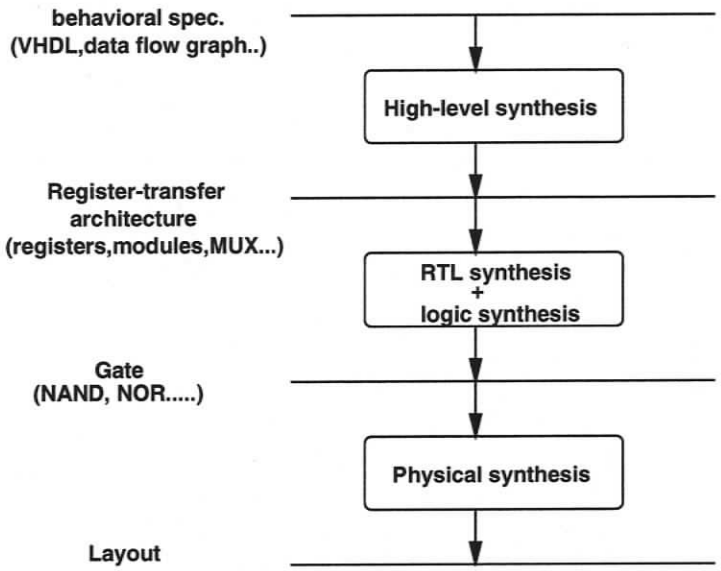


Figure 2.8. *Overall VLSI synthesis flow*

fication, the outcome of high-level synthesis is a data path comprising of *functional modules* such as adders and multipliers, *registers* that store data values and *steering logic circuits* (multiplexors or buses) that transports data to the appropriate destination at the appropriate time. The basic component of the data path will eventually be implemented by some physical modules available in a given technology. The technological parameters, such as silicon area, operation delay and power consumption of the physical modules are usually stored in a module library and made available to the high-level synthesis algorithms. A control part should also be produced if it is not integrated into the data path.

The main steps involved in high-level synthesis are stated as follows:

- **Behavioral Transformations:** These transformations aim at optimizing the behavior of the design. Obvious transformations are compiler optimizations such as constant propagation, dead code elimination, common subexpression elimination and loop unrolling [6]. Other transformations are more specific to high-level synthesis such as substituting multiplication by a power of two with

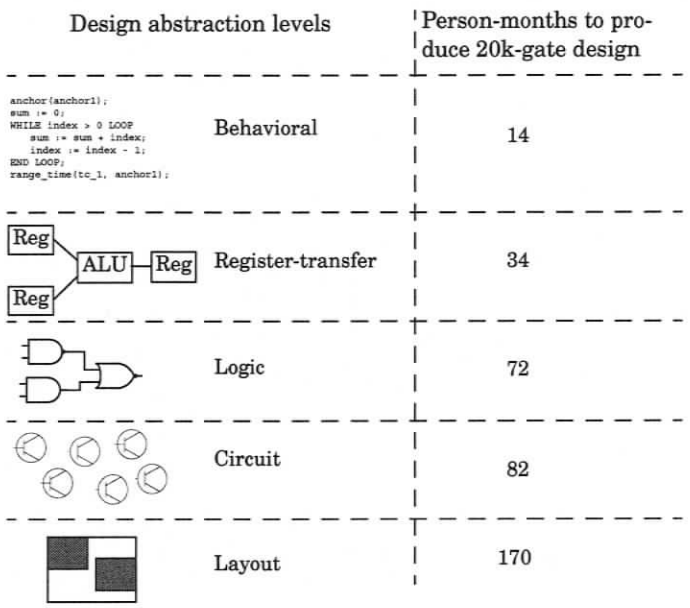


Figure 2.9. *Design abstraction levels and synthesis*

selection of appropriate bits, taking into account commutativity of operators, increasing operator-level parallelism and reducing the number of levels in the data flow graph. Transformations have been studied for a variety of goals in high-level synthesis, including area, performance and fault tolerance [20, 27, 38, 48, 79].

- **Operation Scheduling:** Operation scheduling deals with the assignment of each operation to a time slot corresponding to a clock cycle or time interval. The input to this task usually consists of a control and *data flow graph* (CDFG), a set of available hardware resources and a performance constraint. A schedule will be generated such that the data/control dependency defined by the CDFG will not be violated and the performance constraints is satisfied. Because scheduling determines which operations can be assigned to the same time slot, it affects the degree of concurrency of the resulting design and thus its performance. Furthermore, the maximum number of concurrent operations of

a given type in a schedule is a lower bound on the number of required hardware resources for that operation. Therefore, the choice of a schedule affects the cost of the implementation. Scheduling consequently plays an important role in high-level synthesis. Scheduling is an NP-hard problem [32] and several heuristic scheduling techniques have been proposed such as As-Soon-As-Possible (ASAP) scheduling, As-Late-As-Possible (ALAP) scheduling, list scheduling, force-directed scheduling and path-based scheduling [59].

- **Data Path Allocation:** Data path allocation and binding deal with the problem of which resources to implement in the physical implementation. Such resources include registers, memory units, and different functional units as well as their communication channels. The basic principle is to share resources as much as possible provided that the performance and other design criteria can be satisfied. Allocation and binding carry out selection and assignment of hardware resource for a given design. The allocation selects a number and types of hardware resources for a given design. The binding assigns the particular instance of a selected hardware component to a given data path node. Sharing of hardware modules is allowed if they are not used by different parallel activities at the same time. We use the term allocation to refer to the combined tasks of allocation and binding, unless allocation is being referred to specifically. The selection of a number and type of hardware resources during the allocation step with the general binding problem can usually be formulated as optimization problems. They can be classified into heuristic methods, linear programming approaches and graph-based approaches [59].

Preliminary work in high-level synthesis was carried out as early as the 1960s, but it was not until the end of the 1970s that the first system considering the three subtasks of operation scheduling, data path allocation and binding arrived. During the 1980s the research area of high-level synthesis grew drastically and a wide area of topics was investigated. Many high-level synthesis systems have been developed

in universities. Research in academia has helped create an algorithmic basis for the field of high-level synthesis [30, 58, 67, 88]. More recently, the acceptance of high-level modeling and functional simulation has spurred the development of high-level synthesis systems in the industry. Efforts in the industry by IBM [12], SIEMENS [14], IMEC [94], AT&T [91] and GM [29, 45] have contributed significantly to making high-level synthesis practical and bringing it closer to production use. The transition of high-level synthesis systems to the marketplace has been a slow one, but recently they have come of age and are being used by ASIC designers to significantly cut design cycle times [74, 83, 93].

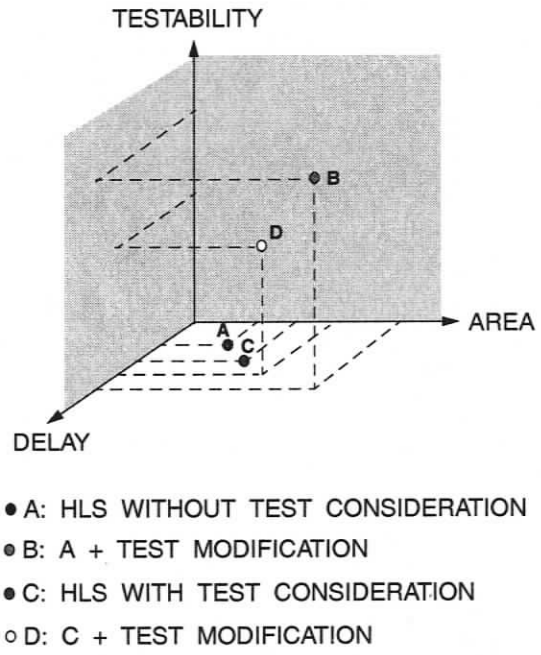


Figure 2.10. *Three dimensional design space*

2.4 High-Level Test Synthesis

Based on the design methodology of high-level synthesis and test synthesis, **High-level Test Synthesis** [28, 54] is becoming a new, emerging and promising area.

The primary goal of high-level test synthesis is to improve a design's testability systematically while keeping performance and area within given constraints. The synthesis is usually carried out by DFT specific transformations together with traditional high-level synthesis transformations. This differs from traditional synthesis methods where the test hardware is added or a test architecture is selected after the given functionality has been implemented.

Typically high-level test synthesis uses testability measures as one term of the cost function to guide the synthesis process. It tries to find a good trade-off among the design testability, performance and area which can be illustrated by Figure 2.10 with the hypothetical three-dimensional design space [68]. The testability axis corresponds to test cost which is reflected by parameters such as test generation time, fault coverage, test area overhead and test application time. In our case the testability axis corresponds to impact on area because of testability. Design point A might have a low area and delay after synthesis but adding testability to this design might result in an inferior design point B. On the other hand if testability is considered during synthesis a design point such as C even though inferior to A will result in a superior design point D after testability has been added.

Assuming the promising BIST techniques, in this thesis, our main focus is on **High-level Test Synthesis** with BIST methodology, which aims to drive an inherently testable architecture at minimal area/best performance overhead. Since such a testable architecture requires far less, or even no, test investment at lower levels, its final implementation can have better overall area and performance than implementations made testable by lower-level DFT methods.

In the following chapter, we start with the introduction of design representation.

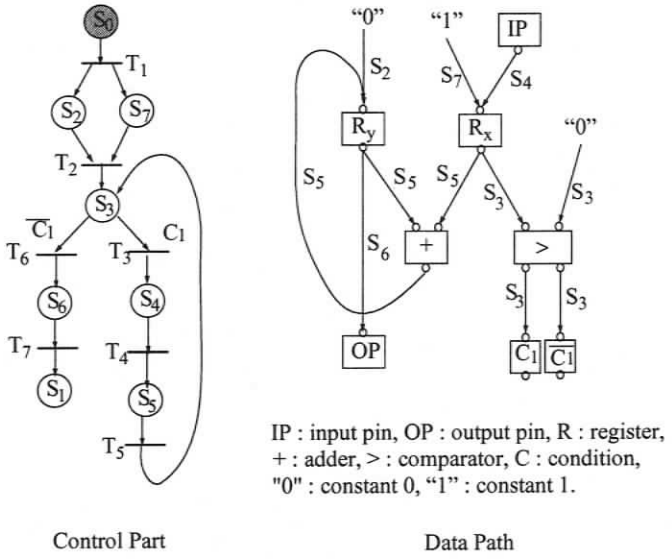
Chapter 3

Design Representation

3.1 Introduction

A VHDL behavioral specification of a digital system and a set of design constraints are taken as the inputs of our test synthesis. It generates a register transfer level hardware implementation which consists of a data path and a controller. The kernel of the system is an intermediate design representation, called Extended Timed Petri Net (ETPN) [76], which can be used both for testability analysis and high-level synthesis. In ETPN, the structural properties of the data path and controller are explicitly captured in order to facilitate accurate analysis of the intermediate design in term of performance, area and testability. Many synthesis systems use several different representation models such as the control flow graph, data flow graph and control/date flow graph during different stages of the synthesis process. In most of the design environments, however, it is more useful to have a unified design representation which can be used to represent the design at different degrees of completeness.

The ETPN design representation is such an unified design representation. A formal notation of semantic equivalence can be defined so as to prove that the design transformations are semantics-preserving. Besides this, using ETPN, explicit information of all stages of high level synthesis, parallel computation or concurrency and timing information can be represented, it can also represent digital hardware with a high degree of concurrency and asynchrony, that supporting the design of locally




```
BEGIN
  y := 0;
  x := 1;
  WHILE x > 0 DO
  BEGIN
     Read(x);
     y := y + x;
  END;
  Write(y);
END;
```

Figure 3.1. *An example of the ETPN design representation*

synchronous, globally asynchronous architectures. It consists of two parts: data path and control part. The data path is a directed graph with nodes and arcs. The node represents storage (registers) and manipulation of data. The arc connecting two nodes represents the flow of data. The control part, on the other hand, is captured as a timed Petri net with restricted transition firing rules [76]. These two parts are related through the control states in the control part controlling the data transfers in the data path, and the condition signals in the data path controlling some transitions in the control part.

An example of the ETPN representation with its VHDL specification is described in Figure 3.1 where Petri net places (S-elements) are the circles while the transitions (T-elements) are the bars. Initially a token is placed at S_0, which is an initial place. A transition is enabled if all its places have at least one token and it may be fired when the transition is enabled and the guard condition is true. Firing an enabled transition removes a token from each of its input places and deposits a token in each of its output places. If no token exists in any of the places, the execution is terminated. When a place holds a token, its associated arcs in the data path will open for data to flow. For instance when S_2 holds a token, the edges controlled by S_2 in the data path activate and data is moved.

3.2 Basic Definitions

The formal definition of ETPN is described as follows.

Definition 1 *A data path, D, is a five-tuple,* $\mathbf{D} = (\mathbf{V}, \mathbf{I}, \mathbf{O}, \mathbf{A}, \mathbf{B})$, *where*
$\mathbf{V} = \{V_1, V_2, \cdots, V_n\}$ *is a finite set of* vertices *each of which represents a data manipulation or storage unit.*
$\mathbf{I} = \mathbf{I}(V_1) \cup \mathbf{I}(V_2) \cup \cdots \cup \mathbf{I}(V_n)$ *with* $\mathbf{I}(\mathbf{V_j}) =$ *the set of input ports associated with vertex* V_j.
$\mathbf{O} = \mathbf{O}(V_1) \cup \mathbf{O}(V_2) \cup \cdots \cup \mathbf{O}(V_n)$ *with* $\mathbf{O}(V_j) =$ *the set of output ports associated with vertex* V_j.
$\mathbf{A} \subseteq \mathbf{O} \times \mathbf{I} = \{< O, I > | O \in \mathbf{O}, I \in \mathbf{I}\}$ *is a finite set of arcs each of which represents a connection from an output port of a vertex to an input port of another vertex or the same vertex;*
$\mathbf{B} : \mathbf{O} \rightarrow \mathbf{2^{OP}}$ *is a mapping from output ports to sets of operations;* $\mathbf{OP} = \{OP_1, OP_2, \cdots OP_m\}$ *is a finite set of operations of the underlying algebraic structure;* **OP** *is divided into the sequential subset* **SEQ** *and the combinational subset* **COM**.

Intuitively, a data path is a directed graph with each vertex having possibly multiple input ports and output ports. The vertices are used to model data manipulation and storage units, while the arcs are used to model their connections.

Definition 2 *A data/control flow system is an eight-tuple,* $\Gamma = (\mathbf{D}, \mathbf{S}, \mathbf{T}, \mathbf{F}, \mathbf{C}, \mathbf{G}, \mathbf{R}, M_o)$, *where*

$\mathbf{D} = (\mathbf{V}, \mathbf{I}, \mathbf{O}, \mathbf{A}, \mathbf{B})$ *is a data path;*

$\mathbf{S} = \{S_1, S_2, \cdots, S_n\}$ *is a finite set of* S*-elements or control places;*

$\mathbf{T} = \{T_1, T_2, \cdots, T_m\}$ *is a finite set of* T*-elements or transitions;*

$F \subseteq (\mathbf{S} \times \mathbf{T}) \cup (\mathbf{T} \times \mathbf{S})$ *is a binary relation, called the control flow relation;*

$\mathbf{C} : \mathbf{S} \rightarrow \mathbf{2^A}$ *is a mapping from control places to sets of arcs of the given data path; an arc* A_i *is* controlled *by a control place* S_j *if* $A_i \in \mathbf{C}(S_j)$*;*

$\mathbf{G} : \mathbf{O} \rightarrow \mathbf{2^T}$ *is a mapping from output ports of data path vertices to set of transitions; a transition* T_i *is guarded by an output port* O_j *if* $T_i \in \mathbf{G}(O_j)$*;*

$\mathbf{R} : \mathbf{S} \rightarrow \mathbf{2^{(O \times OP)}}$ *is a mapping from control places to set of pairs consisting of an output port and an operation; an output port/operation pair* $< O_i, OP_j >$ *is selected by* S_k *if* $< O_i, OP_j > \in \mathbf{R}(S_k)$*;*

$M_o : \mathbf{S} \rightarrow \{0, 1\}$ *is an initial marking function.*

The definition of the data/control flow model is based on the marked Petri net notation. The Petri net **S**-elements are used to capture the control signal concept. When a control place holds a token, a control signal will be sent to control the corresponding arcs in the data path specified by the control mapping function **C**. As there could be more than one control place which holds tokens, there exists multiple control signals that control parallel operations in the data path. The temporal relation between the control signals is defined by a partial ordering structure, which is captured by the control flow relation **F**. To represent the situation where the control flow is influenced by the results of some data path operations, we must be able to use condition signal to direct the control flow. For this purpose, the guarding condition concept is introduced; a transition may be guarded by conditions produced from the

data path represented at the output ports of some vertices, which is defined by **G**. The mapping **R** is used to select the operation to be performed with an output port when it is assigned multiple operations. More details about ETPN can be found from [76]. For illustration purpose, a detailed ETPN control and data path descriptions for benchmark *Ex* are presented in Figure 3.2 and Figure 3.3, respectively.

3.3 Mapping VHDL Specification to ETPN

In this subsection, we briefly outline how VHDL specifications are translated into the ETPN design representation. Compilation of VHDL to ETPN is performed by passing through two internal intermediate forms, namely the program graph and the data/control flow descriptions. The process below shows how a VHDL specification is successively translated to ETPN. It is performed in the following four steps:

1. The first pass of the compiler does a complete syntactic and semantic analysis of the VHDL specification and transforms it into an internal form called a program graph.
2. The second pass performs structural transformations on the program graph and, at the same time, some code optimizations. The transformations performed during this pass are aimed at reducing some high-level constructs, still present in the program graph, to a level which is suitable for the final generation of the ETPN code.
3. The data/control flow description resulting after the second pass reflects already the flow of data and control through the level corresponding to constructs supported by ETPN. The main task of the third pass is to perform parallelization of the internal representation.
4. The last pass transforms the data/control flow descriptions resulting after the previous pass into ETPN, generating the corresponding data path and the control Petri net.

In theory, the ETPN representation resulting from compilation of the VHDL specification can be viewed as a structural specification which can be implemented directly. It would correspond to a very expensive and fast solution since no sharing of resources is introduced. Practically, however, the generated ETPN representation does not indicate how the design is to be implemented in hardware. An scheduling and efficient data path allocation of high-level synthesis process has to perform adequate transformations on the ETPN representation in order to produce a structure with a reasonable sharing of resources. This is also the main focus of the thesis, and these issues are addressed in later chapters.

3.4 Hardware Implementation of ETPN

After high-level synthesis, the ETPN representations have to be mapped into hardware at the register-transfer level. The implementation is divided into two parts. First, the data path is implemented using RTL components, such as registers, functional units and multiplexors and then the control Petri net is converted into a finite state machine.

The transformation of an ETPN data path to a RTL design is carried out by a net-list generation procedure. It is assumed that each ETPN data path vertex has at least one module in a module library which directly implement its functionality.

The timed Petri net description of control logic can not be directly mapped into hardware. It is first transformed into a finite state machine description and then implemented as a microprogram, PLAs or dedicated circuits. More details can be found in [28].

```
(MODULE ex

 ;; The control Petri net is defined as:
 (CP
  (PLACE P0 (IT #) (OT T1))
  (PLACE P1 (IT T15) (OT #))
  (PLACE P2 (IT T1) (OT T2) (%NAME READ) (%LINE 20))
  (PLACE P3 (IT T2) (OT T3) (%NAME READ) (%LINE 21))
  (PLACE P4 (IT T3) (OT T4) (%NAME READ) (%LINE 22))
  (PLACE P5 (IT T4) (OT T5) (%NAME READ) (%LINE 23))
  (PLACE P6 (IT T5) (OT T6) (%NAME :=) (%LINE 24))
  (PLACE P7 (IT T6) (OT T7) (%NAME :=) (%LINE 25))
  (PLACE P8 (IT T7) (OT T8) (%NAME :=) (%LINE 26))
  (PLACE P9 (IT T8) (OT T9) (%NAME :=) (%LINE 27))
  (PLACE P10 (IT T9) (OT T10) (%NAME :=) (%LINE 28))
  (PLACE P11 (IT T10) (OT T11) (%NAME :=) (%LINE 29))
  (PLACE P12 (IT T11) (OT T12) (%NAME :=) (%LINE 30))
  (PLACE P13 (IT T12) (OT T13) (%NAME :=) (%LINE 31))
  (PLACE P14 (IT T13) (OT T14) (%NAME WRITE) (%LINE 32))
  (PLACE P15 (IT T14) (OT T15) (%NAME WRITE) (%LINE 33))

  ;; The transitions are defined as:
  (TRANSITION T1)
  (TRANSITION T2)
  (TRANSITION T3)
  (TRANSITION T4)
  (TRANSITION T5)
  (TRANSITION T6)
  (TRANSITION T7)
  (TRANSITION T8)
  (TRANSITION T9)
  (TRANSITION T10)
  (TRANSITION T11)
  (TRANSITION T12)
  (TRANSITION T13)
  (TRANSITION T14)
  (TRANSITION T15)
 )
```

Figure 3.2. *The ETPN representation (control part) for benchmark Ex*

3.5 Summary

In this chapter, the ETPN design presentation is described. Based on such intermediate design representation, the testability analysis algorithm with its incremental analysis are presented fully in the next chapter.

```
;; The data path is defined as:
(DP

 (NODE N1 Pad ((WL 16) (IP Ip1) (OP Op1)) (%SYMBOL input))
 (NODE N2 Pad ((WL 16) (IP Ip1) (OP Op1)) (%SYMBOL output))
 (NODE N3 Pad ((WL 16) (IP Ip1) (OP Op1)) (%SYMBOL u_in))
 (NODE N4 Pad ((WL 16) (IP Ip1) (OP Op1)) (%SYMBOL v_in))
 (NODE N5 Pad ((WL 16) (IP Ip1) (OP Op1)) (%SYMBOL z_in))
 (NODE N6 Pad ((WL 16) (IP Ip1) (OP Op1)) (%SYMBOL y_in))
 (NODE N7 Pad ((WL 16) (IP Ip1) (OP Op1)) (%SYMBOL x_out))
 (NODE N8 Pad ((WL 16) (IP Ip1) (OP Op1)) (%SYMBOL w_out))
 (NODE N9 Reg ((WL 16) (IP Ip1) (OP Op1)) (%SYMBOL a))
 (NODE N10 Reg ((WL 16) (IP Ip1) (OP Op1)) (%SYMBOL b))
 (NODE N11 Reg ((WL 16) (IP Ip1) (OP Op1)) (%SYMBOL c))
 (NODE N12 Reg ((WL 16) (IP Ip1) (OP Op1)) (%SYMBOL d))
 (NODE N13 Reg ((WL 16) (IP Ip1) (OP Op1)) (%SYMBOL f))
 (NODE N14 Reg ((WL 16) (IP Ip1) (OP Op1)) (%SYMBOL e))
 (NODE N15 Reg ((WL 16) (IP Ip1) (OP Op1)) (%SYMBOL u))
 (NODE N16 Reg ((WL 16) (IP Ip1) (OP Op1)) (%SYMBOL v))
 (NODE N17 Reg ((WL 16) (IP Ip1) (OP Op1)) (%SYMBOL z))
 (NODE N18 Reg ((WL 16) (IP Ip1) (OP Op1)) (%SYMBOL y))
 (NODE N19 Reg ((WL 16) (IP Ip1) (OP Op1)) (%SYMBOL x))
 (NODE N20 Reg ((WL 16) (IP Ip1) (OP Op1)) (%SYMBOL w))
 (NODE N21 Mlt ((WL 16) (IP Ip1 Ip2) (OP Op1)) (%SYMBOL *))
 (NODE N22 Mlt ((WL 16) (IP Ip1 Ip2) (OP Op1)) (%SYMBOL *))
 (NODE N23 Con ((WL 16) (IP #) (OP Op1)) (%SYMBOL 3))
 (NODE N24 Mlt ((WL 16) (IP Ip1 Ip2) (OP Op1)) (%SYMBOL *))
 (NODE N25 Sub ((WL 16) (IP Ip1 Ip2) (OP Op1)) (%SYMBOL -))
 (NODE N26 Con ((WL 16) (IP #) (OP Op1)) (%SYMBOL 5))
 (NODE N27 Sub ((WL 16) (IP Ip1 Ip2) (OP Op1)) (%SYMBOL -))
 (NODE N28 Mlt ((WL 16) (IP Ip1 Ip2) (OP Op1)) (%SYMBOL *))
 (NODE N29 Sub ((WL 16) (IP Ip1 Ip2) (OP Op1)) (%SYMBOL -))
 (NODE N30 Add ((WL 16) (IP Ip1 Ip2) (OP Op1)) (%SYMBOL +))

 ;; The arcs are defined as:
 (ARC A1 16 (N3 Op1) (N15 Ip1) (CP P2) (XT I))
 (ARC A2 16 (N4 Op1) (N16 Ip1) (CP P3) (XT I))
 (ARC A3 16 (N5 Op1) (N17 Ip1) (CP P4) (XT I))
 (ARC A4 16 (N6 Op1) (N18 Ip1) (CP P5) (XT I))
 (ARC A5 16 (N15 Op1) (N21 Ip1) (CP P6) (XT I))
 (ARC A6 16 (N16 Op1) (N21 Ip2) (CP P6) (XT I))
 (ARC A7 16 (N21 Op1) (N9 Ip1) (CP P6) (XT I))
 (ARC A8 16 (N23 Op1) (N22 Ip1) (CP P7) (XT I))
 (ARC A9 16 (N17 Op1) (N22 Ip2) (CP P7) (XT I))
 (ARC A10 16 (N22 Op1) (N10 Ip1) (CP P7) (XT I))
 (ARC A11 16 (N9 Op1) (N24 Ip1) (CP P8) (XT I))
 (ARC A12 16 (N10 Op1) (N24 Ip2) (CP P8) (XT I))
 (ARC A13 16 (N24 Op1) (N11 Ip1) (CP P8) (XT I))
 (ARC A14 16 (N26 Op1) (N25 Ip1) (CP P9) (XT I))
 (ARC A15 16 (N18 Op1) (N25 Ip2) (CP P9) (XT I))
 (ARC A16 16 (N25 Op1) (N12 Ip1) (CP P9) (XT I))
 (ARC A17 16 (N15 Op1) (N27 Ip1) (CP P10) (XT I))
 (ARC A18 16 (N11 Op1) (N27 Ip2) (CP P10) (XT I))
 (ARC A19 16 (N27 Op1) (N14 Ip1) (CP P10) (XT I))
 (ARC A20 16 (N10 Op1) (N28 Ip1) (CP P11) (XT I))
 (ARC A21 16 (N12 Op1) (N28 Ip2) (CP P11) (XT I))
 (ARC A22 16 (N28 Op1) (N13 Ip1) (CP P11) (XT I))
 (ARC A23 16 (N14 Op1) (N29 Ip1) (CP P12) (XT I))
 (ARC A24 16 (N13 Op1) (N29 Ip2) (CP P12) (XT I))
 (ARC A25 16 (N29 Op1) (N19 Ip1) (CP P12) (XT I))
 (ARC A26 16 (N17 Op1) (N30 Ip1) (CP P13) (XT I))
 (ARC A27 16 (N16 Op1) (N30 Ip2) (CP P13) (XT I))
 (ARC A28 16 (N30 Op1) (N20 Ip1) (CP P13) (XT I))
 (ARC A29 16 (N19 Op1) (N7 Ip1) (CP P14) (XT I))
 (ARC A30 16 (N20 Op1) (N8 Ip1) (CP P15) (XT I))
)

;; The conditions are defined as:
(CD
)
```

Figure 3.3. *The ETPN representation (data path) for benchmark Ex*

Chapter 4

Testability Analysis

4.1 Introduction

Recent advances in VLSI technology are motivating changes in the traditional methods of design and test, leading to the integration of design and test activities. Testability, defined as the facility to generate and apply tests, is added as a new constraint to the synthesis process and design modifications are proposed to improve testability. Due to the increasing transistor-to-pin ratios which significantly limit the feasibility of testing digital circuits externally, Built-In Self Test (BIST) techniques have become the most popular method, where pseudo-random pattern generators (PRPG) generate and supply test patterns and multi-input signature registers (MISR) compress test responses.

At the RTL level, circuits consist of interconnections of registers, functional units, multiplexors and buses. Both conventional BIST [4] and circular BIST [78, 80, 85] are well-suited for automatic circuit improvement at the register transfer level. Traditionally, each ALU in a circuit is made directly testable by placing test registers to generate test patterns at the ALU's inputs, and test registers to compact the responses at the ALU's output. However, it may not be necessary to add this many test registers [23]. For example, suppose that the input registers to the ALU are not directly controllable, but they still can generate patterns that are random enough to efficiently test the ALU; in this case, there is no need to replace the normal system

registers with more expensive, slower test registers. Thus, in selecting test registers, cost and speed may be traded-off against test effectiveness.

In this chapter, we describe high level quantitative testability metrics to help to evaluate various BIST configurations described above. We can also use such testability metrics to make improvement decisions in later stages such as in high level synthesis. The early decision about testability improvement gives the possibility that designs can be optimized in later synthesis processes. For example, performance and area are two main metrics in synthesis. The early testability improvements can take into account their influences in the performance and area by not inserting scan registers and test modules at the register transfer level in the critical paths and inserting the appropriate number of scan registers and test modules, respectively. The resulting design is, thus, much closer to a global optimal design. The testability analysis carried out at high level abstraction also reduces the computational complexity, since the complexity of a design at this level is significantly lower than at the gate level.

This chapter is organized as follows. Testability analysis with regards to controllability and observability is described in detail in section 4.2. Then in section 4.3, incremental testability analysis is presented.

4.2 Testability Analysis

The objective of testability metrics is to analyze and quantify testability for a given register transfer level design. There are several different techniques for analyzing testability, namely transformation-based analysis [27, 33], Markov-based analysis [16, 17, 24, 62] and Monte Carlo simulation [9, 16, 53, 61]. Usually these test metrics are derived by exact analysis and/or simulation and a comprehensive library of embedded modules and registers is constructed. The library database is used as the basis of a heuristic for analyzing entire register transfer level design with embedded modules and registers.

Basically, our testability metrics quantify two important testability aspects of data path and control part, namely controllability and observability. Controllability measures the cost of setting up any specific value on a line and observability, on the other hand, measures the cost of observing any specific value on a line. In our approach, we mainly follow the test scheme, minimal behavioral BIST originally proposed in [62]. Both controllability and observability of the data path and the control part are further divided into two factors: combinational factors and sequential factors. The combinational factor is measured in terms of the quality of pseudo-random values as they propagate through embedded modules and registers, and the sequential factor is used for the estimated number of steps or clock cycles to control under test. Similarly the combinational observability is measured in terms of sensitivity of embedded modules and registers to erroneous value propagation, i.e. in terms of how difficult it is to propagate an erroneous value through to an observable output, and the sequential factor is used for the estimated number of steps or clock cycles to observe under test. As a result, our testability metric consists of, four measures: combinational controllability (CC), sequential controllability (SC), combinational observability (CO) and sequential observability (SO).

4.2.1 Controllability of combinational components

The sequential controllability (SC) is mainly based on the number of clock cycles needed for data operation at a functional unit or state transitions in the control part from one state to another needed for data transfer, respectively. We have developed similar computational methods to those in [36] to calculate them. For a combinational node, we suppose that it has inputs $X_1, \ldots, X_p$ with $n_1, \ldots, n_p$ bits respectively, and outputs $Y_1, \ldots, Y_q$ with $m_1, \ldots, m_q$ bits respectively. Since data can be transferred from inputs to outputs in the same control state, the sequential controllability at an

output (line Y_k) of this combinational node is obtained by

$$SC_{Y_k} = SC_{in} + clk(S_i), \quad 1 \leq k \leq q,$$

where

$$SC_{in} = \frac{\sum_{i=1}^{p} SC_{X_i} \times n_i}{\sum_{i=1}^{p} n_i}.$$

Here, $clk(S_i)$ is the number of clock cycles needed for data operation at a functional node or module. Usually we assume it to be zero in the case of a combinational node.

For combinational controllability (CC), we use the metric, randomness to quantify it. For pseudorandom BIST techniques, combinational controllability of a line concerns how random the test patterns generated at that line are. Our approach for evaluating combinational controllability is mainly based on the randomness which can be calculated from the underlying state probability distributions. For any line X, let n denote the width of the line in bits. The current value of the line is the value being stored; at a given time, line X takes any one of the following values: $0, 1, 2, \cdots 2^n - 1$. Let X's value probability distribution be denoted by a row vector P_X; the i^{th} element of the vector, $p_{X,i}$ is the probability that line X has in value i. Here an entropy-based measure of randomness is used. Entropy is a measure of uncertainty about the outcome of an event. The randomness metrics [16, 17, 23, 24, 62, 87] give the ratio between the entropy of line X and that of an ideal random line:

$$R(X) = \frac{I_X}{n} = \frac{1}{n} \sum_{i=0}^{2^n-1} p_{X,i} log_2 \frac{1}{p_{X,i}}. \tag{4.1}$$

Randomness ranges from a value of 0 for a line whose value is constant, to a value of 1 for registers that generate uniformly distributed pseudorandom patterns.

For combinational units such as functional modules with pseudorandom BIST techniques, we have developed an analytical representation of output randomness based on the randomness at a function's inputs originally proposed in [9]. To derive such analytical expressions, a surface fitting technique is applied to computed

data sets simulated by our proposed techniques and formulated as general second order interpolation polynomials. For example, for a 2-input 1-output functional module, given $N = 36$ data points (r_{x_i}, r_{y_i}) where both r_{x_i} and r_{y_i} are taken from $0.5, 0.6, 0.7, 0.8, 0.9, 1$ and Monte Carlo simulation results $r_{z_i}, i = 1, 2, \ldots N$, find a function $r_z(r_x, r_y)$ from P_2, a class of polynomials containing all functions of the form $r_x^i r_y^j$, where $0 \leq i + j \leq 2$ and $i \geq 0, j \geq 0$, so as to minimize

$$\Re(r_z) = \sum_{i=0}^{N} (r_z(r_{x_i}, r_{y_i}) - r_{z_i})^2,$$

where $r_z(r_x, r_y) = a_1 + a_2 r_x + a_3 r_y + a_4 r_x^2 + a_5 r_x r_y + a_6 r_y^2$. By solving the normal equations of least squares problems, we can get the coefficients of the analytical expression such as modulo multiplication as shown in Table 4.2.1.

Table 4.1. *Analytical coefficients for modulo multiplication*

r_z(bits)=	a_1	a_2	a_3	a_4	a_5	a_6
$r_z(3)$=	-0.0287	0.5961	0.5287	0.1673	-0.5969	0.2121
$r_z(4)$=	-0.1607	0.9816	0.9159	0.0197	-0.9095	0.0610
$r_z(5)$=	-0.1585	1.0126	1.0618	-0.0718	-0.8013	-0.0993
$r_z(6)$=	-0.1334	1.0835	1.0642	-0.1030	-0.8725	-0.0909
$r_z(7)$=	-0.1312	1.1115	1.0651	-0.0796	-0.9645	-0.0462
$r_z(8)$=	-0.1037	1.1001	1.0997	-0.1444	-0.8450	-0.1441
$r_z(9)$=	-0.0177	0.9880	0.9875	-0.0458	-0.9004	-0.0454

The relationship between the input and output randomness is unique for any functional unit. We can create the module library for different functional units and save it in databases which can be used for our analytical composition technique. Therefore, the combinational controllability at an output of the combinational component is obtained by:

$$CC_{Y_k} = CC_{in} \times R_U,$$

where

$$CC_{in} = \frac{\sum_{i=1}^{p} CC_{X_i} \times n_i}{\sum_{i=1}^{p} n_i}.$$

Unlike Monto Carlo simulation, our constructive composition technique for combinational controllability operates in a top-down fashion, starting from the controllable points, which have perfect randomness, and working down, functional unit by functional unit and register by register, towards the observable points.

4.2.2 Controllability of sequential components

For sequential components such as registers, the data transfers from input X to output Y are determined by the related state transitions in the control part, from S_i to S_j, where S_i controls input line X and S_j control the output line Y. We estimate this by taking into account the controllability at the condition nodes responsible for the state transitions. The randomness of a sequential component is 1, thus, we have the following:

$$CC_Y = CC_X \times CC_{cond(S_i,S_j)},$$

where $CC_{cond(S_i,S_j)}$ is the product of CCs of all condition nodes for state transitions from S_i and S_j.

For a sequential node, the sequential controllability (SC) can be measured as follows

$$SC_Y = SC_X + clk(S_i, S_j),$$

where $clk(S_i, S_j)$ is the sum of clock cycles for state transitions from S_i to S_j. If there are several paths, we choose the shortest one.

4.2.3 Observability of combinational components

Combinational observability CO of a line X_k is the probability that an arbitrary change in the line's value can be propagated to an observable point. To propagate

a fault at the input line of a component, we need to control some input lines and to observe output lines of a component. The observability at an input line X_k of a component is defined by

$$CO_{X_k} = CO_{out} \times T_U(i,m) \times CC_{in},$$

where

$$CO_{out} = \frac{\sum_{i=1}^{q} CO_{Y_i} \times m_i}{\sum_{i=1}^{q} m_i}.$$

and $T_U(i,m)$ is a transparency measure of input port i of a given combinational component m, also the probability of observing a 1-bit variation of its $i-th$ input port through its output described in [23]. CC_{in} is the average controllability as defined above, but without taking into account CC of the input line X_k. This transparency measure quantifies the sensitivity of a combinational component output on varying its input and depends only on the functionality of the component.

The transparency measure is an estimate of fault effect ability. The exact calculation is not feasible, but Monte carlo simulation gives good estimates. In the following Table 4.2.3 taken from [23], the simulation shows some transparency measures for some combinational components. Since both input ports of these modules have equivalent transparency measures, we only show one value instead.

Based on the table, we have developed a similar analytical representation for transparency measures. To derive such analytical expressions, a similar surface fitting technique is applied to computed data sets simulated by our proposed techniques and formulated as general second order interpolation polynomials, as we did before for randomness. We do not repeat the details here. The relationship between the input and output transparency is unique for any functional unit. We can create the module library for different functional units and save this in databases which can be used for our analytical composition technique.

For a combinational node, sequential observability at an input line X_k

$$SO_{X_k} = SO_{out} + clk(S_i), \quad 1 \leq k \leq q,$$

Table 4.2. *Transparency measures for some combinational components*

Functional component	Bit Width	100 simulations	200 simulations	300 simulations
ADD	2	0.504	0.503	0.501
ADD	8	0.501	0.500	0.500
Multiply	2	0.7516	0.7509	0.7499
Multiply	4	0.9367	0.9378	0.9382
Multiply	8	0.9956	0.9960	0.9960
Multiply	16	0.9999	0.9999	0.9999

$$SO_{out} = \frac{\sum_{i=1}^{q} SO_{Y_i} \times m_i}{\sum_{i=1}^{q} m_i},$$

where $clk(S_i)$ is the number of clock cycles needed for data operation.

4.2.4 Observability of sequential components

Similar to the controllability calculation for sequential components, we consider the controllability at the condition nodes responsible for the state transitions between S_i and S_j, and assume that transparency measure of a sequential component is 1. For a sequential node with input X and output Y whose data transfers from S_i to S_j,

$$CO_X = CO_Y \times CC_{cond(S_i,S_j)},$$
$$SO_X = SO_Y + clk(S_i, S_j),$$

where $clk(S_i, S_j)$ is the sum of clock cycles for state transitions from S_i to S_j. If there are several paths, we choose the shortest one.

4.2.5 Feedback loops

In many applications, feedback loops appear very frequently. Traditionally, testability calculation for feedback loops requires long iterative computations. In this thesis, based on the idea from [36], we use the following two methods, namely when it is possible to calculate the controllability at the condition node that is used to control the termination of the feedback loop execution, and when the condition controlling the termination of the loop execution is difficult to find. We explain these two approaches briefly as follows.

In the first case, we calculate the controllability of all lines within the loop execution by the normal controllability calculation described previously without considering the feedback input from the sequential component within the loop, then we use the following formulae to calculate the final controllability for all lines in the loop:

$$CC_i = CC_i \times CC_{loop},$$
$$SC_i = SC_i + SC_{loop}.$$

CC_{loop} and SC_{loop} are combinational controllability and sequential controllability at the condition node that controls the termination of the loop execution. If there are several conditions controlling the loop, CC_{loop} is the product of all CC_{loop} and SC_{loop} is the maximum of all $SC_{loop}s$.

In the second case, when the condition controlling the termination of the loop execution is difficult to find, we can estimate the number of times (repetitions) that this feedback loop will be executed. There are many discussions on this issue, more details can be found in [36].

4.2.6 Testability analysis algorithm

The testability analysis algorithm calculates the testability $\{CC, SC, CO, SO\}$ for each line in a design. It first assigns ones to CCs and zeros to SCs for all primary

inputs in the data path of the ETPN. These values are propagated according to the testability analysis algorithm until the primary outputs are reached. A similar approach can be used for calculating observability in the reverse direction. A simplified version of the algorithm for CC and SC calculation is presented in Algorithm 4.1.

Algorithm 4.1 Controllability Analysis Algorithm

1: Assign: $CC_{prim-in} = 1, SC_{prim-in} = 0$.
2: **repeat**
3: Calculate controllability at the outputs of node U by the average controllability calculated at its inputs.
4: **if** U is a combinational node **then**
5: $CC_{out} = CC_{in} \times CTF_U$,
6: $SC_{out} = SC_{in} + clk(S_i)$.
7: **else if** U is a sequential node **then**
8: $CC_{out} = CC_{in} \times CC_{cond(S_i,S_j)}$,
9: $SC_{out} = SC_{in} + clk(S_i, S_j)$.
10: **end if**
11: **if** U involve in a feedback loop **then**
12: $CC_{out} = CC_{out} \times CC_{loop}$,
13: $SC_{out} = SC_{out} + SC_{loop}$.
14: **end if**
15: **until** All primary outputs are reached

In Algorithm 4.1, as we have described before, $clk(S_i)$ is the number of clock cycles needed for data operation at a combinational node or module. $CC_{cond(S_i,S_j)}$ is the product of CC's all condition nodes for state transitions from S_i to S_j which in general reflects the cost in the control part to find an input to make these conditions true and $clk(S_i, S_j)$ is the sum of clock cycles for state transitions from S_i to S_j. If there are several paths, we choose the shortest one. CC_{loop} and SC_{loop} are the com-

binational controllability and sequential controllability, respectively, at the condition node that controls the termination of a feedback loop. According to Algorithm 4.1, controllability becomes worse and worse with the increase of depth in the data path from primary inputs.

4.3 Global Testability with Gradients

In this section, we describe the global testability indicator and its gradient with respect to changes of node or line testability. Here the line testability is as defined in the previous sections. The global testability indicator is based on a cost function in [55] and is used to estimate the global testability of an entire design. It is defined as:

$$T = \frac{1}{|S|}\sum_{s\in S}\frac{1}{Pd_s}, \quad Pd_s = C_s \times O_s,$$

where S is a line or node set, $|S|$ is the cardinality of S. C_s denotes the sum of deviations of combinational controllability (CC_s) and sequential controllability (SC_s) for node or line s from their average. In practice, $K = \overline{CC}/\overline{SC}$, which scales the combinational and sequential factors to the same level, is used according to the similar approach in [35, 36]. Here $\overline{CC}$ and $\overline{SC}$ are the average combinational and sequential controllability of the whole design respectively. O_s denotes the sum of deviations of combinational observability (CO_s) and sequential observability (SO_s) at node or line s from their average respectively. In practice, $KK = \overline{CO}/\overline{SO}$, is used to scale the combinational and sequential factors to the same level according to the similar approach in [35, 36]. Here $\overline{CO}$ and $\overline{SO}$ are the average combinational and sequential observability of the whole design respectively. Therefore, we have

$$C_s = \frac{CC_s - \overline{CC}}{\overline{CC}} + K \cdot \frac{\overline{SC} - SC_s}{\overline{SC}},$$

and

$$O_s = \frac{CO_s - \overline{CO}}{\overline{CO}} + KK \cdot \frac{\overline{SO} - SO_s}{\overline{SO}}.$$

Based on the above global testability indicator T, we can calculate the gradients $T_{C_s} = dT/dC_s$ and $T_{O_s} = dT/dO_s$, of T with respect to an infinitely small change of node or line testability. We denote these two gradients T_{C_s} and T_{O_s} as controllability and observability gradients, respectively.

We have also developed a procedure based on the chain rule of derivatives. The procedure computes the quantities T_{C_s} and T_{O_s} for all nodes and lines. The computation begins by initializing T_{O_s} at primary input nodes or lines. The procedure then computes this quantity for all other nodes or lines by iteratively applying the chain rule for derivatives at all nodes and lines in the forward path. Each node or line's observability equations determine the proper chain-rule expression to use which we describe below. Then, evaluation of T_{C_s} for all nodes and lines follows initialization of T_{C_s} at primary outputs. For the computation of T_{C_s}, each node and line in the path from primary outputs to primary inputs applies a chain-rule formula that depends on the particular node or line's type.

For a given node, the global testability changes in the following two ways due to a change of its output line's observability (for simplicity, we assume that this node has only one output line): Pd_s of the node itself change, affecting the cost summation; the global testability also change because the observability of its input lines have been changed. These changes should be propagated backward toward the primary inputs. This propagation follows the chain-rule based on the following formula:

$$\frac{dT}{dO_s} = \frac{-1}{C_s O_s^2} + \sum_{i}^{input} \frac{dT}{dO_i} \frac{dO_i}{dO_s},$$

where the first terms correspond to changes in the output line s itself, and the summation is over all inputs. The quantities $\frac{dO_i}{dO_s}$ represent a transfer function that indicates how a node's input observability changes, given a change in its output line's observability. It is clear that the quantities depend on the different node types. We describe the corresponding methods for controllability gradient of combinational and sequential units respectively below.

After computing dT/dO for all nodes or lines, the dT/dC can be determined similarly. Since changing a node's input line controllability affects both its output line's controllability and the observability of its input lines, application of the chain-rule formula is slightly more complicated than for the dT/dO backward propagation. Propagation of dT/dC proceeds in the reverse direction, beginning with initialized primary inputs. They can be described according to the following formula:

$$\frac{dT}{dC_s} = \frac{-1}{C_s^2 O_s} + \sum_{j,j\neq i}^{input} \frac{dT}{dO_j}\frac{dO_j}{dC_s} + \sum_{k}^{output} \frac{dT}{dC_k}\frac{dC_k}{dC_s}.$$

For a combinational node having inputs $X_1, \ldots, X_p$ with $n_1, \ldots, n_p$ bits respectively, and outputs $Y_1, \ldots, Y_q$ with $m_1, \ldots, m_q$ bits respectively, the observability gradients at the input line X_k due to a change in output line Y_i's observability are obtained as follows:

$$\frac{dCO_{X_k}}{dCO_{Y_i}} = \frac{m_i}{\sum_{i=1}^{q} m_i} \times T_U \times CC_{in},$$
$$\frac{dSO_{X_k}}{dSO_{Y_i}} = \frac{m_i}{\sum_{i=1}^{q} m_i}, \quad 1 \leq k \leq q,$$

where CC_{in} is defined as above. The observability gradients at the input line X_k due to a change in input line X_j's ($j \neq k$) controllability are obtained as follows:

$$\frac{dCO_{X_k}}{dCC_{X_j}} = \frac{n_i}{\sum_{i=1}^{q} n_i} \times T_U \times CO_{out}, \quad \frac{dSO_{X_k}}{dCC_{X_j}} = 0,$$

where CO_{out} is defined as above. The corresponding controllability gradients at the output Y_k due to a change in input line X_i's controllability are obtained as follows:

$$\frac{dCC_{Y_k}}{dCC_{X_i}} = \frac{n_i}{\sum_{i=1}^{p} n_i} \times R_U,$$
$$\frac{dSC_{Y_k}}{dSC_{X_i}} = \frac{n_i}{\sum_{i=1}^{p} n_i}, \quad 1 \leq k \leq q.$$

For a sequential node, we can get the similar expressions for observability gradient at input X due to a change of output Y's observability as follows:

$$\frac{dCO_X}{dCO_Y} = CC_{cond(S_i,S_j)}, \quad \frac{dSO_X}{dSC_Y} = 1.$$

The observability gradients at the input line X due to a change in output line Y's controllability are equal to 0. The corresponding controllability gradients at the output Y due to a change in input line X's controllability are obtained as follows:

$$\frac{dCC_Y}{dCC_X} = CC_{cond(S_i,S_j)}, \quad \frac{dSC_Y}{dSC_X} = 1,$$

where $CC_{cond(S_i,S_j)}$ is the product of CC's of all condition nodes for state transitions from S_i to S_j, as above.

For feedback loops, the observability gradient, due to a change of the controllability and observability, can be calculated in the same manner since the calculation of observability for feedback loops is the same as the normal observability calculation. The corresponding controllability gradient at line Y due to the changes at the input X of the feedback loop can be obtained as follows:

$$\frac{dCC_Y}{dCC_X} = (CTF_U \times CC_{cond(S_i,S_j)})^n, \quad \frac{dSC_Y}{dSC_X} = 1.$$

These gradients can not be applied directly to the applications such as BIST partitioning and improvement, node or register merger of high level synthesis since these transformation steps drastically change the testability of a node or line that violates the assumption of the gradients, namely an infinitely small change of node or line testability. With the threshold mechanism, we can propagate the changes of testabilities due to the transformation steps. Once the changes become small, the gradients are used to estimate the impact of the testability changes of the rest of nodes or lines to avoid high computational complexity of recalculating testability for the entire design.

4.4 Incremental Testability Analysis

In this section, we introduce a new cost function called Incremental Testability Measure (ITM) based on a principle proposed in [90]. ITM is used to estimate the change of global testability due to design transformations.

Given a set S of lines/nodes, the ITM due to a transformation step t can be expressed as follows:

$$\begin{aligned} & ITM^t = \Delta T^t = T^t - T^{org} \\ = & \frac{1}{|S|}\{\sum_{s\in S}(\frac{1}{Pd_s^t} - \frac{1}{Pd_s^{org}})\} \\ = & \frac{1}{|S|}\{\underbrace{\sum_{s\in S_1}(\frac{1}{Pd_s^t} - \frac{1}{Pd_s^{org}})}_{large\ difference} + \underbrace{\sum_{s\in S_2}(\frac{1}{Pd_s^t} - \frac{1}{Pd_s^{org}})}_{small\ difference}\}, \end{aligned}$$

where T^{org}, T^t and Pd_s^{org}, Pd_s^t are global testability and the cost functions as defined in section 4 before and after the transformation t, respectively.

First of all, we divide the line or node set S into two subsets S_1 and S_2. For each node or line s in S_1 which is very close to the place of changes, $1/Pd_s^t - 1/Pd_s^{org}$, the effect on global testability, is quite large. If we use the gradient approximation, then we definitely can not get an accurate estimation. On the other hand, for every node or line s in S_2 which is far away, the effect on global testability, $1/Pd_s^t - 1/Pd_s^{org}$ is small enough so that the gradients we described above can accurately estimate their impact on global testability T. Usually, the size of S_1 is much smaller than that of S_2. Based on the effect on global testability on T caused by a transformation step, we can explicitly calculate $1/Pd_s^t - 1/Pd_s^{org}$ for set S_1 and use the gradients to evaluate the second term of ITM. We can propagate the controllability/observability changes caused by design transformations according to the following mechanism. When the changes drop below a threshold, the changes are not propagated. Whether the propagation continues can be determined by the ratio of $T_{C_s} \cdot \Delta C_s$ to T^{org} for changes of controllability and $T_{O_s} \cdot \Delta O_s$ to T^{org} for changes of observability. The number of nodes or lines to be checked to see whether they are in set S_1 for every transformation step is very small for large design, therefore, the complexity for calculating the first term of ITM is very low. For nodes or lines in set S_2, gradients can provide a very good estimate for the second term of ITM. In the following section, we describe the details of computing ITM due to the controllability and observability changes of a

node or line during transformation steps based on the work proposed in [90].

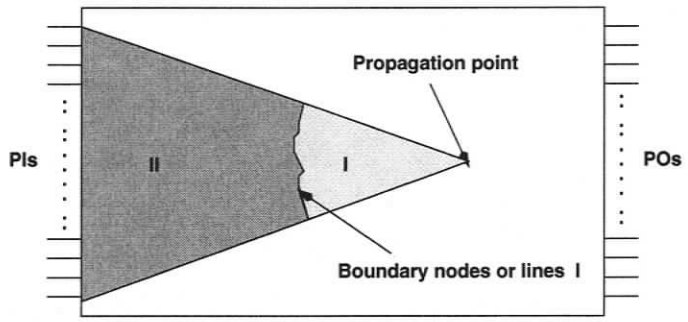


Figure 4.1. *Incremental testability analysis for changes of observability*

We can basically follow the principle that the propagation of the changes of the observabilities starts from the line or node whose observabilities have been changed toward the primary inputs. The details of effect can be seen clearly in Figure 4.1 [90]. For each node or line s inside region I, the ratio of $T_{O_s} \cdot \Delta O_s$ to T^{org} is larger than a given threshold so that we have to further evaluate the new observabilities of its fan-in lines/nodes. Propagation stops at the boundary nodes whose ratios are below the threshold. The ITM of this observability change consists of two parts:

- For each node or line s in region I, we have to explicitly calculate new observabilities and therefore new Pd_s's.
- For nodes or lines s in region II, $\sum_{j \in boundaryI} T_{O_j} \cdot \Delta O_j$ is used to make the estimation of ITM.

Computation of ITM due to the controllability changes is much more complicated because the propagation proceeds in both the forward and backward directions. First of all, starting from the line or node whose observabilities have been changed, the changes will propagate toward the primary outputs shown as region I in Figure 4.2 [90]. During this propagation procedure, we compare the ratios of $T_{C_s} \cdot \Delta C_s$ to T^{org} with a given threshold to decide if we should further calculate the controllabilities of its fanouts. Once the forward propagation stops, we mark the set of nodes or

lines obtained as boundary I in Figure 4.2 [90] and the direction of propagation is then changed to backward toward the primary inputs. We start from nodes or lines in boundary I and proceed the backward propagation similar to what we have done above. Again, if the ratio of $T_{O_s} \cdot \Delta O_s$ to T^{org} is small, the backward propagation stops, and the set of nodes or lines indicated as boundary II in Figure 4.2 [90] is identified. The ITM of this controllability change consists of three parts:

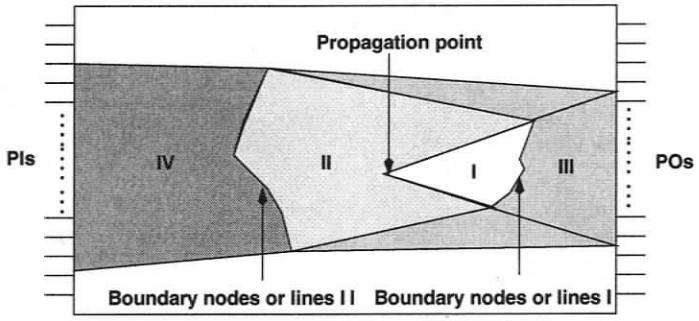


Figure 4.2. *Incremental testability analysis for changes of controllability*

- For each node or line s in region I and II, we have to explicitly calculate the new controllabilities and observabilities and therefore Pd_s.
- For each node or line s in region III, $\sum_{k \in boundary\ I} T_{C_k} \cdot \Delta C_k$ is used as estimations.
- For each node or line s in region IV, $\sum_{j \in boundary\ II} T_{O_j} \cdot \Delta O_j$ is used as estimations.

Actually the changes of controllabilities of nodes or lines boundary I should have effects on all region I to IV. This means that $\sum_{k \in boundary\ I} T_{C_k} \cdot \Delta C_k$ also contains some contribution from region I, II and IV. Because the contribution from region III usually dominates, the error by this approximation is negligible.

4.5 Summary

In this chapter, we have developed data path metrics to analyze and quantify BIST testability for a given design. In the following chapter, these metrics are used to evaluate various BIST configurations such as partitioning, and resource optimization.

Chapter 5

BIST Partitioning and Resource Optimization

The cost of testing complex digital designs has become a large fraction of the total production expenditure as other cost components have decreased. Furthermore, increasing gate-to-pin ratios limit the feasibility of testing digital circuits externally. The incorporation of test structures into the design ameliorates the testability of hardware which is not easily testable through external pins. The Built-In Self-Test (BIST) technique which is the main focus of this thesis has gained acceptance.

The BIST technique involves modification of the hardware on the chip so that the chip has the capability to test itself. Partitioning, an efficient partitioning technique, which decides either which registers should be configured as test registers (conventional BIST) or which registers should be linked in the circular scan path (circular BIST), is very necessary. In this chapter, we first describe an efficient and economic BIST partitioning technique.

5.1 Preliminaries

Before going to the detailed descriptions, we would like to present some basic background on the four main test registers [3], namely Pseudo-Random Pattern Generators (RTPG), Multiple-Input Signature Register (MISR), Built-In Logic Block Observer

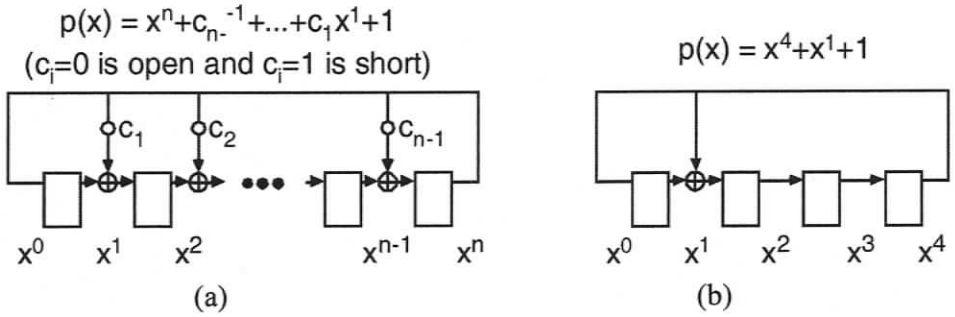


Figure 5.1. *LFSR for (a) generic case and, (b)* $n = 4$

(BILBO) and Concurrent Built-In Logic Block Observer (CBILBO), to be used for the BIST scheme.

5.1.1 Pseudo-Random Pattern Generators (RTPG)

For the test pattern generation, there are two approaches: exhaustive and pseudorandom testing. Exhaustive testing applies all 2^n test patterns generated by counters or linear feedback shift registers (LFSRs). While it can detect all combinational faults, it is feasible only for circuits with a small number of inputs. In pseudo-random testing, test patterns resemble random patterns but are deterministic. The test patterns for pseudo-random testing are generated by LFSRs as shown in Figure 5.1 (a). The connection polynomial function $p(x)$ determines feedback points. Figure 5.1 (b) shows an LFSR with a connection polynomial function, $p(x) = x^4 + x + 1$.

When a primitive polynomial is used to construct an LFSR, the resultant LFSR generates all patterns except the all-zero pattern. Such an LFSR is called a maximum length LFSR. There is at least one primitive polynomial for any order n.

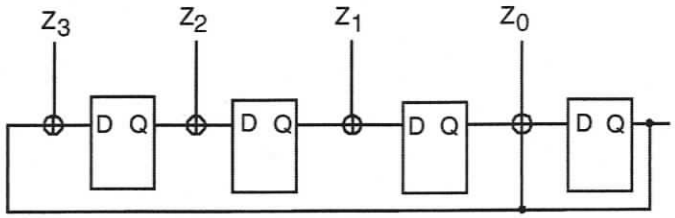


Figure 5.2. *Multiple-input signature register (MISR)*

B1	B2	Modes
1	1	Normal function
0	0	Shift register
1	0	TPG and MISR
0	1	Reset

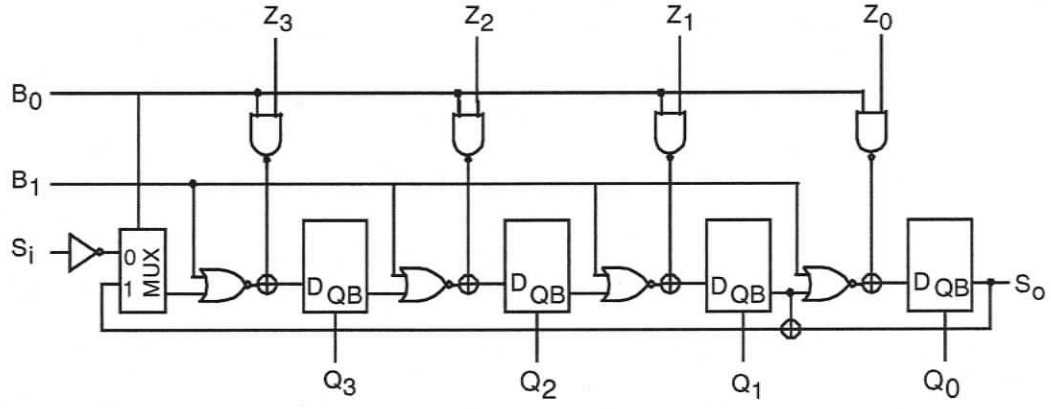


Figure 5.3. *BILBO and its operating modes*

5.1.2 Multiple-Input Signature Register (MISR)

Since test patterns for BIST resemble random patterns, BIST requires a large number of test patterns to achieve a reasonably high fault coverage. The task of storing and comparing a large number of test responses directly on the chip is impractical. Signature analysis is one of the most widely used compression techniques to compress test responses. A signature analyzer performs repetitive division operations and the remainder is called a signature and is compared against the good signature. In general, a circuit under test has multiple outputs. Thus, we need a signature analyzer with multiple-inputs, often referred to as multiple-input signature register (MISR) in BIST, a state is shown in Figure 5.2.

5.1.3 Built-In Logic Block Observer (BILBO)

If a test register behaves as a RTPG in a sub-test session and a MISR in another session, the test register should be implemented as a BILBO as shown in Figure 5.3. Scan in/out functionality of the circuit is necessary for initialization of LFSRs and retrieval of test responses. BILBO has four modes, namely normal function, shift register (scan in/out), TPG/MISR, and reset, and these are shown in Figure 5.3 as well.

5.1.4 Concurrent Built-In Logic Block Observer (CBILBO)

If a test register should be a RTPG and a MISR in the same sub-test session, it should be implemented as a CBILBO. CBILBO, as shown in Figure 5.4, uses two layers of LFSRs to provide RTPG and MISR functionality simultaneously. CBILBO has three modes, as shown in Figure 5.4 as well, normal function, shift register (scan in/out), and TPG/MISR. CBILBO incurs a high area overhead and hence should be avoided, if possible.

B0	B1	Modes
–	0	Normal function
1	1	Shift register
0	1	TPG and MISR

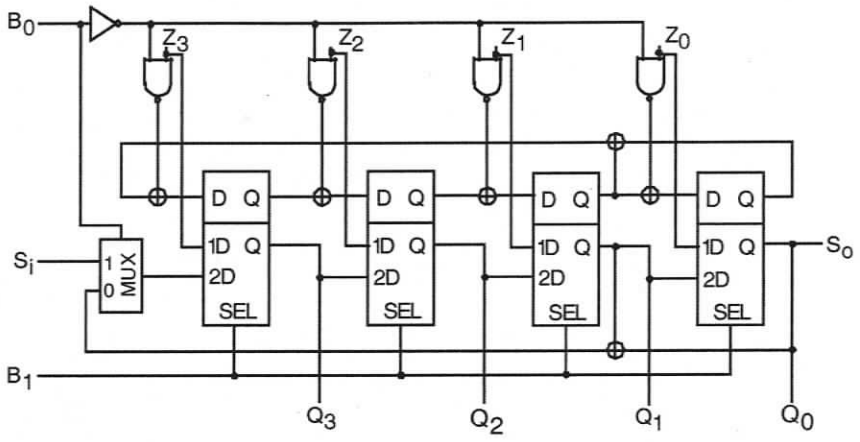


Figure 5.4. *CBILBO and its operating modes*

5.2 BIST Partitioning

For ETPN representation or register transfer level (RTL), digital systems consist of interconnections of registers, functional units (ALUs), multiplexors and buses. Both conventional BIST [4] and circular BIST [78, 80, 85] are well-suited for automatic circuit improvement at the register transfer level. Traditionally, each ALU in a circuit is made directly testable by placing test registers to generate test patterns at the ALU's inputs, and the test registers to compact the responses at the ALU's output. However, it may not be necessary to add this many test registers [23]. For example, suppose that the input registers to the ALU are not directly controllable, but they still can generate patterns that are random enough to efficiently test the ALU; in this case, there is no need to replace the normal system registers with more expensive, slower test registers. Thus, an efficient partitioning technique, which decides either which registers should be configured as test registers (conventional BIST) or which

registers should be linked in the circular scan path (circular BIST), is necessary.

Partitioning for a design can lead to the simplifications of many design procedures such as synthesis and test. Partitioning for testability leads to the simplification of test efforts and the ability to apply different test strategies to different partitions. Our partitioning technique transforms some hard-to-test registers and/or lines to boundary components. These components act as normal registers and/or lines in the normal mode and serve as partitioning boundaries in test mode or test registers. Therefore, a design is partitioned into several sub-circuits and each of them can be tested and controlled based on BIST test schemes. It is, therefore, possible to apply different test strategies, such as scan for deterministic and BIST for random test to different partitions.

The circuit or design partitioning problem can, in general, be formulated as a graph partitioning problem. Given a graph with nodes and arcs, the objective is to partition the nodes into several subsets, such that the total costs of the arcs between nodes in different partitions is minimized. Optimal partitioning is known to be NP-complete [32]. Recent work on partitioning for testability has been reported in [1, 26, 37, 40, 56]. These papers underline the different aspects which can influence the testability of a design. De and Banerjee [26] present a technique which ensures the minimal number of redundant faults after resynthesizing each partitioned block and subsequent re-connection. Maxii and Meo [56] guarantee that each partitioned block is a fanout free region. In [1], Abadir and Newman partition a design to ease testing, but do not take into account whether the partitioning results in the optimal or near-optimal solution for testability. The depth-first strategy they use to search adjacent cells to be in the same cluster does not guarantee the best testability after partitioning. In [40], Gupta partitions a design by clustering gates to clouds. During clustering, the testability issue is not considered. Since full scan is used in their approach, all clouds are combinational.

The main objective of the partitioning presented in this chapter is to use the testa-

bility analysis and other heuristics to find partitioning boundaries and isolate data communication among partitions in the test mode [35, 37]. Partitioning is done in the following two steps: 1) selecting partitioning boundaries, and 2) identifying partitions by clustering a set of components surrounded by the partitioning boundaries.

5.2.1 Boundary selection

There are three types of components, namely transformed registers, test modules (transformed lines) and I/O ports, which can be used as partitioning boundaries. We use two heuristics for the preliminary selection of registers and lines to be transformed. To reduce area overhead, we try to select registers rather than lines. Lines are selected only when the testability or state reachability improvement of registers can not further influence the testability of the hard-to-test region, or further partitioning needs to isolate data communication where no register exists.

Selecting registers and/or lines to be boundary components is based on the following heuristics:

Heuristic 1 *The registers with the worst n testability measure are selected where n is a user-controlled parameter.*

First of all, we run the testability analysis algorithm we described in Chapter 4. Based on the metrics, we select the worst n components as the boundaries first.

5.2.2 The Circular BIST approach

If the Circular BIST technique is applied, a set of registers selected based on the heuristic are linked as a circular scan chain. The detailed techniques of the Circular BIST can be found in [78, 80]. If we plan to use a conventional BIST technique which decides which registers should be configured as test registers such as RTPG, MISR etc., the following partitioning approach, namely component clustering, should be applied.

5.2.3 Component clustering

The actual partitioning procedure is performed by a clustering algorithm which clusters directly interconnected components excluding boundary components. Figure 5.5(a) shows an example of the preliminary partitioning after registers $R1$ and $R2$ are selected as boundary components. The design is partitioned into two partitions ($P1$ and $P2$) in the preliminary partitioning. Accordingly the registers $R1$ and $R2$ are configured as CBILBO test registers, namely the MISR test register for partition $P2$ and the RTPG test register for partition $P2$. Because the area overhead of CBILBO and the size of some partitions such as $P1$ at this stage may still be too large. The transformed registers, such as $R2$, may not be really used for testing purposes as it is not placed between two partitions. Further partitioning by other heuristics is, therefore, necessary. Before presenting these heuristics, we describe some basic definitions from [37]. Assume that $\mathbf{B}$ is the set of all boundary components, $\mathbf{B_i}$ is the set of input boundary components of partition $\mathbf{P_i}$ and $\mathbf{C}$ is the set of components excluding the components in $\mathbf{B}$.

Definition 3 *Given $C_i, C_j \in \mathbf{C} : C_i \Rightarrow C_j$, if the output of components C_i is directly connected to the input of component C_j. $C_i \not\Rightarrow C_j$, otherwise.*

Definition 4 *Given $C_i, C_j \in \mathbf{C} : C_i \stackrel{+}{\Rightarrow} C_j$, if $\exists a_0, a_1, a_2 \cdots, a_{k-1}, a_k \in \mathbf{C}$ such that $(a_0 \Rightarrow a_1) \wedge (a_1 \Rightarrow a_2) \wedge \cdots \wedge (a_{k-1} \Rightarrow a_k)$, $(k \geq 1, C_i = a_0$ and $C_j = a_k)$. $C_i \stackrel{+}{\not\Rightarrow} C_j$, otherwise.*

Definition 5 *A partition $\mathbf{P_i}$ is a non-empty set of components such that*

1. *$C_k \in \mathbf{P_i}$, if $B_i \Rightarrow C_k$, where $B_i \in \mathbf{B_i}$ and $C_k \in \mathbf{C}$.*
2. *$C_k \in \mathbf{P_i}$, if $\exists C_i \in \mathbf{P_i}$, such that $C_i \Rightarrow C_k$, $C_k \in \mathbf{C}$.*
3. *partition $\mathbf{P_m}$ is merged to partition $\mathbf{P_i}$, if $\exists C_i \in \mathbf{P_i}$ such that $C_i \in \mathbf{P_m}$.*

We have developed one additional heuristics for further partitioning, namely Heuristic 2.

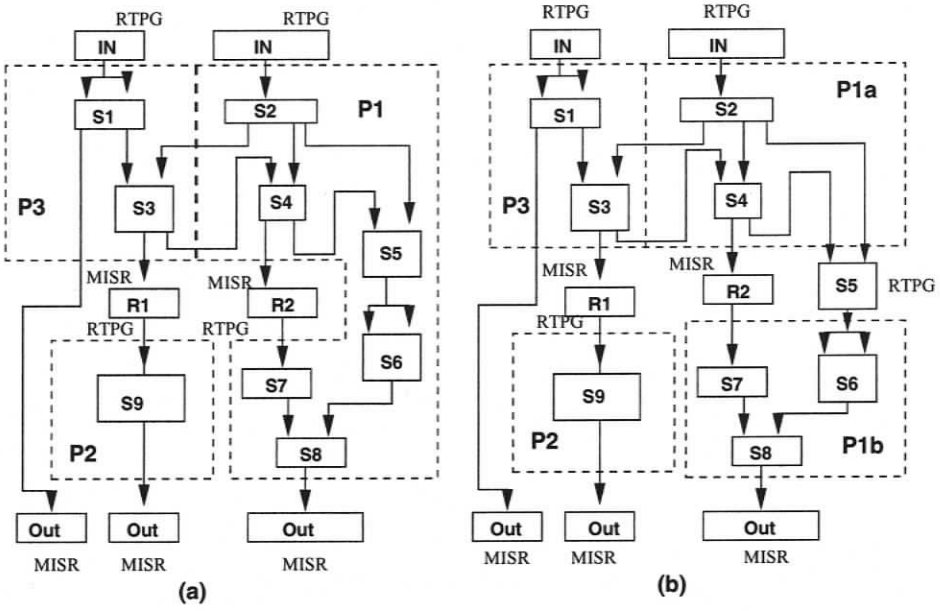


Figure 5.5. *An example of using Heuristic 2*

Heuristic 2 *If* $\exists C_i, C_j \in \mathbf{P_i}$ *and* $\exists B_k \in \mathbf{B}$, *such that* $(C_i \Rightarrow B_k) \wedge (B_k \Rightarrow C_j)$, *then* $\mathbf{P_i}$ *needs to be further partitioned into partitions* $\mathbf{P_m}$ *and* $\mathbf{P_n}$ *such that* $\mathbf{P_m} = \{C_i | (B_k \stackrel{+}{\Rightarrow} C_i) \vee [(C_i \stackrel{+}{\not\Rightarrow} B_k) \wedge (\exists C_j \in \mathbf{P_m} : C_i \stackrel{+}{\Rightarrow} C_j)]\}$, *and* $\mathbf{P_n} = \{C_i | (C_i \stackrel{+}{\Rightarrow} B_k) \vee [(B_k \stackrel{+}{\not\Rightarrow} C_i) \wedge (\exists C_j \in \mathbf{P_n} : C_j \stackrel{+}{\Rightarrow} C_i)]\}$.

This heuristic means that if there is a boundary component used as both the input and output of the same partition, this partition needs to be further partitioned into two such that the boundary component is used only as the input of one partition and as the output of the other. To further divide this partition, we should analyze the partial order of all components in the partition. All components giving direct or indirect output to this component belong to one partition and those receiving direct or indirect input from the boundary component belong to another partition. In the heuristic, components that do not give direct/indirect output from the boundary components and do not receive direct/indirect input from the boundary components have to be decided which partition to be clustered. Based on the above heuristic,

the register $S5$ is selected as RTPG test register for the partition $P1b$. Accordingly register $R2$ can only need to be configured as MISR test register for partition $P1a$. Therefore the size and quality of the partition block can be well-balanced.

5.2.4 Control part modification and implementation

The data transfers and operations specified in the data path are controlled by the control part of ETPN. The partitioning only provides the facility to isolate data communication between partitions. The control part needs to be modified to control the whole design to execute the design function in the normal mode and control the execution of each partition independently in the test mode. We preliminarily follow the method proposed by Gu [37].

All test control registers $T_1, \cdots, T_n$ are stored in the test register TR which is implemented by a special scan path to provide all test control messages. To change the test control message, a new control message needs to be loaded into the scan path. The speed of this scan path is not critical to test speed because the message in the scan path is only used for test control instead of test vectors.

There are two ways to implement the test controller, namely a sequential test controller or a parallel test controller. The sequential test controller only allows partitions to be tested sequentially. A parallel test controller can test several partitions in parallel. The implementation of the parallel test controller needs first to build a low-level controller for each partition. Above these controllers, a top-level controller is built to coordinate these controllers in the normal mode and in the test mode. In these two implementations, all partitions are sequentially executed in the normal mode. The sequential test controller implementation uses less area than the parallel test controller implementation.

5.3 BIST Resource Optimization

As described above, the typical BIST approach tests design components using pseudo-random patterns which are generated by pseudo-random pattern generators (RTPG). Multi-input signature registers (MISR) compact the results of the tests. All test registers are in a shift register chain so that seeds to the RTPGs can be shifted in at the beginning of testing, and compacted results can be shifted out of the MISRs after testing. Table 5.1 based on a macro-cell library from ISI Logic Corp. [68] shows different types of test registers that can be used. *Concurrent built-in logic observation (CBILBO) and built-in logic observation (BILBO)* registers can both generate test patterns and compress test responses, and ensure high fault-coverage. BILBO registers need more test sessions while CBILBO registers require more hardware area. Note that a test register usually has larger hardware area than a normal register (see Table 5.1 where ω is the area scaling factor over a normal register). For example, CBILBOs have an area approximately twice that of the normal registers. One of the main considerations in BIST techniques is, therefore, the extra area for the test circuitry. In this chapter, beside the partitioning technique, we will also present a resource optimization approach before the synthesis algorithm to optimize the BIST test costs as well.

5.3.1 Basic test methodology

We restrict our method to pseudo-random BIST where test patterns are generated using pseudo-random pattern generation registers (RTPGs), based on autonomous Linear Feedback Shift Register (LFSR) design. The test responses are evaluated using multi-input signature registers (MISRs). We base our model on the notion of structural testability at ETPN representation. The key element of structural testability is the Extended Testable Functional Block (XTFBs) [41]. An XTFB consists of a functional block and a set of input and output registers. For the rest of this chapter,

Table 5.1. *Type of test registers*

Register	Test Activity performed	Concurrency of Test Activities	Type of BIST Register	Area (16-bit)	ω
1	-	-	Normal register	432	-
2	Generate test patterns	-	RTPG	528	1.22
3	Compress responses	-	MISR	528	1.22
4	Generate test patterns Compress responses	Non-simultaneous	BILBO	688	1.59
5	Generate test patterns Compress responses	simultaneous	CBILBO	960	2.22

we assume there are two ports for a functional module. Let IR_j^L denote the set of registers connected to the left port of module M_j through only multiplexors and IR_j^R denotes the set of registers connected to the right port of module M_j through only multiplexors. The registers belonging to IR_j^L and IR_j^R are called *input registers* of M_j. Similarly let OR_j denotes any register that collects data from the output port of a module through only multiplexors. Two registers at the input ports should be configured as RTPGs during test. One register connected to the output port should be configured as an MISR in the test mode.

Definition 6 *Given a register-transfer level data path, a* **BIST Embedding** *of a module M_j is a selection of three registers from IR_j^L, IR_j^R and OR_j, respectively, such that each input port of M_j receives test patterns from one of the selected registers and the output port transfers test responses to one of the selected registers.*

Before we perform the self-testable high level synthesis algorithm to be presented in later chapters, every register has to be modified to test registers according to different functionalities. Here we describe an optimal modification approach based on 0-1 Integer Linear Programming formulation [68] to find BIST embeddings in the

data path prepared for the synthesis algorithm such that the cost of modification is minimal.

5.3.2 Pre-synthesis optimization

We now present the pre-synthesis optimization to find embeddings for all modules in a data path such that the cost of modification before the synthesis algorithm is minimal based on [68]. Given a data path with modules $M_1, M_2, \ldots M_m$ and registers $R_1, R_2, \ldots, R_r$, we have the following constraints for every register R_i,

$$R_i^R + R_i^M + R_i^B + R_i^C \leq 1.$$

For every port of M_j, there should be one input register that can generate patterns for the left port and at least one input register that can generate patterns for the right input port. It also has at least one output register that can collect test responses from the module. This give us the following constraints:

for every port of M_j:

$$\sum_{i \in IR_j^L}(R_i^R + R_i^B + R_i^C) \geq 1,$$
$$\sum_{i \in IR_j^R}(R_i^R + R_i^B + R_i^C) \geq 1,$$
$$\sum_{i \in OR_j}(R_i^M + R_i^B + R_i^C) \geq 1$$

We also have the following constraints:

$$\sum_{i \in IR_j^L \bigcup OR_j} R_i^R + R_i^M + R_i^B + 2 \times R_i^C \geq 2,$$
$$\sum_{i \in IR_j^R \bigcup OR_j} R_i^R + R_i^M + R_i^B + 2 \times R_i^C \geq 2.$$

The corresponding objective function is to minimize the total area:

$$\min \sum_{i=1}^{r}(c^R \times R_i^R + c^M \times R_i^M + c^B \times R_i^B + c^C \times R_i^C). \tag{5.1}$$

where C^R, C^M, C^B and C^C are costs for a RTPG, MISR, BILBO and CBILBO, respectively. In our algorithm, we use all the costs for 16-bit area of RTPG, MISR,

BILBO and CBILBO with 528, 528, 688 and 960 cell units by a macro-cell library from ISI Logic Corp. [68].

5.3.3 Case studies

JPEG is a lossy compression method for still images. The lossy compression algorithm operates in three successive stages, the 2-D (2-Dimensional) DCT (Discrete Cosine Transform), quantization, and lossless entropy encoding. The image is partitioned into blocks, and the 2-D DCT is applied to each block. Then the transformed image is quantized and encoded. The key component in the compression process is a mathematical transformation known as the DCT algorithm.

We used the above behavioral-level VHDL description for the benchmark DCT for our case study. Besides, we also use another well-known application FFT as the benchmark to run our resource optimization algorithm. The detailed results can be found in Figure 5.6 and 5.7, respectively.

In the second part of the chapter, we have presented an optimal modification to find the BIST embeddings, the costs for those test registers are still very expensive. In the later 3 chapters, we will show the overhead can be further divided by considering BIST resource requirements early during the high level synthesis process. Therefore the resource optimization can be used for preparation of the high level synthesis or before going to RTL implementation directly.

```
CPU Time for parsing input: 0.1s (0.1s total since program start)
CPU Time for solving: 0.05s (0.15s total since program start)
Value of objective function: 10576

MISR_N10              0
MISR_N11              0
MISR_N12              0
MISR_N13              0
MISR_N14              0
MISR_N15              0
MISR_N16              0
MISR_N17              0
MISR_N18              0
MISR_N19              0
MISR_N20              0
MISR_N21              0
MISR_N22              0
MISR_N23              0
MISR_N24              1
MISR_N25              1
MISR_N26              1

CBILBO_N10            0
CBILBO_N11            0
CBILBO_N12            0
CBILBO_N13            0
CBILBO_N14            0
CBILBO_N15            0
CBILBO_N16            0
CBILBO_N17            0
CBILBO_N18            0
CBILBO_N19            0
CBILBO_N20            0
CBILBO_N21            0
CBILBO_N22            0
CBILBO_N23            0
CBILBO_N24            0
CBILBO_N25            0
CBILBO_N26            0

BILBO_N10             1
BILBO_N11             1
BILBO_N12             1
BILBO_N13             1
BILBO_N14             1
BILBO_N15             1
BILBO_N16             1
BILBO_N17             1
BILBO_N18             1
BILBO_N19             1
BILBO_N20             0
BILBO_N21             0
BILBO_N22             0
BILBO_N23             0
BILBO_N24             0
BILBO_N25             0
BILBO_N26             0


RTPG_N10              0
RTPG_N11              0
RTPG_N12              0
RTPG_N13              0
RTPG_N14              0
RTPG_N15              0
RTPG_N16              0
RTPG_N17              0
RTPG_N18              0
RTPG_N19              0
RTPG_N20              1
RTPG_N21              1
RTPG_N22              1
RTPG_N23              1
RTPG_N24              0
RTPG_N25              0
RTPG_N26              0
```

Figure 5.6. *The BIST resource optimization results for benchmark DCT*

```
CPU Time for parsing input: 0.11s (0.11s total since program start)
CPU Time for solving: 0.07s (0.18s total since program start)
Value of objective function: 12480

RTPG_N3                 1
RTPG_N4                 1
RTPG_N5                 1
RTPG_N6                 1
RTPG_N7                 0
RTPG_N8                 0
RTPG_N9                 0
RTPG_N10                0
RTPG_N11                0
RTPG_N12                0
RTPG_N13                0
RTPG_N14                0
RTPG_N15                0
RTPG_N16                0
RTPG_N17                0
RTPG_N18                0
RTPG_N19                0
RTPG_N20                0
RTPG_N21                0
RTPG_N22                0

MISR_N3                 0
MISR_N4                 0
MISR_N5                 0
MISR_N6                 0
MISR_N7                 1
MISR_N8                 1
MISR_N9                 1
MISR_N10                1
MISR_N11                0
MISR_N12                0
MISR_N13                0
MISR_N14                0
MISR_N15                0
MISR_N16                0
MISR_N17                0
MISR_N18                0
MISR_N19                0
MISR_N20                0
MISR_N21                0
MISR_N22                0

BILBO_N3                0
BILBO_N4                0
BILBO_N5                0
BILBO_N6                0
BILBO_N7                0
BILBO_N8                0
BILBO_N9                0
BILBO_N10               0
BILBO_N11               1
BILBO_N12               1
BILBO_N13               1
BILBO_N14               1
BILBO_N15               1
BILBO_N16               1
BILBO_N17               1
BILBO_N18               1
BILBO_N19               1
BILBO_N20               1
BILBO_N21               1
BILBO_N22               1

CBILBO_N3               0
CBILBO_N4               0
CBILBO_N5               0
CBILBO_N6               0
CBILBO_N7               0
CBILBO_N8               0
CBILBO_N9               0
CBILBO_N10              0
CBILBO_N11              0
CBILBO_N12              0
CBILBO_N13              0
CBILBO_N14              0
CBILBO_N15              0
CBILBO_N16              0
CBILBO_N17              0
CBILBO_N18              0
CBILBO_N19              0
CBILBO_N20              0
CBILBO_N21              0
CBILBO_N22              0
```

Figure 5.7. *The BIST resource optimization results for benchmark FFT*

Chapter 6

Data Path Allocation

6.1 Introduction

Hardware testing is an important activity in hardware design and manufacture. Its main purpose is to check whether a circuit has any manufacturing error. Traditionally, testing digital circuits is performed externally by supplying test patterns on pins and detecting errors by probing the pins. The increasing gate-to-pin ratios limit the feasibility of testing digital circuits externally. *Built-In Self Test (BIST)* techniques offer a solution to the above problems. A BIST technique uses *pseudo-random pattern generators (PRPG)* to generate and supply test patterns and *multi-input signature registers (MISR)* to compress test responses on the chip.

Applying BIST techniques involves modification of the hardware on the chip so that the chip has the capability to test itself. Table 5.1 shows different types of test registers that can be used. *Concurrent built-in logic observation (CBILBO) and built-in logic observation (BILBO)* registers can both generate test patterns and compress test responses, and ensure high fault-coverage. BILBO registers need more test sessions while CBILBO registers require more hardware area as illustrated in Figure 6.1 where register Reg_3 can be configured as either CBILBO or BILBO. Note that a test register usually has larger hardware area than a normal register (see Table 5.1 where ω is the area scaling factor over normal register). For example, CBILBOs have an area approximately twice that of the normal registers. One of

the main considerations in BIST techniques is, therefore, the extra area for the test circuitry. It has been shown that MISR registers can also be used to generate pseudo-random test patterns [10, 13, 18, 31, 49, 96]. This results in both reduction of testing times and reduction of extra registers which reduces hardware area. However, since the actual time required for a MISR register to obtain exhaustive pattern coverage is exponential with respect to the number of bits in the register, as shown in Figure 6.2, the test quality might be reduced. How to reduce this area overhead without sacrificing the test quality is the major concern of chapter.

Considering testability issues in high-level synthesis can lead to a more efficient exploration of the design space, thus resulting in a digital circuit that requires minimal BIST area overhead and has high test concurrency while guaranteeing the test quality. Therefore we move our research attention in this direction.

Investigation of high-level synthesis methods incorporating BIST characteristics has only recently received attention in the literature [8, 41, 42, 60, 63, 64, 69, 72]. With the assumption that every self-adjacent register needs to be modified to be a CBILBO register, which has high hardware area overhead, Avra [8] proposed a register allocation method to minimize the number of self-adjacent registers in the design. Papachristou et al. [63] presented a combined register and ALU allocation method that generates self-testable designs without any self-loops. They also presented a method of generating self-testable designs to include one specific configuration of a self-adjacent register [41, 64]. Recently, Parulkar et al. [69] proposed a data path allocation method which allocates operations, variables and data transfers in such a way that the functional constraints are satisfied and the area overhead required for BIST is minimal. The BIST methodology used is based on the concept of I-path [2] and the sharing of I-paths between modules. However the module allocation is carried out without any testability consideration.

The fact that the contents of signature registers (MISR) can be used as test patterns leads to the following advantages. First, the algorithm produces designs

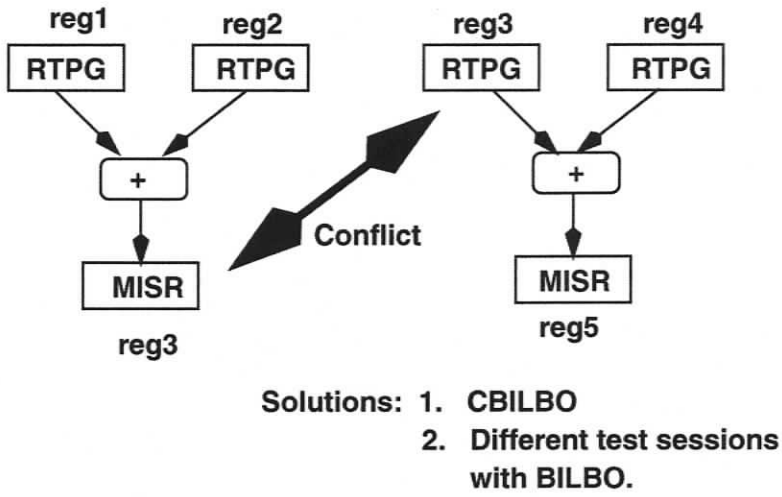


Figure 6.1. *Different BIST solutions*

with high test concurrency which reduces the overall testing time due to increased testing parallelism. Moreover, the number of extra registers for implementing BIST can be reduced. However, since the actual time required for a MISR register to obtain exhaustive pattern coverage is exponential with respect to the number of bits in the register, we consider such template as *incompletely embedded* module. In this chapter, we describe a high-level data path allocation algorithm which generates highly testable data path designs while maximizing the sharing of modules and test registers. Module allocation is guided by a testability balance principle where *incompletely embedded* modules can be mapped into the same function module that is *completely embedded.* In this way, the *incompletely embedded* module after allocation will be fully testable. The register allocation increases in the sharing degrees of registers. The fact that only a small number of registers is modified for BIST reduces the hardware area overhead.

This chapter is organized as follows. Section 6.2 describes the basic test methodology. The corresponding module and register allocation is described fully in section 6.3. In section 6.4, the stepwise refinement data path allocation algorithm is described. Furthermore, we present an improved version of the data path allocation algorithm. The details are described in sections 6.5, 6.6, 6.7, and 6.8 accordingly.

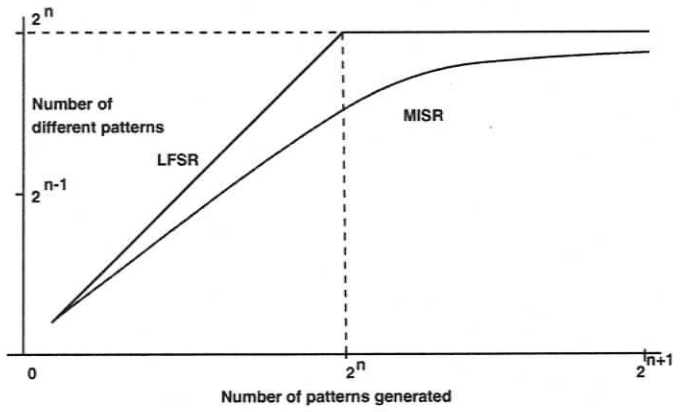


Figure 6.2. *The Effectiveness of MISR as RTPG*

Some estimated results compared with other conventional approaches are presented in section 6.9.

6.2 Basic Test Methodology

Our system takes a VHDL behavioral specification of a digital system and a set of design constraints as input and generates a Register-Transfer Level (RTL) hardware implementation which consists of a data path and a controller. The kernel of the system is an intermediate design representation, called Extended Timed Petri Net (ETPN), which can be used both for testability analysis and high-level synthesis [76].

The objective of the data allocation scheme presented in this chapter is to synthesize a BIST testable register-transfer data path. We base our model on the notion of structural testability at the Register-Transfer level. The key element of structural testability is the Extended Testable Functional Block (XTFBs) [41]. An XTFB consists of a functional block and a set of input and output registers. For the rest of the chapter, we assume there are two ports for a functional module. Let IR_j^L denote the set of registers connected to the left port of module M_j through only multiplexors and IR_j^R denotes the set of registers connected to the right port of module M_j through

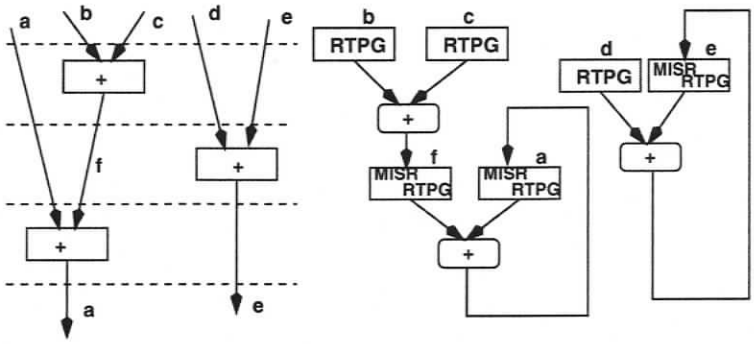


Figure 6.3. *The basic test methodology*

only multiplexors. The registers belonging to IR_j^L and IR_j^R are called *input registers* of M_j. Similarly let OR_j denote any register that collects data from the output port of a module through only multiplexors. Two registers at the input ports should be configured as PRPGs during test. One register connected to the output port should be configured as an MISR in the test mode. Figure 6.3 is an example to illustrate such test methodology.

Definition 7 *Given a register-transfer level data path, a* **BIST Embedding** *of a module M_j is a selection of three registers from IR_j^L, IR_j^R and OR_j, respectively, such that each input port of M_j receives test patterns from one of the selected registers and the output port transfers test responses to one of the selected registers.*

Many operations can be assigned to a module, each operation is called an instance of the module.

Definition 8 *The kth* **Instance** *of module M_j, denoted as $M_{j,k}$ is the kth operation executed by module M_j from the operations assigned to it.*

We use $IR_{j,k}^L$ ($IR_{j,k}^R$) to denote the registers connected to the left (right) port of module M_j corresponding to the instance k. Similarly, $OR_{j,k}$ denotes the register connected to the output port of module M_j corresponding to the instance k.

Definition 9 *The* **Temporal Multiplicity** *of module M_j, denoted as $TM(M_j)$, is the number of operations assigned to module M_j.*

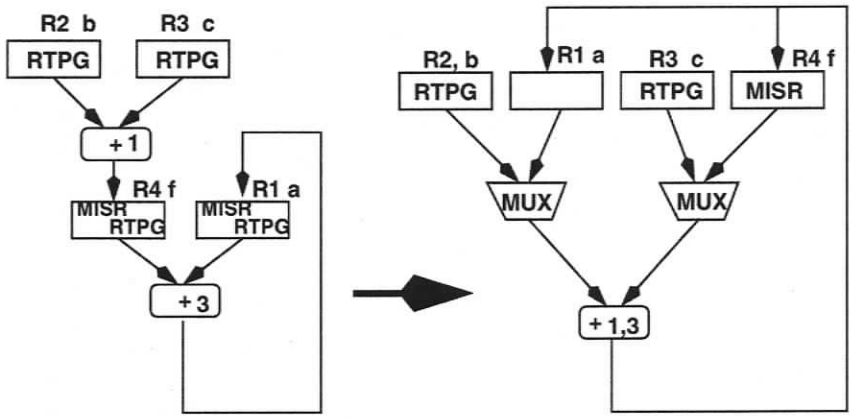


Figure 6.4. *Testability-driven module allocation*

The term *Temporal* here implies that the operations assigned to the same module must be scheduled at different temporal cycles.

Definition 10 *The* **Input Variable Set** *(***Output Variable Set***) of module M_j, denoted as $IVar_{M_j}$ ($OVar_{M_j}$), is the set of all the input (output) variables associated with each instance of module M_j.*

Although a basic BIST embedding typically needs three BIST registers, two PRPGs and one MISR, these BIST registers may be shared by other embeddings in the same data path. Traditionally, some of these registers have to be configured, either as a CBILBO register or as BILBO register. Both CBILBO and BILBO registers ensure high fault-coverage but require large hardware area. This chapter exploits the fact that MISR registers can generate test patterns, which lead to reduction of hardware area and test time, but with reduced test quality. The disadvantage can be avoided by careful design during the high-level data path allocation described below.

6.3 Data Path Allocation

The data path allocation task in high-level synthesis comprises two phases assuming the operation scheduling has been performed already:

- module allocations: assignment of operations to functional modules,
- register allocation: assignment of variables to registers.

In our approach, we begin with a default allocation generated by the VHDL compiler which assumes that each operation instance in the VHDL specification is mapped into an individual data path node. Then re-allocation is carried out by merging data path nodes until a one-by-one mapping to physical hardware is feasible. Note that the merge operations are semantic-preserving transformations.

Conventional allocation approaches often select and merge the data path nodes according to their connectivity or closeness, which aims to minimize interconnections and multiplexors. This generally results in a very hard to test design. In the self-testable version of the data path, registers are modified as BIST resources and interconnect is used to transfer test data to and from the modules. Modules are the hardware components that are under test in the proposed BIST test methodology. Hence the considerations for testability area overhead and test concurrency in data path allocation are fundamentally different from those during the conventional approaches.

6.3.1 Module allocation

Definition 11 *Let $E(R)$, where R is a register, be defined as follows. $E(R) = 0$ if R is modified as a MISR/RTPG; $E(R) = 2$ if R is a normal register and $E(R) = 1$, otherwise. The* **Embedded Degree** *$ED_m(M_j)$ of a module M_j is the sum of the registers from $IR_j^L \bigcup IR_j^R \bigcup OR_j$ which are not configured as MISR/RTPG registers. That is,*

$$ED_m(M_j) = \sum_{k=1}^{TM(M_j)} (ED(IR_{j,k}^L) + ED(IR_{j,k}^R) + ED(OR_{j,k})).$$

where $IR_{j,k}^L$, $IR_{j,k}^R$, and $O(OR_{j,k})$ are the registers connected to the left, right and output port of module M_j corresponding to the instance k.

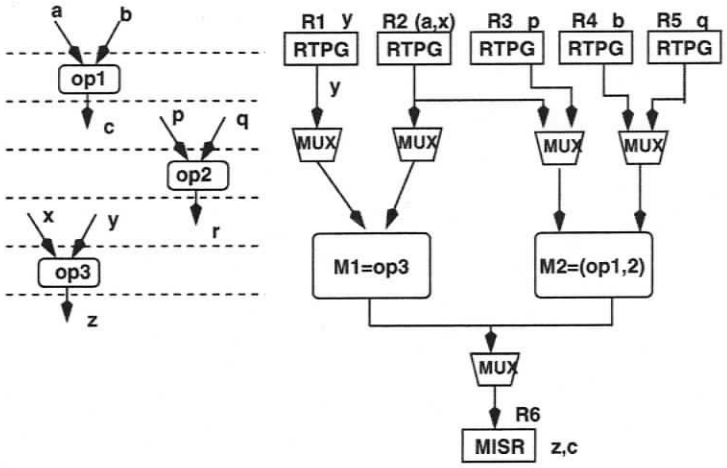


Figure 6.5. *Testability-driven register allocation*

Definition 12 *The module M_j is* **Completely Embedded** *if $ED_m(M_j) \geq 2TM(M_j)$. Otherwise this module is* **Incompletely Embedded**.

Definition 13 *Let $ED_m(M_i \bigcup M_j)$ be the embedded degree of the new module by merging module M_i and module M_j. The* **Incremental Embedded Degree** *of the merger, denoted as $\Delta ED_m(M_i, M_j)$, is*

$$\Delta ED_m(M_i, M_j) = ED_m(M_i \bigcup M_j) - ED_m(M_i) - ED_m(M_j).$$

With regards to module allocation, a testability balance principle has been used. The aim is to reduce or even eliminate *incompletely embedded* modules. The selection of nodes to be merged based on the balance principle where *incompletely embedded* modules can be mapped into the same function embedding with those *completely embedded* modules. In this way, the *incompletely embedded* module after allocation will be fully testable. For example, in the data path example in Figure 6.4, the addition node m_{+3} is *incompletely embedded* with $ED_m(m_{+3}) = 0$ and the addition node m_{+1} is *completely embedded* with $ED_m(m_{+1}) = 2$. Since they can share the same ALU and be scheduled in different control steps, they can be merged to generate a testable new node. Due to the new testable module by test patterns from R_2 and R_3

and signature response to R_4, register R_1 can be changed back to normal register. Therefore the embedded degree of the new module $ED_m(m_{+1} \bigcup m_{+3}) = 5$.

Algorithm 6.1 The Module Allocation Algorithm

1: Select the first k pairs of mergable modules ordered by their *Incremental Embedded Degree*
2: **for** k pairs modules **do**
3: Estimate the incremental hardware cost ΔH
4: Estimate the incremental embedded degree ΔED
5: Compute $\Delta C = \beta \cdot \Delta H - \gamma \cdot \Delta ED$
6: **end for**
7: Select the pair with smallest ΔC
8: Merge the select pair, modify the data path and update the functionality of registers accordingly

The module allocation based on the above ideas can be described in Algorithm 6.1. In each iteration, our module allocation algorithm selects k pairs of modules according to the testability balance allocation principle. A small value of k means that more emphasis is placed on increasing the embedded measure. We estimate the incremental hardware cost ΔH and the incremental embedded degree from k pair candidates of module allocations. We choose the pair with the smallest value of $\Delta C = \beta \Delta H - \gamma \Delta ED$, where β and γ are user-controlled parameters. We present the details of estimation for the execution time and its incremental cost, hardware cost and its incremental cost later, respectively. After the selection, we modify the data path and update the functionality of registers accordingly.

Discussion

It is possible that some *incompletely embedded* modules can not be merged during the synthesis process. In this case, it is still possible to make compromises on test quality,

test concurrency, and area overhead. We can keep such template which results in a reduction of overall test sessions and with smaller area overhead (but with reduced test quality). To guarantee the test quality, we can make the following choices. For area overhead oriented designs, we can configure such a template back to BILBO. For test concurrency oriented designs, we can configure such a template back to CBILBO.

6.3.2 Register allocation

Register allocation is mainly motivated by the idea [69] of maximizing sharing of test registers. An example is shown in Figure 6.5, register R_2 can be used for as RTPG for both modules M_1 and M_2 and register R_6 can be used as MISR for both modules M_1 and M_2. The idea is to merge test registers whose assigned variables have maximized *sharing degree* [69].

Definition 14 *The* **Sharing Degree** *[69]* $SD_{var}(v)$ *of a variable* v *is the sum of the number of modules for which* v *is an input variable and the number of module for which* v *is an output variable. If* $v \in IVar_{M_j}$, *let* $X_j^v = 1$, *else* $X_j^v = 0$; *if* $v \in OVar_{M_j}$, *let* $Y_j^v = 1$, *else* $Y_j^v = 0$. *Then* $SD_{var}(v) = \sum_{j=1}^{m}(X_j^v + Y_j^v)$, *where* m *is the total number of modules assigned.*

Definition 15 *The* **Sharing Degree** $SD_{reg}(R)$ *of a register* R *is defined as follows:* $SD_{reg}(R) = \sum_{v \in R}(SD_{var}(v))$.

The sharing degree reflects the number of modules for which the register can be configured as a RTPG and the number of modules for which it can be configured as a MISR.

We also use the notion of *Increment Sharing Degree* [69], denoted by $\Delta SD_{reg}(R, v)$, which measures the increase in the sharing degree of a register R as a result of assigning variable v to it. Consider a register R that has been assigned some variables. Let the sharing degree of R after another variable v is assigned to it be denoted and

calculated by

$$SD_{reg}(R \bigcup \{v\}) = SD_{reg}(R) + SD_{var}(v) - \sum_{j=1}^{m}(X_j^R * X_j^v + Y_j^R * Y_j^v).$$

Definition 16 *The* **Incremental Sharing Degree** $\Delta SD_{reg}(R, v)$ *is the increase in the sharing degree of a register* R *as a result of assigning variable* v *to it, where*

$$\Delta SD_{reg}(R, v) = SD_{reg}(R \bigcup \{v\}) - SD_{reg}(R).$$

Using this measure the register allocation is guided by choosing mergers that result in large increases in the sharing degrees of registers over those resulting in smaller increases. This would result in registers with high sharing degrees, thereby requiring a smaller number of BIST registers globally in the design. The register allocation based on the above ideas can be described in Algorithm 6.2.

Algorithm 6.2 The Register Allocation Algorithm

1: Do lifetime analysis of variables
2: **for** all mergable test registers **do**
3: Estimate the incremental sharing degree ΔSD
4: **end for**
5: Select one pair with the largest ΔSD
6: Merge the select pair, modify the data path accordingly

We begin by performing the lifetime analysis of variables which are the interval between their *birth times*, the control steps in which variables are first defined, and their *death times*, the control steps in which variables are last used. In each iteration, for all mergable test registers (their lifetimes do not overlap), we estimate the incremental sharing degree ΔSD and choose one pair with largest ΔSD. Note that another criteria for mergable test registers is that they are of the same type (RTPG, MISR or RTPG/MISR). After the selection, we merge the selected pair and modify the data path accordingly.

6.3.3 Estimation of performance and cost

The hardware cost H is based on calculation of the cost of the data path. The cost of the data path is calculated according to the cost of each data path unit. The cost of data path units which performs logic, arithmetic, or storage operations is given by the corresponding module parameters stored in the module library. The cost of data path units for communication such as arcs and bus vertices depends on the placement of the components as well. To make a more accurate estimation, we follow the floor-planning algorithm proposed in [76] to estimate the hardware cost which takes into account the geometrical information. This algorithm basically makes use of simple heuristics based on the connectivity between the data path vertices. These heuristics provide a relatively precise indication of how much it will cost in the hardware implementation.

Given a floor-plan, the cost for an ETPN data path is estimated as follows:

$$H = \sum_i Area(V_i) + \sum_j Len(A_j) \times Wid(A_j),$$

- $Area(V_i)$ is the area cost of the module corresponding to a data path node V_i.
- $Len(A_j)$ is the length of the connection represented as a data path connection A_j.
- $Wid(A_j)$ is the width of the connection represented as A_j, which is the bit width of the connection multiplied by a given weighted factor.

Based on the above estimations and test register library of LSI [72] described in Table 5.1, when a pair of modules or registers is merged, ΔH is equal to the increase in data path area cost. The scaling factor ω listed in Table 5.1 are used for the estimation due to some registers being modified to test registers.

6.4 Data Path Allocation Algorithm

Our data path allocation algorithm accepts a scheduled ETPN design representation as input and generates a scheduled BIST testable data path.

Generally speaking, our data path allocation is an iterative one. In each iteration, the algorithm selects pairs of modules and registers in the data path and merges them to generate a new data path. Here merging two modules into one implies that all the operations which were assigned to the two modules are reassigned into one single module. Therefore, mergable modules mean that all the operations which were assigned to these modules must be scheduled in a different control step. A similar argument holds for the case of register merger. Mergable registers mean that the lifetime of all variables which were assigned in these registers merging. Note that merger of nodes (modules and registers) leads to a reduction in the number of hardware components in the design and do not lead to an increase in the execution time of the data path. The iterative synthesis algorithm is described as Algorithm 6.3.

First of all, we accept a scheduled ETPN design representation as input and perform a simple default allocation. Then for all registers, we have to modify them to different test registers (RTPG, MISR, or RTPG/MISR) according to their functionalities. According to the data path allocation technique presented above, we perform module and register allocation, respectively. Some test registers have been modified back to normal registers after the data path allocation. η is a user-controlled parameter. Note that the merging criterion for test registers is to maximize sharing degree while that for normal registers is to minimize incremental hardware cost. This process is repeated until no merger is possible in the data path.

Discussion

So far, we have presented an efficient BIST data path allocation method. However there are still some possibilities to improve the data path allocation algorithm such as having the pre-allocation optimization, introducing redundant transformations etc. All these techniques together with the improved data path allocation are presented in the later part of this chapter.

Algorithm 6.3 The Self-Testable Data Path Allocation Algorithm

1: Perform a simple default allocation
2: **for** all normal registers **do**
3: Do lifetime analysis of variables
4: **end for**
5: **for** all registers **do**
6: Change to test registers according to their functionalities
7: **end for**
8: **repeat**
9: Perform module allocation
10: Perform test register allocation
11: **for** all normal registers **do**
12: Estimate the incremental hardware cost ΔH
13: Compute $\Delta C = \eta \cdot \Delta H$
14: **end for**
15: Select the pair with smallest ΔC
16: Merge the selected pair, modify the data path accordingly
17: **until** no merger exists

6.5 Pre-Allocation Optimization

Before we perform the self-testable data path allocation algorithm, every register has to be modified to be a test register with different functionality. In order to optimize such modification, in our approach as described in chapter 5, an optimal modification idea based on an Integer Linear Programming formulation to find BIST embeddings in the data path is prepared for the data path allocation algorithm such that the cost of modification to RTPGs, MISRs, BIBLOs and CBIBLOs respectively is minimum.

6.6 Redundant Transformations with Properties

Based on the definition described in [68, 71, 73], an I-module is a module in the data path whose operation type has an identity value associated with it and one of it operands or input ports is set to a constant corresponding to the identity value. An I-module can be added in the data path without changing the functionality of the data path. Parulkar et al. introduce the definition of *I-transformation* as follows. Introduce an I-module o_k^I between any two operation nodes o_i and o_j. The module is introduced such that the output of o_i becomes one of the inputs of o_k^I and the output of o_k^I becomes the input of o_j. The other input of o_k^I corresponds to a constant, the identity value of the I-module operation. It is easy to see that the data path obtained after the I-transformation is functionally equivalent to the original data path.

In [68, 71, 73], they introduce redundant paths that can be used to either transport test data, or enable sharing of non-redundant paths in transporting test data. Correspondingly they present two types of I-transformations, namely type 1 and type 2 transformations. Although the introduction of an I-module does not change the functional behavior, it does change the structure and some properties of a data path. The following two important results from [68, 71, 73] are essential to characterize the effect of the I-transformations on the properties of the data path when we want to apply these two types of transformations in an efficient and beneficial way.

Property 1 *If an I-transformation is introduced in a non-critical path, then the control step or minimum latency for which the data path can be scheduled remains unchanged.*

Property 2 *An I-transformation increases the number of variables in the data path by 1, but the minimum number of registers required to store all the variables remains the same.*

From their results, we see clearly that I-modules can be introduced such that the number of modules, number of registers and the control step (the latency) remains the

same. However, the I-transformation can have an adverse effect on the interconnect complexity [68, 71, 73]. In order to trade-off the reduction in BIST area overhead and the increased interconnect complexity, we make use of an accurate hardware estimation based on the floor-planning algorithm described later.

6.7 Improved Module and Register Allocation

In our improved approach, the basic ideas are the same as in the original algorithm presented as above.

We still make use of the same basis that the contents of signature registers can also be used as test patterns [49]. A signature register collects test responses of a module as it is intended. At the same time, the contents or patterns of the signature register can be applied to test other modules or the module being analyzed by the signature register. According to the analysis in [49], signature registers can be effectively used as pseudo-random pattern generators. This results in a reduction of overall test sessions through increased parallelism in testing. The number of extra registers could also potentially be reduced. The detailed idea of this basic test methodology has been discussed above.

However, since the actual time required for a MISR register to obtain exhaustive pattern coverage is exponential with respect to the number of bits in the register, we may consider this template, directly modified from BIBLO or CBIBLO test registers resulted from the pre-allocation optimization, as an *incompletely embedded* module. Although it can generate test patterns, which lead to reduction of hardware area and test time, it also leads to reduced test quality. This disadvantage can be avoided by careful design during the high-level data path allocation described below.

6.7.1 Improved module allocation

With regards to module allocation, a testability balance principle has been used in previous chapters. The aim is to reduce or even eliminate the *incompletely embedded* modules. The selection of modules to be merged is based on the balance principle where the *incompletely embedded* modules can be mapped into the same function embedding with those *completely embedded* modules. The re-allocation is carried out by semantic-preserving transformations such as *merger* which compact the data path nodes until a one-by-one mapping to physical hardware is feasible. In this way, the *incompletely embedded* module after allocation will be fully testable.

The module allocation based on the above ideas can be described in Algorithm 6.4. In each iteration, k pairs of modules according to the testability balance allocation principle are selected first. If we want to put more emphasis on increasing the embedded measures, we can use a small value of k. The incremental hardware cost ΔH and the incremental embedded degree from k pair candidates of module allocations are estimated in next step. Then we choose the pair with the smallest value of $\Delta C = \beta\Delta H - \gamma\Delta ED$, where β and γ are user-controlled parameters. The details of estimation for the execution time and its incremental cost, hardware cost and its incremental cost have been described fully in section 6.3. After the selection, we modify the data path and update the functionality of the registers accordingly.

It is possible that some *incompletely embedded* modules can not be merged during the synthesis process. In this case, we apply two types of I-transformations for different modules. For those *incompletely embedded* modules whose $ED = 0$, the type 1 I-transformation is applied with the same type module. For those *incompletely embedded* modules whose $ED = 1$, the type 2 I-transformation is applied with the same type module. After the redundant transformations, all *incompletely embedded* modules will be merged and become fully testable. In this way, we can avoid using BIBLO and CBIBLO instead to cover those *incompletely embedded* modules.

Algorithm 6.4 The Module Allocation Algorithm

1: **repeat**
2: select the first k pairs of mergable modules ordered by the Embedded Degree difference
3: **for** k pairs modules **do**
4: estimate the incremental hardware cost ΔH
5: estimate the incremental embedded degree ΔED
6: compute $\Delta C = \beta \cdot \Delta H - \gamma \cdot \Delta ED$
7: **end for**
8: select the pair with smallest ΔC
9: merge the select pair, modify the data path and update the functionality of registers accordingly
10: **if** no completely embedded modules **then**
11: **if** module $ED = 0$ **then**
12: apply the type 2 I-transformation
13: **end if**
14: **if** module $ED = 1$ **then**
15: apply the type 1 I-transformation
16: **end if**
17: **end if**
18: **until** no incompletely embedded modules

6.7.2 Improved register allocation

The improved register allocation is mainly based on the same idea of maximizing sharing of test registers presented previously. The idea is to merge test registers whose assigned variables have maximized *sharing degree* [69]. Therefore for maximizing the sharing of RTPG between the input ports of modules, a register allocation is desirable such that for each test register R_i the number of input variable sets with which it has at least one variable in common is maximized. Similarly the number of output variable sets with which each R_i has at least one common variable should be maximized. The sharing degree reflects the number of modules for which the register can be configured as a RTPG and the number of modules for which it can be configured as a MISR. We also use the notion of *Increment Sharing Degree* [69], denoted by $\Delta SD_{reg}(R, v)$, which measures the increase in the sharing degree of a register R as a result of assigning variable v to it. Using this measure the register allocation is guided by choosing mergers that result in large increases in the sharing degrees of registers over those resulting in smaller increases. This would result in registers with high sharing degrees, thereby requiring a smaller number of BIST registers globally in the design.

The register allocation algorithm [100, 101] begins by performing the lifetime analysis of variables which are the interval between their *birth times*, the control steps in which variables are first defined, and their *death times*, the control steps in which variables are last used. In each iteration, for all mergable test registers (their lifetimes do not overlap), we estimate the incremental sharing degree ΔSD and choose one pair with largest ΔSD. Note that another criteria for mergable test registers is that they are of the same type (RTPG, MISR or RTPG/MISR). After the selection, we merge the selected pair and modify the data path accordingly.

Table 6.1. *Estimated results on the Diffeq benchmark*

Synthesis techniques	Control step	Register	Modular				BIST estimated area overhead	%Decrease in total estimated area
			Mul	Add	Sub	Les		
SWT-AWT	6	5	3	1	1	1	29.25%	-
SFT-AWT	6	5	3	1	1	1	21.40%	5.50 %
SFT-AFT	6	5	3	1	1	1	18.66%	6.53 %
Basic method	6	5	3	1	1	1	17.04%	7.01%
Improved method	6	5	3	1	1	1	15.04%	8.56 %

6.8 Improved Data Path Allocation

Our improved allocation algorithm accepts a scheduled ETPN design representation as input and generates a BIST testable data path. Similarly, our allocation algorithm is an iterative one. In each iteration, the algorithm selects pairs of modules and registers in the data path and merges them to generate a new data path. The iterative allocation algorithm is described in the Algorithm 6.5.

First of all, we accept a scheduled ETPN design representation as input and perform a simple default allocation. Then for all registers, we have to modify them to different test registers (RTPG, MISR, BILBO or CBILBO then convert to RTPG/MISR) by performing the above resource optimization according to their functionality. Module and register allocation then are performed, respectively. Some of the test registers have been modified back to normal registers after the data path allocation. η is a user-controlled parameter. Note that the merging criterion for test registers is to maximize sharing degree while that for normal register is to minimize incremental hardware cost. This process is repeated until no merger is possible in the data path.

Algorithm 6.5 The Improved Data Path Allocation Algorithm

1: perform a simple default allocation
2: **for** all normal registers **do**
3: perform lifetime analysis of variables
4: **end for**
5: **for** all registers **do**
6: perform the pre-allocation optimization
7: change CBIBLO or BIBLO test registers to LSFR/MISR according to their functionality
8: **end for**
9: **repeat**
10: perform improved module allocation
11: perform improved register allocation
12: change back test registers to normal one if any
13: **for** mergeable normal registers **do**
14: estimate the incremental hardware cost ΔH
15: compute $\Delta C = \eta \cdot \Delta H$
16: **end for**
17: select the pair with smallest ΔC
18: merge the select pair, modify the data path
19: **until** no merger exists
20: perform high-level BIST configuration scheme

6.9 Cost Estimation

To demonstrate the use of the proposed technique, the estimated cost according to the CAMAD [76] and LSI [68] library have been conducted with a variety of benchmarks such as the 2nd order differential equation, Diffeq [75]; the auto regression filter element, AR filter [46]; an 8-point FIR filter, FIR filter [65] and the elliptic wave filter, EWF filter [52]. The proposed data path allocation algorithm is applied to these benchmarks and the results are compared with our previous and those conventional approaches.

Tables 6.1, 6.2 and 6.3 show the characteristics of the data paths synthesized from the three selected examples, Diffeq [75], EWF [52] and FIR filter [65], respectively. In the tables, SWT-AWT stands for the approach with a traditional scheduling (as-soon-as-possible) that does not consider BIST, and traditional register and module allocations that do not consider BIST. SFT-AWT denotes the approach with scheduling described in [72] with BIST considerations, and traditional module and register allocations without BIST consideration. SFT-AFT is the scheduling approach presented in [72] with BIST considerations, and module and register allocations that search for low BIST area overhead [70]. We compare the proposed approach, namely Basic method (presented in section 6.4 and [100, 101]) and its improved version (presented in section 6.8 and [107, 110] with regards to the number of control steps, register and functional module requirements and total estimated BIST area overhead. Since this algorithm is only for data path allocation, assuming the operation scheduling has been performed in advance, we use the same operation scheduling as our compared methods for three benchmarks in order to make the comparison fair. In this case we use the same scheduling approach as the SFT-AWT and SFT-AFT methods. The estimated results indicate the advantages of our approach over conventional ones and our previous ones in term of total estimated BIST area and hardware costs.

Table 6.2. *Estimated results on the EWF benchmark*

Synthesis techniques	Control step	Register	Modular		BIST estimated area overhead	%Decrease in total estimated area
			Add	Mul		
SWT-AWT	14	10	5	3	16.28%	-
SFT-AWT	14	8	5	3	6.94%	10.54 %
SFT-AFT	14	8	5	3	6.90%	10.13%
Basic method	14	8	5	3	6.79%	11.06%
Improved method	14	8	5	3	4.96%	15.46%

Table 6.3. *Estimated results on the FIR benchmark*

Synthesis techniques	Control step	Register	Modular		BIST estimated area overhead	%Decrease in total estimated area
			Add	Mul		
SWT-AWT	5	8	4	4	46.06%	-
SFT-AWT	5	8	4	4	32.00%	0.6 %
SFT-AFT	5	8	4	4	27.42%	4.0%
Basic method	5	8	4	4	20.14%	6.3%
Improved method	5	8	4	4	16.59%	8.9%

6.10 Summary

In this chapter, we present a high-level data path allocation algorithm to generate highly testable data path designs while maximizing the sharing of modules and test registers. The data path allocation is performed after the operation scheduling is fixed. In order to further explore the testability issues to achieve more self-testable designs, we consider integrating testability consideration into both operation scheduling and data path allocation stages. Two different built-in self-test synthesis approaches for integrating both operation scheduling and data path allocation are presented in later chapters.

Chapter 7

Integrated Synthesis Algorithm

7.1 Introduction

Considering testability issues at high-level synthesis can lead to a more efficient exploration of the design space, thus resulting in a digital circuit that requires minimal BIST area overhead and has high test concurrency while meeting area, throughput and other requirements. In this chapter, we integrate operation scheduling and data path allocation by performing them simultaneously so that the effects of high-level synthesis on testability are exploited more effectively and completely.

Incorporating BIST characteristics into the high-level synthesis process has only recently been investigated [8, 41, 63, 64, 69]. However all above listed work assumed that the operation scheduling has been completed prior to data path allocation. Harris et al. [42, 60] first presented a scheduling and allocation method that improves the test concurrency and hence the test time required to test data paths. The effect on the number of BIST resources was not considered in work. Recently, Parulkar et al. [72] presented a scheduling technique and a module assignment technique that produce schedules to ensure a small number of BIST resources and area overhead.

In this chapter, we describe a high-level test synthesis algorithm for both operation scheduling and data path allocation. It generates highly testable data path designs while maximizing the sharing of test registers, which means that only a small number of registers is modified for BIST. The algorithm also produces designs with high test

concurrency, thereby decreasing test time. In our approach, the data path allocation uses the same approach presented in chapter 6. Module allocation is guided by a testability balance principle where *incompletely embedded* modules can be mapped into the same function module that is *completely embedded.* In this way, the *incompletely embedded* module after allocation will be fully testable. The register allocation is guided by an incremental sharing measurement which chooses merges that result in large increases in the sharing degrees of registers. Scheduling, on the other hand, is carried out by rescheduling transformations which change the default scheduling to improve testability. Contrary to other approaches in which the scheduling and allocation tasks are performed independently, our approach integrates scheduling and allocation by performing them simultaneously so that the effects of scheduling and allocation on testability are exploited more effectively and completely.

This chapter is organized as follows. Section 7.2 briefly describes data path allocation including module and register allocation. In section 7.3, the stepwise refinement synthesis algorithm and the introduction of scheduling constraints imposed by the process of data path synthesis are described fully. Furthermore, we also present an improved version of the synthesis algorithm in section 7.5 and 7.6, with some estimated results compared with other conventional approaches in section 7.7, respectively.

7.2 Data Path Allocation

In chapter 6, we described a high-level data path allocation algorithm which generates highly testable data path designs while maximizing the sharing of modules and test registers. Module allocation is guided by a testability balance principle where *incompletely embedded* modules can be mapped into the same function module that is *completely embedded.* In this way, the *incompletely embedded* module after allocation will be fully testable. The register allocation is primarily decided by the idea of maximizing sharing of test registers. There are two terms that were introduced

in the previous chapter, namely *sharing degree* and *incremental sharing degree.* The *sharing degree* reflects the number of modules for which the register can be configured as RTPG and the number of modules for which it can be configured as a MISR. Accordingly the *Increment Sharing Degree* measures the increase in the *sharing degree* of a register as a result of assigning a variable to it. Using these measures, the register allocation is guided by choosing mergers that result in larger increases in the sharing degrees of registers over those resulting in smaller increases. This would result in registers with high sharing degrees, thereby requiring a smaller number of BIST registers globally in the design. The detailed algorithm can be found in chapter 6. Additionally, we presented an improved version of the algorithm in different aspects in chapter 6 as well.

7.3 Test Synthesis Algorithm

7.3.1 The basic algorithm

Our test synthesis algorithm [102, 104] accepts an unscheduled ETPN design representation as input and generates a BIST testable data path.

Generally speaking, our test synthesis is an iterative one. In each iteration, the algorithm selects pairs of modules and registers in the data path and merges them to generate a new data path. Here merging two modules into one implies that all the operations which were assigned to the two modules are reassigned into one single module. Therefore, merging two modules into one imposes a scheduling constraint that all the operations which were assigned to these modules must be scheduled in different control step. A similar argument holds for the case of register merger. Merging two registers into one imposes a scheduling constraint that the lifetimes of all variables which were assigned in these registers must be disjoint. Note that the merger of nodes (modules and registers) leads to a reduction in the number

of hardware components in the design, whereas the additional scheduling constraints may lead to an increase in the execution time of the data path. The iterative synthesis algorithm is described as Algorithm 7.1.

First of all, since we accept an unscheduled ETPN design representation as input, we have to perform a simple default scheduling/allocation. Then for all registers, we need to modify them to different test registers according to their functionalities. According to the data path allocation technique presented in chapter 6, we perform module and register allocation, respectively. Rescheduling, if needed, is performed by a merge-sort algorithm, after we finish the lifetime analysis of variables. Since some of test registers have been modified back to normal registers after the data path allocation, we have to do the corresponding allocation for them as well where τ and η are two user-controlled parameters. This process is repeated until no further merger possible in the data path.

7.3.2 Operation scheduling

As described above, when two modules are merged, the operations executed on these modules must be scheduled in different control steps so that they can share the same component. Similarly for registers, the variables stored in these registers must be disjoint. We present the rescheduling transformation which is performed by a merge-sort algorithm originally proposed in [50]. These transformations change locally the execution orders of some operations in the current schedule in order to improve the testability and satisfy the scheduling constraints imposed by the merger.

7.3.3 Rescheduling by module merger

Based on the notation introduced in [50], suppose we would like to merge two modules m_i and m_j. Assume the s operations scheduled for execution in module m_i are $o_{i_1}, o_{i_2}, \cdots, o_{i_s}$ and t operations scheduled for execution in module m_j are $o_{j_1}, o_{j_2}, \cdots, o_{j_t}$.

Algorithm 7.1 The Self-Testable Synthesis Algorithm

1: Perform a simple default scheduling/allocation
2: **repeat**
3: **for** all registers **do**
4: Perform the pre-synthesis optimization
5: Change CBILBO or BILBO test registers to LSFR/MISR according to their functionalities
6: **end for**
7: Perform module allocation
8: Perform register allocation
9: **for** selected module or test registers **do**
10: Perform rescheduling transformations imposed by data path synthesis using a merge-sort algorithm
11: **end for**
12: **for** all normal registers **do**
13: Do lifetime analysis of variables
14: Estimate the incremental execution time ΔE
15: Estimate the incremental hardware cost ΔH
16: Compute $\Delta C = \tau \cdot \Delta E + \eta \cdot \Delta H$
17: **end for**
18: Select the pair with smallest ΔC
19: Merge the select pair, modify the data path accordingly
20: **until** no merger exists
21: Perform high-level BIST configuration scheme

Let us first consider the simplest case where $s = t = 1$, then the operations o_{i_1} and o_{j_1} must be scheduled in different control steps for execution. If they have been scheduled already in different control steps, we do not need to introduce any scheduling constraints. Otherwise, we have to consider two possibilities:

- execute o_{i_1} before o_{j_1},
- execute o_{j_1} before o_{i_1}.

We select the pair which results in the smallest increase in the length of the critical path of the data path. Rescheduling is performed by introducing dummy control steps (places in a Petri net) [76] if necessary so as to change the default scheduling of operations.

For the general case in which operations $o_{i_1}, o_{i_2}, \cdots, o_{i_s}$ are scheduled for execution in module m_i and $o_{j_1}, o_{j_2}, \cdots, o_{j_t}$ for execution in module m_j, respectively, we know that there is a sequential order among $o_{i_1}, o_{i_2}, \cdots, o_{i_s}$ because they already share the module m_i. Without loss of generality, we assume the order to be $o_{i_1} \rightarrow o_{i_2} \rightarrow \cdots \rightarrow o_{i_s}$. Similarly, let the sequential order for operations in module m_j to be $o_{j_1} \rightarrow o_{j_2} \rightarrow \cdots \rightarrow o_{j_t}$. The main goal is to merge these two sequential orders into one. First we examine the pair of operations o_{i_1} and o_{j_1} to decide the more suitable order which results in the smaller increase in the length of the critical path of the data path. Then we decide the remainder using a merge-sort heuristic to achieve a single execution order.

7.3.4 Rescheduling by register merger

With regards to the scheduling constraints imposed by a register merger, we can use a similar merge-sort algorithm. For this case, we still use the same notation introduced in [50], by supposing we want to merge two registers r_i and r_j. Let $v_{i_1}, v_{i_2}, \cdots, v_{i_s}$ denote the s variables which are stored in register r_i and $v_{j_1}, v_{j_2}, \cdots, v_{j_t}$ denote the t variables which are stored in register r_j. Let us also first consider the simplest case

where $s = t = 1$, then the operations which determine the lifetimes of variables v_{i_1} and v_{j_1} must be scheduled in a way that the lifetimes of v_{i_1} and v_{j_1} do not overlap.

First we examine whether some operations, which determine the lifetime of v_{i_1} and v_{j_1}, have been scheduled to be executed in a certain order. Based on our observations, it is very natural to check first whether these operations are scheduled such that two lifetimes are never disjoint. There are two cases:

- there are two arcs, one is from some of the operations that determine the lifetime of v_{i_1} to the operations that determine the lifetime of v_{j_1}, the other is from some of the operations that determine the lifetime of v_{j_1} to the operations that determine the lifetime of v_{i_1}.
- there is an operation which uses both of the values of v_{i_1} and v_{j_1} as inputs.

If both cases do not hold, a scheduling constraint to ensure that the lifetime of v_{i_1} and v_{j_1} will be disjoint is added to the representations. If there are no arcs between any of the operations which determine the lifetime of v_{i_1} and any of the operations which determine the lifetime of v_{j_1}, we consider two possibilities:

- variable v_{i_1} expires before variable v_{j_1} is created.
- variable v_{j_1} expires before variable v_{i_1} is created.

Our algorithm selects, between the two possibilities, the one which results in the smaller increase in the length of the critical path of the data path.

For the general case in which variables $v_{i_1}, v_{i_2}, \cdots, v_{i_s}$ and $v_{j_1}, v_{j_2}, \cdots, v_{j_t}$ share registers r_i and r_j, respectively, the procedure for adding lifetime disjoint arcs is similar to that of the case of module merger.

7.4 Estimation of Performance and Costs

For a given data path, the minimum execution time E is equal to the length of the critical path which consists of a sequence of control states which dominates the time

needed for a token to flow from the initial state to the final state [76].

The method to detect the critical path is based on the reachability tree of the Petri net model [77]. A reachability tree represents the reachability set of the given Petri net. That is, it shows all markings which can be reached from the initial markings. The critical path algorithm first constructs such a reachability tree and, analyzing the reachability tree, extracts the critical path.

The hardware cost H is based on the calculation of the cost of the data path. The cost of the data path is calculated according to the cost of each data path unit. The cost of data path units which perform logic, arithmetic, or storage operations is given by the corresponding module parameters stored in the module library. The cost of data path units for communication such as arcs and bus vertices depends on the placement of the components as well. To make a more accurate estimation, as described in chapter 6, we combine the floor-planning algorithm proposed by Peng et al. [76] and test register library of LSI [68] to estimate the hardware cost which takes into account the geometrical information. The scaling factor ω listed in Table 5.1 in [100, 101] is used for the accurate estimation due to some registers being modified to test registers and the increased interconnect complexity. Based on the estimations, when a pair of modules or registers is merged, ΔH is equal to the increase in data path area cost.

But for ΔE, it is more complicated than estimating ΔH. As we described above, for a given data path, the minimum execution time E is equal to the length of the critical path. When a pair of modules or registers is merged, we have to estimate the length of the critical path, which is mainly determined by the scheduling of the data path after a module or register merger. Before we perform rescheduling transformations, we do not have any knowledge about the length of the critical path after a module or register merger.

To reduce the high computational complexity of estimation on the length of the critical path, we propose an efficient method described as follows. Suppose we would

like to merge two modules m_i and m_j. Assume the s operations scheduled for execution in module m_i are $o_{i_1}, o_{i_2}, \cdots, o_{i_s}$ and t operations scheduled for execution in module m_j are $o_{j_1}, o_{j_2}, \cdots, o_{j_t}$. It is easy to know that at most there are $\min(s,t)$ operations to be possibly scheduled in the same control step. It means that we have to introduce at most $\min(s,t)$ dummy control steps [76]. We can consider this as the incremental length of the critical path. If the number of operations scheduled in the same control step is known, we can estimate the incremental length of the critical path more accurately. A similar approach can be extended to the case of register merger.

7.4.1 Discussion

So far, we have presented an efficient built-in self-testable synthesis algorithm to further consider built-in self-test issues in both operation scheduling and data path allocation process. However there are still some possibilities to improve the algorithm such as having the pre-allocation optimization, introducing redundant transformations for both data path allocation and operation rescheduling, etc. All these techniques together with the improved test synthesis algorithm [108, 110, 111] are presented in the remainder of this chapter.

7.5 Improved Test Synthesis Algorithm

Generally speaking, our improved version synthesis algorithm [108, 110, 111] is almost the same as Algorithm 7.1. The main differences are the pre-allocation optimization, improved module and register allocations, and some enhancement techniques such as introducing redundant transformations for the operation rescheduling and data path allocation.

7.5.1 Pre-allocation optimization

In the integrated synthesis algorithm, before we perform the data path allocation, every register has to be modified to be a test register with different functionality. In our approach [100, 101, 102, 104], as discussed in chapter 5, an optimal modification idea based on an Integer Linear Programming formulation to find BIST embeddings in the data path is prepared for the allocation algorithm such that the cost of modification to RTPGs, MISRs, BIBLOs and CBIBLOs respectively is minimized.

7.5.2 Improved module and register allocation

With regards to module allocation, a testability balance principle has been used [100, 101, 102, 104], as described in chapter 6. The aim is to reduce or even eliminate the *incompletely embedded* modules. The selection of modules to be merged is based on the balance principle where the *incompletely embedded* modules can be mapped into the same function embedding with those *completely embedded* modules. The reallocation is carried out by semantic-preserving transformations such as *merger* which compacts the data path nodes until a one-to-one mapping to physical hardware is feasible. In this way, the *incompletely embedded* module after allocation will be fully testable.

The module allocation based on the above ideas is described in Algorithm 7.2. In each iteration, our module allocation algorithm selects k pairs of modules according to the testability balance allocation principle. A small value of k means that more emphasis is placed on increasing the embedded measure. Then we estimate the incremental hardware cost ΔH and the incremental embedded degree from k pairs candidates of module allocations. Then we choose the pair with the smallest value of $\Delta C = \beta\Delta H - \gamma\Delta ED$, where β and γ are user-controlled parameters. The details of estimation for the execution time and its incremental cost, hardware cost and its incremental cost can be found in [100, 101, 102, 104]. If the pair to be

Algorithm 7.2 The Improved Module Allocation Algorithm

1: **repeat**
2: select the first k pairs of mergable modules ordered by the Embedded Degree difference
3: **for** k pairs modules **do**
4: estimate the incremental hardware cost ΔH
5: estimate incremental embedded degree ΔED
6: compute $\Delta C = \beta \cdot \Delta H - \gamma \cdot \Delta ED$
7: **end for**
8: select the pair with smallest ΔC
9: **if** the pair is from the same control step **then**
10: **if** the incomplete module $ED = 0$ **then**
11: apply the type 2 I-transformation
12: **end if**
13: **if** the incomplete module $ED = 1$ **then**
14: apply the type 1 I-transformation
15: **end if**
16: **end if**
17: merge the select pair, modify the data path and update the functionality of registers accordingly
18: **if** no completely embedded modules **then**
19: **if** module $ED = 0$ **then**
20: apply the type 2 I-transformation
21: **end if**
22: **if** module $ED = 1$ **then**
23: apply the type 1 I-transformation
24: **end if**
25: **end if**
26: **until** no incompletely embedded modules

merged is from the same control step, we have to introduce dummy places in the later operation rescheduling process which have negative impacts or increase the control steps leading to longer execution time or slow performance. Therefore two types of I-transformations for different modules are introduced to avoid the type of operation pair, but still keep the same useful merger features. After the selection, we modify the data path and update the functionality of registers accordingly.

It is possible that some *incompletely embedded* modules can not be merged during the allocation and synthesis process. In this case, we apply two types of I-transformations for different modules. For those *incompletely embedded* modules whose $ED = 0$, the type 1 I-transformation is applied with the same type module. For those *incompletely embedded* modules whose $ED = 1$, the type 2 I-transformation is applied with the same type module. After the redundant transformations, all *incompletely embedded* modules will be merged and become fully testable.

7.5.3 Register allocation

Register allocation is driven by an attempt to maximize the sharing of test registers [68]. The idea is to merge test registers whose assigned variables have maximized *sharing degree* [68], reflects the number of modules for which the register can be configured as RTPG and the number of modules for which it can be configured as a MISR, and *Increment Sharing Degree* [68], measures the increase in the sharing degree of a register as a result of assigning variables to it. The details can be found in [100, 101, 102, 104], also discussed in chapter 6.

7.6 Improved Operation Rescheduling

As pointed out before, when two modules are merged, the operations executed on these modules must be scheduled in different control steps so that they can share the same component. Similarly for registers, the variables stored in these registers

must be disjoint. We have presented the rescheduling transformations by a merge-sort algorithm in section 7.3. These transformations change locally the execution orders of some operations in the current schedule in order to improve the testability and satisfy the scheduling constraints imposed by the merger.

Before we present the main improvement or enhancement techniques, there are two important concepts that have to be introduced first from [68] although they have been widely used.

Definition 17 *The* **as-soon-as-possible** *value $ASAP_i$ of an operation o_i is the earliest control step in which the operation can be scheduled. The* **as-late-as-possible** *value $ALAP_i$ of an operation o_i is the latest control step in which o_i can be scheduled.*

Definition 18 *The* **mobility** $\Pi(o_i)$ *of an operation o_i is the interval between $ASAP_i$ and $ALAP_i$, and the* **slack** *of an operation is $slack(o_i) = |\Pi(o_i)| - 1$.*

7.6.1 Rescheduling by module merger

Based on the notations introduced in section 7.3, suppose we would like to merge two modules m_i and m_j. Assume the s operations scheduled for execution in module m_i are $o_{i_1}, o_{i_2}, \cdots, o_{i_s}$ and t operations scheduled for execution in module m_j are $o_{j_1}, o_{j_2}, \cdots, o_{j_t}$. Let us first consider the simplest case where $s = t = 1$, then the operations o_{i_1} and o_{j_1} must be scheduled in different control steps for execution. Here we would like to make use of three different concepts introduced in [68], namely *strictly sequential, strictly concurrent* and *weakly sequential* (or *weakly concurrent*) pairs of operation:

Definition 19 *Operation o_{i_1} and o_{j_1} are*

- **strictly sequential** *if* $\forall S$, $S(o_{i_1}) \neq S(o_{j_1})$,
- **strictly concurrent** *if* $\forall S$, $S(o_{i_1}) = S(o_{j_1})$,
- **weakly sequential or concurrent** *if* $\exists S_1$, *such that* $S_1(o_{i_1}) = S_1(o_{j_1})$, *and* $\exists S_2$, *such that* $S_2(o_{i_1}) \neq S_2(o_{j_1})$,

where S, S_1 and S_2 are schedules.

If they have been scheduled already in different control steps, we do not need to introduce any scheduling constraints. Otherwise, we have to consider two possibilities:

- execute o_{i_1} before o_{j_1},
- execute o_{j_1} before o_{i_1}.

In other words, we have to either introduce an arc from operation o_{i_1} to operation o_{j_1} or introduce an arc from operation o_{j_1} to operation o_{i_1} in the design representation. Such an arc is referred to as an execution dependency. Introducing extra arcs obviously will have negative impacts or will increase the control steps which leads to longer execution time or slow performance. Before we decide the order of these two operations, we can check if these are **weakly sequential** pairs based on the following theorem [68]:

Theorem 3 *If operations o_{i_1} and o_{j_1} are a* **weakly sequential** *operation pair, then adding a temporal constraint or dependency arc still can guarantee the same schedule without increasing the control steps.*

The above theorem allows us to add constraints to the scheduling in a manner that is beneficial for BIST overhead without increasing the control steps. In order to make use of this type of special operation pair, we need to identify them first. The following theorem introduced in [68] can be used to identify pairs of the **weakly sequential** operations.

Theorem 4 *Operations o_{i_1} and o_{j_1} are* **weakly sequential** *if and only if all of the following conditions are true:*

- *there is no path from operation o_{i_1} to operation o_{j_1} or operation o_{j_1} to operation o_{i_1},*
- $[ASAP_i, ALAP_i] \bigcap [ASAP_j, ALAP_j] \neq \emptyset$,
- *at least one of $slack(o_i)$ and $slack(o_j)$ is not equal to 0.*

Otherwise, we select the pair which results in the smallest increase in the length of the critical path of data path. If they are **strictly concurrent**, rescheduling has to be performed by introducing dummy control steps (places in Petri net) [76] if necessary so as to change the default scheduling of operations. Here we apply two types of I-transformations. For the pair whose *incompletely embedded* module $ED = 0$, the type 1 I-transformation will be applied with the same type module. For the pair whose *incompletely embedded* module $ED = 1$, the type 2 I-transformation will be applied with the same type module. Therefore this can avoid the increase of scheduling steps because one of the operations can be merged with the introduced redundant operations at different scheduling steps.

For the general case in which operations $o_{i_1}, o_{i_2}, \cdots, o_{i_s}$ are scheduled for execution in module m_i and $o_{j_1}, o_{j_2}, \cdots, o_{j_t}$ for execution in module m_j, respectively, we know that there is a sequential order among $o_{i_1}, o_{i_2}, \cdots, o_{i_s}$ because they already share the module m_i. Without loss of generality, we assume the order to be $o_{i_1} \rightarrow o_{i_2} \rightarrow \cdots \rightarrow o_{i_s}$. Similarly, let the sequential order for operations in module m_j to be $o_{j_1} \rightarrow o_{j_2} \rightarrow \cdots \rightarrow o_{j_t}$. The main goal is to merge these two sequential orders into one. First we examine the pair of operations o_{i_1} and o_{j_1} to decide the more suitable order which results in the smaller increase in the length of the critical path of the data path. Then we decide the rest using a merge-sort heuristic to achieve a single execution order.

7.6.2 Rescheduling by register merger

With regards to the scheduling constraints imposed by a register merger, we can use a merge-sort algorithm similar to the one described already in section 7.3.4.

Table 7.1. *Estimated results on the Diffeq benchmark*

Synthesis techniques	Control step	Register	Modular				BIST estimated area overhead	%Decrease in total estimated area
			Mul	Add	Sub	Les		
SWT-AWT	6	5	3	1	1	1	29.25%	-
SFT-AWT	6	5	3	1	1	1	21.40%	5.50 %
SFT-AFT	6	5	3	1	1	1	18.66%	6.53 %
Basic method	6	5	3	1	1	1	14.36%	9.21%
Improved method	6	5	3	1	1	1	12.42%	11.46 %

Table 7.2. *Estimated results on the EWF benchmark*

Synthesis techniques	Control step	Register	Modular		BIST estimated area overhead	%Decrease in total estimated area
			Add	Mul		
SWT-AWT	14	10	5	3	16.28%	-
SFT-AWT	14	8	5	3	6.94%	10.54 %
SFT-AFT	14	8	5	3	6.90%	10.13%
Basic method	14	8	5	3	4.96%	15.46%
Improved method	14	8	5	3	3.65%	18.62%

Table 7.3. *Estimated results on the FIR benchmark*

Synthesis techniques	Control step	Register	Modular		BIST estimated area overhead	%Decrease in total estimated area
			Add	Mul		
SWT-AWT	5	8	4	4	46.06%	-
SFT-AWT	5	8	4	4	32.00%	0.6 %
SFT-AFT	5	8	4	4	27.42%	4.0%
Basic method	5	8	4	4	8.76%	16.81%
Improved method	5	8	4	4	6.51%	18.37%

7.7 Cost Estimation

To demonstrate the use of the proposed high-level test synthesis technique, the estimated cost according to the CAMAD [76] and LSI [68] library have been conducted with a variety of benchmarks such as the 2nd order differential equation, Diffeq [75]; the auto regression filter element, AR filter [46]; an 8-point FIR filter, FIR filter [65] and the elliptic wave filter, EWF filter [52]. The proposed high-level built-in self-test synthesis algorithm with its improved version is applied to these benchmarks and the results are compared with those conventional approaches.

Tables 7.1, 7.2 and 7.3 show the characteristics of the data paths synthesized from the three selected examples, Diffeq [75], EWF [52] and FIR filter [65], respectively. In the tables, SWT-AWT stands for the approach with a traditional scheduling (as-soon-as-possible) that does not consider BIST, and traditional register and module allocations that do not consider BIST. SFT-AWT denotes the approach with scheduling described in [72] with BIST considerations, and traditional module and register allocations without BIST consideration. SFT-AFT is the scheduling approach presented in [72] with BIST considerations, and module and register allocations that search for low BIST area overhead [70]. We compare the proposed test synthesis

approach, namely Basic method (presented in section 7.3 and [102, 104]) and its improved version (presented in section 7.5 and [108, 110, 111]) with them with regards to the number of control steps (scheduling), register and functional module requirements and the total estimated BIST area overhead. The estimated results indicate the advantages of our approaches over conventional ones in terms of BIST area and hardware costs.

7.8 Summary

In this chapter, we present a high-level built-in self-test synthesis algorithm integrating operation scheduling and data path allocation simultaneously. In the next chapter, we present a similar approach for such integration, but based on our proposed BIST testability measurements and its incremental computations described in chapter 4.

Chapter 8

Testability Metrics-based Synthesis

8.1 Introduction

In chapter 7, we have described a series of high-level built-in self-test synthesis algorithms for integrating operation scheduling and data path allocation. Contrary to all existing approaches in the literature in which the scheduling and allocation tasks are performed independently, our approaches presented in chapter 7 and 8 integrate scheduling and allocation simultaneously so that the effects of scheduling and allocation on testability are exploited more effectively and completely. In this chapter, we present a different methodology from that in chapter 7, namely a BIST testability metrics-based algorithm for integrating operation scheduling and data path allocation. In our approach, module and register allocation are guided by a testability balance technique. Scheduling, on the other hand, is still carried out by rescheduling transformations which change the default scheduling to improve testability.

This chapter is organized as follows: in section 8.2 the BIST testability analysis is described briefly (the details can be found in chapter 4). The data path allocation principle and the corresponding built-in self-test synthesis algorithm including operation scheduling are described in section 8.3 and section 8.4, respectively. Furthermore, the introduction of the incremental testability analysis and an improved version of the synthesis algorithm are presented in section 8.5 and section 8.6, with some estimated results compared with other existing work in section 8.7, respectively.

8.2 BIST Testability Analysis

In chapter 4, we have described high-level quantitative metrics to evaluate various BIST configurations and make corresponding improvement decisions. The early decision about BIST testability improvement gives the possibility that designs can be optimized in later synthesis processes. The testability analysis carried out at a high-level of abstraction will also reduce the computational complexity, since the complexity of a design at this level is significantly lower.

The main idea of testability metrics we defined in chapter 4 is to analyze and quantify BIST testability for a given design. Basically, our BIST testability metrics quantify two important testability aspects, namely controllability and observability. Both of controllability and observability are further divided into two factors: combinational factor and sequential factor. The combinational factor is measured in terms of the quality of the pseudorandom values as they propagate through embedded modules and registers, and the sequential factor is used for the estimated number of steps or clock cycle to control under test. Similarly the combinational observability is measured in terms of sensitivity of embedded modules and registers to erroneous value propagation, i.e. in terms of how difficult it is to propagate an erroneous value through to an observable output, and the sequential factor is used for the estimated number of steps or clock cycles to observe under test. As a result, our testability metric consists of, therefore, four measures:*combinational controllability* (CC), *sequential controllability* (SC), *combinational observability* (CO) and *sequential observability* (SO), and provides a means of measuring the effect of test improvement with regards to BIST test quality.

8.3 Allocation Principle

Our approach uses a controllability/observability balance allocation technique to perform the date path allocation task. Conventional allocation approaches often select

and merge the data path nodes according to their connectivity or closeness, which aims to minimize interconnections and multiplexors. This usually results in a very hard to test design because many loops, especially self-loops, are generated. Further, nodes with good controllability and bad observability are merged together since they are very close to the primary inputs. Similar merger for nodes with good observability and bad controllability will also occur. As a result, the data path consists of many nodes which are very difficult either to control or observe.

In our approach, the selection of nodes to be merged is based on the BIST testability measures generated by the testability analysis algorithm described in chapter 4. The main goal is to generate a data path with good controllability and observability for all the nodes and with as few loops as possible. The basic idea is to fold nodes with good controllability and bad observability to nodes with good observability and bad controllability. In this way, the new node will inherit the good controllability from one of the old nodes and the good observability from the other.

8.4 Metrics-based Synthesis Algorithm

8.4.1 The basic algorithm

Our synthesis algorithm [99, 106] accepts an unscheduled ETPN design representation as input and generates a highly testable data path. It iteratively applies transformation to the current ETPN. In each iteration, the algorithm selects pairs of modules and registers in the data path based on the BIST testability analysis and merges them to generate a new data path according to the above described controllability/observability balance principle. Here merging two modules into one implies that all the operations which were assigned to the two modules are reassigned into one single module. Therefore, merging two modules into one imposes a scheduling constraint that all the operations which were assigned to these modules must be scheduled in

different control steps. A similar argument holds for the case of register merger. Merging two registers into one imposes a scheduling constraint that the lifetime of all variables which were assigned in these registers must be disjoint. Note that the merger of nodes (modules and registers) leads to a reduction in the number of hardware components in the design, whereas the additional scheduling constraints may lead to an increase in the execution time of the data path. The iterative algorithm is described in Algorithm 8.1.

Algorithm 8.1 The Metrics-based Synthesis Algorithm

1: Perform a simple default scheduling/allocation
2: **repeat**
3: **for** all modules and registers **do**
4: Run the BIST testability analysis algorithm
5: **end for**
6: Select k pairs of mergable nodes according to the controllability/observability balance principle
7: **for** k pairs modules or registers **do**
8: Estimate the incremental execution time cost ΔE
9: Estimate the incremental hardware cost ΔH
10: **end for**
11: Select the pair with smallest $\Delta C = \alpha \cdot \Delta E + \beta \cdot \Delta H$
12: Merge the selected pair and modify the data path
13: Do lifetime analysis of variables
14: Perform rescheduling imposed by data path synthesis using a merge-sort algorithm
15: **until** no merger exists

In each iteration, our algorithm runs the BIST testability analysis algorithm to select k pairs of modules and registers according to the controllability/observability

balance allocation principle. Here k is a number chosen by the user which is used to control the trade-offs between the testability and execution time and hardware cost. A small value of k means that more emphasis is placed on improving the testability measure. For each of k pairs of modules and registers, we estimate the incremental execution time cost ΔE and the incremental hardware cost ΔH. Then we choose the pair with the smallest value of $\Delta C = \alpha\Delta E + \beta\Delta H$, where α and β are two user-controlled parameters. The selected pair is merged and the data path is modified accordingly. A rescheduling transformation is performed, after we finish the lifetime analysis of variables. This process is repeated until no further merger is possible in the data path.

8.4.2 Operation rescheduling

The rescheduling transformations are performed by a merge-sort algorithm described fully in section 7.3. These transformations locally change the execution orders of some operations in the current schedule in order to both improve the testability and satisfy the scheduling constraints imposed by both module and register mergers.

8.4.3 Estimation of performance and costs

For a given data path, the minimum execution time E is equal to the length of the critical path. The hardware cost H is based on the calculation of the cost of the data path. To make a more accurate estimation, as described in chapter 6, we combine the floor-planning algorithm and test register library of LSI to estimate the hardware cost which takes into account the geometrical information. Based on the estimations, when a pair of modules or registers is merged, ΔH is equal to the increase in data path area cost. For ΔE, it is more complicated than estimating ΔH. In section 7.4, we propose an efficient method for estimating the incremental length of the critical path.

8.4.4 Discussion

So far, we have presented a testability metric-based synthesis algorithm to further explore built-in self-test issues in both operation scheduling and data path allocation process. However there are still some possibilities to improve the algorithm such as making use of incremental testability analysis, introducing redundant transformations, and some enhancement techniques for operation scheduling. The incremental testability analysis and different enhancement techniques together with the improved test synthesis algorithm [109, 110, 112] are presented in section 8.5 and section 8.6, respectively.

8.5 Incremental Testability Analysis

Due to the large computational complexity of testability analysis and the need to perform such analysis after each module and register merger, we have developed a systematic technique for the present BIST technique to accurately approximate the repeated testability calculation and evaluation in chapter 4. In order to get the better module and register merger, the global testability of a data path is used to estimate the global testability of an entire design as defined in chapter 4 as well.

8.6 Improved Metric-based Synthesis

Our improved synthesis algorithm [109, 110, 112] also accepts an unscheduled ETPN design representation as input and generates a BIST testable data path. The main difference compared with the previous one is that there are several enhancement techniques introduced in different phases of the synthesis algorithm. We describe them briefly one by one in this section.

8.6.1 Improved allocation principle

Our approach uses a similar idea to that presented in section 8.3 with the controllability/observability balance allocation technique to perform the date path allocation task. Furthermore, in our improved approach, the selection of nodes to be merged is based on the incremental measures on testability generated by the testability analysis algorithm. We select several pairs which lead to the most significant changes on the global testability of the whole data path. The main goal is to generate a data path with good controllability and observability for all the nodes and with as few loops as possible.

It is possible that some modules with bad controllability or observability can not be merged during the allocation and synthesis process. In this case, we apply two types of I-transformations for different modules as described in section 6.7. After the redundant transformations, all modules with bad controllability or observability will be merged and become testable.

8.6.2 Improved metric-based synthesis

In each iteration, the algorithm selects pairs of modules and registers in the data path and merges them to generate a new data path based on the allocation principle. Here merging two modules into one implies that all the operations which were assigned to the two modules are re-assigned into one single module. Therefore, merging two modules into one imposes a scheduling constraint that all the operations which were assigned to these modules must be scheduled in different control steps. A similar argument holds for the case of register merger. Merging two registers into one imposes a scheduling constraint that the lifetime of all variables which were assigned in these registers must be disjoint. Note that the merger of nodes (modules and registers) leads to a reduction in the number of hardware components in the design, whereas the additional scheduling constraints may lead to an increase in the execution time

Algorithm 8.2 The Improved Metrics-based Synthesis

```
1: Perform a simple default scheduling/allocation
2: repeat
3:    for all modules and registers do
4:       Run the BIST testability analysis algorithm
5:       Run the incremental testability analysis algorithm
6:    end for
7:    Select k pairs of mergable nodes according to the allocation balance principle
8:    for k pairs modules or registers do
9:       Estimate the incremental execution time cost ΔE
10:      Estimate the incremental hardware cost ΔH
11:   end for
12:   Select the pair with smallest ΔC = α · ΔE + β · ΔH
13:   Merge the selected pair and modify the data path
14:   Do lifetime analysis of variables
15:   Perform rescheduling imposed by data path allocation using a merge-sort algo-
      rithm
16: until no merger exists
```

of the data path. The iterative synthesis algorithm is described as Algorithm 8.2. Rescheduling if needed is performed by a merge-sort algorithm described later, after we finish the lifetime analysis of variables. This process is repeated until no further merger is possible in the data path and control part.

8.6.3 Operation rescheduling

When two modules or registers are merged, the rescheduling transformation needs to be performed based on a merge-sort algorithm similar to the one described in chapter 7. These transformations change locally the execution order of some operations in

the current schedule in order to improve the testability and satisfy the scheduling constraints imposed by the merger.

Introducing extra constraints obviously has negative impacts such as increasing the control steps which leads to longer execution time or slow performance. Before we decide the order of these operations to be merged, we can check similarly as we described in section 7.6 if they are **weakly sequential** pairs because if operations are **weakly sequential** operation pair, then adding a temporal constraints or dependency arc still can guarantees the same schedule without increasing the control steps. This allows us to add constrains to the scheduling in a manner that is beneficial for BIST overhead without increasing the control steps. Otherwise, we will select the pair which results in the smallest increase in the length of the critical path of data path described in section 8.4. If they are **strictly concurrent**, rescheduling has to be performed by introducing dummy control steps (places in the Petri net) if necessary so as to change the default scheduling of operations. Here we apply two types of I-transformations as we did in section 7.6. Therefore this can avoid the increase of scheduling steps because one of the operations can be merged with the introduced redundant operations at different scheduling steps.

With regards to the scheduling constraints imposed by a register merger, we can use the above merge-sort algorithm similar to the one described in section 7.6.

8.7 Cost Estimation

To demonstrate the use of the proposed high-level test synthesis technique, the estimated cost according to the CAMAD [76] and LSI [68] library have been conducted with a variety of benchmarks such as the 2nd order differential equation, Diffeq [75]; the auto regression filter element, AR filter [46]; an 8-point FIR filter, FIR filter [65] and the elliptic wave filter, EWF filter [52]. The proposed high-level built-in self-test synthesis algorithm with its improved version are applied to these benchmarks and

Table 8.1. *Estimated results on the Diffeq benchmark*

Synthesis techniques	Control step	Register	Modular				BIST estimated area overhead	%Decrease in total estimated area
			Mul	Add	Sub	Les		
SWT-AWT	6	5	3	1	1	1	29.25%	-
SFT-AWT	6	5	3	1	1	1	21.40%	5.50 %
SFT-AFT	6	5	3	1	1	1	18.66%	6.53 %
Basic method	6	5	3	1	1	1	14.36%	9.21%
Improved method	6	5	3	1	1	1	12.42%	11.46 %

Table 8.2. *Estimated results on the EWF benchmark*

Synthesis techniques	Control step	Register	Modular		BIST estimated area overhead	%Decrease in total estimated area
			Add	Mul		
SWT-AWT	14	10	5	3	16.28%	-
SFT-AWT	14	8	5	3	6.94%	10.54 %
SFT-AFT	14	8	5	3	6.90%	10.13%
Basic method	14	8	5	3	3.65%	18.62%
Improved method	14	8	5	3	2.45%	24.25%

Table 8.3. *Estimated results on the FIR benchmark*

Synthesis techniques	Control step	Register	Modular		BIST estimated area overhead	%Decrease in total estimated area
			Add	Mul		
SWT-AWT	5	8	4	4	46.06%	-
SFT-AWT	5	8	4	4	32.00%	0.6 %
SFT-AFT	5	8	4	4	27.42%	4.0%
Basic method	5	8	4	4	12.37%	10.21%
Improved method	5	8	4	4	8.76%	16.81%

the results are compared with those conventional approaches.

Tables 8.1, 8.2 and 8.3 show the characteristics of the data paths synthesized from the three selected examples, Diffeq [75], EWF [52] and FIR filter [65], respectively. In the tables, SWT-AWT stands for the approach with traditional scheduling (as-soon-as-possible) that do not consider BIST, and traditional register and module allocations that does not consider BIST. SFT-AWT denotes the approach with scheduling with BIST considerations as described in [72], and traditional module and register allocations without BIST consideration. SFT-AFT is the scheduling approach with BIST considerations presented in [72], and module and register allocations that search for low BIST area overhead [70]. We compare the proposed test synthesis approach, namely Basic method (presented in section 8.4 and [99, 106]) and its improved version (presented in section 8.6 and [109, 110, 112]) with them with regards to the number of control steps (scheduling), register and functional module requirements and total estimated BIST area overhead. The estimated results indicate the advantages of our approaches over conventional ones in terms of BIST area and hardware costs.

From the cost estimations, we can see that this approach can produce comparable results with the approach described in chapter 7. Clearly both are better than other

methods published in the literature. If we compare explicitly with the approach presented in chapter 7, this approach is comparable, it is hard to decide which one is likely to lead to better results. For the EWF benchmark, it is better, but for the FIR benchmark, it is worse. The possible explanation is that both approaches use the greedy-type optimization method. More elaborate approaches or heuristics are to be investigated in the future.

8.8 Summary

In this chapter, we present an approach for the integration of operation scheduling and data path allocation, based on our proposed BIST testability measurements and its incremental computations described in chapter 4.

Chapter 9

Conclusion

High level synthesis consists of a variety of synthesis tasks, namely operation scheduling and data path allocation. Each of these has the potential to influence a number of design parameters such as area, power and performance. The testability of a design has traditionally been addressed after the design has been synthesized to the logic level or to the high level. A need to consider testability during high-level synthesis has clearly been established. It is also widely recognized that testability of a circuit is dependent on the testing methodology adapted. Due to the increasing gate-to-pin ratios of embedded systems and system-on-chip which limit the feasibility of testing embedded systems and system-on-chip designs externally, BIST techniques have gained the wide acceptance as a solution to the above problems.

The main contribution of this dissertation is addressing testability during high-level synthesis for a built-in self-test methodology of testing digital system designs. This dissertation develops new systematic techniques to integrate testability considerations into the high-level synthesis process and to make it possible for an automatic synthesis tool to predict testability of the synthesized circuits accurately in the early stage and to optimize the designs in terms of test cost as well as performance, area cost and hopefully power consumption.

A built-in self-test synthesis system has been built in this dissertation. The following are the highlights of the system:

- **Design representation and transformations**: the system takes a VHDL

behavioral specification of a digital system and a set of design constraints as input and generates a Register-Transfer Level (RTL) hardware implementation which consists of a data path and a controller. The kernel of the system is an intermediate design representation, called Extended Timed Petri Net (ETPN) [76], which can be used for testability analysis and high-level synthesis.

- **Testability analysis**: based on the design representation, we have developed a high level data path testability metrics to evaluate various BIST configurations and make improvement decisions. Our testability metric consists of four measures and provides a means of measuring the effect of test improvement with regards to BIST test quality.
- **Incremental testability**: due to the large computational complexity of testability analysis and the need to perform such analysis after each synthesis step, we have proposed a systematic technique to approximate accurately the repeated testability calculation and evaluation.
- **BIST partitioning**: based on the above testability measurements, an efficient partitioning technique, which decide either which registers should be configured as test registers (conventional BIST) or which registers should be linked in the circular scan path (circular BIST) is necessary. This process will be iterated until the design is partitioned into several disjoint sub-circuits and each of them can be tested independently.
- **Resource optimizations**: Before going to RTL implementation or performing the high-level data path allocation or synthesis algorithm, a pre-synthesis resource optimization can be applied. It mainly involves modification of the hardware on the chip so that the chip has the capability to test itself. We describe an approach to find BIST embeddings in the data path prepared for the allocation or synthesis algorithm or before going to RTL implementation such that the cost of modification is minimal.

- **Data path allocation**: a high-level data path allocation algorithm to facilitate the Built-In Self Test is developed. It generates self-testable data path design while maximizing the sharing of modules and test registers. The sharing of modules and test registers enables only a small number of registers are modified for BIST, thereby decreasing the hardware area which is one of the major overheads for BIST technique.
- **Redundant transformations**: Two types of redundant transformations [68, 73] are used, which add redundancy that improves test resources to be shared in the data path without affecting the scheduling step (latency) and functional resource requirement of the behavior. They also improve our data path allocation algorithm, even later the scheduling operations, and the integrated synthesis approach, and make all incompletely embedded modules become fully testable.
- **Integrated synthesis algorithm**: A high-level test synthesis algorithm is proposed based on BIST testing methodology for integrating operation scheduling and data path allocation. It generates highly self-testable data path design while maximizing the sharing of test registers, which means only a small number of registers are modified for BIST. The algorithm also produces design with high test concurrency, thereby decreasing test time. Besides, two types of redundant transformations for the integrated synthesis algorithm are used, which add redundancy that improves test resources to be shared in the data path and operation scheduling.
- **Testability metrics-based synthesis**: A different BIST synthesis methodology is proposed, namely a BIST testability metrics-based algorithm for integrating operation scheduling and data path allocation. It is based on the BIST data path testability analysis, with the help of an incremental BIST testability analysis. Similarly redundant transformations have been used to enhance the testability consideration not only during the data path allocation but also the

operation scheduling process.

This thesis has addressed the high-level test synthesis for digital system designs. The scope of future work is likely to follow the following two primary directions.

Transformational Optimized Test Synthesis

Targeting the same high-level test synthesis problem, the approach is based on an algorithm which applies a sequence of semantics-preserving transformations, such as allocation, de-allocation, operation scheduling and redundant transformations, to a design to generate an efficient register-transfer level implementation from a VHDL behavioral specification. The task of finding a sequence of transformations that will change the initial design into the optimal or near optimal solution in terms of testability, area and performance is a combinatorial problem. For reasonably-sized problems, the greedy type of approaches we have developed in this thesis are going to be successful. Therefore, for large and practical designs, more elaborate approaches, such as simulated annealing, Tabu search or genetic algorithms, will have to be explored.

Low Power BIST Synthesis

Low power consumption is one of the crucial factors determining the success of embedded systems and system-on-chip. The typical examples are mobile computing systems and biomedical implantable devices. The power consumption is a basic constraint to be met, since their operability in the time domain depends on limited energy storage. However, extended battery life is not the only motivation. Due to the increasing integration density of the system-on-chip and the scaling up of operating frequencies, the overheating produced by power consumption is becoming one of the most important factors determining the duration of the electronic components of the embedded systems and system-on-chip. All the high-level synthesis approaches presented in this thesis have considered low power and testability as two separated

optimization objectives.

One promising way to approach low power synthesis is to extend the above built-in self-testable synthesis system for low power consumption. It is anticipated that a data path allocation and an integrated synthesis algorithms are to be developed in the future work to achieve fully testable design and low power consumption without resulting in serious performance degradation and with minimal area overhead.

This thesis has demonstrated the importance of considering high-level test synthesis to boost design quality and shorten the development cycle. With regards to high-level test synthesis in general, other testability criteria and methodologies remain a very rich and interesting area to be explored in the future.

Bibliography

[1] M. Abadir and J. Newman. Partitioning hierarchical designs for testability. In *Proceedings of International Test Conference*, pages 174–183, 1991.

[2] M.S. Abadir and M. A. Breuer. A knowledge-based system for designing testable VLSI chips. *IEEE Design and Test of Computers*, pages 56–58, 1985.

[3] M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital System Testing and Testable Design.* IEEE Press, 1990.

[4] V. D. Agrawal, C. R. Kime, and K. K. Saluja. A tutotial on Built-in Self-test, part 1: principles. *IEEE Design and Test of Computers*, March 1993.

[5] V. D. Agrawal, C. R. Kime, and K. K. Saluja. A tutotial on Built-in Self-test, part 2: applications. *IEEE Design and Test of Computers*, June 1993.

[6] A. V. Aho and R. Sethi amd J. D. Ullman. *Compiler Principles, Techniques and Tools.* Addison-Wesley Publishing Company, 1986.

[7] Semiconductor Industry Association. The national technology roadmap for semiconductors. Technical report, Semiconductor Industry Association, 181 Metro Drive, Ste 450, San Jose, California 95110, 1997.

[8] L. Avra. Allocation and assignment in high-level synthesis for self-testable data paths. In *Proceedings of the International Test Conference*, pages 463–472, 1991.

[9] M. Baklashov. *Behavioral-structural testability modeling and synthesis for BIST with application to high-level synthesis.* PhD thesis, Department of Computer Engineering and Science, Case Western Reserve University, January 1997.

[10] P. H. Bardell and W. H. McAnney. Self-testing of multichip logic modules. In *Proceedings of International Test Conference*, pages 200–204, 1982.

[11] P.H. Bardell, W. H. McAnney, and J. Savir. *Built-In Self-Test for VLSI: Pseudo-random Techniques.* John Wiley & Sons Inc., 1987.

[12] R. A. Bergamaschi, R. A. C'Connor, L. Stok, M. Z. Moricz, S. Prakash,

A. Kuehlmann, and D. S. Rao. High-level synthesis in an industrial environment. *IBM Journal of Research Development*, 39:131–148, 1995.

[13] F. P. Beucler and M. J. Manner. HILDO: the highly integrated logic device observer. *VLSI Design*, pages 88–96, June 1984.

[14] J. Biesenack, M. Koster, T. Langmaier, S. Ledeux, S. Marz, M. Pilsl, S. Rumler, H. Soukup, A. Stoll, N. Wehn, and P. Duzy. The Siemens high-level synthesis system CALLAS. *IEEE Transactions on VLSI Systems*, 1(3):24–253, September 1993.

[15] R. Camposano and W. H. Wolf. *High-Level VLSI Synthesis.* Kluwer Academic Publishers, 1991.

[16] J. Carletta and C. A. Papachristou. Testability analysis and insertion for RTL circuits based on pseudorandom BIST. In *Proceedings of International Conference on Computer Design*, 1995.

[17] J. Carletta and C. A. Papachristou. Behavioral testability insertion for datapath/controller circuits. *Journal of Electronic Testing: Theory and Applications*, 11:9–28, 1997.

[18] J. L. Carter. The theory of signature testing for VLSI. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 66–76, 1982.

[19] S. Chakradhar, A. Balakrishnan, and V. D. Agrawal. An exact algorithm for selecting partial scan flip-flops. In *Proceedings of Design Automation Conference*, pages 81–86, June 1994.

[20] A. P. Chandrakasan, M. Potkonjak, R. Mehra, and J. Rabaey. Optimizing power using transformations. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, 14:12–31, 1995.

[21] T. H. Chen and M. A. Breuer. Automatic design for testability via testability measures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 4(1):3–11, January 1985.

[22] K. T. Cheng and V. D. Agrawal. A partial scan method for sequential circuits with feedbacks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(4):538–544, April 1990.

[23] S. Chiu and C. A. Papachristou. A design for testability scheme with applications to data path synthesis. In *Proceedings of Design Automation Conference*, pages 271–277, June 1991.

[24] C. C. Chuang and A. K. Gupta. The analysis of parallel BIST by the combined

markov chain. In *Proceedings of International Test Conference*, pages 337–343, 1989.

[25] P. Clemente, R. Crevier, and P. Runstadler. RTL and bahavioral synthesis: a case study. *VHDL Times*, 5:56–60, 1996.

[26] K. De and P. Banerjee. A system for logic partitioning and resynthesis for testability. *IEEE Transactions on VLSI Systems*, 1(4), 1993.

[27] S. Dey and M. Potkonjak. Transforming behavioral specifications to facilitate synthesis of testable designs. In *Proceedings of International Test Conference*, pages 184–193, 1994.

[28] P. Eles, K. Kuchcinski, and Z. Peng. *System Synthesis with VHDL*. Kluwer Academic Publishers, 1998.

[29] T. Fuhrman. Industrial extensions to university high-level synthesis tools: making it work in the real world. In *Proceedings of the 28th Design Automation Conference*, pages 520–525, 1991.

[30] D. D. Gajski, N. D. Dutt, A. C-H. Wu, and Steve Y-L. Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.

[31] J. W. Gannet. *VLSI Electronics*, chapter Self-testing by integrated feedback, pages 107–110. Academic Press, New York, 1986.

[32] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., San Francisco, 1979.

[33] M. Gentil, D. Crestani, A. Rhalibi, and C. Durante. A new high-level testability: description and evaluation. In *Proceedings of IEEE VLSI Test Symposium*, 1994.

[34] L. H. Goldstein. Controllability/observability analysis of digital circuits. *IEEE Transactions on Circuits and Systems*, CAS-26(9):685–693, 1979.

[35] X. Gu. *Register-transfer level testability improvement by testability analysis and transformations*. PhD thesis, Department of Computer Science, Linköping University, 1996.

[36] X. Gu, K. Kuchcinski, and Z. Peng. Testability analysis and improvement from VHDL behavioral specifications. In *Proceedings of the European Design Automation Conference with EURO-VHDL*, 1994.

[37] X. Gu, K. Kuchcinski, and Z. Peng. An efficient and economic partitioning approach for testability. In *Proceedings of International Test Conference*, Washington D. C., October 1995.

[38] L. Guerra, M. Potkonjak, and J. Rabaey. High-level synthesis for reconfigurable

datapath structures. In *Proceedings of International Conference on Computer-Aided Design*, pages 26–29, November 1993.

[39] R. Gupta, R. Gupta, and M. A. Breuer. The BALLAST methodology for structured partial scan design. *IEEE Transaction on Computers*, 39(4):538–544, April 1990.

[40] R. Gupta, R. Srinivasan, and M. A. Breuer. Reorganizing circuits to aid testability. *IEEE Design and Test of Computers*, pages 49–57, 1991.

[41] H. Harmanani and C. Papachristou. An improved method for RTL synthesis with testability tradeoffs. In *Proceedings of the International Conference on Computer-Aided Design*, pages 30–35, 1993.

[42] I. G. Harris and A. Orailoglu. Microarchitectural synthesis of VLSI designs with high test concurrency. In *Proceedings of 31st ACM/IEEE Design Automation Conference*, pages 206–211, June 1994.

[43] J. P. Hayes. On modifying logic networks to improve their diagnosability. *IEEE Transactions on Computer*, C-23:56–62, January 1974.

[44] J. P. Hayes and A. D. Friedman. Test point placement to simplify fault detection. In *Proceedings of the 1973 IEEE International Symposium on Fault-Tolerant Computing*, pages 73–78, June 1973.

[45] R. W. Hunter, T. Fuhrman, and D. E. Thomas. Working chips from high-level synthesis: a case study from industry. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 144–147, May 1994.

[46] R. Jain, M. Mlinar, and A. Parker. Area-time model for synthesis of non-pipelined designs. In *Proceedings of International Conference on computer-Aided Design*, pages 48–51, 1990.

[47] S. Jian and C. Stroud. Built-in self testing of embedded memories. *IEEE Design and Test*, pages 27–37, 1986.

[48] R. Karri and A. Oraiglu. Transformation-based high-level synthesis of fault tolerant ASICs. In *Proceedings of the 29th Design Automation Conference*, pages 662–665, June 1992.

[49] K. Kim, D. S. Ha, and J. G. Tront. On using signature registers as pseudorandom pattern generators in built-in self-testing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 7(8):919–928, August 1988.

[50] T. Kim. *Scheduling and allocation problems in high-level synthesis.* PhD thesis,

Department of Computer Science, University of Illinois at Urbana-Champaign, 1993.

[51] K. Kuchcinski and Z. Peng. Testability analysis in a VLSI high-level synthesis system. *The Euromicro Journal, Microprocessing and Microprogramming*, 28(1-5):295–300, March 1990.

[52] S. Y. Kung, H. J. Whitehouse, and T. Kailath. *VLSI and Modern Signal Processing.* Prentice-Hall, 1985.

[53] K. Lai, C. A. Papachristou, and M. Baklashov. High-level test synthesis across the boundary of behavioral and structural domains. In *Proceedings of International Conference on Computer Design*, 1997.

[54] T. C. Lee. *Behavioral synthesis of highly testable data path in VLSI digital circuits.* PhD thesis, Department of Electrical Engineering, Princeton University, 1993.

[55] R. Lisanke, F. Braglez, A. J. Degues, and D. Gregory. Testability-driven random test pattern generation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6:1082–1087, 1987.

[56] E. Macii and A. R. Meo. A test generation program for sequential circuits. *Journal of Electronic Testing: Theory and Applications*, 5, 1994.

[57] M. C. McFarland and A. C. Parker. The high-level synthesis of digital systems. In *Proceedings of IEEE*, volume 78, pages 301–318, 1990.

[58] P. Michel, U. Lauther, and P. Duzy. *The Synthesis Approach to Digital System Design.* Kluwer Academic Publishers, 1992.

[59] G. D. Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill Inc., 1994.

[60] A. Orailoglu and I. Harris. Microarchitectural synthesis for rapid BIST testing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(6):573–586, June 1997.

[61] C. A. Papachristou and M Baklashov. A test synthesis technique using redundant register transfers. In *Proceedings of International Conference on Computer-Aided Design*, 1997.

[62] C. A. Papachristou and J. Carletta. Test synthesis in the behavioral domain. In *Proceedings of International Test Conference*, October 1995.

[63] C. A. Papachristou, S. Chiu, and H. Harmanani. A data path synthesis method for self-testable designs. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 378–384, 1991.

[64] C. A. Papachristou, S. Chiu, and H. Harmanani. SYNTEST: a method for high-level SYNthesis with self-TESTability. In *Proceedings of the IEEE International Conference on Computer Design*, pages 458–462, October 1991.

[65] N. Park and A. C. Parker. SEHWA: a program for synthesis of pipelines. In *Proceedings of the 23rd Design Automation Conference*, pages 454–460, 1986.

[66] S. Park and S. B. Akers. A graph theoretic approach to partial scan design by k-cycle elimination. In *Proceedings of International Test Conference*, pages 303–311, November 1992.

[67] A. C. Parker, J. T. Pizarro, and M. Mlinar. MAHA: a program for datapath synthesis. In *Proceedings of the 23th Design Automation Conference*, pages 496–499, June 1986.

[68] I. Parulkar. *Optimization of BIST resource during high-level synthesis.* PhD thesis, University of South California, May 1998.

[69] I. Parulkar, S. Gupta, and M. Breuer. Data path allocation for synthesizing RTL designs with lower BIST area overhead. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, 1995.

[70] I. Parulkar, S. Gupta, and M. Breuer. Lower bounds on test resources for data flow graphs. In *Proceedings of the 33rd ACM/IEEE Design Automation Conference*, pages 143–148, 1996.

[71] I. Parulkar, S. Gupta, and M. Breuer. Introducing redundant computations in a behavior for reducing BIST resources. In *Proceedings of the 35th ACM/IEEE Design Automation Conference (DAC-98)*, pages 548–553, San Francisco, USA, June 15-18, 1998.

[72] I. Parulkar, S. Gupta, and M. Breuer. Scheduling data flow graphics for minimizing BIST resources in data paths. In *Proceedings of Design, Automation and Test in Europe (DATE-98)*, Paris, France, 1998.

[73] I. Parulkar, S. Gupta, and M. Breuer. Introducing redundant computations in RTL data paths for reducing BIST resources. *ACM Transactions on Design Automation of Electronic Systems*, 6(3):423–445, 2001.

[74] P. G. Paulin. DSP design tool requirements for the nineties: an industrial perspective. In *Proceedings of the 6th International Workshop on High-level Synthesis*, November 1992.

[75] P. G. Paulin and J. P. Knight. Forced-directed scheduling for the behavioral synthesis of ASIC's. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8:661–678, June 1989.

[76] Z. Peng and K. Kuchcinski. Automated transformation of algorithms into register-transfer level implementations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 150–166, 1994.

[77] J. Peterson. *Petri Net Theory and the Modeling of System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[78] S. Pilarski, A. Krasniewski, and T. Kameda. Estimating testing effectiveness of the circular self-test path technique. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(10):1301–1316, October 1992.

[79] M. Potkonjak and J. Rabaey. Optimizing resource utilization using transformations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(3):277–292, March 1994.

[80] M. M. Pradhan, E. J. Brrien, S. L. Lam, and J. Beausang. Circular BIST with partial scan. In *Proceedings of International Test Conference*, pages 719–729, October 1988.

[81] S. M. Reddy and R. Dandapani. Scan design using standard flip-flops. *IEEE Design and Test*, pages 52–54, 1987.

[82] L. Rosqvist. *Application Specification Integrated Circuit (ASIC) Technology*, chapter 8, Test and testability of ASICs. Academic Press, San Diego, California, 1991.

[83] E. Roza, J. Biesterbos, B. De Loore, and J. Van Meerbergen. On the application of architectural synthesis in the design of high-volume production ICs for consumer applications. In *Proceedings of the 6th International Workshop on High-level Synthesis*, pages 2–15, November 1992.

[84] J. Savir and P. H. Bardell. Built-in self-test: milestones and challenges. *VLSI Design*, 1(2):119–225, May 1992.

[85] C. E. Stroud. Automated BIST for sequential logic synthesis. *IEEE Design and Test of Computers*, pages 22–32, December 1988.

[86] M. Sugihara. *A study on test methodologies for system-on-chip.* PhD thesis, University of Kyushu, Japan, 2001.

[87] K. Thearling and J. Abraham. An easily computed functional level testability measure. In *Proceedings of International Test Conference*, pages 381–390, 1989.

[88] D.E. Thomas, E. D. Lagnese, R. A. Walker, J. A. Nestor, J. V. Rajan, and R. L. Backburn. *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench.* Kluwer Academic Publishers, 1990.

[89] E. Trischler. Incomplete scan path with an automatic test generation approach.

In *Proceedings of International Test Conference*, pages 153–162, November 1980.

[90] H. C. Tsai, K. T. Cheng, C. J. Lin, and S. Bhawmik. A hybrid algorithm for test point selection for scan-based BIST. In *Proceedings of Design Automation Conference*, 1997.

[91] C-J. Tseng, R-S. Wei, S.G. Rothweiler, M. M. Tong, and A. K. Rose. Bridge: a versatile behavioral synthesis system. In *Proceedings of the 25th Design Automation Conference*, pages 415–420, July 1988.

[92] F. F. Tsui. *LSI/VLSI Testability Design.* McGraw-Hill, 1988.

[93] B. Tuck. Finally, behavioral synthesis is production-ready. *Computer Design*, pages 57–63, July 1997.

[94] J. Vanhoof, K. V. Rompaey, I. Bolsens, G. Goossen, and H. De Man. *High-level Synthesis for Real-time Digital Signal Processing.* Kluwer Academic Publishers, 1993.

[95] B. Vinnakota and N. K. Jha. Synthesis of sequential circuits for parallel scan. In *Proceedings of European Conference of Design Automation*, Brussels, Belgium, March 1992.

[96] L. T. Wang and E. J. McCluskey. Built-in self-test for sequential machines. In *Proceedings of International Test Conference*, pages 334–341, 1987.

[97] M. J. Y. Williams and J. B. Angell. Enhancing testability of large scale integrated circuit via test points and additional logic. *IEEE Transactions on Computers*, C-22:46–60, January 1973.

[98] T. W. Williams and K. P. Parker. Design for Testability, A Survey. In *Proceedings of IEEE*, volume 71, January 1983.

[99] L. T. Yang and J. Muzio. A BIST testability metric-based algorithm to integrate scheduling and allocation in high-level test synthesis. In *Proceedings of the 9th International Symposium on Integrated Circuits, Devices and Systems (ISIC-01)*, pages 409–413, Singapore, September 3-5, 2001.

[100] L. T. Yang and J. Muzio. Built-in self-testable data path synthesis. In Smailagic A. and De Man H., editors, *Proceedings of the 2001 IEEE Computer Society Workshop on VLSI (WVLSI-01)*, pages 78–84, Orlando, Florida, April 19-20, 2001.

[101] L. T. Yang and J. Muzio. High-level data path synthesis for built-in self-testable designs. In *Proceedings of the IEEE Pacific Rim Conference on Communica-*

tion, Computers and Signal Processing (PARCIM-01), volume 1, pages 279–282, Victoria, Canada, August 26-28, 2001.

[102] L. T. Yang and J. Muzio. An improved high-level built-in self-test synthesis algorithm. In *Proceedings of the 8th IEEE International Conference on Electronics, Circuits and Systems (ICECS-01)*, volume 1, pages 549–552, Msida, Malta, September 2-5, 2001.

[103] L. T. Yang and J. Muzio. An improved register transfer level built-in self-test partitioning. In *Proceedings of the 9th International Symposium on Integrated Circuits, Devices and Systems (ISIC-01)*, pages 414–417, Singapore, September 3-5, 2001.

[104] L. T. Yang and J. Muzio. An integrated high-level test synthesis algorithm for built-in self-testable designs. In *Proceedings of the XIV International Symposium on Integrated Circuits and System Designs (SBCCI-01)*, pages 115–121, Brasilia, Brazil, September 10-15, 2001.

[105] L. T. Yang and J. Muzio. A register-transfer level BIST partitioning approach for ASIC designs. In *Proceedings of the 2001 IEEE Pacific Rim Conference on Communication, Computers and Signal Processing (PARCIM-01)*, volume 1, pages 275–278, Victoria, Canada, August 26-28, 2001.

[106] L. T. Yang and J. Muzio. An improved BIST testability metric-based high-level test synthesis approach. In *Proceedings of the 2002 International Conference on VLSI (VLSI-02)*, pages 78–85, Las Vegas, USA, June 24-27, 2002.

[107] L. T. Yang and J. Muzio. Introducing redundant transformations for built-in self-testable data path allocation. In *Proceedings of the 2002 IEEE International Conference on Communications, Circuits and Systems (ICCCAS-02)*, volume 2, pages 1346–1350, Chengdu, China, June 29-July 1, 2002.

[108] L. T. Yang and J. Muzio. Introducing redundant transformations for high-level built-in self-testable synthesis. In *Proceedings of the 9th IEEE International Conference on Electronics, Circuits and Systems (ICECS-02)*, volume 2, pages 475–479, Dubrovnik, Croatia, September 15-18, 2002.

[109] L. T. Yang and J. Muzio. Redundant transformations for the testability metrics-based high-level built-in self-testable synthesis. In *Proceedings of the XVII International Conference on Design of Circuits and Integrated Systems (DCIS-02)*, Santander, Spain, November 19-22, 2002.

[110] L. T. Yang and J. Muzio. Testing methodologies for embedded systems and systems-on-chip. In *Proceedings of the 2004 International Conference on Em-*

bedded Software and Systems (ICESS-04), pages 15–25, Hangzhou, China, December 9-10, 2004.

[111] L. T. Yang and J. Muzio. High level built-in self-testable synthesis for system-on-chip. *Submitted to Integration, VLSI Journal*, 2007.

[112] L. T. Yang and J. Muzio. Testability metrics-based high level built-in self-testable synthesis of digital systems. *Submitted to Microelectronic Engineering Journal*, 2007.