A Methodology for Partitioning and Hierarchical Reorganization of Sequential Circuits for DFT and BIST

BY

Rajiv Gupta, Rajagopalan Srinivasan, and Melvin A. Breuer

Technical Report CENG 90-19

Electrical Engineering - Systems Department

University of Southern California

Los Angeles, CA. 90089-0781

A Methodology for Partitioning and Hierarchical Reorganization of Sequential Circuits for DFT and BIST *

Rajiv Gupta, Rajagopalan Srinivasan and Melvin A. Breuer

July 1990

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089-0781.

*This work was supported by the Defense Advanced Research Projects Agency and monitored by the Office of Naval Research under Contract No. N00014-87-K-0861. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U. S. Government.

A Methodology for Partitioning and Hierarchical Reorganization of Sequential Circuits for DFT and BIST

Abstract

Several design-for-testability (DFT) and Built-In Self-Test (BIST) techniques exist for VLSI circuits to make them easily testable. These techniques typically assume register/gate level decomposition of the overall circuit. In general, the given user hierarchy is not appropriate for embedding various testable design methodologies (TDMs). This paper describes a new canonical partitioning of a circuit into disjoint subcircuits, referred to as *clouds* and registers. Procedures for transforming any user hierarchy into this canonical representation are given. A salient feature of this transformation is the attempt to preserve the user hierarchy as much as possible. This enables easy identification of equivalence among various clouds of the circuit. This information can be used in the TDM embedding process. We also show how this canonical representation can be used for three specific TDMs, viz. full scan, partial scan and BILBO designs. For the case of full scan, deterministic tests are generated for one cloud in each equivalence class, and replicated for all clouds in that class. These tests are organized to form a test set for the entire circuit. Finally, the test vectors are edited to correspond to the order of flip-flops in the scan paths of the circuit.

Keywords: Test generation, clouds, register identification, structural equivalence, DFT, BIST

1 Introduction

Test generation for sequential circuits is known to be a computationally intensive task [1]. To reduce test generation complexity, design-for-testability (DFT) techniques such as full scan [2], partial scan designs [3,4] and Built-In Self-Test (BIST) [5] have been proposed. These techniques attempt to reduce the test generation effort by providing better controllability and/or observability over the internal state of the sequential circuit under test. Typically, scan based approaches consider gate and flip-flop (FF) model of the circuit while BIST techniques, such as BILBO [6], manipulate user given partitioning of the circuit and assume a clustering based on blocks of logic separated by registers.

In general, the user given hierarchy may not be ideally suited for applying Testable Design Methodologies (TDM). In addition, many TDMs result in very large combinational circuits (in the case of full scan and BIST) or sequential circuits (in the case of partial scan). Scan techniques tend to be especially time consuming as test vectors have to be scanned through the scan paths. Therefore, methodologies for (1) partitioning the circuits to make them more manageable, (2) reducing the test generation effort, and (3) reducing the number of test vectors to lower the test application time, are very desirable.

In this paper we present a methodology called CRETE [1] that attempts to achieve the above mentioned goals. CRETE reorganizes the user-provided hierarchical description of the circuit by clustering gates and FFs separately to derive a canonical representation of a circuit. Various DFT or BIST techniques can then be applied to this new hierarchy which represents a different view of the same circuit.

There are four basic procedures embodied in CRETE: partitioning and hierarchical reorganization, equivalence determination, test methodology embedding, and vector editing. These basic procedures are quite general. TDM specific aspects influence only the last two stages of CRETE. Test methodology embedding procedures alter the canonical representation according to a predefined TDM, and generate tests for individual clouds. For example, while CRETE may identify n registers in a circuit, a BILBO TDM embedding subsystem may either combine some of these registers or partition one or more of these registers into separate registers. The last

[1] CRETE is an acronym for Clouding, hierarchical Reorganization, Equivalence determination, Test methodology embedding, and Editing

step in CRETE edits and composes tests of all clouds to generate a complete test for the entire circuit.

CRETE can be interpreted as a meta-methodology that enables application of and experimentation with any specific TDM. It acts as a shell which facilitates implementation of any specific TDM by standardizing on a canonical representation of the circuit. At the very least, it can be viewed as a library of preprocessing routines which relieves the user from the burden of implementing many often repeated tasks.

The remainder of this paper is organized as follows: Section 2 presents an overview of the CRETE methodology. Section 3 describes the circuit representation and the software environment used to implement CRETE. The basic steps executed by CRETE are detailed in the subsequent sections. Sections 4 and 5 deal with the cloud identification and cloud equivalence problems, respectively. TDM embedding and vector editing are discussed in Sections 6 and 7. Section 8 presents a case study of the Viterbi Decoder circuit processed using CRETE. Finally, concluding remarks are presented in the last section.

2 Overview of the CRETE Methodology

In this section an overview of the various processing steps involved in CRETE is presented. In order to make the discussion concrete, we will illustrate the CRETE methodology in the context of full scan system. As we shall see, even for a simple TDM such as full scan, one can benefit from circuit partitioning in terms of both the test generation time and the quality of test vectors. Application of CRETE to the BALLAST partial scan technique [4] and the well known BIST technique BILBO will also be described briefly.

In full scan designs, all the storage elements in the circuit are made scannable during the test mode. Full scan designs require only combinational automatic test pattern generation (ATPG). In partial scan designs, only a subset of the storage elements are made scannable. Since the complete internal state of the circuit is not always initializable, sequential ATPG may be required [3]. However, it has been shown that by appropriately choosing the subset of storage elements to be made scannable, the circuit may be partitioned into sequential structures for which combinational ATPG suffices [4]. Thus from the standpoint of test generation these structures may be regarded as combinational.

Conventional full scan systems apply combinational ATPG to the entire combinational logic of the circuit. The size of combinational logic circuit, however, can be large and processing all the logic simultaneously may be computationally intensive. Many families of circuits such as data paths, Digital Signal Processing (DSP) chips, and bit-slice architectures can be naturally partitioned into disjoint subcircuits. It will be shown that even if the circuit is connected, for the purpose of test generation, it may still be possible to partition it by removing global input signals. For BIST designs, to test a large combinational logic circuit and achieve a high level of fault coverage using pseudo-random test patterns may require an enormous number of test patterns. It would be simpler if the logic could be divided into disjoint subcircuits and tested as separate entities.

The first step in CRETE clusters the logic gates in the circuit into disjoint partitions. We will refer to these partitions of the combinational logic as *clouds*. The formal definition of a cloud is presented in Section 4. A scan based circuit with multiple scan paths, or a circuit to be tested using a BIST TDM can be partitioned into clouds. An algorithm for clouding a hierarchical circuit without flattening it to its gate-level description is presented in this paper.Instead of processing the whole combinational logic block as one entity, a more efficient way is to apply combinational ATPG to individual clouds and combine the test sets for the clouds to form the test set for the whole combinational logic. We will show that partitioning generates clouds that are more manageable in terms of test generation effort, and thus reduces the overall test generation time.

Besides clustering gates into clouds, CRETE clusters FFs into registers. A complete definition of a register and the corresponding FF clustering rules are given in Section 4. Essentially, a register is a group of "similarly" connected FFs. It should be emphasized that both clouds and registers are reorganization of original circuit and no information is lost in the process of hierarchical reorganization. At the top-most level, a reorganized circuit hierarchy contains only clouds and registers.

Consider a typical bit-sliced circuit BS shown in Figure 1(a). The circuit consists of n identical modules M1, $\cdots$, Mn. Each module consists of a combinational logic block C which can be hierarchical, and a pair of FFs. Clouding the circuit BS leads to a configuration shown in Figure 1(b). However, the FFs of the circuit can be clustered to form registers as shown in Figure 1(c). For scan based circuits, deterministic tests can be generated for the cloud and

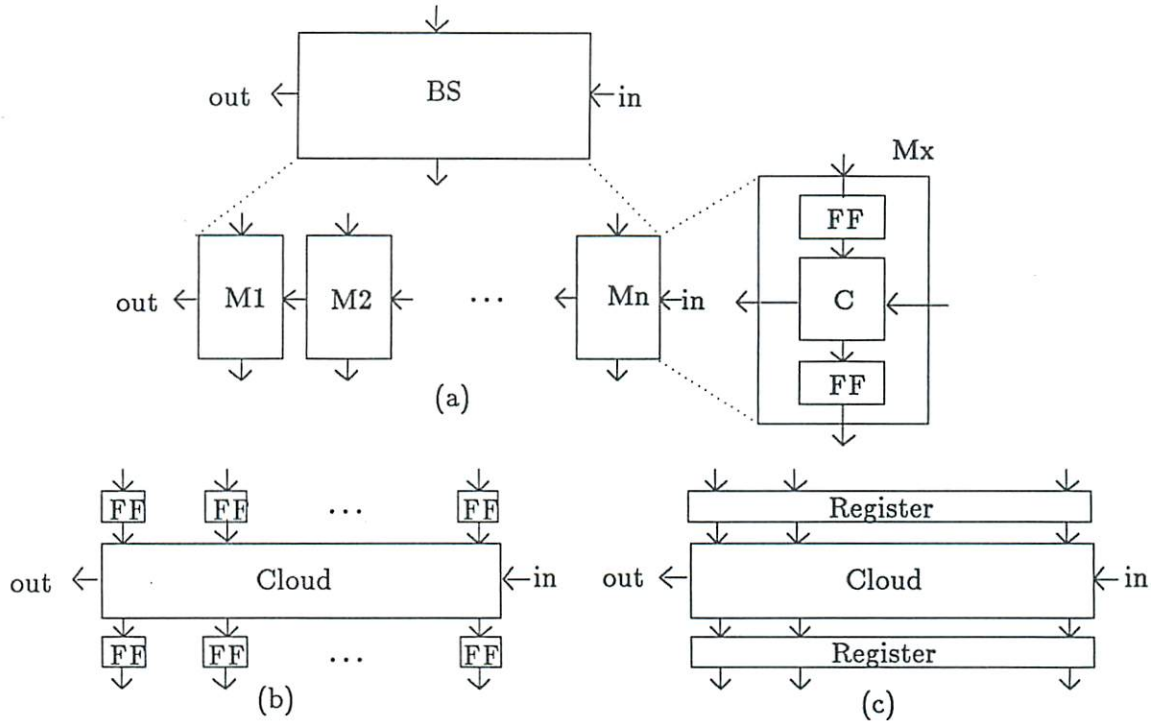

Figure 1: (a) A bit-sliced circuit. (b) After cloud formation. (c) After register formation.

the registers can be modified to have scan capabilities. For BIST designs, the cloud can be tested using pseudo random patterns by modifying registers into linear feedback shift registers (LFSRs). Thus CRETE helps in reorganizing a circuit to aid in TDM embedding process.

Clouds and registers identified by CRETE lead to a canonical partitioning of the circuit. These partitions can be used for TDM embedding and processing. For example, for BILBO and other BIST TDMs, register clustering produces natural candidates for LFSRs. In the case of full scan, instead of processing the entire combinational logic associated with a circuit as one entity, a more efficient way is to apply combinational ATPG to individual clouds and combine the test sets for the clouds to form a complete test set. We will show — both analytically as well as through experiments — that the CRETE methodology leads to a significant reduction in both the test generation effort and the number of test vectors. The reduction in the number of test patterns is $O(\log k)$, where k is the number of clouds in the circuit. Experimental data validates this claim and correlates well with analytical predictions. This is significant for scan-based circuits that often require considerable time to test.

One can further reduce the test effort by realizing that in some cases there may exist clouds in a circuit that are structurally equivalent. The information about cloud equivalence can be

used to reduce the processing effort required for a TDM. For example, in the case of BILBO, if it is known that two clouds are equivalent, the same polynomial can be used for their TPG and SA registers. Also, if the testing is non-exhaustive, only one of them need to be fault-simulated. Similarly, in full scan designs, equivalent clouds can be tested using the same test vectors. CRETE groups equivalent clouds into equivalence classes which are used in subsequent stages of processing.

Testing for structural equivalence, unfortunately, is equivalent to determining graph isomorphism between the corresponding clouds. For the latter problem, to date, no efficient algorithm is known and the problem has indeterminate complexity [7]. We define a new type of equivalence, called *hierarchical equivalence* or *H-equivalence*. It uses the fact that each cloud is a hierarchical entity and checking isomorphism between hierarchical graphs is easier than that for flat graphs because a divide and conquer approach is feasible for the former.

The TDM embedding step in CRETE invokes a user-specified procedure on the partitioned circuit. For full scan, this step simply replaces the original FFs with scan FFs; for partial scan, an appropriately chosen set of FFs are replaced by scan FFs; for BILBO, appropriate registers are converted into LFSRs.

The final step in CRETE, namely test vector editing, is also TDM specific. It results in the derivation of logic values that need be applied to the primary I/O of the circuit in order to exercise a given TDM. For full scan and partial scan TDMs, it consists of composing individual test vectors and responses to form (1) bit streams that can be scanned into the scan paths, (2) the vectors that can be applied at the primary inputs, and (3) the expected responses at the primary outputs and scan paths. For BIST TDMs it involves composing the seeds that need to be scanned into the LFSRs to initialize the internal state of the circuit.

The above steps constitute the basic CRETE methodology. All the steps outlined above, with the exception of equivalence determination and register formation, have been implemented for a full scan TDM in a program called CRETE. In the remainder of this paper we shall use the words 'CRETE methodology' and the 'CRETE system' interchangeably. In order to validate these concepts, CRETE was used to generate test vectors for a complex circuit — a Viterbi decoder chip — obtained from JPL [8]. The results of this experiment, which are presented in Section 8, confirm the basic advantages claimed above.

3 Circuit Representation and System Organization

The circuits processed by CRETE are modeled and stored in an object-oriented database called Cbase. Cbase allows structural aspects of VLSI circuits to be represented in an integrated persistent schema [9]. Readers are referred to [9,10] for details of Cbase data model, the merits of this data organization, and the support provided by the framework. We briefly discuss them here.

Both hierarchical and flat descriptions of circuits can be stored in the database. The basic types used in describing a circuit are *Cell*, *Port* and *Bus*. **Cells** are the fundamental component abstraction. Cells can have attributes specifying whether they represent combinational, register or sequential (i.e., mixture of combinational and register elements) components. The interface to a cell is provided by a set objects of type **Port** which correspond to the input/output ports of logic element represented by the cell. A port can be assigned the attributes 'input', 'output' or 'bidirectional' and can be further refined into individual pins or **terminals**. **Buses** interconnect ports and can be refined into **nets**.

The logic circuit is entered into the database schematically in a hierarchical manner. The CRETE system implemented for full scan circuits, works directly on the database and is organized as four sub-program modules. The first module forms clouds and reorganizes the circuit hierarchy. The output of the first module are hierarchical descriptions of all clouds in the given circuit. The second module identifies the equivalence classes among clouds. This module is currently being implemented. The third part of CRETE system deals with generation of test patterns for all the clouds using our combinational ATPG system called Test Generation System (TGS) [11]. Replication of tests for all the clouds in each equivalent class is also carried out by this module. The final module determines logical ordering of FF bits and physical ordering of the scan FFs. It then permutes test vector sequences to comply with the scan paths. The final output of the CRETE program is a set of reorganized test vectors sorted in four categories: input stimuli and output responses for the primary I/Os, and input stimuli and output responses for each scan path.

4 Hierarchical Reorganization

Any partitioning scheme for the purpose of test generation should satisfy the following two criteria. (1) The partitions should be disjoint (i.e. they should not contain any common logic gates) and faults in the partitions should cover all the faults in the original circuit. This condition imposes a notion of minimality on the generated test as no component is tested twice. (2) The partitions should be independent from the point of view of test generation. In other words, it should be able to generate tests for each partition separately and then combine the tests "easily" to test the whole circuit.

The first step performed by CRETE is to reorganize the circuit hierarchy by clustering combinational gates and FFs into clouds and registers, respectively, in a manner consistent with the above criteria.

4.1 Cloud Formation

The entire combinational logic of the given sequential circuit is partitioned into disjoint subcircuits known as *clouds*. A **cloud** is a maximal group of directly interconnected combinational logic blocks in a circuit. The inputs to a cloud are either primary inputs or outputs of FFs; the outputs from a cloud are either primary outputs or inputs to FFs. Thus a cloud is surrounded by primary I/Os and/or FFs. Clouds can be constructed by clustering combinational logic blocks using the following rules.

Clouding Rules: *Two combinational logic blocks A and B belong to the same cloud if any of the following criteria is satisfied.*

1. *One block directly feeds the other block*
2. *There exists a common input signal that fans out to both the blocks.*
3. *One block is fed by the output of a FF (Q) and the other block is fed by the inverted output of the same FF ($\overline{Q}$) (thereby introducing a dependency between the two blocks).*

4.2 Register Formation

Registers are formed by grouping FFs together. FFs are *homogeneous* if they correspond to identical library elements. Examples of such elements are: (1) D-type FF with no asynchronous

reset or set; (2) D-type FF with scan I/O; (3) JK-type FF, edge triggered with asynchronous preset and set. FFs can have control lines, such as hold enable signals. Equivalent clock signals correspond to two clock signals CLK1 and CLK2 where CLK1 is equal to CLK2.

A **register** is a maximal group of FFs satisfying the following rules.

Register Rules: *Two FFs A and B belong to the same register if all of the following criteria are satisfied.*

1. *They feed exactly one cloud and are being fed by exactly one cloud.*
2. *They are homogeneous and driven by an equivalent clock signal.*
3. *They are controlled by the same or equivalent mode control signals, if any exist.*

It should be noted that the registers defined above have nothing to do with the user-defined registers. Our definition of register simply enables us to process a set of identically connected and homogeneous FFs as a group. Clustering of FFs into registers have many useful applications such as identifying LFSRs for circuits employing BIST, and simplifying the graph model of the clouded circuit used in BALLAST partial scan methodology. The legal configurations for a cloud and a register are illustrated in Figure 2. An example illustrating the clouding rules is given below.

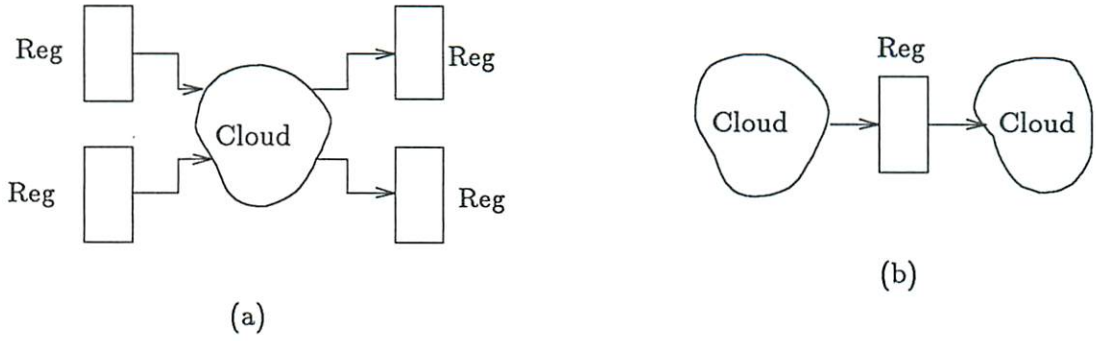

Figure 2: Legal configurations for (a) a cloud, and (b) a register.

The above definitions imply that a cloud can be fed by (can feed) one or more registers; however, a register can be fed by exactly one cloud and can feed exactly one cloud.

Example : Consider the circuit shown in Figure 3(a), where C1 through C6 are combinational logic blocks, and interconnected via FFs. For the FFs fed by C2 and feeding C5, some have hold enable mode while others do not have one. The combinational blocks C1, C2, C3 and C4

are connected together, and hence form a cloud A1. The combinational blocks C5 and C6 form another cloud A2. These clouds are shown in Figure 3(b). FFs are clustered to form registers. The FFs fed by C1 and feeding C3 form the register R1. Registers R2 and R5 are formed in a similar manner. The FFs fed by C2 and feeding C5 are classified into two types based on having hold enable mode, and hence form two registers R3 and R4. The two clouds A1 and A2 are totally disjoint and are surrounded by only primary I/Os and/or registers. □

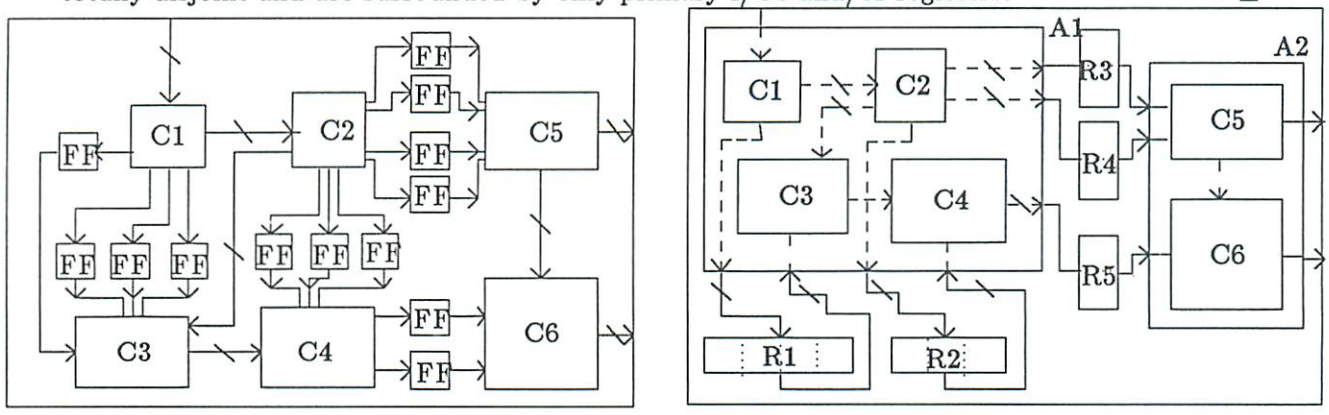

Figure 3: Two versions of a sequential circuit. (a) the unclouded version, and (b) the clouded version

While partitioning the circuit, two special types of clouds can be recognized, viz. *v-cloud* and *r-cloud*. A **v-cloud** *(vacuous cloud)* is a cloud comprising of only a single net and no combinational logic. V-clouds can be formed when (1) a primary input directly feeds a register or/and a primary output, or (2) when a register directly feeds another register or/and a primary output.

Further savings in test time can be accrued by recognizing the micro-structure of a cloud. In particular, when identical logic blocks that are connected only via primary inputs comprise a cloud, one can easily generate tests for the cloud. An **r-cloud** *(repetitious cloud)* is a cloud containing identical combinational logic blocks, referred to *pseudo clouds (p-clouds)* and which are fed by one or more common input signals. The test set for an r-cloud is formed by generating tests for a p-cloud and replicating them. Examples of v-cloud, p-clouds and r-cloud are given in Figures 4(a) and 4(b) respectively.

Clouds, as defined above, satisfy both of the partitioning criteria mentioned earlier. Since each cloud is a maximal group of directly interconnected combinational logic, the disjointness

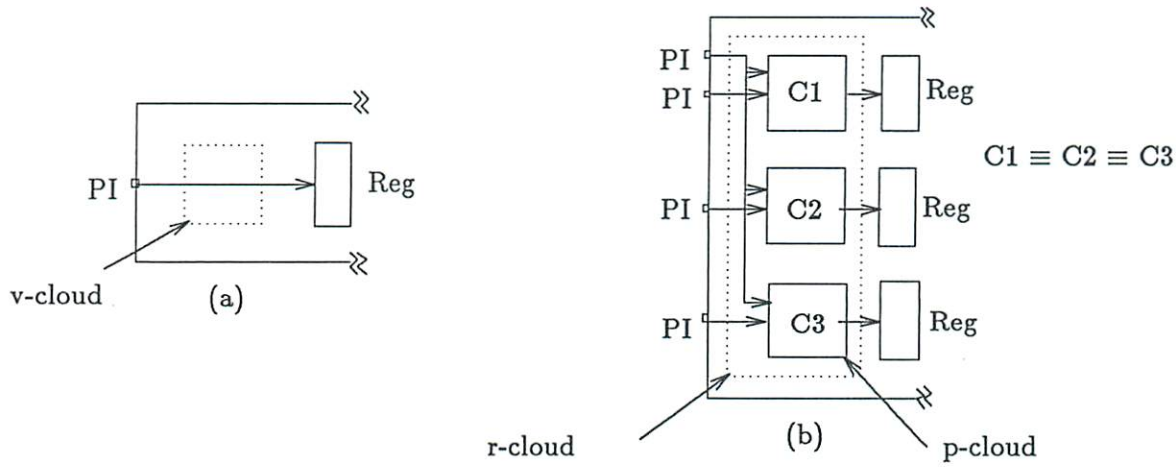

Figure 4: Types of special clouds. (a) v-cloud (vacuous cloud), and (b) r-cloud and p-clouds (repetitious cloud and pseudo clouds).

condition is trivially satisfied. Also the clouds are independent because they do not share any gates or nets (except in the case of r-clouds). Thus, any conflict in the test vector arising due to fanning out of input signals to two clouds is avoided, and the test sets can be easily combined. As the combinational logic portion of the circuit can be constructed by juxtaposing all its clouds, a test set can be obtained by juxtaposing the test sets of all its clouds. A test can be generated for a p-cloud and replicated to create a test for an r-cloud.

Clouding of a flat circuit description is a straightforward task as it is equivalent of finding maximal connected components in the circuit graph without the FFs. However, for equivalence testing (see Section 5), it is desirable to preserve the original circuit hierarchy as far as possible while performing clouding. In the process of clouding, the hierarchical circuit is selectively flattened and is reorganized into a new hierarchy. The algorithm for clouding is given in Appendix 1.

For combinational circuits described hierarchically, clusters of maximally connected logic (clouds) are identified. If an hierarchical circuit is sequential, the procedure is invoked recursively on its subcircuits, and the subcircuits are moved up one level in the hierarchy by flattening the sequential circuit. Finally, at the top level of the hierarchy there will be only registers or hierarchical clouds.

5 Cloud Equivalence

As mentioned earlier, one can reduce the test generation effort by observing that in some cases there may exist clouds in a circuit which are structurally equivalent. There can be significant reduction in test generation time if equivalent clouds are determined in advance.

Interchangeability of test vectors is guaranteed by structural equivalence. Unfortunately, no efficient test exist for determining structural equivalence between two circuits. This problem can be easily mapped on to the graph isomorphism problem, a well known problem whose complexity is as yet unknown [7]. We circumvent this hard problem by solving a slightly restricted version of this problem. This new version of equivalence, though easy to test, preserves the basic notion of equivalence as far as test vector generation is concerned. However, it is pessimistic in that in some rare instances, it may report structurally equivalent clouds as being non-equivalent.

Because each cloud is hierarchical, the problem of structural equivalence reduces to the problem of determining equivalence between two hierarchical graphs. We refer to two component hierarchies which are isomorphic, i.e. they are identical except for renaming of their constituent objects, as *H-equivalent.*

In the following, for any cell C, $C.ports$, $C.buses$ and $C.subcells$ refer to the set of ports, buses and subcells of C respectively. For a port P, $P.isOnCell$ refer to the cell which contains P; for a bus B, $B.ports$ and $B.isInCell$ refer to the set of ports connected by B and the cell containing subcells connected by B, respectively.

Two clouds C1 and C2 are **H-equivalent** if any of the following two conditions hold.

1. C1 and C2 are instances of the same library component. A one to one correspondence exists between the ports on C1 and those on C2.

2. There exist two bijections,
   $\sigma_1 : C1.subcells \rightarrow C2.subcells$ and
   $\sigma_3 : C1.ports \rightarrow C2.ports$,
   such that $\forall c_i \in C1.subcells$, c_i and $\sigma_1(c_i)$ are H-equivalent, and buses interconnecting subcells of C1 and C2 have $1-1$ correspondence (denoted as σ_2) in terms of the ports they interconnect. That is, $\forall b_i \in C1.buses$ with $b_i.ports = \{p_1 \ldots p_k\}$, $\exists b'_j \in C2.buses$ with $b'_j.ports = \{p'_1 \ldots p'_k\}$ such that $\forall m$, $1 \leq m \leq k$ either $\sigma_2(p_m) = p'_m$, or p_m and p'_m

are corresponding ports on the H-equivalent subcells $p_m.isOnCell$ and $p'_m.isOnCell$.

An algorithm for checking H-equivalence is presented in Appendix 2. Two cloud hierarchies are compared for equivalence recursively. Fast pruning is possible because of the tree structure used to represent circuit hierarchy and that the general graph isomorphism problem has to be solved only at each level, rather than for the whole graph. Also, the information about multiple instantiations of library components helps making the equivalence procedure more efficient by increasing the granularity of the basic cells.

The determination of the mapping σ_3 is linear in the number of ports since they are ordered. C1 and C2 are equivalent only under this mapping. At the higher levels, the ports associated with two clouds C1 and C2 should be used in a manner consistent with σ_3, otherwise these cells would not be equivalent. The bulk of processing in *H-equivalence* is done in determining three mappings, σ_1, σ_2 and σ_3, which map subcells, buses, and ports of C1 to those of C2, respectively. In particular the following three checks are performed.

1. σ_1 should map cells in C1.subcells to only H-equivalent subcells in C2.subcells.

2. σ_2 should map one bus to another only if their cardinality is the same and the sets of cells connected by these buses should be equivalent under σ_1. In other words, if b_i connects cells $\{c_1, \ldots c_k\}$, then $\sigma_2(b_i)$ should connect cells $\{\sigma_1(c_1), \ldots \sigma_1(c_k)\}$.

3. The checks performed on σ_3 constitute (1) a check on the cardinalities and directions of the mapped ports, (2) constraint that a port on C1 (and not on one of the subcells of C1) should map on to a port on C2, and (3) the ports on subcells of C1 should be mapped to H-equivalent subcells of C2, and should be consistent with σ_3 returned while checking the H-equivalence of subcells.

Once suitable σ_1, σ_2 and σ_3 are found, the final check simply ascertains that for each bus the σ_2 bijection is consistent with σ_1 and σ_3. Partial results are stored to avoid recomputation and the transitivity property of H-equivalence is exploited. If **H-equivalence**(c_i, c_j) and **H-equivalence**(c_j, c_k) have been established, then it is true (and return an appropriate σ_3) for **H-equivalence**(c_i, c_k) without any computation. A detailed analysis of the complexity of the H-equivalence algorithm is given in Appendix 3.

6 Test Methodology Embedding

We shall next describe embedding a few DFT and BIST techniques into a hierarchically reorganized circuit. We will only present details for the full scan methodology.

6.1 Full scan

For scan based circuits, test for each cloud is generated automatically by invoking a combinational ATPG system called TGS [11]. Since TGS can only process flat gate-level descriptions, each hierarchical cloud is converted to its flattened description before invoking TGS. Tests need only be generated for one cloud per equivalent class. This test set is duplicated for all the clouds in that class.

The ATPG for combinational circuits has been shown to be an NP-complete problem [12]. All known algorithmic procedures for it have an exponentially increasing worst-case run-time as the circuit complexity increases. However, in practice, empirical studies place the average case complexity in the neighborhood of $O(n^2)$, where n is the number of gates in the circuit [12]. The following analysis establishes the benefits of partitioning both for test generation and test application (i.e., number of test vectors).

6.1.1 Reduction in Test Generation Time

Assume that the average-case time complexity of ATPG for a circuit with n gates is cn^α, where α has a value ranging from 2 to 3 and c is a constant. If the circuit is partitioned into k disjoint partitions, each containing n_i gates $(1 \leq i \leq k)$ such that $\sum_{i=1}^{k} n_i = n$, then the overall test generation time will be $c\sum_{i=1}^{k} n_i^\alpha$. It is easy to see that $cn^\alpha > c\sum_{i=1}^{k} n_i^\alpha$ if $\alpha > 1$, and hence, partitioning reduces the total test generation time.

Test generation time is further reduced by doing ATPG only for distinct partitions. Clearly, the savings accrued because of this are circuit dependent. In the worst case, all the partitions in the circuit may be distinct and ATPG must be carried out for each individual partition. However, in practice, most VLSI chip designs such as DSP chips, bit-slice architectures, and vector processors tend to be repetitious in nature and there can be significant reduction in ATPG time if equivalences are determined *a priori*. At the same time, since this may lead to

wasted computation for the circuits that do not fall into the above category. Thus, equivalence determination should be carried out based on a user supplied flag.

6.1.2 Reduction in Number of Test Vectors

It can be shown that the size of the test set obtained using CRETE, in general, is smaller than that for an unpartitioned circuit. Intuitively, the composed test vector set tends to be smaller due to following fact. In CRETE, every test pattern can, if required, simultaneously detect at least one new fault in every cloud. One can think of the CRETE methodology as *concurrent test pattern generation* for independent portions of the circuit. However, in the unclouded circuit every new test pattern is guaranteed to cover only one additional fault. Circuit partitioning thus leads to maximum parallelism in test generation and a significant reduction in both the number of test vectors and test generation time is achieved.

The above intuitive idea is formalized below. Analytical expressions for computing the saving in test patterns are derived. They are validated through experimental results and a good match is observed between theory and practice.

6.1.3 Fault Coverage Analysis

Let C be the circuit under test with F detectable faults. Since we are considering deterministic testing, every test pattern is generated for a target fault in the circuit. In general, a test pattern also detects other as yet undetected faults. If the circuit does not contain any random-pattern-resistant faults, it has been shown empirically that every new pattern detects a constant fraction r of as yet undetected faults. The value of r is circuit dependent and it typically ranges from 0.01 to 0.2 [13].

Let $f(p)$ be the number of faults detected and $t(p)$ be the fault coverage after applying p patterns to the circuit C. Then,

$$
\begin{aligned}
f(0) &= 0, \text{ and} \\
f(p) &= \begin{cases} \lfloor 1 + f(p-1) + r \times (F - f(p-1) - 1) \rfloor & \text{if } f(p-1) < F \\ F & \text{if } f(p-1) = F. \end{cases}
\end{aligned} \tag{1}
$$

In Eq. (1), the constant term 1 is due to the target fault for the pattern p, the term $f(p-1)$ de-

notes the number of faults detected by the first $(p-1)$ patterns, and the expression $F - f(p-1) - 1$ is the number of yet undetected faults after $(p-1)$ patterns, not including the target fault. The solution to this recurrence relation yields

$$f(p) = F \times (1 - s^p) + (1 - s^p) \times s/(1-s), \tag{2}$$

where $s = (1-r)$. The fault coverage $t(p)$ for the unclouded circuit after applying p patterns will be $t(p) = f(p)/F$, and the number of vectors required to obtain 100 % coverage for the unclouded circuit, denoted as p_{100}, can be shown to be

$$p_{100} = 1 + \frac{\log((1-s)F + s)}{\log(1/s)}. \tag{3}$$

Let the circuit C be partitioned into k clouds, $C_1, C_2, \ldots, C_k$, by invoking the clouding procedure. Since the clouds are disjoint partitions, it follows that $C_i \cap C_j = \phi, \forall i, j, i \neq j$. Let the total number of possible faults in cloud C_i be $F_i (where \sum_{i=1}^{k} F_i = F)$. Also, let $f_i(p)$ denote the faults in cloud C_i detected after applying p patterns to the clouded circuit. Then for the cloud C_i, we have

$$\begin{aligned} f_i(0) &= 0, \text{ and} \\ f_i(p) &= \begin{cases} \lfloor 1 + f_i(p-1) + r \times (F_i - f_i(p-1) - 1) \rfloor & \text{if } f_i(p-1) < F_i \\ F_i & \text{if } f_i(p-1) = F_i. \end{cases} \end{aligned} \tag{4}$$

The fault coverage $t_i(p)$ for the cloud C_i after applying p patterns is given by $t_i(p) = f_i(p)/F_i$, and the fault coverage $t(p)$ for the entire clouded circuit will be

$$t(p) = \frac{\sum_{i=1}^{k} f_i(p)}{F}. \tag{5}$$

Let F_{max} be the maximum value among the F_i's, i.e. $F_{max} = \max(F_1, F_2, \ldots, F_k)$. Both lower and upper bounds on the value of F_{max} can be derived by the following arguments.

Lower Bound: The circuit is partitioned into k clouds. Thus the average number of faults per cloud is given by F/k. Since all the clouds cannot contain less than the average number of faults, $F_{max} \geq F/k$.

Upper Bound: Consider the other extreme case where most of the circuit elements form a single cloud and each of the remaining $(k-1)$ clouds is a vacuous cloud containing a single net. Each single net gives rise to 2 faults and thus $F_{max} \leq F - 2(k-1)$.

The largest cloud determines the overall number of test patterns to achieve a required fault coverage. This is due to the fact that by the time the cloud with F_{max} faults achieves

the required fault coverage all the remaining clouds would have already achieved this coverage. Thus the value of p required to obtain 100 % coverage for the clouded circuit, denoted as p'_{100}, is given by

$$p'_{100} = 1 + \frac{\log((1-s)F_{max} + s)}{\log(1/s)} \tag{6}$$

The reduction in the number of patterns due to clouding in achieving 100 % coverage is given by the expression

$$\begin{aligned} p_{100} - p'_{100} &= \frac{\log\left(\frac{(1-s)F+s}{(1-s)F_{max}+s}\right)}{\log(1/s)} \\ &\simeq \frac{\log\left(\frac{F}{F_{max}}\right)}{\log(1/s)} \text{ if } F \gg 1 \end{aligned} \tag{7}$$

For circuits partitioned into k equal size clouds, $F_{max} = F/k$, and the number of patterns reduced is given by

$$p_{100} - p'_{100} = \frac{\log k}{\log(1/s)} \tag{8}$$

The savings in the number of test patterns (from Eq. (8)) as a function of the number k of clouds is plotted on a semilog scale in Figure 5. The analytical curves are straight lines since the savings are proportional to $\log k$.

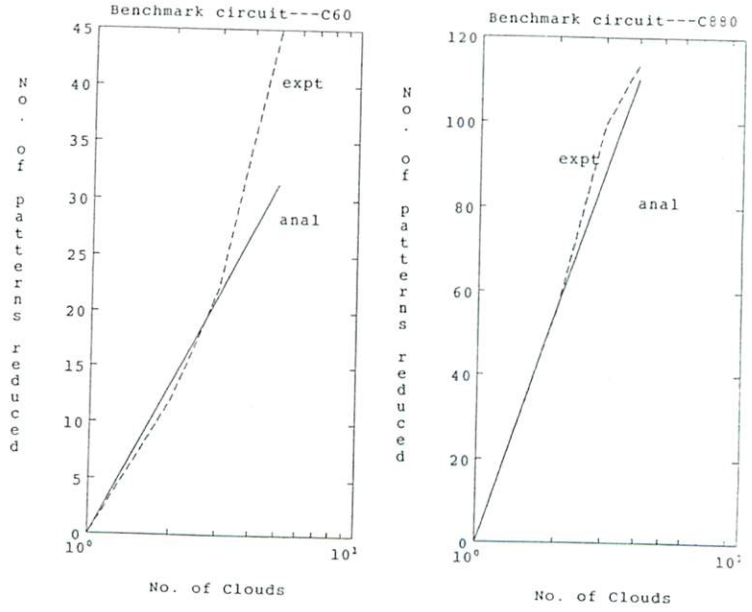

Figure 5: Plot of reduction in test patterns vs. number of clouds

6.1.4 Experimental Results

Two combinational circuits C60 and C880 (a circuit from the ISCAS-85 benchmarks) were used in an experiment to validate the above analytical results. Each of the circuits forms a single cloud by itself. For each circuit, tests (say T_1) were determined using TGS. To obtain a circuit with k clouds each with the same value of r, the circuits were replicated k times. Tests (say T_k) were also obtained for the k-clouded circuit by treating it as a single large circuit. In this case, since the circuit contains k identical clouds, T_1 can be replicated k times and juxtaposed to make up the tests for the large circuit. Thus the reduction in test patterns is given by $T_k - T_1$.

These results are plotted in the Figure 5 for different values of k. The analytical equation for test vector reduction for equal clouds (Eq. (8)) results in a straight line. The experimental results when plotted on the same axes also result in a curve close to a straight line. The analytic results shown are for $r = 0.0125$ and $r = 0.05$ for C880 and C60, respectively.

The table given below presents the results of the experiments conducted on various ISCAS-85 benchmark circuits. In Table 1, Column 1 refers to various circuits. To generate hybrid circuits with more than one cloud, benchmark circuits were juxtaposed to form bigger circuits (for example, circuits c60 and c880 were juxtaposed to produce the hybrid circuit c60c880). Column 2 indicates the total number of faults in the circuit. The number of test vectors required to test all irredundant faults of each circuit is given in Column 3. The corresponding fault coverage in percentage is given in Column 4. For hybrid circuits, the reduction in test vectors due to clouding is calculated as follows: Let T_x, T_y and T_{xy} be the number of test vectors generated using TGS for circuits cx, cy, and the hybrid circuit $cxcy$, respectively. If the hybrid circuit is processed by CRETE, it requires only $\max(T_x, T_y)$ test vectors. Thus the reduction in the number of test vectors is given by $T_{xy} - \max(T_x, T_y)$ and is listed in Column 5. As we observe, the reduction becomes significant for large circuits. Thus, the experimental data correlates well with the analytical results.

6.2 Partial Scan

For the BALLAST partial scan methodology [4], the circuit is partitioned into clouds and modeled as a directed graph. Both clouding and register clustering simplifies the graph model of the original circuit. BALLAST attempts to optimally select a subset of storage elements to

Table 1: CRETE results on benchmark circuits

1	2	3	4	5
Circuit	Faults	Test Vectors	Fault Coverage	% Reduction
c60	120	21	93.33	-
c880	1760	100	100	-
c1908	3816	205	99.55	-
c3540	7080	540	95.96	-
c60c880	1880	102	99.57	2
c880c1908	5576	267	99.68	24
c1908c3540	10896	617	97.29	12

be made scannable, so that tests for the circuit resulting from the removal of the scan elements can be generated using a combinational test generation algorithm.

6.3 BIST designs

For circuits using BIST TDMs, such as BILBO, test sessions can be independently generated for clouds by selecting appropriate registers (obtained by register clustering) and modifying to be LFSRs to function as pseudorandom pattern generators and/or signature analysers [5]. For equivalent clouds, LFSRs with the same characteristic polynomial can be used.

7 Test Vector Editing

Each input to a cloud comes either from a primary input or from a storage element in one of the scan chains. Similarly, each cloud output drives either a primary output or a storage element in one of the scan chains. In order to apply a test vector to each cloud one has to determine the bit streams that need to be scanned into each scan chain, and the vector that needs to be applied to the primary inputs. The process of splicing and composing individual test vectors into bit streams for scan chains, the primary inputs, and the expected primary and scan outputs is referred to as *test vector editing*.

For a cloud, each test pattern consists of input and output test vectors, referred to as *input stimulus* and expected *output response*. Among input stimuli (output responses), some bits correspond to primary inputs (outputs) called *I/O bits*, and rest to scan FF outputs (inputs) called *FF bits*.

For the test set generated for a cloud the ordering of FF bits in a test pattern is referred to as the *logical ordering*, and the ordering of the corresponding FFs in a scan path is referred to as the *physical ordering*. The logical ordering is obtained by separating the FF bits from I/O bits in test patterns. The physical ordering of FFs is determined by tracing all scan paths of the circuit. Generally, the logical ordering does not match the physical ordering and test vector editing is required.

Since the physical ordering is fixed by the circuit design, the FF bits in the logical ordering have to be permuted to comply with the physical ordering of FFs in the scan paths. This concept is illustrated in Figure 6. The permuted test patterns can then be reformatted as a test sequence that can be loaded into the scan paths so that the FF bits occur in the correct order. The composed I/O bits are applied to primary inputs. Similarly, the expected response that will be latched into the scan chains after the application of the test vector, and that at the primary outputs, is computed.

Referring to Figure 6, cloud 2 requires the maximum number of test vectors and determines the total number of test vectors required for the whole circuit. The vectors for other clouds are filled with random 0s and 1s after all their deterministically derived vectors have been applied. In this Figure, since cloud n is equivalent to cloud 1, the same test set is used for both.

8 Case Study—Viterbi Decoder Circuit

To validate the concepts underlying CRETE, a Viterbi decoder circuit designed by JPL was processed. The circuit contains 16 butterfly processors, each of which has a complexity of about 1800 gates. To enhance testability, much of this design uses a scan based architecture. Since the circuit is made up of 16 identical butterfly processors, only one butterfly processor needs to be processed for test vector generation.

The hierarchical description of one butterfly processor was entered into the Cbase database

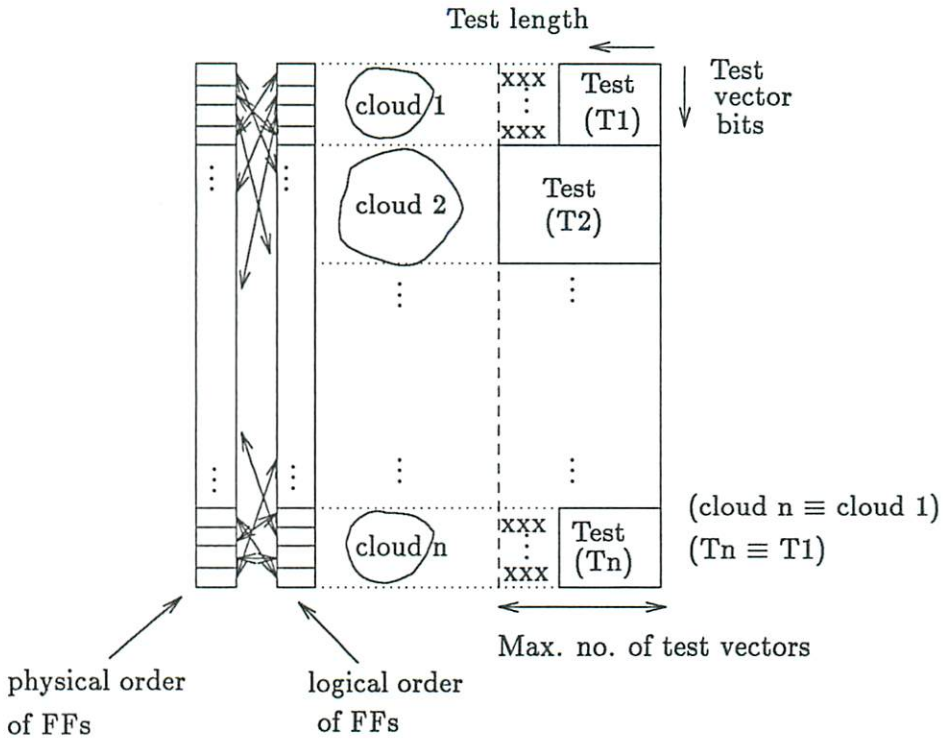

Figure 6: Test Vector Editing

using the graphical user interface. The butterfly processor circuit contains two scan paths with 88 and 6 flip-flops, respectively. The long scan path is used for enhancing controllability and observability while the short scan path is used for scanning in the butterfly ID. During test mode both the scan paths are used to scan test data. Since the circuit is not strictly a classical full scan design, special models for parts of the processor were employed.

CRETE software was invoked on this butterfly processor to form clouds. The combinational logic of the circuit was partitioned into 9 clouds which were classified into 5 equivalence classes. CRETE determined that the entire butterfly processor could be tested with a test set containing just 34 test patterns.

This circuit is peculiar in that under CRETE it gets transformed into a circuit with a very large number of inputs as compared to the number of gates in it when the circuit is made scannable. The combinational logic portion of the circuit does not contain any random pattern resistant faults. As a result, only 39 test vectors are required to test the logic when modeled as only one combinational block rather than by using clouds. This still leads to 15 % reduction in test time.

The test set generated for the butterfly processor holds good for all 16 identical butterfly processors of the decoder circuit. Thus the test set of a single butterfly processor can be replicated to form a test set for the entire decoder circuit. Further details can be found in [14].

9 Conclusions

This paper presents CRETE methodology for canonical partitioning of sequential circuits. This leads to reorganization of circuit hierarchy aiding in various TDM embedding process. An implementation of this methodology for full scan based circuits is also described. It is shown that this approach results in significant reduction in test generation effort. This reduction is a direct consequence of circuit partitioning and equivalence determination.

Besides reducing the test generation effort, the CRETE methodology also achieves a reduced number of test vectors without a loss in fault coverage. An analytic expression for computing the reduction in the number of test vectors is presented. Experimental results that substantiate our model are presented. This approach can be easily extended to various partial scan and BIST TDMs.

Acknowledgements: The authors wish to thank Rajesh Gupta for his comments and contributions to the clouding procedure.

References

[1] M. A. Breuer and A. D. Friedman. *Diagnosis and Reliable Design of Digital Systems.* Computer Science Press, Rockville, Maryland, 1976.

[2] T. W. Williams and K. P. Parker. Design for Testability—A Survey. *Proc. IEEE*, 71(1):98–112, January 1983.

[3] V. D. Agrawal, K. T. Cheng, D. D. Johnson, and T. Lin. A Complete Solution to the Partial Scan Problem. In *Proc. Int'l. Test Conf.*, pages 44–51, August 1987.

[4] Rajesh Gupta, Rajiv Gupta, and M. A. Breuer. BALLAST: A Methodology for Partial Scan Design. In *Proc. 19th Int'l. Symp. on Fault-Tolerant Computing*, pages 118–125, August 1989.

[5] E. J. McCluskey. Built-In Self-Test Techniques. *IEEE Design & Test*, 2(2):21–28, April 1985.

[6] B. Koenemann, J. Mucha, and G. Zwiehoff. Built-In Logic Block Observation Techniques. In *Proc. Int'l. Test Conf.*, pages 37–41, August 1979.

[7] M. R. Garey and D. S. Johnson. *Computers and Intractability.* W. H. Freeman and Co., New York, NY, 1979.

[8] I. S. Hsu. *On Testing VLSI Chips for the Big Viterbi Decoder.* Technical Report TDA Progress Report, Jet Propulsion Laboratory, October-December 1988.

[9] Rajiv Gupta, W. H. Cheng, Rajesh Gupta, I. Hardonag, and M. A. Breuer. An Object-Oriented VLSI CAD Framework: A Case-Study in Rapid Prototyping. *IEEE Computer*, 22(5):28–37, May 1989.

[10] M. A. Breuer, W. H. Cheng, R. Gupta, I. Hardonag, E. Horowitz, and S. Y. Lin. Cbase 1.0: A CAD Database for VLSI Circuits Using Object-Oriented Technology. In *Proc. Int'l. Conf. on Computer-Aided Design*, pages 392–395, November 1988.

[11] *Test Generation System (TGS) User's Manual—Version 1.0.* Technical Report CENG 89-03, University of Southern California, 1989.

[12] H. Fujiwara. *Logic Testing and Design for Testability.* MIT Press, Cambridge, Massachusetts, 1985. chapter 4.

[13] M.S. Abadir and M.A. Breuer. Scan Path with Look Ahead Shifting SPLASH. In *Proc. Int'l. Test Conf.*, pages 696–704, September 1986.

[14] M. A. Breuer, M. Driscoll, Rajesh Gupta, Rajiv Gupta, Sheng Lin, and R. Srinivasan. *Test Generation for the JPL Viterbi Decoder Chip.* Technical Report CENG 90-08, University of Southern California, 1990.

Appendix I

Algorithm 1 *formClouds (CELL)*

INPUT: *A hierarchical circuit CELL;*
OUTPUT: *Clouded circuit with reorganized hierarchy*

```
procedure formClouds (CELL);
if CELL has no subcell
then Determine whether CELL is either combinational or register;
     if CELL is combinational formConnectedComponents (CELL);
else for all sequential subcells of CELL formClouds (subcell) and flatten1 (subcell);
     for all combinational subcells of CELL formClusters (subcell) and flatten1 (subcell);
     formConnectedComponents (CELL); }

procedure flatten1 (CELL);
if CELL has subcells
        for all ports of CELL
                Determine the netlist connecting internal subcells
                        of the CELL and external cells via the port;
                Remove the port from the netlist and delete the port;
        for all subcells of CELL
                make the parent of CELL as the parent of subcell;
        Delete CELL;

procedure formConnectedComponents (CELL);
if CELL has no subcell
then S ← { s | s is a combinational gate of CELL; }
else S ← { s | s is a combinational subcell of CELL; }
∀ s, s ∈ S, unmark(s); ConnectedComponent ← 0;
while S ≠ ∅
      C ← ∅; Select s, such that s ∈ S and s is unmarked; C ← C ∪ s;
      while ∃ s, such that s ∈ C and s is unmarked
            Select s, such that s ∈ C and s is unmarked; mark(s);
            Determine { x | x ∈ S, and x is connected to s };
            S ← S − {x}; C ← C ∪ {x}; }
      if C ≠ S
          Create a new cell and make it the parent of all elements in C;
          Make CELL the parent of the new cell;
      ConnectedComponent ← ConnectedComponent + 1;

procedure formClusters (CELL);
if CELL has subcells
        for all combinational subcells of CELL
                formClusters (subcell) and flatten1 (subcell);
formConnectedComponents (CELL);
```

Appendix II

Algorithm 2 *H-equivalence (C1, C2);*

INPUT: *Hierarchical clouds C1, C2;*
OUTPUT: *Determines if C1 and C2 are equivalent. Returns TRUE/FALSE and the port mapping σ_3 under which C1 and C2 are H-equivalent.*

procedure *H-equivalence* (C1, C2);

if C1 and C2 are copies of same library cell
then determine σ_3 : C1.ports $\rightarrow$ C2.ports;
  return (TRUE, σ_3);
if |C1.subcells| $\neq$ |C2.subcells| or |C1.ports| $\neq$ |C2.ports| or |C1.buses| $\neq$ |C2.buses|
then return (FALSE, $\emptyset$);
for σ_1 : C1.subcells $\rightarrow$ C2.subcells s.t.
  $\forall c_i \in$ C1.subcells, **H-equivalence** $(c_i, \sigma_1(c_i))$ = (TRUE, σ_{i3})
 for σ_2 : C1.buses $\rightarrow$ C2.buses s.t.
  $\forall b_i \in$ C1.buses,
  (1) $|b_i$.ports$| = |\sigma_2(b_i)$.ports$|$, and
  (2) **if** b_i connects $\{c_1, \ldots c_k\}$ **then** $\sigma_2(b_i)$ connects $\{\sigma_1(c_1), \ldots \sigma_1(c_k)\}$
  A $\leftarrow$ C1.ports $\cup$ c_i.ports $\forall c_i \in$ C1.subcells;
  B $\leftarrow$ C2.ports $\cup$ c_j.ports $\forall c_j \in$ C2.subcells;
  for σ_3 : A $\rightarrow$ B s.t. $\forall p_i \in$ A,
    (1) p_i.width $= \sigma_3(p_i)$.width
    (2) p_i.direction $= \sigma_3(p_i)$.direction
    (3) **if** $p_i \in$ C1.ports, **then** $\sigma_3(p_i) \in$ C2.ports
    (4) **if** $p_i \in c_i$.ports, $c_i \in$ C1.subcells **then** $\sigma_3(p_i) \in \sigma_1(c_i)$.ports **and**
      H-equivalence $(c_i, \sigma_1(c_i))$ = (TRUE, σ_{i3})
     then $\sigma_3(p_i) = \sigma_{i3}(p_i)$
   if $\forall b_i \in$ C1.buses, $\forall p_j \in b_i$.ports, σ_1 (p_j.isOnCell) $= \sigma_3(p_j)$.isOnCell **and**
    $\sigma_3(p_j) \in \sigma_2(b_i)$
   then return (TRUE, σ_3);

Appendix III
Complexity Analysis of H-equivalence algorithm

Clearly, the H-equivalence algorithm is enumerative for worst-case examples where no pruning can be done. Also, if the original circuit is flat, the clouding procedure will return flat clouds and the algorithm will require exponential run time. However, if the user given hierarchy is well constructed, the clouding procedure will yield cloud hierarchies that lead to a good partitioning of the H-equivalence problem.

If in the original circuit hierarchy each component has a bound on the number of subcomponents and buses at any hierarchical level, then H-equivalence can be checked in polynomial time. In particular, if no component in the user hierarchy has more than $O(\log n)$ subcomponents or

buses (where 'n' is the total number of components in the circuit), and the reorganization preserves the branching factor in the clouded hierarchy, then H-equivalence can be established in polynomial time.

The bound is estimated as follows. The main enumerative steps in this algorithm rely on picking appropriate bijections σ_1, σ_2 and σ_3. Let s_{max}, b_{max} and p_{max} be the maximum number of subcells of any cell, maximum number of buses of any cell and maximum number of ports connected by any bus, respectively. If the total number of cells in the circuit hierarchy is n, then the worst time complexity is bounded by $O(n.2^{s_{max}}.2^{b_{max}}.2^{p_{max}})$. Since buses in typical circuits have bounded fanout, the effect of p_{max} in this expression can be ignored. The values s_{max} and b_{max} are computed from the clouded hierarchy. However, we are interested in estimating these parameters in the user given hierarchy.

Consider, once again, the clouding procedure. It flattens one level of the hierarchy and then reclusters the combinational block at this level. If the combinational blocks are well connected they will get reclustered. For those circuits for which the amount of reclustering leads to less than $O(\log n)$ subcells and buses at each level, the H-equivalence algorithm will run in polynomial time. This is due to the fact that s_{max} and b_{max} are bounded by $O(\log n)$. Without making any further assumptions about the circuit, nothing more general can be said about the run-time of the equivalence algorithm.

It should be noted that there can be structurally equivalent clouds that are not H-equivalent. Such a situation can arise if the circuit description contains equivalent clouds that are split differently across various hierarchy levels in the user given hierarchy. However, if the user has consistently used library components whenever two or more copies of a component are needed, this phenomenon will seldom occur. It is rare that disjoint partitions in different parts of the circuit will give rise to identical flattened structures if their hierarchies are different. Of course one can flatten the hierarchies and test directly for structural equivalence, but as mentioned earlier, no efficient algorithms are known for this approach.