TALLINN TECHNICAL UNIVERSITY

Faculty of Information Technology
Department of Computer Engineering
Chair of Computer Engineering and Diagnostics

Bachelor Thesis
IAF34LT

Test Time Minimization for Hybrid BIST of Systems-on-Chip

by

Maksim Jenihhin

Supervisors: **Raimund Ubar**
Gert Jervan

Tallinn 2003

Annotatsioon

Käesoleva töö põhieesmärgiks oli eksperimentaalse keskkonna loomine mikroelektroonika testimisele kuluva aja vähedamiseks. Töö põhineb hybriidsel ise-testivuse arhitektuuril ja on mõeldud kaasaegsete SoC disainide jaoks. Käesolev töö põhineb töö käigus välja töötatud metodoloogial ja demonstreerib selle sobivust antud probleemi lahendamiseks koos eksperimentaalsete tulemustega. Esimesed kaks peatükki kirjeldavad probleemi aktuaalsust ja pakuvad taustainformatsiooni. Järgnevalt kirjeldatakse testimisele kuluva aja vähedamist kombinatoorsete disainide korral. Kirjeldatakse ka sobivad hübriidset ise-testivat arhitektuuri. Kuna enamus kaasaegseid disaine põhineb aga järjestikskeemidel, siis järgmine peatükk kirjeldabki sama probleemi lahendamist järjestikskeemide korral. Viimasena esitletakse graafilist demonstraatorit, mida saab kasutada käesoleva probleemi illustreerimiseks.

Käesolev töö on toimunud Tallinna Tehnikaülikooli Arvutitehnika instituudi ja Linköpingi ülikooli Embedded Systems Laboratory koostöös.

Abstract

The main goal of this thesis was to develop an experimental environment for the test time minimization problem. It assumes Hybrid BIST architecture and targets System-on-Chip designs. The thesis is based on methodology developed during the work and demonstrates the feasibility of the proposed methodology together with experimental results. First two sections of this thesis explore the actuality of the problem and provide background information. Further, the proposed methodology is discussed for the case when a SoC consists only of combinational cores. An appropriate Hybrid BIST architecture is proposed as well. However, real life System-on-Chip designs contain mostly sequential cores, and this is taken into account in the next part of this thesis, where Hybrid BIST for SoCs with sequential cores is examined. At the end of this thesis a small demonstrational program is presented, which may be interpreted as a useful add-on for the rest of the material reporting results of the research.

The thesis is a result of research carried out in cooperation between Tallinn Technical University, Department of Computer Engineering (Estonia) and Embedded Systems Laboratory of Linköping University (Sweden).

Contents

Chapter 1 Introduction *6*

Motivation **7**

Thesis Overview **7**

Chapter 2 Background *9*

Systems-on-Chip **9**

BIST **10**

LFSR **11**

Hybrid BIST **12**

Scan Design **13**

Conclusion **15**

Chapter 3 Test Time Minimization for Hybrid BIST of Systems-on-Chip with Combinational Cores *16*

3.1 Theoretical Description of the Proposed Approach *16*

Hybrid BIST Architecture **16**

Basic Definitions and Problem Formulation **19**

Hybrid Test Sequence Computation Based on Cost Estimates **22**

Test Length Minimization under Memory Constraints **24**

3.2 Experiments *26*

Setup **26**

Pseudorandom Pattern Generation **27**

Deterministic Pattern Generation **28**

Reporting Numbers of Faults Covered by Test Patterns **29**

Estimations' Generation **30**

Exhaustive Simulation **34**

CPU Time Measurement for Performed Computations **36**

Conclusion **39**

Chapter 4 Test Time Minimization for Hybrid BIST of Systems-on-Chip with Sequential Cores ***40***

4.2 Theoretical discussion ***40***

Hybrid BIST Architecture **40**

Differences in the Testing Approach **42**

4.1 Experiments ***43***

Setup **43**

Test Pattern Generation **44**

Estimation **45**

Exhaustive Simulation and CPU Time Measurement **47**

Conclusion **49**

Chapter 5 Demonstrational program ***51***

Motivation **51**

User Interface **52**

Initialization **55**

Reaction on Events **56**

Conclusion **57**

Chapter 6 Conclusions and Future Work ***58***

Conclusion **58**

Future work **59**

References and Bibliography **60**

Appendix A **61**

Appendix B **65**

Appendix C **67**

Chapter 1
Introduction

Due to constant development of microelectronics design technology, testing techniques of new integrated circuits should be updated with the same speed. At present, such a style of design, when a number of functional blocks are combined in one single integrated circuit (IC), provides designers many convenient possibilities. It is usually referred as System-on-Chip (SoC) or Core-Based System approach. Systems-on-chip look very attractive from designing point of view, mainly, because they allow reusing previous designs, what in its turn leads to reduced cost and shorter time-to-market. At the same time, testing of such systems is very complex and insufficiently explored task.

General tendency shows decrease in minimal size of transistors and as a result increase in ICs' complexity and density, as well as their working frequency. Traditional testing approaches use both test pattern source and sink off-chip, and therefore require external Automatic Testing Equipment (ATE). Unfortunately, such kind of ATE can not be any longer considered as a good solution for modern SoC testing. The reason is unacceptable growth in their price and memory size requirements and often their disability to perform testing at speed. Consequently, we need another solution for today's ICs.

One of the possible solutions, according to the facts above, is to use Hybrid Built-In Self-Test approach. It performs all testing operations at-speed and does not need any external equipment, while assuming some extra logic integration into IC itself. However, Hybrid BIST approach still has some parameters, which have not been examined enough. This thesis is a result of a research carried out to try to find a suitable solution for one of the problems that we may face while implementing Hybrid BIST in real life.

Motivation

General idea of Built-In Self-Test (BIST) is to generate, apply and analyze pseudorandom test patterns internally. But in real life, due to unacceptably long test sequences and pseudorandom pattern resistant faults, it may not always be efficient enough for some embedded cores. Therefore, Hybrid BIST approach was proposed, what mainly adds deterministic pattern sequence to pseudorandom one, used in BIST.

One of the most important parameters influencing efficiency of Hybrid BIST becomes, hence, the ratio of pseudorandom and deterministic patterns in the final test set. In other words, this is a trade-off between longer total test time, when more pseudorandom test patterns are used, and higher memory requirements, when more deterministic test patterns must be stored.

There have been a number of researches in the area of SoC testing, but the main emphasis has been so far on scheduling, TAM design and testability analysis. Consequently, the research, which results are described in this thesis may, be considered as one of the first, where, while determining parameters for Hybrid BIST of a SoC, the system under test is handled as one whole, but not as its separate cores.

This thesis contains new methodology of test time minimization, where memory constraints are taken into account, and maximum possible fault coverage is guaranteed. To avoid exhaustive search, two algorithms will be introduced: one to estimate the cost of deterministic part of the test, the second one to adjust the estimations to quasi exact values.

Thesis Overview

The rest of the thesis is organized as follows. The second section describes some background theories, such as general information about SoCs, Hybrid BIST, LFSR, scan design and STUMPS. These basic concepts are necessary for overall understanding of the proposed ideas.

Chapter 3 is the most important one. It describes our proposed solution, or being more exact, the algorithms for deterministic component cost estimation, and further for its

adjustment. In this chapter all the theoretical explanations and carried out experiments are provided for SoCs with combinational cores.

Chapter 4 discusses about additional difficulties related to testing SoCs with sequential cores in comparison to combinational ones. And as a consequence, it describes the differences in our approach implementation in case of SoCs with sequential cores.

Chapter 5 presents an ActiveX control that was created mainly for demonstration of general principles of this approach. This chapter may be considered as a separate part of this thesis, but very useful for at first sight understanding of the processes referred in the proposed methodology.

Finally, Chapter 6 concludes this thesis by summarizing the ideas and discussing possible directions for the future work.

Chapter 2
Background

In this chapter a number of basic concepts are discussed. It starts with presentation of Systems-on-Chip. Then it is followed by description of BIST, while making emphasis on pseudorandom pattern generation with LFSR and Hybrid BIST approach. A short discussion about sequential ICs' testing particularity is provided in Scan Design subsection.

Systems-on-Chip

Recent miniaturization tendency in microelectronics technology and increase in designs' complexity have encouraged designers to accept a new approach to design [5]. The main innovation in it is implementation of whole systems, consisting of modules with different functions, on one single chip. These systems are usually referred as *Systems-on-Chip (SoC), Core-Based Systems*, or *Multi-Core Systems*. An example of a SoC is shown in Fig. 2.1.

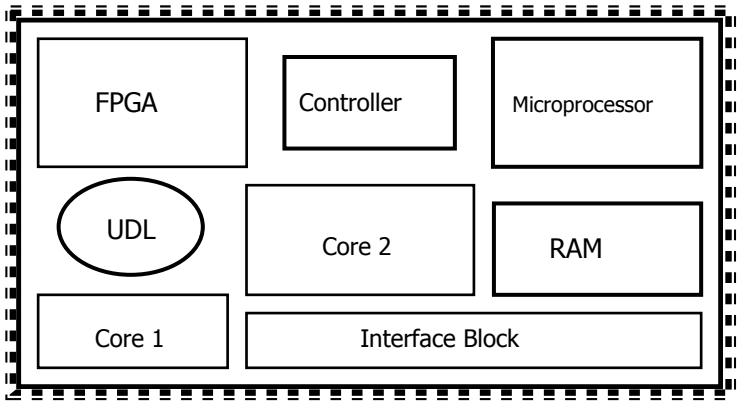


Figure 2.1 System-on-chip

SoC approach provides to designers possibility to reuse their previously designed modules, usually referred as *embedded cores*, as well as integrating in their systems cores, designed by others. Usually a SoC contains at least one microprocessor and one RAM module, as well as such called *user-defined logic* (UDL). The last one is used to "glue" various cores in the system and requires different approaches of testing.

However, UDL is beside of this thesis's point. The attractive cores for our approach are *Microprocessor* and *RAM*, the rest are just cores that should be tested. As it will be described latter the first one may be used for pseudorandom patterns generation and as a test controller. A possible usage of RAM module for testing is, obviously, storing deterministic test patterns. Although, for the possible test architecture, this thesis provides an implementation of these Hybrid BIST components as additional logic on IC.

BIST

Built-In Self-Test (BIST) is a design technique in which parts of a circuit are used to test the circuit itself [6]. Although, there are various BIST schemes, any of those performs test pattern generation, test application and response verification [5]. BIST uses mostly pseudorandom test patterns. They are usually generated by Linear Feedback Shift Registers (LFSR), as it will be described further. These test patterns are easy and cheap to generate, but in reality, a pure pseudorandom test should be very long, hence too expensive in terms of time, to obtain the highest possible fault coverage. Moreover, it is not even always guaranteed that the highest possible fault coverage will be achieved even with extremely long pseudorandom test pattern sequence. The reason is *random pattern resistant faults* (RPR).

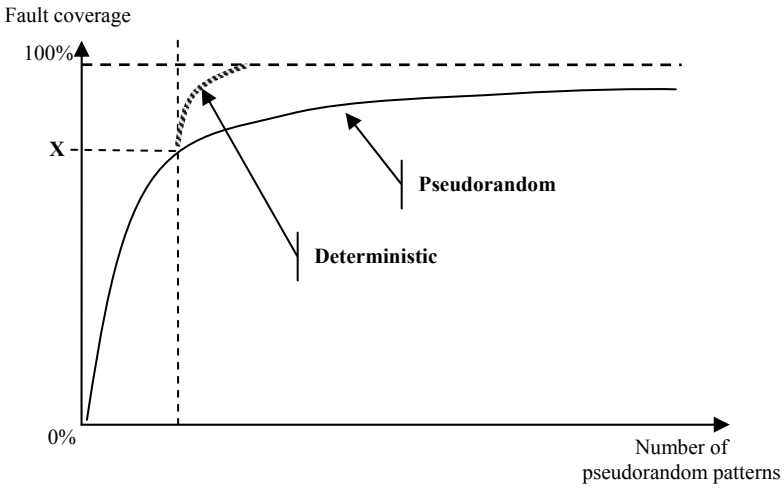


Figure 2.2 Pseudorandom test behavior.

In the following pseudorandom test pattern generation based on LFSR-s will be described.

LFSR

As its name implies, the LFSR (Linear Feedback Shift Register) is a shift register with feedback from the last stage and others [5]. A general structure of it is shown on Fig 2.4.

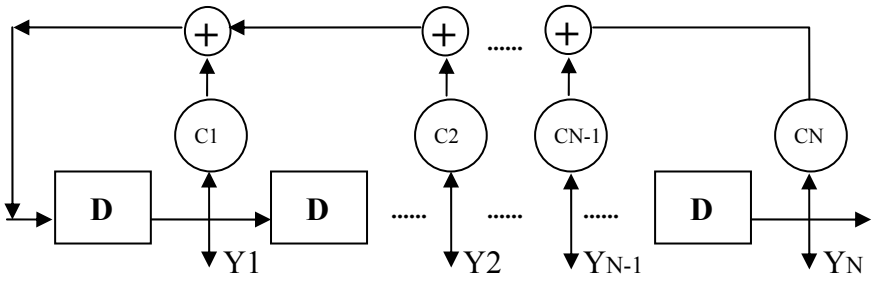


Figure 2.4 Representation of a standard LFSR.

Two important parameters of every LFSR are *initial vector* and *characteristic polynomial*. At the starting clock cycle, every flip-flop D contains a bit from initial vector. On the next clock a D flip-flop shifts its value to the next one in the chain and to the output Yi. The very last flip-flop D shifts its bit to the first one, while on the way XOR operation may be performed between this bit and values of some other D flip-flops. Polynomial coefficients C_1, C_2, …, C_N determine which of D flip-flop values will participate in the operation. An example result of 3-flip-flop LFSR work is shown below:

Step	*0:*	*0*	*1*	*1 -initial vector*
	1:	*0*	*0*	*1*
	2:	*1*	*0*	*0*
	3:	*0*	*1*	*0*
	4:	*1*	*0*	*1*
	5:	*1*	*1*	*0*
	6:	*1*	*1*	*1*
	---	---	---	---
7 =	*0:*	*0*	*1*	*1*

Here C_1= 0; C_2= 1; C_3= 1.

In case of a good polynomial, LFSR will repeat its state on 2n-1 step, otherwise it will happen earlier. The vector with all "0" never can be generated by LFSR, and no vector at all can be generated, if all "0" are used for the initial vector.

Although the test patterns generated by LFSR are still *pseudo*-random, the randomness provided by them is acceptable for BIST technique, considering very low generation cost. In our approach we assume, that PRPG and MISR are implemented on Linear

Feedback Shift Registers (LFSR).

A simple program in C language was written for the experiments with sequential cores, described in this thesis, and may be considered as a sample software LFSR implementation. The source code is enclosed in *Appendix B*.

Hybrid BIST

To avoid the problems related to pseudorandom patterns, a solution, known as *Hybrid BIST*, was introduced. In this case, we can dramatically reduce the length of the initial pseudorandom sequence by completing it with stored deterministic test patterns, and guarantee the highest possible fault coverage.

Figure 2.2 demonstrates usual rapid increase in fault coverage obtained by pseudorandom test in the beginning, and further saturation. Consequently, it is reasonable to terminate pseudorandom test sequence, as soon as fault coverage X is achieved, and to continue with deterministic test patterns till the highest possible fault coverage.

Determining the optimal ratio of pseudorandom and deterministic tests in the final test set is a complex task even for one single core [4]. While considering the core as a part of a SoC with additional constraints makes this task significantly more difficult. However this thesis describes a possible solution for the problem.

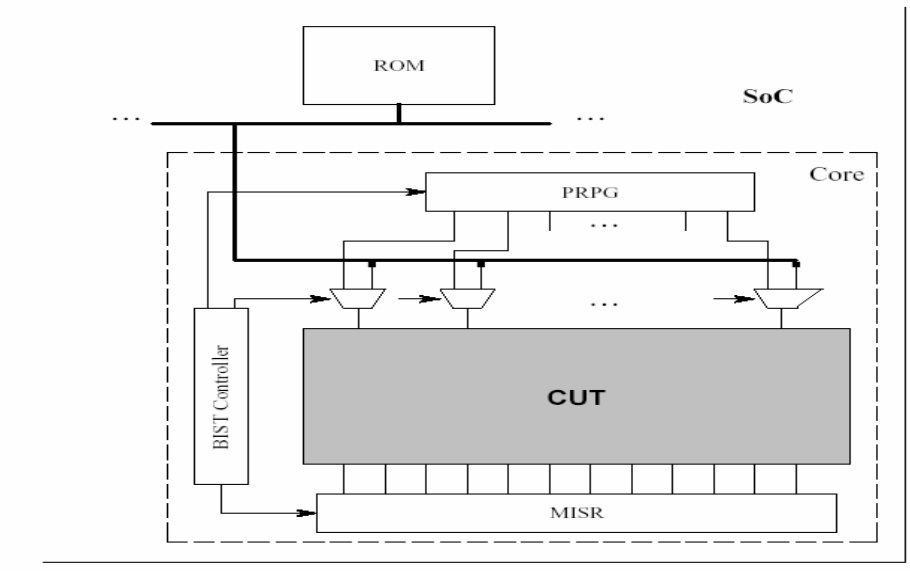


Figure 2.3 An example of hardware-based Hybrid BIST architecture.

Fig. 2.3 shows the main components of Hybrid BIST. Pseudorandom Pattern Generator

(PRPG) and Multi Input Signature Analyzer (MISR) may be implemented using any appropriate structure able to provide pseudorandom test vectors with required degree of randomness; however, the usual implementation is based on LFSRs. As it follows from their names, the first one is needed to generate vectors, and the second one is for test verification. BIST Controller supervises the testing process, and ROM stores deterministic patterns, generated off-line.

At the same time there is another well known Hybrid BIST implementation, when instead of additional logic on IC, existing one is used for BIST components. For instance, it is possible to add some instructions to microprocessor, what will allow using it as a test controller and/or PRPG.

Scan Design

Testing a System-on-Chip with sequential cores is supplemented with additional difficulties, due to feedback loops in the cores. Moreover, the complexity grows with the number of these loops and their lengths. Several Design for Testability (DFT) techniques were proposed to solve the problem, and one of them is *internal scan*. The general idea behind it is to break the feedback paths and to improve observability and controllability of memory elements by integrating an over-laid shift register called scan path [4]. However, this technique forces designers to accept several aspects increasing total cost of IC, such as increase in silicon area, larger number of pins needed, increase in test application time etc. In order to manage the influence of these disadvantages, *partial scan* was introduced in addition to *full scan*. As it implies from the name, only a subset of memory elements is included in the scan path, in this case. On the other hand, full scan allows achieving higher fault coverage.

In our approach, we assume that full scan is used for sequential cores of a SoC. A number of particular differences of testing a SoC with sequential cores instead of combinational ones are provided by Chapter 4 of this thesis.

For a coherent implementation of Hybrid BIST and scan path, STUMPS may be used. The *Self-Test Using MISR and Parallel SRSG* (STUMPS) architecture is shown in Figure 2.5. The acronym SRSG (Shift Register Sequence Generator) may be considered

as equivalent to PRPG, mentioned above.

In case of full scan, every memory element of a core under test (CUT) should be included in the scan path. Often, a scan path is split into several scan chains for a large CUT. The multiplicity of scan chains speeds up test application, because the length of one test cycle is determined by the length of scan path. At the same time, it equals only to the length of the longest scan chain for a CUT with multiple scan chains. However, there is always a trade-off: the more scan chains a core has the more additional scan inputs are required for it. From our point of view, it means increase in the LFSR length and in the amount of memory we need to store one deterministic pattern.

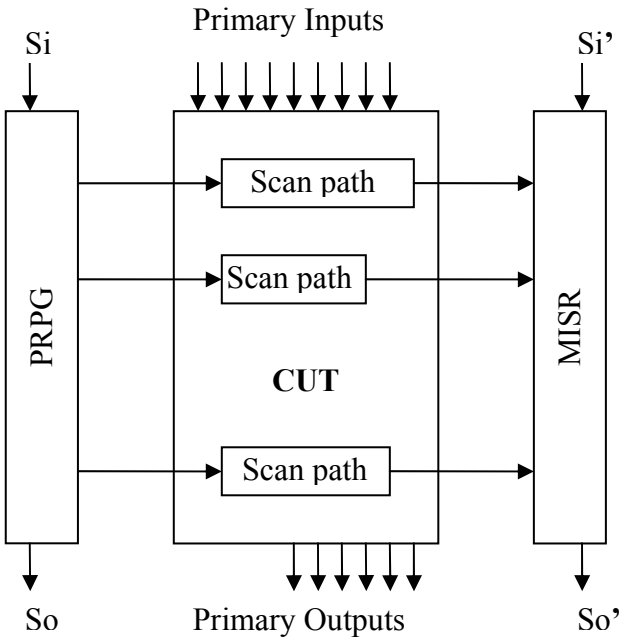


Figure 2.5. Self-Test Using MISR and Parallel SRSG (STUMPS)

The general idea of the STUMPS approach is following. PRPG, MISR and scan registers are clocked simultaneously. All scan chains registers are loaded from PRPG for the number of clock cycles equal to the longest scan chain. Then the Test Controller sends *Scan Enable* signal, the data captured by scan registers is scanned out, and later the results are analyzed by MISR.

As before, the sequences obtained from LFSR are periodic and not linearly independent. The fact, that they are not really random, may affect resulting fault coverage and test performance. Nevertheless, the STUMPS architecture is widely used and, hence,

considered in our approach.

Conclusion

This section has briefly presented testing methodologies that are common nowadays for testing modern integrated circuits. At the same time, a number of necessary terms used to describe our approach further were introduced. The explanations provided in this section do not claim to be comprehensive but, quite the contrary, were aimed to their usage in the following chapters. More specific information about these concepts is available from the literature listed in *References and Bibliography* section.

Chapter 3
Test Time Minimization for Hybrid BIST of Systems-on-Chip with Combinational Cores

This chapter presents a solution to test time minimization problem for core-based systems consisting only combinational cores. We assume a Hybrid BIST approach, where a test set is assembled, for each core, from pseudorandom test patterns that are generated online, and deterministic test patterns that are generated off-line and stored in the system. In this chapter we propose an iterative algorithm to find the optimal combination of pseudorandom and deterministic test sets of the whole system, consisting of multiple cores, under given memory constraints, so that the total test time is minimized. This approach employs a fast estimation methodology in order to avoid exhaustive search and to speed-up the calculation process. Experimental results have shown the efficiency of the algorithm to find near optimal solutions.

This chapter is divided into two sections. First, theoretical presentation of our approach is given [1], followed by section presenting experimental work.

3.1 Theoretical Description of the Proposed Approach

Hybrid BIST Architecture

Recently it was proposed a Hybrid BIST optimization methodology for a single core designs [3]. Such a Hybrid BIST approach starts with a pseudorandom test sequence of length L. At the next stage, the stored test approach takes place: precomputed deterministic test patterns are applied to the core under test to reach the desirable fault coverage. For off-line generation of the deterministic test patterns, arbitrary software test generators may be used, based on deterministic, random or genetic algorithms.

In a Hybrid BIST technique the length of the pseudorandom test is an important parameter that determines the behavior of the whole test process. It is assumed here that

for the Hybrid BIST the best polynomial for the pseudorandom sequence generation will be chosen. By using the best polynomial, we can achieve the maximal fault coverage of the CUT. In most cases this means that we can achieve 100% fault coverage if we run the pseudorandom test long enough. With the Hybrid BIST approach we terminate the pseudorandom test in the middle and remove the latter part of the pseudorandom sequence, which leads to lower fault coverage achievable by the pseudorandom test. The loss of fault coverage should be compensated by additional deterministic test patterns. In general a shorter pseudorandom test set implies a larger deterministic test set. This requires additional memory space, but at the same time, shortens the overall test process, since deterministic test vectors are more efficient in covering faults than the pseudorandom ones. A longer pseudorandom test, on the other hand, will lead to longer test application time with reduced memory requirements. Therefore it is crucial to determine the optimal length L_{OPT} of the pseudorandom test sequence, in order to minimize the total testing cost. The previously proposed methodology enables us to find the most cost-effective combination of the two test sets not only in terms of test time but also in terms of tester/on-chip memory requirements. The efficiency of such approach has been demonstrated so far for individual cores. In this chapter we propose an approach to extend the methodology also for complex systems containing more than one core. We take into account the constraints (memory size) imposed by the system and minimize the testing time for the whole system with multiple cores, while keeping the high fault coverage.

In this chapter we assume the following test architecture: Every core has its own dedicated BIST logic that is capable to produce a set of independent pseudorandom test patterns, i.e. the pseudorandom test sets for all the cores can be carried out simultaneously. The deterministic tests, on the other hand, can only be carried out for one core at a time, which means only one test access bus at the system level is needed. An example of a multi-core system, with such test architecture is given in Figure 3.1.

This example system consists of 5 cores (different ISCAS benchmarks). Using the Hybrid BIST optimization methodology for single core [3] we can find the optimal combination between pseudorandom and deterministic test patterns for every individual core (Figure 3.2). Considering the assumed test architecture, only one deterministic test set can be applied at any given time, while any number of pseudorandom test sessions

can take place in parallel.

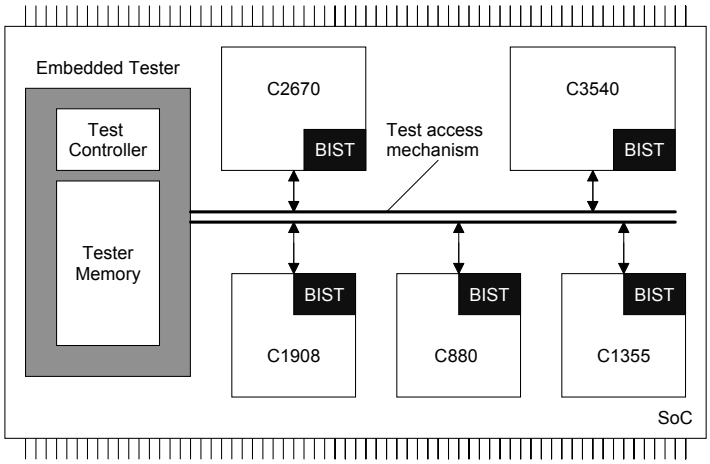


Figure 3.1 A core-based system example with the proposed test architecture

To enforce the assumption that only one deterministic test can be applied at a time, a simple ad-hoc scheduling can be used. The result of this scheduling defines the starting moments for every deterministic test session, the memory requirements, and the total test length *t* for the whole system. This situation is illustrated on Figure 3.2.

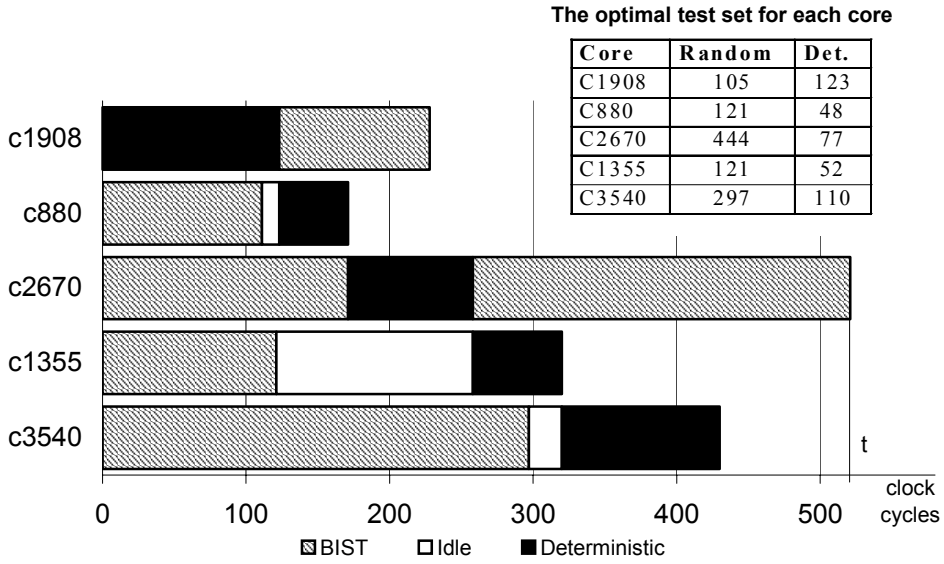


Figure 3.2 Ad-hoc test schedule for Hybrid BIST of the core-based system

As it can be seen from Figure 3.2, the solution where every individual core has the best possible combination between pseudorandom and deterministic patterns usually does

not lead to the best system-level test solution. In the example we have illustrated three potential problems:

- The total test length of the system is determined by the single longest individual test set, while other tests may be substantially shorter;
- The resulting deterministic test sets do not take into account the memory requirements, imposed by the size of the on-chip memory or the external test equipment;
- The proposed test schedule may introduce idle periods, due to the test conflicts between the deterministic tests of different cores;

There are several possibilities for improvement. For example the ad-hoc solution can easily be improved by using a better scheduling strategy. This, however, does not necessarily lead to a significantly better solution as the ratio between pseudorandom and deterministic test patterns for every individual core is not changed. Therefore we have to explore different combinations between pseudorandom and deterministic test patterns for every individual core in order to find a solution where the total test length of the system is minimized and memory constraints are satisfied. In the following sections we will define this problem more precisely, and propose a fast iterative algorithm for calculating the optimal combination between different test sets for the whole system.

Basic Definitions and Problem Formulation

Let us assume a system *S,* consisting of *n* cores $C_1, C_2, ..., C_n$. For every core $C_k \in S$ a complete sequence of deterministic test patterns TD^F_k and a complete sequence of pseudorandom test patterns TP^F_k will be generated. It is assumed that both test sets can obtain by itself maximum achievable fault coverage F_{max}.

Definition 1: A *hybrid BIST* set $TH_k = \{TP_k, TD_k\}$ for a core C_k is a sequence of tests, constructed from the subsets of pseudorandom test sequence $TP_k \subseteq TP^F_k$, and a deterministic test sequence $TD_k \subseteq TD^F_k$. The sequences TP_k and TD_k complement each other to achieve the maximum achievable fault coverage.

Definition 2: A pattern in a pseudorandom test sequence is called *efficient* if it detects at

least one new fault that is not detected by the previous test patterns in the sequence. The ordered sequence of efficient patterns form an *efficient pseudorandom test sequence* $TPE_k = (P_1, P_2, ..., P_n) \subseteq TP_k$. Each efficient pattern $P_j \in TPE_k$ is characterized by the length of the pseudorandom test sequence TP_k, from the start to the efficient pattern P_j, including P_j. Efficient pseudorandom test sequence TPE_k, which includes all efficient patterns of TP^F_k is called *full efficient pseudorandom test sequence* and denoted by TPE^F_k.

***Definition 3*:** The cost of a hybrid test set TH_k for a core C_k is determined by the total length of its pseudorandom and deterministic test sequences, which can be characterized by their costs, $COST_{P,k}$ and $COST_{D,k}$ respectively:

$$COST_{T,k} = COST_{P,k} + COST_{D,k} = \alpha|TP_k| + \beta_k|TD_k|$$

and by the cost of recourses needed for storing the deterministic test sequence TD_k in the memory:

$$COST_{M,k} = \gamma_k|TD_k|$$

The parameters α and β_k can be introduced by the designer to align the application times of different test sequences. For example, when a test-per-clock BIST scheme is used, a new test pattern can be generated and applied in each clock cycle and in this case $\alpha = 1$. The parameter β_k for a particular core C_k is equal to the total number of clock cycles needed for applying a deterministic test pattern from the memory. In a special case, when deterministic test patterns are applied by external test equipment, application of deterministic test patterns may be up to one order of magnitude slower than applying BIST patterns. The coefficient γ_k is used to map the number of test patterns in the deterministic test sequence TD_k into the memory recourses, measured in bits.

Definition 4: When assuming the test architecture described above, a hybrid test set $TH = \{TH_1, TH_2, ..., TH_n\}$ for a system $S = \{C_1, C_2, ..., C_n\}$ consists of hybrid tests TH_k for each individual core C_k, where pseudorandom components of the *TH* can be scheduled in parallel, whereas the deterministic components of *TH* must be scheduled in sequence due to the shared test resources.

Definition 5: $J = (j_1, j_2, ..., j_n)$ is called the *characteristic vector* of a hybrid test set $TH = \{TH_1, TH_2, ..., TH_n\}$, where $j_k = |TPE_k|$ is the length of the efficient pseudorandom test sequence $TPE_k \subseteq TP_k \subseteq TH_k$.

According to Definition 2, for each j_k corresponds a pseudorandom subsequence $TP_k(j_k) \subseteq TP^F_k$, and according to Definition 1, any pseudorandom test sequence $TP_k(j_k)$ should be complemented with a deterministic test sequence, denoted with $TD_k(j_k)$, that is generated in order to achieve the maximum achievable fault coverage. Based on this we can conclude that the characteristic vector J determines entirely the structure of the hybrid test set TH_k for all cores $C_k \in S$.

Definition 6: The test length of a hybrid test $TH = \{TH_1, TH_2, ..., TH_n\}$ for a system $S = \{C_1, C_2, ..., C_n\}$ is given by:

$$COST_T = \max\{\max_k(\alpha|TP_k| + \beta_k|TD_k|), \sum_k \beta_k|TD_k|\}$$

The total cost of resources needed for storing the patterns from all deterministic test sequences TD_k in the memory is given by:

$$COST_M = \sum_k \gamma_k|TD_k|$$

Definition 7: Let us introduce a generic cost function $COST_{M,k} = f_k(COST_{T,k})$ for every core $C_k \in S$, and an integrated generic cost function $COST_M = f_k(COST_T)$ for the whole system S.

The functions $COST_{M,k} = f_k(COST_{T,k})$ will be created in the following way. Let us have a hybrid BIST set $TH_k(j) = \{TP_k(j), TD_k(j)\}$ for a core C_k with j efficient patterns in the pseudorandom test sequence. By calculating the costs $COST_{T,k}$ and $COST_{M,k}$ for all possible hybrid test set structures $TH_k(j)$, i.e. for all values $j = 1, 2, \ldots, |TPE^F_k|$, we can create the cost functions $COST_{T,k} = f_{T,k}(j)$, and $COST_{M,k} = f_{M,k}(j)$. By taking the inverse function $j = f'_{T,k}(COST_{T,k})$, and inserting it into the $f_{M,k}(j)$ we get the generic cost function $COST_{M,k} = f_{M,k}(f'_{T,k}(COST_{T,k})) = f_k(COST_{T,k})$ where the memory costs are directly related to the lengths of all possible hybrid test solutions.

The integrated generic cost function $COST_M = f(COST_T)$ for the whole system is the sum of all cost functions $COST_{M,k} = f_k(COST_{T,k})$ of individual cores $C_k \in S$.

From the function $COST_M = f(COST_T)$ the value of $COST_T$ for every given value of $COST_M$ can be found. The value of $COST_T$ determines the lower bound of the length of the hybrid test set for the whole system. To find the component j_k of the characteristic vector J, i.e. to find the structure of the hybrid test set for all cores, the equation $f_{T,k}(j) = COST_T$ should be solved.

The objective of this chapter is to find a shortest possible ($min(COST_T)$) hybrid test sequence TH_{opt} when the memory constraints are not violated $COST_M \leq COST_{M,LIMIT}$.

Hybrid Test Sequence Computation Based on Cost Estimates

By knowing the generic cost function $COST_M = f(COST_T)$, the total test length $COST_T$ at any given memory constraint $COST_M \leq COST_{M,LIMIT}$ can be found in a straightforward way. However, the procedure to calculate the cost functions $COST_{D,k}(j)$ *and* $COST_{M,k}(j)$ is very time consuming, since it assumes that the deterministic test set TD_k for each $j = 1, 2, \ldots, |TPE^F_k|$ has to be available. This assumes that after every efficient pattern $P_j \in TPE_k \subseteq TP_k$, $j = 1, 2, \ldots, |TPE^F_k|$ a set of not yet detected faults $F_{NOT,k}(j)$ should be calculated. This can be done either by repetitive use of the automatic test pattern generator or by systematically analyzing and compressing the fault tables for each j. Both procedures are accurate but time-consuming and therefore not feasible for larger designs. To overcome the complexity explosion problem we propose an iterative algorithm, where costs $COST_{M,k}$ and $COST_{D,k}$ for the deterministic test sets TD_k can be found based on estimates. The estimation method is based on fault coverage figures and does not require accurate calculations of the deterministic test sets for not yet detected faults $F_{NOT,k}(j)$.

In the following we will use $FD_k(i)$ and $FPE_k(i)$ to denote the fault coverage figures of the test sequences $TD_k(i)$ and $TPE_k(i)$, correspondingly, where i is the length of the test sequence.

Procedure 1: Estimation of the length of the deterministic test set TD_k.

1. Calculate, by fault simulation, the fault coverage functions $FD_k(i)$, $i = 1, 2, \ldots, |TD^F_k|$, and $FPE_k(i)$, $i = 1, 2, \ldots, |TPE^F_k|$. The patterns in TD^F_k are ordered in such

the way that each pattern put into the sequence contributes with maximum increase in fault coverage.

2. For each $i^* \leq |TPE^F_k|$, find the fault coverage value F^* that can be reached by a sequence of patterns $(P_1, P_2, \ldots, P_{i^*}) \subseteq TPE_k$ (see Figure 3).

3. By solving the equation $FD_k(i) = F^*$, find the maximum integer value j^* that satisfies the condition $FD_k(j^*) \leq F^*$. The value of j^* is the length of the deterministic sequence TD_k that can achieve the same fault coverage F^*.

4. Calculate the value of $|TD^E_k(i^*)| = |TD^F_k| - j^*$ which is the number of test patterns needed from the TD^F_k to reach to the maximum achievable fault coverage.

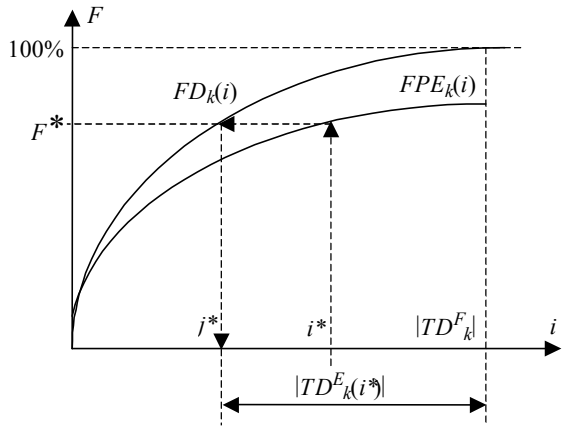


Figure 3.3 Estimation of the length of the deterministic test sequence

The value $|TD^E_k(i^*)| = |TD^F_k| - j^*$, calculated by the Procedure 1, can be used to estimate the length of the deterministic test sequence TD_k in the hybrid test set $TH_k = \{TP_k, TD_k\}$ with i^* efficient test patterns in TP_k, ($|TPE_k| = i^*$).

By finding $|TD^E_k(j)|$ for all $j = 1, 2, \ldots, |TPE^F_k|$ we get the cost function estimate $COST^E_{D,k}(j)$. Using $COST^E_{D,k}(j)$, other cost function estimates $COST^E_{M,k}(j)$, $COST^E_{T,k}(j)$ and $COST^E_{M,k} = f^E_k(COST^E_{T,k})$ can be created according to the Definitions 3 and 7.

Finally, by adding cost estimates $COST^E_{M,k} = f^E_k(COST^E_{T,k})$ of all cores, we get the hybrid BIST cost function estimate $COST^E_M = f^E(COST^E_T)$ for the whole system.

Test Length Minimization under Memory Constraints

As described above, the exact calculations for finding the cost of the deterministic test set $COST_{M,k} = f_k(COST_{T,k})$ are very time-consuming. Therefore we will use the cost estimates, calculated by Procedure 1 in the previous subsection, instead. Using estimates can give us a quasi-minimal solution for the test length of the hybrid test at given memory constraints. After obtaining a quasi-minimal solution, the cost estimates can be improved and another, better, quasi-minimal solution can be calculated. This iterative procedure will be continued until we reach the final solution.

Procedure 2: Test length minimization.

1. Given the memory constraint $COST_{M,LIMIT}$, find the estimated total test length $COST^{E*}_{T}$ as a solution to the equation $f^{E}(COST^{E}_{T}) = COST_{M,LIMIT}$.

2. Based on $COST^{E*}_{T}$, find a candidate solution $J^* = (j^*_1, j^*_2, ..., j^*_n)$ where each j^*_k is the maximum integer value that satisfies the equation $COST^{E}_{T,k}(j^*_k) \leq COST^{E*}_{T}$.

3. To calculate the exact value of $COST^*_M$ for the candidate solution J^*, find the set of not yet detected faults $F_{NOT,k}(j^*_k)$ and generate the corresponding deterministic test set TD^*_k by using an ATPG algorithm.

4. If $COST^*_M = COST_{M,LIMIT}$, go to the Step 9.

5. If the difference $|COST^*_M - COST_{M,LIMIT}|$ is bigger than that in the earlier iteration make a correction $\Delta t = \Delta t/2$, and go to Step 7.

6. Calculate a new test length $COST^{E,N}_{T}$ from the equation $f^{E}_{k}(COST^{E}_{T}) = COST^{*}_{M}$, and find the difference $\Delta t = COST^{E,*}_{T} - COST^{E,N}_{T}$.

7. Calculate a new cost estimate $COST^{E,*}_{T} = COST^{E,*}_{T} + \Delta t$ for the next iteration.

8. If the value of $COST^{E,*}_{T}$ is the same as in an earlier iteration, go to Step 9, otherwise go to Step 2.

9. *END:* The vector $J^* = (j^*_1, j^*_2, ..., j^*_n)$ is the solution.

To illustrate the above procedure, in Figures 3.4 and 3.5 an example of the iterative search for the shortest length of the hybrid test is given. Figure 3.4 represents all the basic cost curves $COST^E_{D,k}(j)$, $COST^E_{P,k}(j)$, and $COST^E_{T,k}(j)$, as functions of the length j of TPE_k where j_{min} denotes the optimal solution for a single core hybrid BIST optimization problem [3].

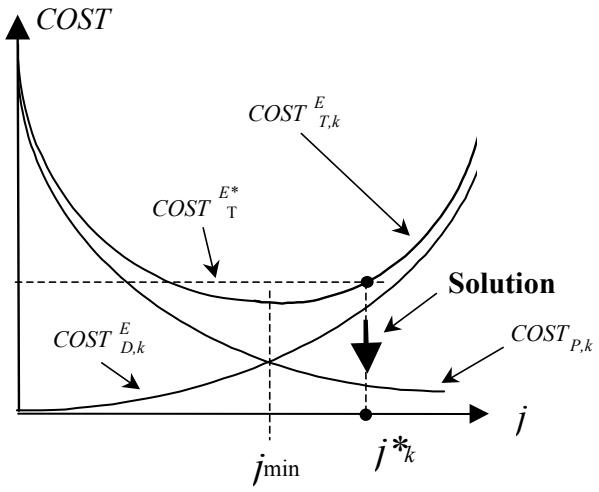


Figure 3.4 Cost curves for a given core C_k

Figure 3.5 represents the estimated generic cost function $COST^E_M = f^E(COST^E_T)$ for the whole system. At first (Step 1), the estimated $COST^{E*}_T$ for the given memory constraints is found (point 1 on Figure 3.5). Then (Step 2), based on $COST^{E*}_T$ the length j^*_k of TPE_k for the core C_k in Figure 4 is found. This procedure (Step 2) is repeated for all the cores to find the characteristic vector J^* of the system as the first iterative solution. After that the real memory cost $COST^{E*}_M$ is calculated (Step 3, point 1* in Figure 3.5). As we see in Figure 3.5, the value of $COST^{E*}_M$ in point 1* violates the memory constraints. The difference Δt_1 is determined by the curve of the estimated cost (Step 5). After correction, a new value of $COST^{E*}_T$ is found (point 2 on Figure 3.5). Based on $COST^{E*}_T$, a new J^* is found (Step 2), and a new $COST^{E*}_M$ is calculated (Step 3, point 2* in Figure 3.5). An additional iteration via points 3 and 3* can be followed in Figure 3.5.

It is easy to see that Procedure 2 always converges. By each iteration we get closer to the memory constraints level, and also closer to the minimal test length at given constraints. However, the solution may be only near-optimal, since we only evaluate solutions derived from estimated cost functions.

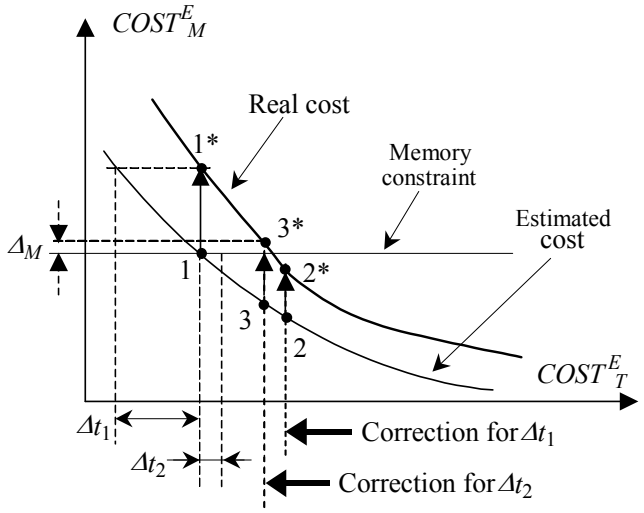


Figure 3.5 Minimization of the test length

The theory provided above was proven with experiments described in the next section.

3.2 Experiments

Setup

The experiments for this part of the research were performed with *ISCAS'85* benchmarks as sample combinational cores for virtual SoCs and listed in Table 3.1.

Table 3.1 ISCAS'85 Benchmarks used for the experiments.

Design name	Number of inputs	Number of outputs	Used in systems
c432	36	7	S1, S2
c499	41	32	2xS1, S2
c880	60	26	S1, S2, 2xS3
c1355	41	32	S2, S3
c1908	33	25	S2, S3
c3540	50	22	S3
c5315	178	123	2xS1, S2, S3
c6288	32	32	S2

For simulation software, *Turbo Tester* tools were chosen (Tallinn Technical University). These tools were run on Solaris machine with Sun OS 5.8. The final charts were created with *Microsoft Excel XP*. A number of additional programs were written in C language, especially for these experiments.

The rest of this section describes step by step actions performed during the experiments in such a way that anyone could repeat them.

Pseudorandom Pattern Generation

Our approach assumes that for cost estimation algorithm we have pseudorandom and deterministic test sequences generated for each core. Moreover, it is desirable, but not necessary, that both test sequences would be able to obtain fault coverage as close to 100% as possible.

We have used ***bist*** tool to generate pseudorandom sequences. The execution command for it is following:

bist -rand -glen 36 -alen 7 -simul bilbo –count 3000 c432

Here option *–rand* means that *initial vector* and *characteristic polynomial* for LFSR, emulator used in the program, will be generated randomly. *–glen* determines the length of PRPG generated vector, which should not be shorter then desired resulting patterns. The length of a test pattern for combinational circuits equals to the number of its inputs. That is why in the command we use *36* – the number of primary inputs of c432 (Table 3.1). *–alen* determines the length of MISR and may not be shorter then number of outputs of the circuit. Option *–simul bilbo* chooses BILBO BIST architecture, where two different LFSRs are used one for PRPG and another one for MISR. *–count* defines the number of generated vectors.

After several attempts for every core we succeeded to achieve pseudorandom test pattern sets with 100% fault coverage for most of the cores. The resulting sets were stored in *.tst files.

It is important that during the rest of the experiments always the same test patterns are

used. However, it is enough to save only initial vector and characteristic polynomial and remember the number of patterns for each core, to be sure that next time the same test pattern sequence is generated.

Deterministic Pattern Generation

Tool ***generate*** was used to obtain deterministic test pattern sequences. A sample command for its execution is:

generate –backtracks 300 c432

Here the larger number of backtracks allows to achieve the higher fault coverage, although execution time increases. The resulting pattern sequences were saved as files, but assuming that program test generation algorithm always works in the same way, we should get the same sequences on any other run of the program.

Further, the tools ***optimize*** and ***analyze*** were used, to minimize the resulting test sets. The first one finds and eliminates from the set of generated patterns such ones that do not influence the final fault coverage. The second tool performs fault simulation and saves the results in the format we need for our experiments.

The results of test pattern generation are presented in Table 3.2.

Table 3.2 Number of pseudorandom and deterministic patterns for each core.

Design name	Number of pseudorandom patterns	Fault coverage	Number of generated deterministic patterns	Number of compacted deterministic patterns	Fault coverage
c432	3000	100%	67	39	100%
c499	12000	100%	96	80	100%
c880	15000	100%	60	41	100%
c1355	7000	100%	104	82	100%
c1908	20000	100%	69	34	100%
c3540	20000	98,99%	228	122	100%
c5315	7000	100%	127	90	100%
c6288	7000	100%	62	37	100%

While generating pseudorandom patterns it was not our goal to achieve the minimal

number of them in test sets for the highest possible coverage. As it will be shown further, even if we have some extra patterns at the end of pseudorandom test set, which do not increase the final fault coverage, it does not influence our experimental results.

On the other hand, we have to work with the minimal set of deterministic patterns that achieve 100% fault coverage (or the highest possible), in order to guarantee that the minimal amount of memory is used to store the deterministic test set of every core.

Reporting Numbers of Faults Covered by Test Patterns

At this moment our task is to determine fault coverage obtained by every pattern in the test sets. While generating pseudorandom patterns, tools *bist* and *generate* also perform fault simulation. Another Turbo Tester tool ***report*** with option *–progress* allows extracting the needed information. Parts of report files generated by this tool for *c432* core are provided bellow:

Pseudorandom test simulation report

Coverage progress report:

*Pattern **1**: coverage 9.615385 % (**60**/624)*
Pattern 2: coverage 16.826923 % (105/624)
Pattern 3: coverage 24.839744 % (155/624)
Pattern 4: coverage 33.173077 % (207/624)
Pattern 5: coverage 40.865385 % (255/624)
...
Pattern 234: coverage 98.237179 % (613/624)
Pattern 274: coverage 98.717949 % (616/624)
Pattern 288: coverage 99.679487 % (622/624)
Pattern 314: coverage 99.839744 % (623/624)
*Pattern **465**: coverage 100.000000 % (**624**/624)*

Deterministic test simulation report

Coverage progress report:

*Pattern **1**: coverage 16.826923 % (**105**/624)*
Pattern 2: coverage 27.083333 % (169/624)
Pattern 3: coverage 35.897436 % (224/624)
Pattern 4: coverage 41.666667 % (260/624)
Pattern 5: coverage 45.993590 % (287/624)
...
Pattern 35: coverage 96.794872 % (604/624)
Pattern 36: coverage 97.916667 % (611/624)
Pattern 37: coverage 98.237179 % (613/624)
Pattern 38: coverage 98.717949 % (616/624)
*Pattern **39**: coverage 100.000000 % (**624**/624)*

Every line of this report shows the number of efficient test pattern and amount of faults, covered by it and previous patterns together, from all possible faults of the CUT.

At this moment it is possible to modify the report file with Excel or to write a simple program in C (the second way is more convenient, because this operation will be repeated for a number of cores) to have only the following information:

Table 3.3 Reporting the amount of faults covered.

For pseudorandom test		**For deterministic test**	
1	*60*	*1*	*105*
2	*105*	*2*	*169*
3	*155*	*3*	*224*
4	*207*	*4*	*260*
5	*255*	*5*	*287*
...		...	
234	***613***	*35*	*604*
274	*616*	*36*	*611*
288	*622*	*37*	*613*
314	*623*	*38*	*616*
465	*624*	*39*	*624*

The first column of the both sequences from Table 3.3 represents the order number of an efficient pattern in the whole test set. The second column shows amount of faults covered. The table for pseudorandom patterns may always be extrapolated:

...	
234	***613***
235	*613*
236	*613*
...	
272	*613*
273	***613***
274	*616*
...	

At the same time, every deterministic pattern is efficient.

Now, we save the data described in Table 3.3 as files for every core for both pseudorandom and deterministic tests. As a result, we have 16 files.

Estimations' Generation

As it may be seen from Table 3.3, we need 234 pseudorandom patterns (boxed) or 37 deterministic ones to cover the same amount of faults (613) for *c432* core. It means that if we terminate pseudorandom test just after applying pattern number 234, then we may probably need to apply 2 deterministic patterns to cover the remaining faults (39-37=2). For our estimation we do not consider what faults exactly we covered by the moment and take into account only their amount. If the exact amount of faults covered by certain number of pseudorandom patterns could not be found in the saved table for deterministic, then we take the line with that many deterministic patterns that cover the closest larger amount of faults.

This operation does not require much calculation time and may be implemented by this small C function:

```
for (i=1; i<=total_num_of_random_patterns; i++){
      for(j=1; j<=total_num_of_deterministic_patterns; j++){
            if(random_faults[i]<=deterministic_faults[j]){
                  det=total_num_of_deterministic_patterns -
                        deterministic_patterns[i];
            }
      }
      fprintf(FP_output_file, "%d %d\n", random_patterns[i],
det);
   }
```

The resulting estimation table should look like this:

Table 3.4 Estimations for numbers of required deterministic patterns for c423.

1	39	19	25	43	13	125	7
2	38	22	24	44	12	130	6
3	38	23	23	48	12	135	6
4	37	24	22	50	12	139	6
5	36	25	22	52	9	143	6
6	33	26	22	55	9	159	5
7	32	27	20	66	9	185	4
8	32	28	20	69	9	195	4
9	32	29	20	78	9	197	4
10	28	31	18	79	8	216	3
11	28	34	18	85	8	234	2
12	28	35	18	***97***	***8***	274	1
13	27	38	14	***102***	***7***	288	1
14	26	39	13	105	7	314	1
17	25	41	13	119	7	465	0
18	25	42	13	123	7		

The first column of Table 3.4 represents number of pseudorandom patterns applied; the second shows how many deterministic patterns we estimate we may need.

Now, we need to extrapolate this table so that estimations not only for efficient, but for all pseudorandom patterns would be represented.

1	39
2	38
. . .	
97	***8***
98	8
99	8
100	8
101	8
102	***7***
. . .	
463	1
464	1
465	0
466	0
. . .	
2999	0
3000	0

Using the previously saved data we generate a file with such a table as shown above for every core (***Program 1***).

The next step is to find out how much memory would be used by a certain core at every possible total test length. Total test length is the sum of time we need too apply desired number of pseudorandom patterns and the time needed to apply predetermined number of deterministic patterns, if any, for the core. That means, if we have chosen a test for a core with combination of deterministic and pseudorandom parts as shown in the first line of Table 3.4, then the total test length will be 39+1=**40** clock cycles.

To store 1 deterministic pattern for a combinational core the amount of bits needed equals to number of this core's inputs. For example, for c432 it is 36 bits (from Table 3.1). For 39 patterns we need 39*36= 1404 (bits). Using this principal, we calculate estimated memory and total test length for every line of stored tables.

40	1404
40	1368
41	1368
41	1332
41	1296
39	1188
…	

Further, we just choose for every total test length the minimal value of memory may be used, and sort the results by the first column. Finally, we obtain a table like the following Table 3.5:

Table 3.5 Memory estimation for every total test length.

38	1008	50	648	62	324	74	324	2993	0
39	1008	51	648	63	324	75	324	2994	0
40	936	52	468	64	324	76	324	2995	0
41	936	53	468	65	324	77	324	2996	0
42	900	54	468	66	324	78	324	2997	0
43	900	55	468	67	324	…		2998	0
44	900	56	432	68	324			2999	0
45	900	57	432	69	324	2988	0	3000	0
46	792	58	432	70	324	2989	0		
47	720	59	432	71	324	2990	0		
48	720	60	432	72	324	2991	0		
49	648	61	324	73	324	2992	0		

8 files were created to store similar data for every core from Table 3.1. For all the calculations, another program was implemented in C language (***Program 2***).

In order to have a graphical representation of the data, we insert it into Excel and build charts. If we sum up estimated memory cost for all cores used in a SoC at some particular total test length, the result will represent the memory needed for the whole SoC.

For our experiments we have chosen 3 virtual systems, showed bellow:

Table 3.6 List of cores for the experimental SoCs.

System name	**S1** 6 cores	**S2** 7 cores	**S3** 5 cores
List of used cores	c5315	c432	c880
	c880	c499	c5315
	c432	c880	c3540
	c499	c1355	c1908
	c499	c1908	c880
	c5315	c5315	
		c6288	

Due to lack of available benchmarks, we had to use sometimes the same ones twice in one system, however we consider them as different cores, and it does not influence the final results.

The resulting chart with memory cost estimation curves for one system is shown in Figure 3.5.

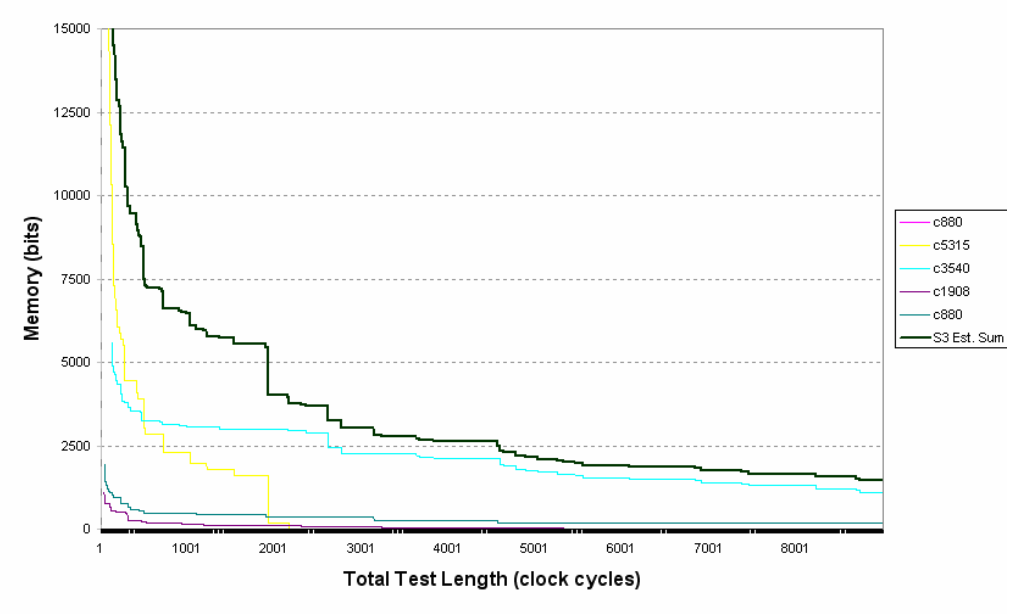


Figure 3.5 Memory cost estimation curves for system S1.

Similar charts were built for experimental systems S2 and S3 in the same way.

Exhaustive Simulation

As soon as estimation costs are calculated, we need to obtain the real results, in order to verify the estimates. A program ***sub_faults***, written by Elmet Orasson, was used for this purpose. However, usual deterministic patterns generation, after applying every efficient pseudorandom pattern for a core, can be used as well. As its input the program uses 2 files containing pseudorandom and deterministic test pattern sets. The files also contain fault tables, which provide information about what faults every particular test pattern covers. This program finds out what faults were covered by first N pseudorandom patterns, and then looks from the deterministic patterns set for those patterns, which cover the rest of the faults. If we run the program for N=1, 2, … , *last_random_pattern*, we should get the real behavior of memory cost. A script was written for this purpose. Its execution is very time consuming, because for every core thousands of iterations should be performed. After every run of *sub_faults*, the script also executes *optimize* and *analyze* tools to obtain the minimal set of deterministic patterns.

A sample report of the script used for c5315 is provided bellow:

Step #1 1 vectors opt=90
Step #2 2 vectors opt=89
Step #3 3 vectors opt=89
Step #4 4 vectors opt=89
Step #5 5 vectors opt=88
Step #6 6 vectors opt=86
Step #7 7 vectors opt=86
Step #8 8 vectors opt=86
Step #9 9 vectors opt=86
...

At this point, we achieved the similar data that we did, while estimating number of required deterministic patterns in the previous subsection after the extrapolation. Therefore, after handling this information with ***Program 2*** mentioned above, we will have real memory values at each possible total test length, for every core.

Further, we just build charts with Excel, based on these results. Final charts for all three experimental systems S1, S2 and S3, containing both estimation and real values, are presented in Figures 3.6-3.8.

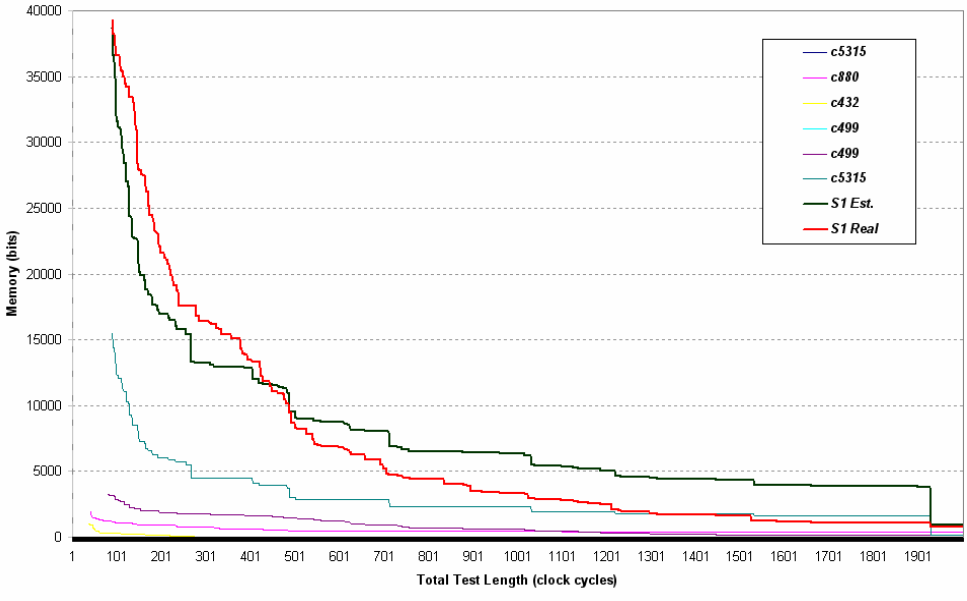


Figure 3.6 Memory Cost estimations and real values for S1

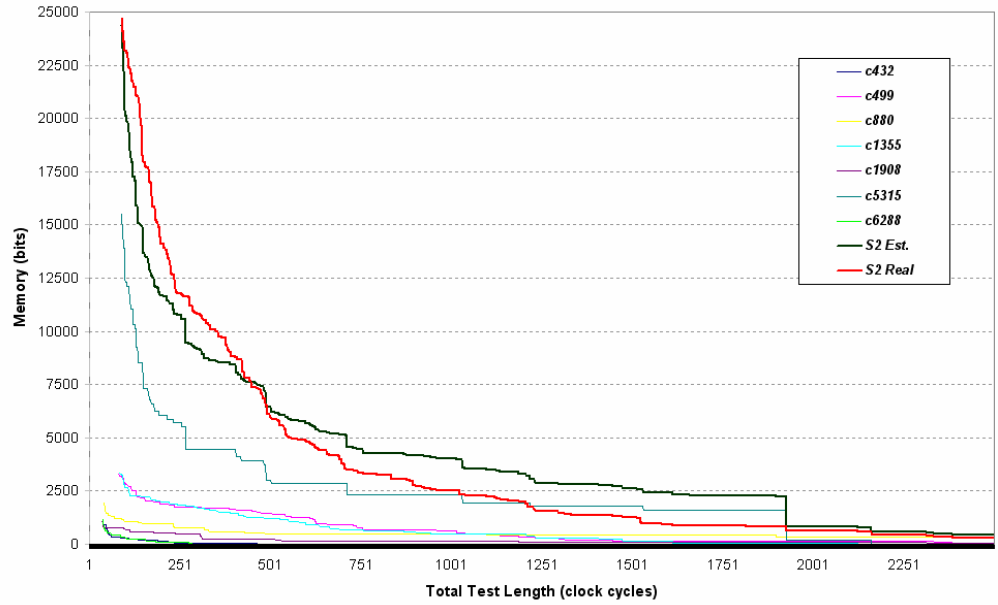


Figure 3.7 Memory Cost estimations and real values for S2

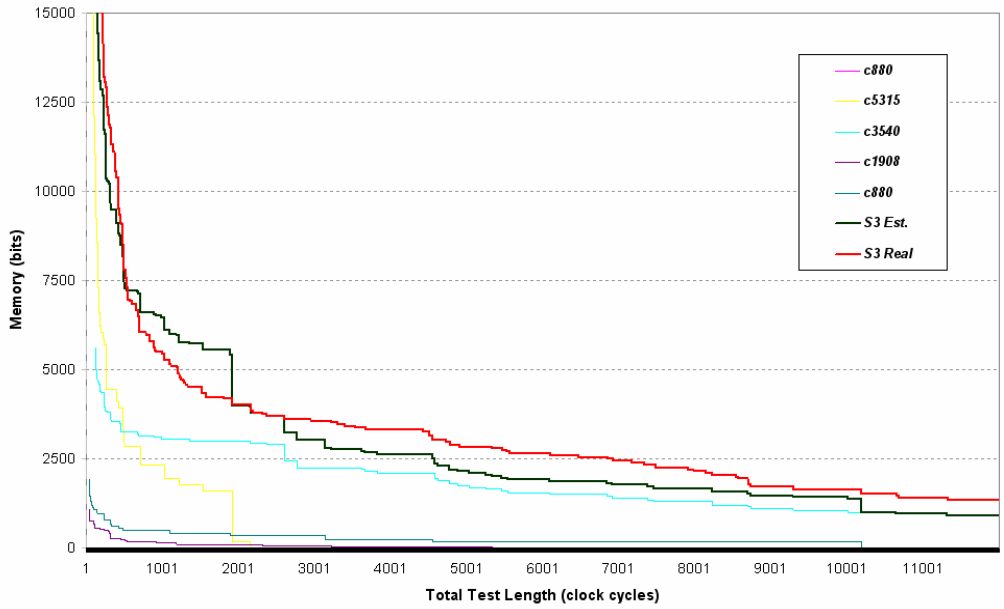


Figure 3.8 Memory Cost estimations and real values for S3

CPU Time Measurement for Performed Computations

The adjustment algorithm is well described in the theoretical part; this section provides a small piece of manual verification carried out. The results are shown in Table 3.7.

Table 3.7 Manual adjustment for S1 with Memory Constraint equal to 20000 bits.

System name	S1					
Memory Constraint	20 000 bits					
Step #	Real Memory	Clock for real value	Estimated Memory	Clock for est. value	Delta	New clock
Initial			19844	162		
1	27440	162	26490	121	41	***203***
2	21592	***203***	20912	148	55	217
3	20716	217	20200	150	67	229
4	19114	229	19808	164	65	227
5	19114	227	19808	164	63	225
6	19530	225	19808	164	**61**	223
7	19886	223	19844	162	**61**	

We use previously calculated, and applied in Excel, data for this algorithm. As it implies from the Table 3.7, it is needed 7 iterations until we reach the same *delta* as in the previous step (boxed). In terms of time, one iteration means one call of particular number of pseudorandom patterns simulation and generation deterministic patterns for the remaining faults. The both actions should be implemented for all the cores used in an experimental SoC. The values of time, required to perform one such iteration for the whole experimental systems are presented in Table 3.8.

Table 3.8 CPU time used to perform one adjustment iteration for every system.

System name	Time for one iteration (seconds)
S1	28.54
S2	33.46
S3	58.28

We have chosen several memory constraints for every experimental SoC and manually emulated work of the adjustment algorithm. While this operation, the numbers of steps needed for every adjustment were found. Therefore, by multiplying them with the values from Table 3.8, we obtain the time our approach needs to find a solution (Table 3.9). Here, we do not consider the CPU time used by estimation process, because it is much less then the time spent for the adjustment, and would not influence the presented numbers.

For the CPU time required by Exact Approach we report the time used by our script

(mentioned in previous subsection) to calculate the complete test cost data for every system.

Table 3.9. Experimental results. Final table.

System	Number of cores	Memory Constraint (bits)	Exact Approach		Our Approach	
			Total Test Length (clocks)	CPU Time (seconds)	Total Test Length (clocks)	CPU Time (seconds)
S1	6	20 000	222	3772.84	223	199.78
		10 000	487		487	57.08
		7 000	552		599	114.16
S2	7	14 000	207	3433.10	209	167.3
		5 500	540		***542***	133.84
		2 500	1017		1040	200.76
S3	5	7 000	552	10143.14	586	174.84
		3 500	3309		3413	291.40
		2 000	8549		8 556	407.96

In Table 3.9 we compare our approach where the test length is found based on estimates with an exact approach, where deterministic test sets have been found by manipulating the fault tables for every possible switching point between pseudorandom and deterministic test patterns. As it can be seen from the results, our approach can give significant speedup (more than order of magnitude), while retaining acceptable accuracy.

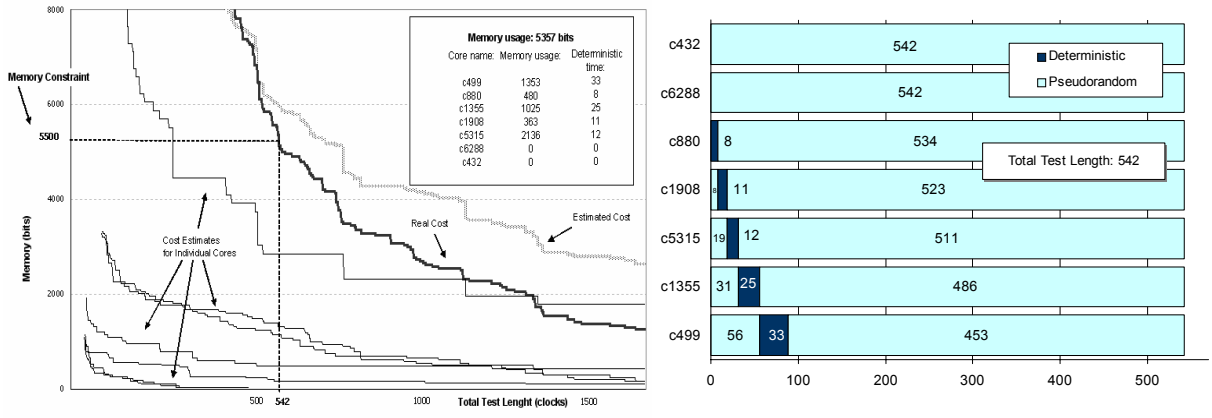


Figure 3.9 A test schedule for a found solution (S2, M_{LIMIT} = 5500).

Figure 3.9 provides a graphical representation of the solution found for the system S2 with the memory constraint equal to 5500 bits (bold in Table 3.9) with a possible test

schedule for this case.

To sum up the experimental part of this chapter, a list of performed steps is presented bellow:

1. *Test pattern sets generation*
   - *1.1. Pseudorandom pattern generation*
   - *1.2. Deterministic pattern generation*
2. *Estimation*
   - *2.1. Reporting number of faults covered by test patterns*
   - *2.2. Estimation of required number of deterministic patterns after every new pseudorandom one applied*
   - *2.3. Estimation of required memory for every possible total test length*
3. *Exhaustive simulation*
   - *3.1. Script execution to obtain the number of required deterministic patterns after every new pseudorandom one is applied*
   - *3.2. Extracting information about required memory for every possible total test length*
4. *Manual emulation of adjustment algorithm*
5. *CPU time used for all operations measurement (summing up separate values)*
6. *Results representation*

Conclusion

Chapter 3 has presented an approach to the test time minimization problem for Systems-on-Chip with combinational cores. A heuristic algorithm was proposed to minimize the test length for a given memory constraint. The algorithm is based on the analysis of different cost relationships as functions of the hybrid BIST structure. To avoid the exhaustive exploration of solutions, a method for the cost estimation of the deterministic component of the hybrid test set was proposed. It also provides an iterative algorithm, based on the proposed estimates, to minimize the total test length of the hybrid BIST solution under the given memory constraints. Experimental results show very high speed of the algorithm compared to the exact calculation method.

Chapter 4
Test Time Minimization for Hybrid BIST of Systems-on-Chip with Sequential Cores

This chapter examines the test time minimization problem for Systems-on-Chip, containing sequential cores with STUMPS architecture. As in the previous discussion, we assume a Hybrid BIST approach, where a test set is assembled, for each core, from pseudorandom test patterns that are generated online, and deterministic test patterns that are generated off-line and stored in the system. This chapter will mostly describe the differences in the approach presented above for Systems-on-Chip with combinational cores.

The first part of the chapter discuses additional difficulties caused by sequential cores usage and a possible solution for them [2]. The second part of it presents experiments carried out.

4.2 Theoretical discussion

Hybrid BIST Architecture

As it was shown before, generally, a Hybrid BIST approach combines two different types of tests. It starts with a pseudorandom test sequence of length L and continues with precomputed deterministic test patterns, stored in the system, in order to reach the desirable fault coverage.

There are two widely used BIST schemes: test-per-clock and test-per-scan. Our earlier discussion was concentrated on systems with combinatorial cores and therefore a test-per-clock scheme could be used. In this chapter our objective is to provide a solution to the test time minimization problem in case of sequential cores. As testing of sequential cores is very complex process and development of efficient test pattern generation

algorithm for sequential cores is outside the scope of this thesis then it is assumed here that every core contains one or several scan paths (full scan). Therefore a test-per-scan scheme has to be used, and for every individual core, the Self-Test Using MISR and Parallel Shift Register Sequence Generator (STUMPS) architecture is assumed.

While every core has its own STUMPS architecture then at the system level we assume the following architecture. Every core's BIST logic is capable to produce a set of independent pseudorandom test patterns, i.e. the pseudorandom test sets for all the cores can be carried out simultaneously. The deterministic tests, on the other hand, can only be carried out for one core at a time, what means only one test access bus at the system level is needed. An example of a multi-core system, with such test architecture is given in Figure 4.1.

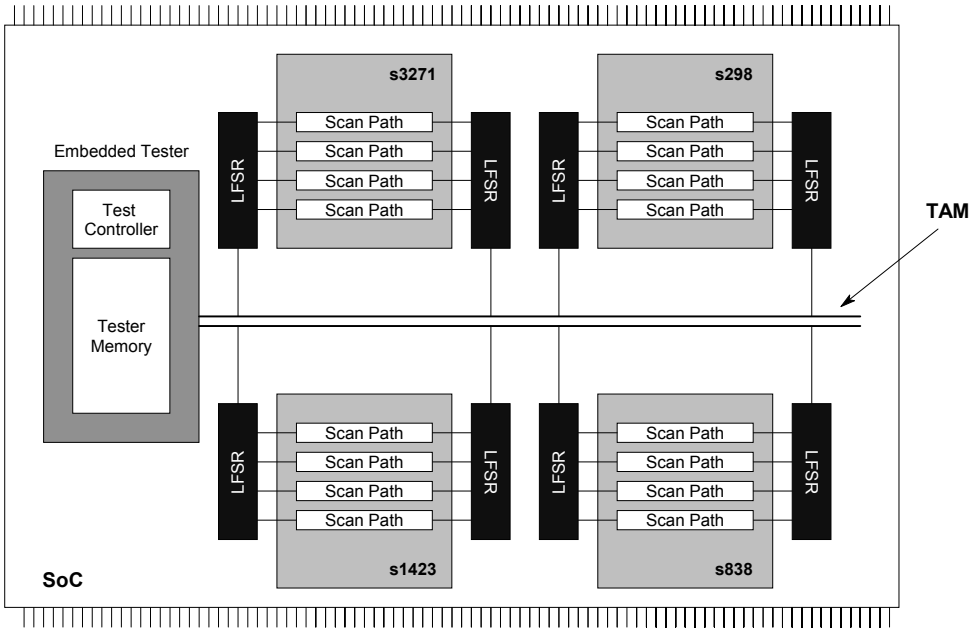


Figure 4.1 A core-based system example with the proposed test architecture

This example system consists of 4 cores (different ISCAS benchmarks). As we have shown earlier, the solution where every individual core has the best possible combination between pseudorandom and deterministic patterns usually does not lead to the best system-level test solution. Several reasons for this were named in section 3.1.

The problem can be solved in a straightforward way if the supplementary deterministic test set for every possible length of the pseudorandom set is available, what requires very expensive in terms of time exhaustive simulation. Another, cheaper solution was proposed in chapter 3 and can be used for systems with sequential cores, if differences

discussed bellow are taken into account.

Differences in the Testing Approach

Due to different Hybrid BIST scheme used for sequential cores, one test cycle no longer equals to one clock cycle and depends on the longest scan chain of a particular CUT. It means for us, that a solution can exist only at some certain total test lengths (measured in clock cycles), when the test cycles for each core participating in the test are accomplished. The same reason introduces some additional limitations for the moments of time when the deterministic part of the total test for a core from a system can start. In fact, it may cause idle periods in the final schedule. An example for this situation is shown in Figure 4.2.

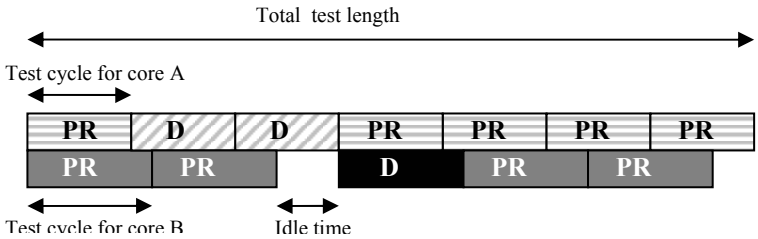


Figure 4.2 Additional scheduling difficulties for sequential cores.

Let us assume, 2 cores *A* and *B* belong to one hypothetical SoC and it was determined that, to test this SoC with some particular memory constraint, core *A* needs to apply 5 pseudorandom patterns and 2 deterministic ones, while core *B* requires 4 pseudorandom and 1 deterministic pattern. A possible schedule for this process is provided on Figure 4.2. If the number of clocks needed to apply a test cycle for core *A* differs from the one of core *B*, then at the switching moment, from pseudorandom to deterministic test for core *B,* a short idle period occurs.

We would not like to emphasize the scheduling problem in this thesis. The example above is provided only to illustrate the testing differences caused by using test-per-scan BIST scheme instead of test-per-clock. There are a number of other particularities of testing SoC with sequential cores, caused by STUMPS architecture. For instance, the additional inputs for a CUT, such as *Scan In* (for each scan chain) and *Scan Enable,*

increase the length of LFSR and amount of memory required to store 1 deterministic pattern.

At the same time it is necessary to notice, that even we need to consider these differences (and it will be shown in the *Experiments* section) while calculating estimates and obtaining exact values, they do not influence the general idea of the approach described in chapter 3 and more concern to the practical issue.

4.1 Experiments

Setup

The experiments for this chapter were carried out with ISCAS' 89 benchmarks listed in Table 4.1, as sequential cores:

Table 4.1 Sequential benchmarks used in experimental SoCs.

Design name	Number of inputs (w/o clk)	Number of scan chains	Max. length of scan chains
s1423	23	5	15
s208	12	1	8
s298	5	1	14
s3271	32	5	24
s420	20	1	16
s526	6	2	11
s641	38	2	9
s838	37	2	16

All of them were redesigned with *Mentor Graphics DFTAdvisor* to have scan paths (full scan). Number of resulting scan chains and maximum lengths of them for each core are provided in the Table 4.1 too.

The experiments for this chapter were carried out on the same machines as in the previous one. This time, for fault simulation and deterministic pattern generation commercial tool *Mentor Graphics FlexTest* was chosen (Figure 4.1 provides a screen shot of this program). The rest of this section describes step by step operations performed, concentrating mostly on new ones while omitting explanations provided by the chapter 3.

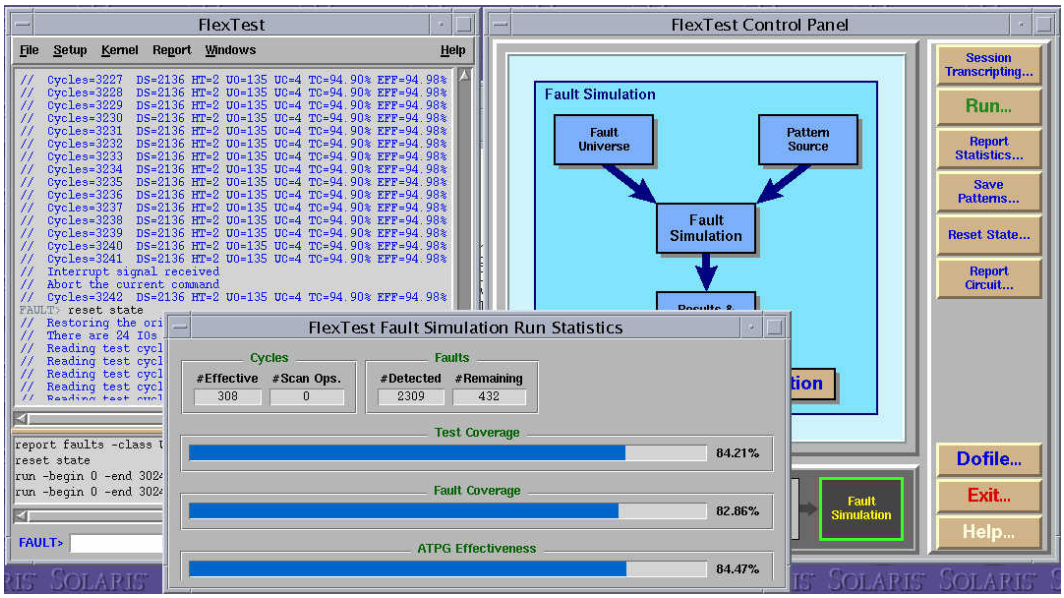


Figure 4.1 Screen shot of Mentor Graphics FlexTest program.

Test Pattern Generation

Test pattern generation for circuits containing STUMPS architecture should consider that during one test cycle at first the patterns will be uploading for *n* clock cycles and only then scanned within the next clock cycle (*n = Max(scan chains length)*). Therefore, test pattern sequence must be divided into groups. Generally, a sample sequence looks like it is shown in Figure 4.2:

```
1.   P0110110110 0 1      - one test pattern
2.   P0011011011 0 1
3.   P1001101101 1 1
4.   P0100110110 1 1
5.   P1010011011 0 1
6.   P0101001101 1 1
7.   P1010100110 1 1
8.   P1101010011 0 1
9.   P1110101001 1 0      - the test pattern applied for primary inputs, when the data is scanned
10.  P0111010100 1 1
11.  P0011101010 0 1
12.  P1001110101 0 1
13.  P1100111010 1 1
14.  P1110011101 0 1
15.  P0111001110 1 1
16.  P0011100111 0 1
17.  P1001110011 1 1
18.  P0100111001 1 0
```

Figure 4.2 Test pattern format for Mentor Graphics. A sample sequence.

This sequence was generated for *s208* core. The maximum scan chain length for the core is 8 bits (Table 4.1). Consequently, the length of 1 test cycle is 8 clock cycles to upload the data from *scan_in* input to the registers plus 1 clock cycle to scan the data. The last bit of a test pattern in the sequence above represents the value of *scan_enable* ("0" – enables scan operation). Next to the last bits (boxed) are the *scan_in* values.

FlexTest regards the *clk* input in a special way, therefore it demands for an input test pattern sequence the char "P" instead of its value.

Due to FlexTest features, we had to implement a separate program in C for pseudorandom pattern generation. The source code of it is provided in *Appendix B*. The program emulates LFSR work and outputs test sequences in the format described above based on the input parameters, such as maximum scan chain length for a core, initial vector, characteristic polynomial, desired number of test cycles and others.

Deterministic test patterns sets were generated by FlexTest. The results of test pattern generation are shown in Table 4.2.

Table 4.2 The results of test pattern generation for sequential cores.

Design name	Number of pseudorandom test cycles	Fault coverage (%)	Number of deterministic test cycles	Fault coverage (%)
s1423	1000	97.34	90	99.27
s208	1200	96.72	36	98.59
s298	500	99.24	41	100.00
s3271	500	98.01	161	99.70
s420	1411	89.00	67	99. 86
s526	1000	97.92	45	99.50
s641	1000	96.77	73	98.63
s838	1411	69.98	137	99.62

Later, we do not consider hypertrophic faults, what increases reported fault coverage, makes the further calculations easier, while does not affect the final experimental results.

Estimation

During test pattern simulation FlexTest allows saving session transcripts to external

files. A part of such a file is listed in Figure 4.3. Necessary for estimation procedure information may be extracted from the data manually with Excel help or by some specially implemented program. The fault coverage is calculated based on values of Total Faults, UO (unobserved faults) and UC (uncontrolled faults).

```
...
//  Cycles=13 Scan=10  DS=166 HT=10 UO=90 TC=70.31% EFF=70.68% 0:00:01
//  Cycles=14 Scan=11  DS=169 HT=10 UO=87 TC=71.25% EFF=71.60% 0:00:01
//  Cycles=15 Scan=12  DS=170 HT=11 UO=85 TC=71.72% EFF=72.07% 0:00:01
//  Cycles=16 Scan=13  DS=171 HT=11 UO=84 TC=72.03% EFF=72.38% 0:00:01
//  Cycles=17 Scan=14  DS=171 HT=12 UO=83 TC=72.19% EFF=72.53% 0:00:01
//  Cycles=18 Scan=15  DS=179 HT=12 UO=75 TC=74.69% EFF=75.00% 0:00:01
//  Cycles=19 Scan=16  DS=185 HT=12 UO=69 TC=76.56% EFF=76.85% 0:00:01
//  Cycles=20 Scan=17  DS=189 HT=12 UO=65 TC=77.81% EFF=78.09% 0:00:01
...
```

Figure 4.3 An extract from Mentor Graphics FlexTest fault simulation transcript.

The estimation procedure itself was step by step described in chapter 3 and uses the same programs.

However, there are 2 differences. The first one concerns the amount of memory we need to store one deterministic test cycle.

M_{DET}= *{max_scan_length*(nr_of_chains + 1)} + {nr_of_primary_inputs + nr_of_chains + 1}*

(The "1" in this expression represents *scan_enable* bit.)

If we take one test cycle for core *s208* from Figure 4.3, then the information, that should be stored, is the boxed one in the sequence below:

```
1.  P0110110110 0 1
2.  P0011011011 0 1
3.  P1001101101 1 1
4.  P0100110110 1 1
5.  P1010011011 0 1
6.  P0101001101 1 1
7.  P1010100110 1 1
8.  P1101010011 0 1
9.  P1110101001 1 0
```

The second difference is also caused by test-per-scan BIST usage. Earlier we did not need to discriminate between *test cycle*, *time for one test pattern application* and *clock cycle*, because they were equal in terms of time. In case of sequential cores, when the estimation procedure is accomplished, we will have all the information for every core expressed in test cycles. At the same time, in order to use the data for each core in a

representation of a whole SoC, we need this information expressed in clock cycles. Other words, we need one more extrapolation. For this reason one more program in C was written.

Table 4.3 Memory and time needed for one test cycle of the experimental cores.

Design name	Memory for 1 det. test cycle (bits)	Time for 1 test cycle (clock cycles)
s1423	113	16
s208	28	9
s298	33	15
s3271	176	25
s420	52	17
s526	39	12
s641	65	10
s838	85	17

Exhaustive Simulation and CPU Time Measurement

As earlier, in order to compare the obtained estimation results we had to execute an exhaustive simulation for every number of pseudorandom test cycle. A script was used for this purpose, which was iteratively calling FlexTest program (without GUI) for pseudorandom pattern simulation and deterministic pattern generation.

For these experiments we have determined three virtual Systems-on-Chips containing only the cores from Table 4.1. The systems are represented in Table 4.4:

Table 4.4 List of sequential cores for the experimental SoCs.

System name	J 6 cores	K 6 cores	L 6 cores
List of used cores	s838	s3271	s838
	s3271	s1423	s1423
	s298	s208	s526
	s641	s641	s420
	s526	s298	s208
	s526	s526	s298

Based on the methodology described in chapter 3, illustrative charts for each experimental system were built with Excel to represent both estimated and exact results. The charts are provided in Figures 4.4 -4.6.

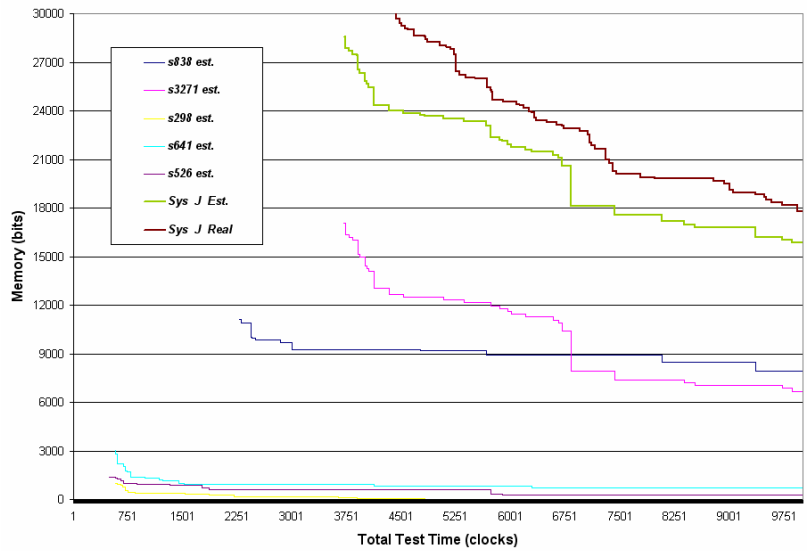


Figure 4.4 Memory cost estimations and real values for system J.

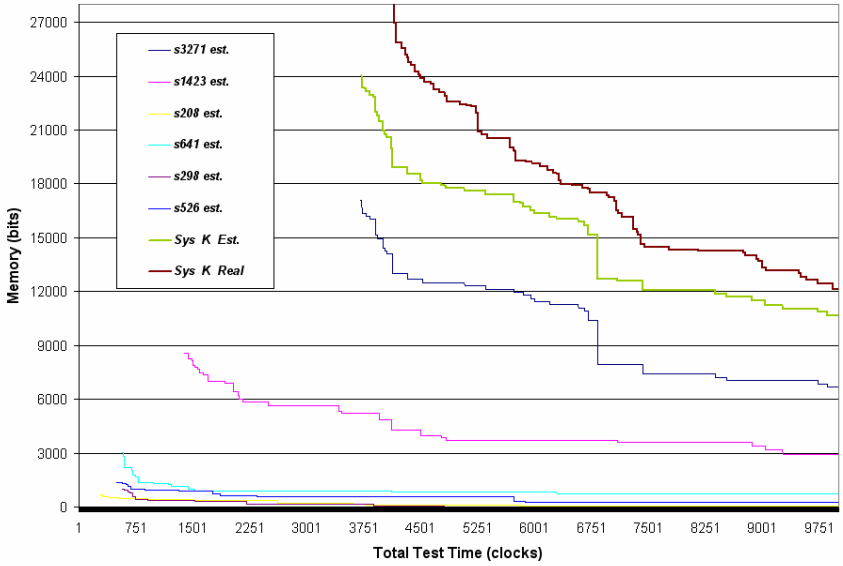


Figure 4.5 Memory cost estimations and real values for system K.

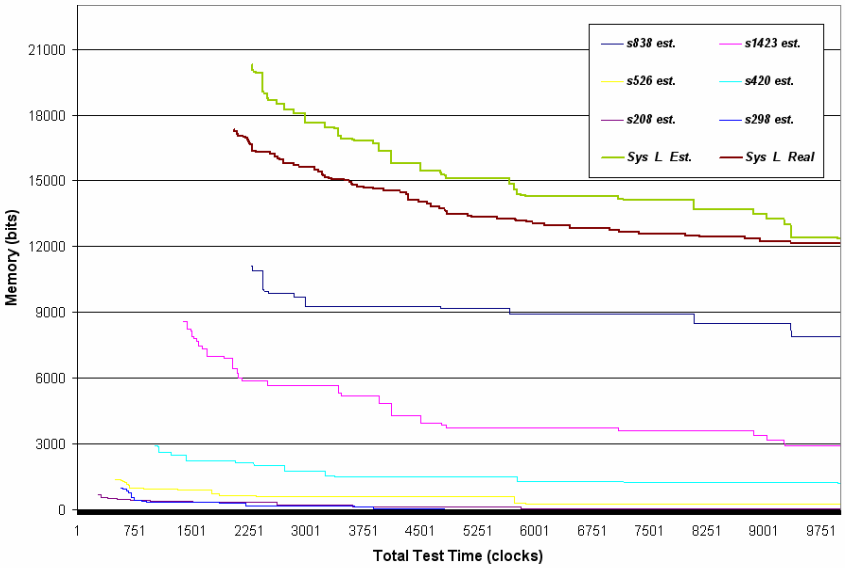


Figure 4.6 Memory cost estimations and real values for system L.

(Figure 4.4 shows a good example of the situation when memory requirements for Hybrid BIST of a SoC can be dictated by one or two dominated cores.)

Finally, three random memory constraints were chosen for each experimental SoC. Adjustment algorithm verification was carried out in the same way, like it is shown in section 3.1 (Table 3.7). CPU time measurement for performed calculations does not have any differences too.

Table 4.5 compares our proposed approach where the test lengths for experimental systems are found based on the estimation methodology and further adjustment, with an exact approach where they were obtained by iterative pseudorandom patterns simulation and appropriate number of deterministic patterns generation for every reasonable number of test cycles. As it can be seen from the table, our approach gives significant speedup and high accuracy for SoCs with sequential core as well.

Table 4.5. Experimental results with sequential cores. Final table.

System Name	Number of Cores	Memory Constraint (bits)	Exhaustive Approach		Our Proposed Approach	
			Total Test Length (clocks)	CPU Time (seconds)	Total Test Length (clocks)	CPU Time (seconds)
J	6	25 000	5750	57540	5775	270
		22 000	7100		7150	216
		19 000	9050		9050	335
K	6	22 000	5225	53640	5275	168
		17 000	7075		7075	150
		13 000	9475		9475	427
L	6	15 000	3564	58740	3570	164
		13 500	4848		4863	294
		12 200	9350		9350	464

Conclusion

In this chapter we have presented an approach to the test time minimization problem for Systems-on-Chip, containing sequential cores with STUMPS architecture. To avoid the

exhaustive exploration of solutions, the cost estimation method for the deterministic component of the hybrid test set is used. An iterative algorithm, based on cost estimates is thereafter applied in order to minimize the total test length of the hybrid BIST solution under the given memory constraints. As in the previous chapter, experimental results show the very high speed and accuracy of the proposed method compared to the exact calculation approach.

Chapter 5
Demonstrational program

This chapter presents a demonstrational ActiveX control, which was specially created to give some visual representation of the proposed approach for test time minimization for Hybrid BIST of Systems-on-Chip. The previous chapters of this thesis contain many mathematical expressions and hardware testing terminology, what makes them sometimes not that easy to get the general idea of the internal processes for a reader who is not very familiar with the topic. The main purpose of this demonstrational program is to improve the situation. However, the control does not explain the procedure of test time minimization itself, and works with precalculated values.

Motivation

ActiveX technology is convenient for visualization of a dynamic process, such as, for example, a SoC testing. ActiveX controls are easy to insert into a web page, any Microsoft Office document, or another program, that could be written in a programming language different from the one used to create the control. ActiveX provides good functionality, and in spite of the disadvantage that it works only with Microsoft Windows Environment, ActiveX is a good solution for such kind of task, as visualization.

To create the control *Visu_mchbist.ocx*, ***Visual Basic 6*** was used. As an example of ActiveX integration it was inserted in a web-page (http://www.tud.ttu.ee/~t990834/Project_MC-HBIST/Visu_mchbist.HTM) and Microsoft Power Point presentation.

To show advantages of ActiveX and cooperation, this control uses as one of the components a small third party ActiveX control “Advanced Progress Bar”. (All specifications and source files are referenced).

Visu_mchbist.ocx ActiveX control shows the dependency of the test schedule and the whole testing process on the chosen Memory Constraint.

The rest of the chapter is structured as follows. *User Interface* section describes components layout and provides some basic instructions about the control usage. The following two sections describe implemented functions of the program. Next part provides general overview of the paper and conclusions.

Appendix A and *Appendix C* of this thesis provide the source code listing of the program and screenshot representatively.

User Interface

The control uses 2 forms. The main one called *Visu_ctrl* and Visual Basic provided *frmAbout.* The second one is used only to output some general "about" information for the program.

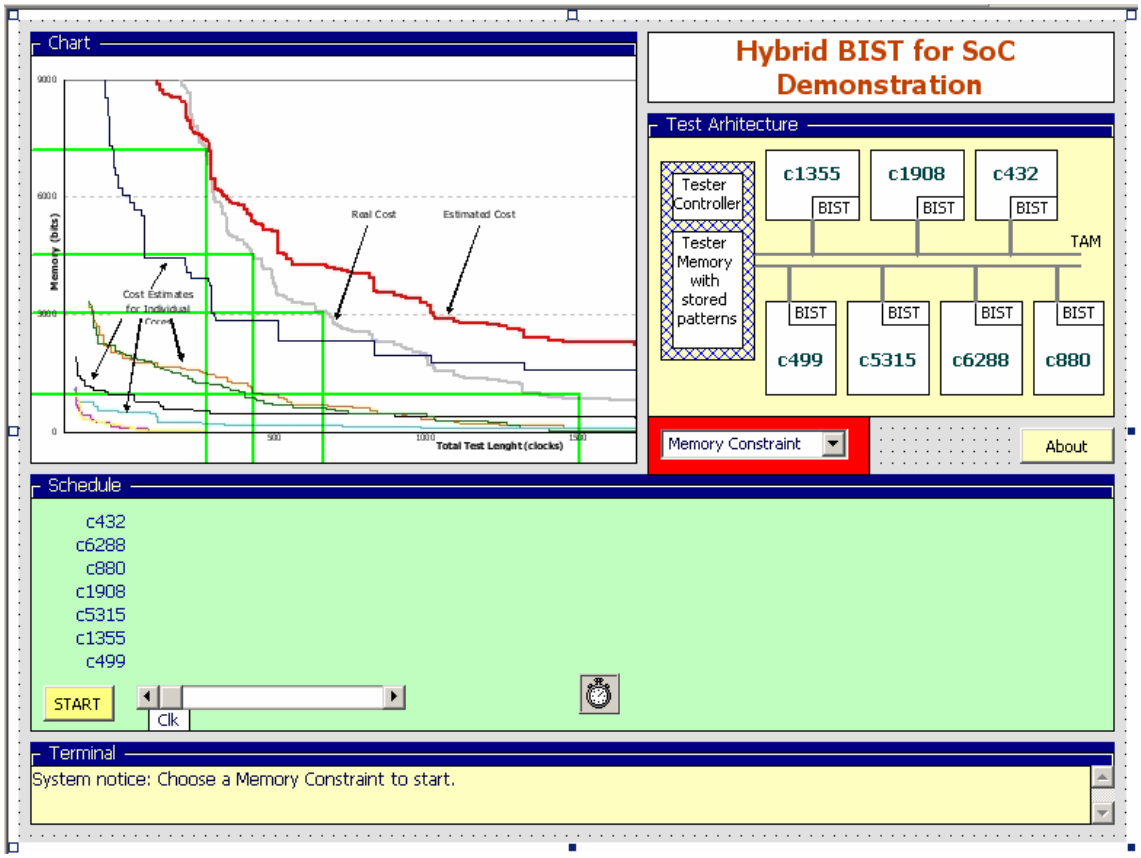


Figure 5.1. Main Form *Visu_ctrl*.

The Main Form consists of 4 frames, a Title, ComboBox *Cmb_mem* and an *About* CommandButton.

The tasks of the frames are as follows:

- Frame *Chart* represents graphical diagram (a part of Figure 3.7), which shows how much time (horizontal axis) we will need to test our System-on-Chip if we have certain amount of Memory (vertical axis) available. Green lines show our position on the curve. During the operation time only 1 set of green lines is visible.

- Frame *Test Architecture* shows general structure of a SoC from testing point of view. Here "BIST" means the part of a core responsible for generating and applying pseudorandom test patterns. "TAM" (Test Access Mechanism) is a set of internal components including a bus, used to transport deterministic patterns from the Tester Memory to a core. In the operation mode all the active components are highlighted.

- Frame *Schedule* is used to represent a test schedule based on chosen Memory Constraint and to control an imitation of test execution. Here we can find an *array* of labels with the cores' names; 3 arrays of "Advanced Progress Bar" controls, where the first *PB_1p1* and second *PB_1p2* are introduced to represent first and second parts of pseudorandom test sets correspondingly, third - *PB_1d* represents deterministic parts. Scrollbar *HScrll_manual_run* allows the user to follow the process of testing, when program imitates test execution, and manually scroll till the needed time moment of the process. One multi-purpose command button is located in the left corner of the frame. It is used to *start*, *stop* and *reset* test execution imitation, according to the situation. A timer *Timer1* was used to imitate the testing process. Its task is to increase the *HScrll_manual_run.Value* every predetermined interval of time, when it is allowed.

- Frame *Terminal* provides a dialog with the user and consists mainly of the TextBox *Text1*. In the current version of this control it is possible only to read "system notices" and tips concerning next user actions. However, in future the user will be able to type his/her commands to the program here.

In the middle of the form *Visu_ctrl* a ComboBox *Cmb_mem* is located. It is used to initiate the control work and to choose a desired Memory Constraint from the list (4 options). All the other components (except CommandButton *About*) are disabled until the user chooses a Memory Constraint. To make the control more friendly for first time users, the area around the *Cmb_mem* is highlighted with red color before any Memory Constraint is chosen.

Each of the frames listed above have *PictureBox* as a base for other components. That allows managing colors in easier way and makes it more convenient to work with other components on the frame.

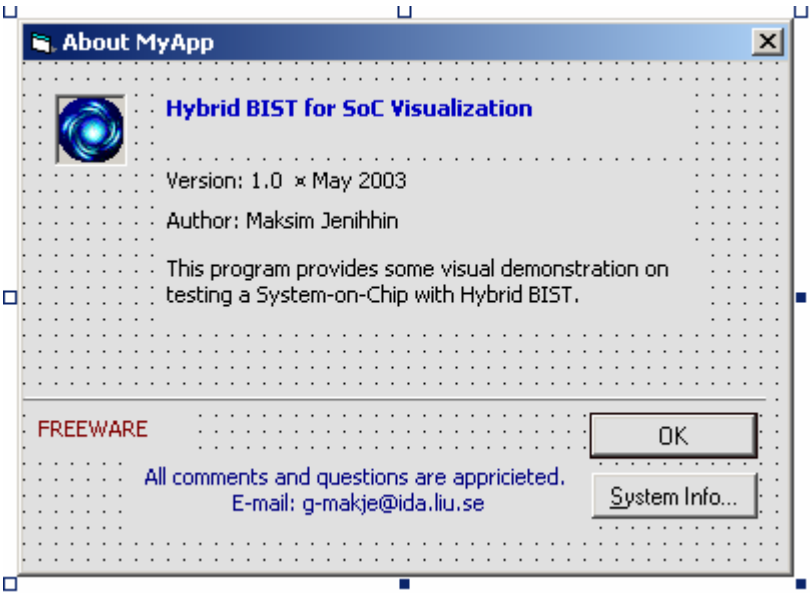


Figure 5.2. About form *frmAbout.*

Second form used by the control is *frmAbout.* It is a standard *About* form provided by Visual Basic 6, and it can be called by clicking CommandButton *About*. The form includes also CommandButton *System Information,* which allows user to see the resources of the computer he/she is using.

To conclude User Interface part, it is necessary to mention that all the components were placed on one form to give the whole impression of the process of testing, and to allow the user to observe it from different points of interest at the same time. Although it has caused a lack of space on the form to show some components more detailed.

The size of the form was chosen to consider also 800x600 display resolution.

Initialization

The control *Visu_mchbist.ocx* uses two stages of setting up initial states for the components used during the main work of the program, when it imitates test execution.

The first stage is obvious. Some initial values for parameters of the components are predetermined during the Main Form *Visu_ctrl* design (such as text *Labels'* captions and position of most of the components).

The second stage is *Private Function* ***Initia.*** This function is called only when a Memory Constraint is chosen from *Cmb_mem* ComboBox. Choosing a Memory Constraint actually means for us choosing:

- Total length of the test, or a value for the variable *tln*
- Values for the *DET* array elements, which represent the lengths of deterministic part of the test for every core
- Scaling value *mm* for the schedule representation. It is used to fit longer schedule with a low Memory Constraint into the Schedule frame, and on the other hand to extend short schedules with a higher Memory Constraint in order to give user a better view
- Which of the green lines *Line1 ... Line8* should be visible to point the right position in the *Chart*

Based on the values listed above, function *Initia* calculates values for some parameters of *PB_1p1, PB_1p2, PB_1d* arrays' elements and for *HScrll_manual_run* parameters (such as *.Value*, *.Max*, *.Width*). Most of the computations for the schedule are determined by the rules mentioned in chapter 2 of this thesis.

Many similar components in this control are combined in arrays to make some necessary assignments easier.

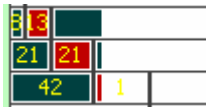
The function outputs summary information in the terminal window for the user convenience.

Reaction on Events

After initial states of the components are defined, the user has a choice to run the imitation of testing process or to observe the SoC state at any moment of time during the test process by scrolling *HScrll_manual_run* manually.

A special function *Private Function SM()* was introduced in the program. It is called every time *HScrll_manual_run.Value* is changed. Based on the current position of the *HScrll_manual_run,* the function calculates which of the *ProgressBars* from the arrays *PB_1p1*, *PB_1p2* and *PB_1d* must be changed. It is implemented so, that the user has such an impression of the elements from the arrays as if they were one whole.

Generally, we can start changing the value of a ProgressBar on the right from currently being changed only when it has reached its maximum value:



By the precalculated data, the testing of the first two cores, used in this particular SoC, does not need any deterministic parts (when the user can choose only from these 4 provided Memory Constraints). Therefore, the elements with indexes 0 and 1 from *PB_1p1*, *PB_1p2*, *PB_1d* arrays are treated separately and do not participate in the *for*-cycles of function *SM*. We have a similar situation with the core *c880* related elements (index = 2), although now the reason is that here we start testing not with pseudorandom part, but with deterministic one.

The timer *Timer1* is allowed to (and does) increase the value of *HScrll_manual_run.Value* only when the *Boolean* variable *chk* is true. The value of the variable changes to the opposite every time a *Click* on CommandButton *Cmd_start* is registered. This feature gives us an additional advantage: the same CommandButton initiates several actions, depending on the current state of the process. If *Timer1* recognizes that *HScrll_manual_run.Value* has become equal to its maximum possible value (*.Max*) it assigns *reset* action to *Cmd_start,* and stops.

The control does not have any special properties except the standard ones.

Conclusion

This chapter has presented an ActiveX control *Visu_mchbist.ocx,* which provides a simplified easy-to-use demonstration of testing a System-on-Chip with Hybrid Built-In Self Test. The control may be used for illustrative purposes, as a helpful addition to the documentation related to the research. It can be easily integrated into any Microsoft Office Document, web page or used as a component of another program, developed for Microsoft Windows Platform.

[A third-party control *AdvProgressBar v.1.0* was used in the current program to show cooperation between ActiveX controls. The original package can be found on *http://www.activex.net.ru*]

Chapter 6
Conclusions and Future Work

The main goal of this thesis was to develop an experimental environment for the test time minimization problem. It assumes Hybrid BIST architecture and targets System-on-Chip designs. The thesis is based on methodology developed during the work and demonstrates the feasibility of the proposed methodology together with experimental results. First, the proposed methodology was discussed for the case when a SoC consisted only of combinational cores. An appropriate Hybrid BIST architecture was proposed as well. However, real life System-on-Chip designs contain mostly sequential cores, and this was taken into account in the following part of this thesis, where Hybrid BIST for SoCs with sequential cores was examined. At the end of this thesis a small demonstrational program was presented, which may be interpreted as a useful add-on for the rest of the material reporting results of the research.

In this section we summarize the thesis and outline possible directions for the future work.

Conclusion

Nowadays, many modern convenient design techniques are available, and as a consequence, manufactured integrated circuits become more and more complex. This tendency demands, in its turn, development of existing testing techniques for the circuits. Therefore, new test methods and approaches are highly appreciated. The approach, proposed in this thesis, deals with one particular, but quite actual problem that was chosen for our research activities.

Hybrid BIST is recognized as one of the most sufficient solutions in testing core-based systems. However, even if it is implemented on one, most likely it does not consider

time and memory costs, and, hence, becomes too expensive. The test time minimization for Hybrid BIST is a complex task even for separate cores. When the cores are combined into one system, the task of test time minimization at system level becomes much more difficult, because we need to consider all cores of the system simultaneously.

A naïve approach for this problem would be an exact computation of every possible switching moment between deterministic and pseudorandom test for every core of a system, what requires iterative pseudorandom patter simulation and deterministic pattern generation. Moreover, an exhaustive search should be used then. Obviously, these operations are very time consuming.

Our proposed approach uses fast cost estimation algorithm, based on fault coverage reports, which may be obtained by only one for every core pseudorandom and deterministic pattern simulation. Further, second algorithm iteratively adjusts the estimated values to near optimal results.

We have carried out experiments for both combinational and sequential core-based systems to compare these two approaches. Their results show significant speedup for the proposed one, while retaining very high accuracy.

Future work

The following are some possible directions for future research:

- ✓ The approach described in this thesis does not consider *power consumption*. At the same time the last remains very important factor in design. Therefore, it would be highly beneficial to include power constraints into test time minimization algorithm.
- ✓ In addition to full scan STUMPS architecture, it would be quite innovative to investigate the possibilities to apply the same approach also to the sequential cores with *partial scan*.
- ✓ Also, it would be interesting to examine more *complex test architectures*.

References and Bibliography

[1] G. Jervan, P. Eles, Z. Peng, R. Ubar, M. Jenihhin, "Test Time Minimization for Hybrid BIST of Core-Based Systems", *submitted to the Asian Test Symposium 2003*, Xian, China, 2003

[2] G. Jervan, Z. Peng, R. Ubar, M. Jenihhin, "Hybrid BIST Test Time Minimization for Core-Based Systems with STUMPS Architecture", *submitted to the 18th IEEE Int. Symposium on Defect and Fault Tolerance in VLSI Systems,* Cambridge, USA, 2003

[3] G. Jervan, Z. Peng, R. Ubar, "Test Cost Minimization for Hybrid BIST," *IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems (DFT'00)*, pp.283-291, Yamanashi, Japan, October 2000.

[4] G. Jervan, "High-Level Test Generation and Built-In Self-Test Techniques for Digital Systems" Linkoping, October 2002

[5] S. Mourad, Y. Zorian, "Principles of Testing Electronic Systems", Johan Willey & Sons, 2000

[6] M. Abramovichi, M. Breuer, A. Friedman, "Digital Systems Testing and Testable Design", IEEE Press, 1990.

[7] C. Reeves, "Modern Heuristic Techniques for Combinatorial Problems", Oxford, 1993

[8] Adam Blum, "ActiveX Web Programming. ISAPI, Controls, and Scripting", New York 1997

[9] D. Benage, A.Mirza, "Building Enterprise Solutions with Visual Studio 6", SAMS 1999

[10] C. Franklin, "3. Visual Basic 6.0 Internet Programming", New York 1999

[11] MSDN Library http://www.microsoft.com/

[12] Turbo Tester Reference Manual, Tallinn Technical University. http://www.pld.ttu.ee/tt

[13] C. Pappas, W. Murray, "The Complete Reference C++", McGraw-Hill, 1998

Appendix A

Visu_mchbist.ocx control source code (used in chapter 5)

```
Option Explicit
Dim i, tln, mm, j, tmr As Integer, D, DET, INP, chk As Boolean, c As String
Private Sub Cmb_mem_Click()                    'Choice of the Memory Constraint
INP = Array(36, 32, 60, 33, 178, 41, 41)       'number of inputs for every core
                                               'to calculate memory used for every core
Select Case Cmb_mem.ListIndex                  'Assigns some parameters,
   Case 0                                      'after a Memory Constraint is chosen
      HScrll_manual_run.Enabled = True
      DET = Array(0, 0, 8, 13, 21, 28, 36)
      tln = 468
      mm = 15
      Initia
      Line1.Visible = True
      Line2.Visible = True
   Case 1
      DET = Array(0, 0, 8, 11, 12, 25, 33)
      tln = 542
      mm = 15
      Initia
      HScrll_manual_run.Enabled = True
      Line5.Visible = True
      Line6.Visible = True
   Case 2
      DET = Array(0, 0, 7, 7, 6, 14, 17)
      tln = 879
      mm = 10
      Initia
      HScrll_manual_run.Enabled = True
      Line7.Visible = True
      Line8.Visible = True
   Case 3
      DET = Array(0, 0, 5, 4, 2, 2, 3)
      tln = 1527
      mm = 6
      Initia
      HScrll_manual_run.Enabled = True
      Line9.Visible = True
      Line10.Visible = True
End Select
End Sub

Private Sub Cmd_About_Click()
   frmAbout.Show vbModal
End Sub

Private Sub Cmd_start_Click()                  'Multi-purpose button; chk - allows the Timer1
   If chk = False Then
       If Cmd_start.Caption = "RESET" Then
       HScrll_manual_run.Value = 0
       Text1.Text = "Ready"
       Else
       Cmd_start.Caption = "STOP"
       chk = True
       End If
       Else
       Text1.Text = "To continue the simulation start the system clock again"
       Text1.Text = Text1.Text + " or scroll the test manualy."
       Cmd_start.Caption = "START"
       chk = False
   End If
End Sub
```

```
Private Sub HScrll_manual_run_Change()
   SM
   Rst_c
End Sub

Private Sub HScrll_manual_run_Scroll()
     SM
     Rst_c
End Sub

Private Function SM()       'Changes values of the ProgressBars, according to current position
Lbl_Clk.Caption = HScrll_manual_run.Value
Lbl_Clk.Left = HScrll_manual_run.Left + 100 + HScrll_manual_run.Value * mm * (1 - 600 / (tln * mm))
                                'the formula keeps the label exactly under the scrollbar cursor
PB_1p2(0).Value = HScrll_manual_run.Value
PB_1p2(1).Value = HScrll_manual_run.Value
PB_1p2(0).Caption = PB_1p2(0).Value
PB_1p2(1).Caption = PB_1p2(1).Value
If HScrll_manual_run.Value <= PB_1d(2).Max Then
PB_1d(2).Value = HScrll_manual_run.Value
i = 2
D_1
PB_1p2(2).Value = 0
Else
   PB_1d(2).Value = PB_1d(2).Max
   PB_1p2(2).Value = (HScrll_manual_run.Value - PB_1d(2).Max)
End If
PB_1d(2).Caption = PB_1d(2).Value
PB_1p2(2).Caption = PB_1p2(2).Value
For i = 3 To 6
   If HScrll_manual_run.Value <= PB_1p1(i).Max Then       '1st pseudorandom
   PB_1p1(i).Value = HScrll_manual_run.Value
   PB_1d(i).Value = 0
   PB_1p2(i).Value = 0
   Else
      PB_1p1(i).Value = PB_1p1(i).Max                     'deterministic part
      PB_1p2(i).Value = 0
      If (HScrll_manual_run.Value - PB_1p1(i).Max) <= PB_1d(i).Max Then
      PB_1d(i).Value = (HScrll_manual_run.Value - PB_1p1(i).Max)
      D_1
      Else
         PB_1d(i).Value = PB_1d(i).Max                    '2nd pseudorandom
         PB_1p1(i).Value = PB_1p1(i).Max
         PB_1p2(i).Value = (HScrll_manual_run.Value - PB_1p1(i).Max - PB_1d(i).Max)
         If i = 6 Then
           D_rst
         End If

      End If
   End If
   PB_1p1(i).Caption = PB_1p1(i).Value
   PB_1d(i).Caption = PB_1d(i).Value
   PB_1p2(i).Caption = PB_1p2(i).Value
Next
End Function

Private Function Initia()     'Sets up some initial values, considering Memory Constraint choice
   chk = False
   PB_1p2(0).Max = tln
   PB_1p2(1).Max = tln
   PB_1p2(0).Width = mm * (PB_1p2(0).Max)
   PB_1p2(1).Width = mm * (PB_1p2(1).Max)
   PB_1p2(0).Visible = True
   PB_1p2(1).Visible = True
   Pic_mem.BackColor = &HFFC0C0

   PB_1d(2).Max = DET(2)
   PB_1d(2).Width = mm * (PB_1d(2).Max)
   PB_1d(2).Visible = True
   PB_1p2(2).Max = tln - PB_1d(2).Max
```

```
   PB_1p2(2).Left = PB_1d(2).Left + (PB_1d(2).Width)
   PB_1p2(2).Width = mm * (PB_1p2(2).Max)
   PB_1p2(2).Visible = True
   For i = 3 To 6
   PB_1p1(i).Max = tln - PB_1p2(i - 1).Max
   PB_1p1(i).Width = mm * (PB_1p1(i).Max)
   PB_1p1(i).Visible = True
   PB_1d(i).Max = DET(i)
   PB_1d(i).Left = PB_1p1(i).Left + PB_1p1(i).Width
   PB_1d(i).Width = mm * (PB_1d(i).Max)
   PB_1d(i).Visible = True
   PB_1p2(i).Max = tln - PB_1d(i).Max - PB_1p1(i).Max
   PB_1p2(i).Left = PB_1d(i).Left + (PB_1d(i).Width)
   PB_1p2(i).Width = mm * (PB_1p2(i).Max)
   PB_1p2(i).Visible = True
   Next i
   HScrll_manual_run.Max = tln
   HScrll_manual_run.Min = 0
   HScrll_manual_run.Value = 0
   HScrll_manual_run.Width = (HScrll_manual_run.Max) * mm
   Cmd_start.Enabled = True
   c = tln                              'here all the schedule information is outputted to the terminal
   Text1.Text = "Total clock cycles for the test: " + c & vbCrLf & "Deterministic patterns for the cores: "
   For i = 0 To 6
      c = DET(i)
      Text1.Text = Text1.Text + Label8(i).Caption + ": " + c + ";  "
   Next i
   Text1.Text = Text1.Text + "(... scroll down)" & vbCrLf & "Memory used for every core: "
   For i = 0 To 6
      c = INP(i) * DET(i)
      Text1.Text = Text1.Text + "" & vbCrLf & "" + Label8(i).Caption + ": " + c + " bits;  "
   Next i
   Line1.Visible = False                'green lines
   Line2.Visible = False
   Line5.Visible = False
   Line6.Visible = False
   Line7.Visible = False
   Line8.Visible = False
   Line9.Visible = False
   Line10.Visible = False
   All_rst
End Function

Private Sub Timer1_Timer()              'Moves the scrollbar
   If chk = True Then
      If HScrll_manual_run.Value < HScrll_manual_run.Max Then
         HScrll_manual_run.Value = HScrll_manual_run.Value + 1
         Text1.Text = "Simulating..."
         Else
         chk = False
         Cmd_start.Caption = "RESET"
         Text1.Text = "System notice: Simulation is completed"
         All_rst
      End If
   End If
End Sub

Private Function D_1()                  'Determines which core in TA to highlight
 For j = 0 To 6
 Label10(j).BackColor = &HFF&
 Shape4(j).BorderColor = &H0
 Shape4(j).BorderWidth = 1
 Line13(j).BorderColor = &HE0E0E0
 Next
 Label10(i).BackColor = &H80000005      'i - value is taken from SM
 Shape4(i).BorderColor = &HFF&          'i - index of core using memory at the moment
 Shape4(i).BorderWidth = 3
 Line13(i).BorderColor = &HFF&
 Line3.BorderColor = &HFF&
 Line4.BorderColor = &HFF&
 Label22.ForeColor = &HFF&
```

```
 Label23.ForeColor = &HFF&
End Function

Private Function D_rst()                    'resets TA, when none of the cores uses the memory
 For j = 0 To 6
 Label10(j).BackColor = &HFF&
 Shape4(j).BorderColor = &H0
 Shape4(j).BorderWidth = 1
 Line13(j).BorderColor = &HE0E0E0
 Next
 Line3.BorderColor = &HE0E0E0
 Line4.BorderColor = &HE0E0E0
 Label23.ForeColor = &H0
End Function

Private Function All_rst()                  'none of SoC components are active
 For j = 0 To 6
 Label10(j).BackColor = &H80000005
 Shape4(j).BorderColor = &H0
 Shape4(j).BorderWidth = 1
 Line13(j).BorderColor = &HE0E0E0
 Next
 Line3.BorderColor = &HE0E0E0
 Line4.BorderColor = &HE0E0E0
 Label22.ForeColor = &H0
 Label23.ForeColor = &H0
End Function


Private Function Rst_c()                    'a patch on found bug
    If Cmd_start.Caption = "RESET" And chk = False Then
        Cmd_start.Caption = "START"
        Text1.Text = "System notice: Current position was changed manualy."
    End If
End Function
```

Appendix B

LFSR emulator source code (used in section 4.1)

```
#include <stdio.h>
#include <stdlib.h>

#define name_length 40
#define max_size 100

int SR(int *reg, int b, int length) //shift right operation
{   int i;                          //b to the highest bit
    for (i=0; i<length-1 ; i++)
    {   reg[i]=reg[i+1];}
    reg[length-1]=b;
    return 0;
}

int xor(int a, int b)               //XOR operation
{
    if (a==b){return 0;} else {return 1;}
}

int main(void)
{
    FILE *fp_in, *fp_out;
    int i, j, k, c, size, count, scan, vec_size;
          int reg1[max_size], reg2[max_size];
          char  tmp[name_length], tmp2[name_length];
    char name_in[name_length], name_out[name_length];


    printf("\nOutput file name: \n");   //file name request for generated patterns
    scanf("%s", &name_out);
    printf("\nInput file name: \n");    //file name request for configuration
    scanf("%s", &name_in);
    printf("\nInput number of cycles for pattern generation: \n");
    scanf("%d", &count);

    if ((fp_in=fopen(name_in, "r")) == NULL)
    {   printf("Cannot open the file.\n");
        exit(1);
    }

    fscanf(fp_in, "%d", &vec_size);     //length of a generated pattern
    fscanf(fp_in, "%d", &size);         //length of LFSR (>= vec_size)
    fscanf(fp_in, "%d", &scan);         //max length of scan chains

    for (i=size-1; i>=0; i--)           //read characteristic polynomial
    {   fscanf(fp_in, "%d", &reg1[i]);}
    for (i=size-1; i>=0; i--)           //read initial vector
    {   fscanf(fp_in, "%d", &reg2[i]);}
```

```
    if ((fp_out=fopen(name_out, "w")) == NULL)
    {   printf("Cannot open the file.\n");
        exit(1);
    }
    fprintf(fp_out,"//GENERATOR\n//PLYNOMIAL     ");
    for (i=size-1; i>=0; i--)
    {   fprintf(fp_out,"%d",reg1[i]);}

    fprintf(fp_out,"\n//INITIAL_STATE ");
    for (i=size-1; i>=0; i--)
    {   fprintf(fp_out,"%d",reg2[i]);}
    fprintf(fp_out,"\n");

    fscanf(fp_in, "%s %s", &tmp, &tmp2);    //copy inputs info from configuration  file to output
    while(!feof(fp_in))
    {   fprintf(fp_out,"%s %s\n", tmp, tmp2);
        fscanf(fp_in, "%s %s", &tmp, &tmp2);
    }
    fprintf(fp_out,"\n");

    c=0;
    for (j=count*(scan+1); j>0; j--)            //pseudorandom pattern generation
    {   for (i=0; i<size; i++)
        {   if (reg1[i]==1)
            {   reg2[0]=xor(reg2[i], reg2[0]);
            }
        }
        SR(reg2, reg2[0], size);
        fprintf(fp_out, "P");
        for (i=size-1; i>(size-vec_size-1); i--)
        {   fprintf(fp_out,"%d",reg2[i]);}
        c++;
        if (c==(scan+1)){fprintf(fp_out, "0\n"); c=0;}
        else {fprintf(fp_out, "1\n");}
    }

    fclose(fp_in);
    fclose(fp_out);

    return 0;
}
```

Sample configuration file (used for core *s298*):

```
4                                   <number of inputs w/o clk and scan_en>
20                                  <LFSR length>
14                                  <max. scan chain length>
0 1 0 0 0 1 0 1 0 0 1 0 1 0 1 0 1 1 0 0   <characteristic polynomial>
0 0 1 1 1 0 1 1 0 1 1 0 0 0 1 0 0 1 1 0   <initial vector>
PI h                                <
PI inp(0)                           <
PI inp(1)                           <inputs order (FlexTest requirement)>
PI inp(2)                           <
PI scan_in1                         <
PI scan_en                          <
```

Appendix C

ActiveX control Visu_mchbist.ocx screenshot.