Comput. Math. Applic. Vol. 13, No. 5/6, pp. 519–536, 1987
Printed in Great Britain.

0097-4943/87 $3.00 + 0.00

TEST SCHEDULES FOR VLSI CIRCUITS HAVING BUILT-IN TEST HARDWARE†

M. S. ABADIR‡ and M. A. BREUER
Department of Electrical Engineering, University of Southern California, Los Angeles, CA 90089-0781, U.S.A.

Abstract—Numerous built-in test techniques exist for testing structures within a VLSI chip. In general these techniques deal with a repeated application of the following steps: (1) generate a test vector; (2) transmit it to the structure being tested; (3) process the test through the structure; (4) obtain the response from the structure; and (5) process the response. These steps constitute a test schema. Because these steps must be repeated for each test vector, it is possible that steps in processing one test vector can overlap those used in processing another vector. The manner of overlapping this testing process leads to the concept of a test schedule. In this paper we first present a model for built-in test techniques and for describing test schemas and schedules. We introduce the new concept of an I-path which is used to transfer data from one place in a circuit to another, without modifying the data. Finally results are presented describing how to create test schedules that minimizes the total testing time. Lower bounds on the minimal test time are also derived.

1. INTRODUCTION

The emergence of VLSI has led to significant problems of chip and system testing. One approach to addressing this problem is to adopt *testable design methodologies* (TDMs) [1, 2, 3] during the early stages of chip design. A TDM deals with the complete process of (1) designing an easily testable structure, and (2) testing the structure using external and/or built-in test hardware. Examples of well known TDMs are scan-path, LSSD, scan/set, BILBO, syndrome testing, and autonomous testing [4]. Over 20 TDMs exist for PLAs alone [5].

A *test schema* specifies how a test methodology is to execute. The main aspects specified by a test schema are: (1) the generation of a test vector; (2) the transfer of the test vectors to the circuit structure, referred to as the kernel, to be tested; (3) the propagation of the test vectors through that structure; (4) the transfer of response data to some response analysis circuit; and (5) the processing of the response data.

When a test methodology is employed in an actual circuit, many transformations may occur to a test schema. For example, there may be many ways to transfer the test vectors from the vector generator to the kernel and from the kernel to the response evaluator. These transfer paths are called identity paths, or *I-paths*, since the data does not get transformed. The result of this mapping or embedding of a test methodology into an actual circuit produces what we call a *test plan* for the circuit. A test plan specifies how the test methodology is to be executed in the given circuit.

In most cases the processing of test vectors can be pipelined through a circuit, thus reducing the total test time. The manner of overlapping the processing of test vectors leads to the concept of a *test schedule*. Clearly, it is of interest to construct test schedules that optimizes total test time.

This paper deals with the following subjects: (1) modeling TDMs and their test schemas; (2) embedding a TDM into a circuit containing a structure to be tested and generating an appropriate test plan; and (3) creating a schedule for the given test plan which minimizes the total time required to test the structure. The later concept is similar to that used in optimizing static pipelines [6]. Yet, there are key differences between the two problems, and hence our approach and results are tailored to the problem in hand. Those differences will be discussed at the end of the paper.

†This work was supported in part by the Defense Advanced Research Projects Agency (DoD) under Contract No. N00014-84-K-0649.
‡Now with the Microelectronics and Computer Technology Corporation, Austin, Tex., U.S.A.

2. TESTABLE DESIGN METHODOLOGIES

In this section we will introduce a framework for a methodology that incorporates structural, behavioral, quantitative, and qualitative aspects of known design-for-test techniques, and can support a systematic design for testability synthesis process [1, 2].

A testable design methodology (TDM) deals with the complete process of designing and testing an easily testable circuit structure. There are three major components that form a TDM. The first deals with the structural aspect of the methodology and how the structures involved are interconnected. The second deals with the operational aspect of the TDM. The last component deals with measures and criteria that reflect the various costs associated with the TDM and its merits. In the next three subsections we will discuss the TDM components in more detail.

2.1. The TDM structural template

The *template* of a TDM describes its structural architecture. It conveys information about the type, design style, and size of a circuit structure to which the TDM is applicable. This structure is referred to as the *kernel* of the TDM. The template of a TDM also describes the built-in-test (BIT) structures needed by the methodology and the connection paths that must exist between them and the kernel. These BIT structures are employed by a TDM to carry out one or more of the following tasks: (1) generate the test stimuli for the kernel; (2) process the test responses; (3) gain access to the kernel input and/or output ports; (4) control the testing process; (5) modify or add features to the kernel to make it easily testable.

As an example, consider the structural template of the BILBO TDM [7] shown in Fig. 1. The labels associated with the kernel indicate that the BILBO TDM is applicable to PLAs or random combinational logic. The template also indicates that the BILBO TDM employs two linear feedback shift registers B1 and B2, the first acts as a pseudo random number generator and the second as a signature analyzer.

It it important to note that an arc between two structures in a TDM template has a broader meaning than just a wire or a set of wires. It represents a data transfer path between the two structures. Such a path can be as simple as a wire, or a complex path through a number of busses, registers and MUXs. This simple yet powerful concept represents an important departure from previous descriptions of TDMs.

2.2. The TDM test schema

A test schema describes how a test methodology is to execute. In general, a test schema consists of three sections: a head, a body, and a tail. The body of a test schema describes the on-chip actions and constitutes the life cycle of a single test vector from the time it is generated until its effects

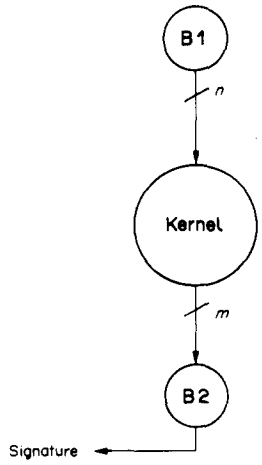

Fig. 1. The BILBO structural template. Register B1—Clear; pseudo random number generator (RNG). Kernel—Combinational structure; n inputs and m outputs; design style: PLA or random logic. Register B2—Clear; signature analyzer (SA).

on the kernel have been captured and processed. The main aspects specified in the body of a test schema have been previously defined. In addition to the actions in the body of a schema which must be executed once for every test vector, a test schema often has a head (tail) section in which initialization (closing) actions are specified.

There are two kinds of actions within a test schema: data transfer actions and data processing action. A data transfer action calls for transferring data (without modifying it) between two structures. The format of a data transfer action is as follows:

Transfer (*source* → *destination*)

A data processing action calls for propagating data through a structure. If that structure has different modes of operation, one of these modes is specified in the action. The format of a data processing action is as follows:

Structure (*mode of operation*)

As an example consider the BILBO test schema shown below:

```
HEAD
  B1 (clear)
  B2 (clear)
BODY
  Execute T times:
    B1 (random number generation)
    Transfer (B1 output→kernel input)
    Kernel (–)
    Transfer (kernel output→B2 input)
    B2 (signature analysis)
TAIL
  Transfer (B2 output→scan-out output pin)
```

The body of the schema has to be executed T times, where T is the number of test vectors to be used. First, the input BILBO register, B1, is clocked while in the random number generation (RNG) mode to produce a new test vector. Next the output of B1 is transferred to the kernel inputs, the test vector then propagates through the kernel logic, and finally the response is transferred to the second BILBO register, B2, which is clocked while in the signature analysis (SA) mode. The head and tail sections are both self explanatory.

2.3. The TDM measures

The TDM measures reflects the various cost of implementing the methodology as well as its merits. Examples of these measures are area overhead, test time, effects on circuit speed, number of extra I/O pins required, and fault coverage. These measures can be used to evaluate and compare different methodologies.

The values of many of these measures are often functions of the circuit to be made testable. For example, the number of random test vectors T required by the BILBO TDM to get 95% single stuck-at fault coverage is a function of the kernel. Moreover there are usually several ways of implementing a TDM, each with a different set of values of the measures. The latter concept is discussed further in the next section.

In this paper we will focus on techniques for calculating and optimizing the value of one of the key measures associated with a TDM, namely the test time. For further details on the different kinds of measures associated with TDMs and how they can be computed and used in a system that assists designers in producing testable VLSI chips refer to [2].

3. EMBEDDING TDMs INTO VLSI CIRCUITS

Let us assume that we are interested in one of the TDMs to a given structure within a circuit under consideration (C). We will refer to this structure as the kernel. The circuit C is assumed to be complex, consisting of registers, RAMs, ROMs, PLAs, busses, and random combinational

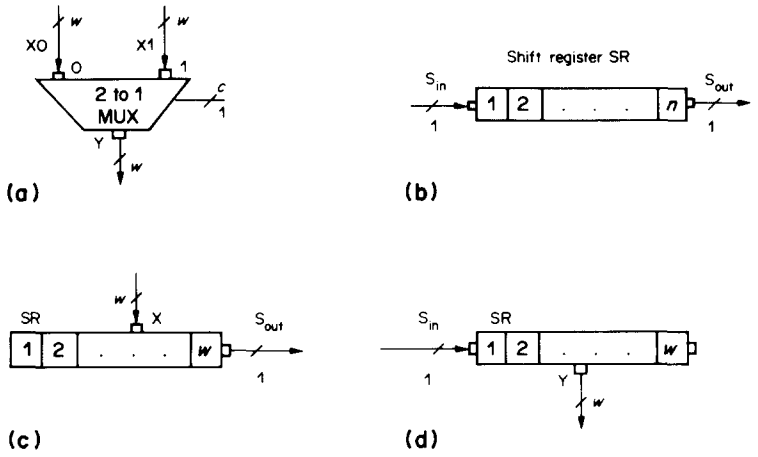

Fig. 2. Examples of structures with various I-modes. (a) *P/P I-mode* M(MUX: X0 → Y)—*activation plan*: MUX(select 0); *P/P I-mode* M(MUX: X1 → Y)—*activation plan*: MUX(select 1). (b) *S/S I-mode* M(SR: $S_{in} \rightarrow S_{out}$)—*activation plan*: SR(Shift), . . . , SR(Shift) → (*n* times). (c) *P/S I-mode* M(SR: X → S_{out})—*activation plan*: SR(Latch) →, SR(Shift) →, . . . , SR(Shift) → (*w* − 1 times). (d) *S/P I-mode* M(SR: S_{in} → Y)—*activation plan*: SR(Shift), . . . , SR(Shift) (*w* times).

logic. Often, structures in the neighborhood of a kernel can be used—possible after simple modifications—to aid in the testing of the kernel. For example, a register connected to the kernel inputs can be used to gain access to the kernel, or it can be modified to become a BILBO register which can be used to supply test patterns to the kernel. The same statement still applies if the register is connected to the kernel input through a multiplexer. This concept of passing data unchanged through circuit structures is very important and plays a major role in the process of designing testable circuits. In the next subsection this concept will be formally defined.

3.1. Data transfer paths

Definition. A structure S with an input port X and an output port Y is said to have an *identity mode* (*I-mode*), denoted by M(S: X → Y), if the data on port X can be transferred, possibly after clocking one or more times, to port Y. A time tag t and an activation plan p are associated with every I-mode, where t is the time, in clock cycles and gate delays, for the data to be transferred from X to Y, and p indicates the sequence of data processing actions that have to be carried out by structure S in order to activate the mode. The format for these data processing actions is the same as the one used in Section 2.2 for describing test schemas.

There are 4 different types of I-modes depending on the size of ports involved and the scheme used for transferring the data. Let w denote the width of the data being transferred, measured in units of bits, and let x and y denote the width of ports X and Y, respectively.

(1) *Parallel to parallel I-mode* (P/P I-mode): In this mode $x = y = w$, and the data at X is transferred in parallel to Y. An example of a structure with parallel I-modes is a 2-to-1 multiplexer as shown in Fig. 2a.

(2) *Serial to serial I-mode* (S/S I-mode): In this mode $x = y = 1$, and the w data bits are transferred serially from X to Y. The time tag t associated with an S/S I-mode indicates the time for one of the data bits to travel from X to Y. A shift register with an S/S I-mode is shown in Fig. 2b.

(3). *Parallel to serial I-mode* (P/S I-mode): In this mode $x = w$ and y = 1. Only structures with internal memory elements, such as registers, can have such a mode. The data is first entered in parallel via X and latched, than it is transferred serially to Y. The time tag of a P/S I-mode indicates the time for the last data bit to appear at Y. A register with a P/S I-mode is shown in Fig. 2c. The arrows (→) in the activation plan of the I-mode indicate the appearance of the data bits on the serial output port.

(4) *Serial to parallel I-mode* (S/P I-mode): In this mode $x = 1$ and $y = w$. Again only structures with internal memory elements, such as registers, can have such a mode. The data is entered serially via X and held within the structure, then it is made available in parallel on port Y. A register with an S/P I-mode is shown in Fig. 2d.

Definition. There exists an *identity transfer path* (*I-path*) from output port X of structure S1 to input port Y of structure S2, denoted by P(S1:X→S2:Y) if the data at port X can be transferred unchanged to port Y. Every I-path has a time tag and an activation plan. The time tag specifies the time, in clock cycles, for the data to be transferred from X and Y, while the activation plan indicates the sequence of actions which must take place to establish the I-path. Again the format used for describing activation plans is the same as the one used for describing data processing actions of TDM test schemas.

Similar to the way we classified I-modes, there are also 4 types of I-paths depending on the width of the ports involved and the scheme of transferring the data. Let w denote the width of the data being transferred, and let x and y denote the width of ports X and Y, respectively.

(1) *Parallel to parallel I-path* (P/P I-path): $x = y = w$.
(2) *Serial to serial I-path* (S/S I-path): $x = y = 1$.
(3) *Parallel to serial I-path* (P/S I-path): $x = w$ and $y = 1$.
(4) *Serial to parallel I-path* (S/P I-path): $x = 1$ and $y = w$.

Let W(S1:X→S2:Y) denote a physical connection between output port X of structure S1 and input port Y of structure S2. Clearly, X and Y should have the same width.

We formally define an α/β I-path, where α and $\beta \in \{P, S\}$, as follows.

Definition. There exist and α/β I-path P(S1:X→S2:Y), where α and $\beta \in \{P, S\}$ if either (1) $\alpha = \beta$ and W(S1:X→S2:Y), or (2) W(S1:X→S3:Z), α/γ I-mode M(S3:Z→Q) where $\gamma \in \{P, S\}$, and γ/β I-path P(S3:Q→S2:Y).

Figure 3 illustrates the above definition.

The activation plan of an I-path can be formed by concatenating the activation plans of all the I-modes encountered along the path.† As an example of I-paths, consider the circuit of Fig. 4. There is a P/P I-path between port X of K and port Y of R2. There is also an S/S I-path P(R2:Z→R3:Q). Linking these two I-paths with the P/S I-mode of R2 produces a P/S I-path P(K:X→R3:Q) whose activation plan is given in Fig. 4. The symbol →R3:Q indicates the arrival of the data bits at the serial port of R3. R3 might be a signature analyzer that is processing the test responses of K.

The I-mode concept can be generalized to incorporate *transformation modes* (*T-modes*) in which there is a one-to-one correspondence between the inputs and the outputs of a structure. The

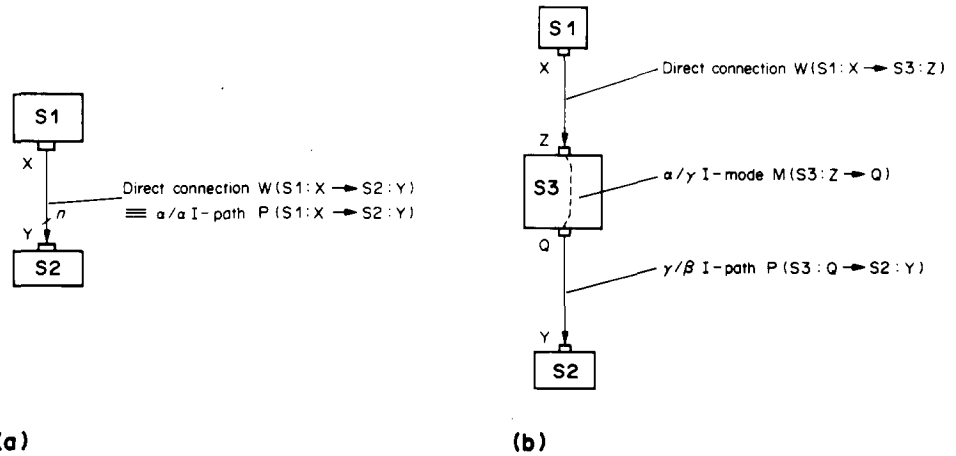

Fig. 3. α/β I-path P(S1:X→S2:Y). (a) The trivial case; (b) the recursive case.

†We assume that an I-path exists only if its activation plan is feasible, that is the plan does not call for actions in the same clock cycle that require conflicting control signals.

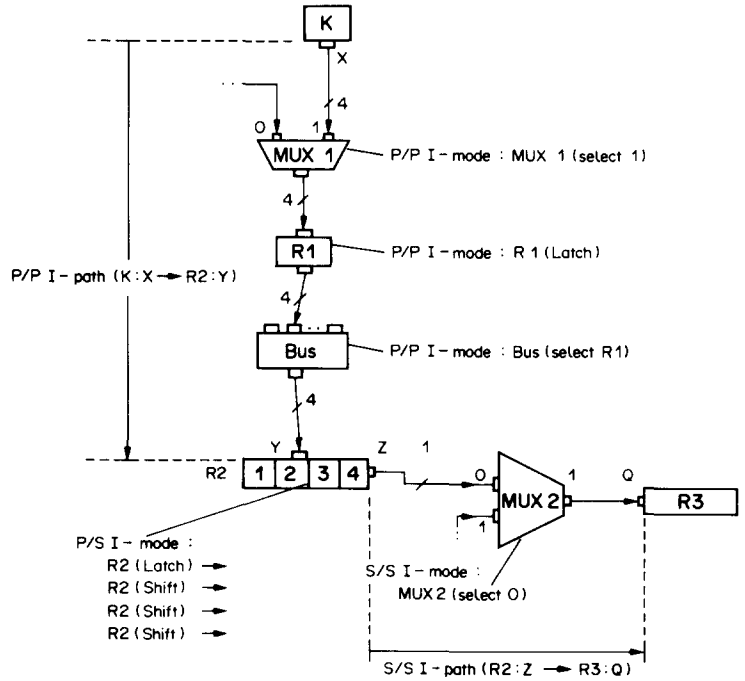

Fig. 4. A P/S I-path P(K:X → R3:Q). The activation plan of P(K:X → R3:Q) is as follows:

MUX1(select 1), R1(Latch).
Bus(select R1), R2 (Latch).
R2(Shift), MUX2 (select 0), → R3:Q.
R2(Shift), MUX2 (select 0), → R3:Q.
R2(Shift), MUX2 (select 0), → R3:Q.
MUX2(select 0), → R3:Q.

simplest structure with a T-mode is an inverter. Having an inverter in the path between a BILBO register and a kernel does not violate the applicability of the BILBO TDM. The same argument is true for scan path TDMs, were a complemented version of the test vectors can be scanned in. It is interesting to note that paths which incorporate I and T-modes can be regarded as generalized forms of sensitization paths. In the rest of our discussion we restrict our attention to I-modes and I-paths.

3.3. The embedding process

The process of applying a given TDM to a kernel in a circuit C is equivalent to that of embedding the TDM structural template into C. Usually this embedding process requires modifying and/or adding structures to C. Clearly, there might be more than one way of embedding a TDM into C. A feasible embedding should, by definition, satisfy the following two conditions:
(1) for every structure in the TDM template there is a matching structure in C; and (2) for every connection between structures in the TDM template there is an I-path between the corresponding circuit structures.

The test schema of a TDM can be customized to a given embedding producing a *test plan* for the kernel under consideration. The process of generating a test plan from a test schema can be simply described as follows: (1) replace the names of the TDM template structures with the corresponding circuit structures; (2) replace every data transfer action in the schema with the activation plan of the corresponding I-path.

Clearly, a test plan has the same general form as a test schema, i.e. it has a head, a tail, and a body. However, the actions in a test plan are all data processing actions since the data transfer actions are replaced by activation plans that in turn consists of data processing actions (structures operating in their I-modes). The actions that form the body of a test plan can be organized in a natural way into a number of steps, such that the actions in one step can all be executed in one clock period. The steps are numbered 1, 2, . . . , S where S is the total number of steps.

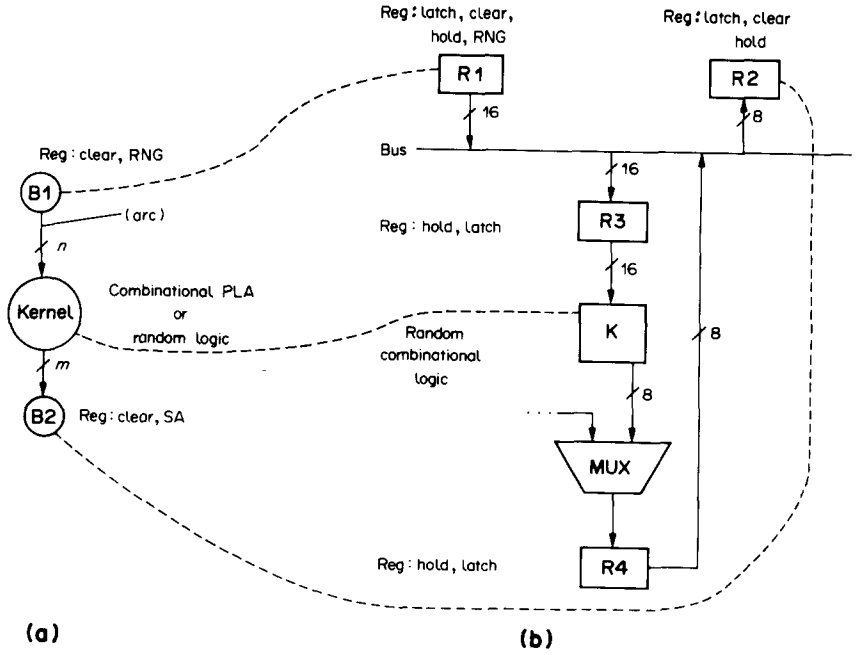

Fig. 5. BILBO structural template (a) and a circuit (b). Matching results are as follows:

This concept	*maps into*	*this real object*
Kernel	⇒	K
n	⇒	16
m	⇒	8
B1	⇒	R1
B2	⇒	R2 Modified (+SA)
P(B1 → Kernel)	⇒	P(R1 → K) ≡ Bus(select R1), R3(Latch).
P(Kernel → B2)	⇒	P(K → R2) ≡ MUX(select K), R4(Latch). Bus(select R4).

A test plan for K is generated as follows:

(1) R1(RNG).
(2) Bus(select R1), R3(Latch).
(3) K(–), MUX(select K), R4(Latch).
(4) Bus(select R4), R2(SA).

} execute *T* times

We assume that we are dealing with synchronous circuits, and that registers are triggered by the trailing edge of every clock pulse. We also assume that all registers are in a hold state unless otherwise specified in the test plan.

Example 1 illustrates the embedding process and how to generate a test plan for a circuit kernel.

Example 1. Assume we want to apply the BILBO TDM that has the structural template as shown in Fig. 5a to structure K of the circuit shown in Fig. 5b. The labels associated with the structures in both figures indicate their key attributes and modes of operation.

The first step is to match K with the TDM kernel to make sure that the methodology is applicable. The next step is to find a match for B1. Any register in the circuit with a P/P I-path to the kernel is a candidate. R1, R3, and R4 qualify, but only R1 has all the functional capabilities required of B1. Three registers R2, R3 and R4 are candidates for matching B2. Both R3 and R4 lack the SA and clear modes, while R2 is only missing the SA mode. To minimize area overhead, assume R2 is selected. The matching results are shown in the caption to Fig. 5. Note that the arcs of the TDM template are mapped into I-paths in the circuit. The activation plans for the two I-paths are also shown in the caption to Fig. 5. By doing the necessary substitution in the BILBO test schema, a test plan for K is generated as described in Fig. 5. The period (.) between actions signals the end of one clock period.

4. EXPLORING PARALLELISM IN TEST PLANS

Let T be the number of test patterns to be used. To test the kernel, the S steps of the test plan body have to be executed (iterated) T times, and in addition the actions in the head and tail section have to be executed once. These initialization and closing actions play a very minor role in determining the total execution time of a test plan, and hence they will be ignored. Unless explicitly stated otherwise, in the rest of our discussion we will use the term test plan to refer to the body of a test plan.

The simplest way to execute a test plan is in a sequential manner, hence requiring $S \times T$ clock cycles. On the other hand, it is often possible to overlap the execution of consecutive iterations, hence reducing the overall test time. In other words, it might be possible to start the execution of one iteration before the end of the previous one.

Let D (stands for *initiation delay*) be the number of clock periods between the initiation of two consecutive iterations. $D = 1$ indicates maximum overlapping, while $D = S$ means sequential execution. Figure 6 displays all possible *executions schedules* for a *generic* five steps test plan. The numbers in the leftmost column represent the time in clock cycles, while the other numbers represent the steps. The third line of Fig. 6a indicates that during the 3rd clock cycle, step 3 of the 1st iteration, step 2 of the 2nd iteration, and step 1 of the 3rd iteration will be executing in parallel.

4.1. *Cyclic behavior of the execution schedules of test plans*

By analyzing execution schedules of test plans with different values of D, one can observe the following cyclic behavior of test plans. Assuming T is greater than S, after the first $\lfloor S/D \rfloor \times D$ clock periods every schedule starts to *cycle* with a period equal to D. We will refer to the D time slots of one cycle as the D *phases* of the schedule. For example, consider the execution schedule shown in Fig. 7. The schedule has a cycle with two phases.

Note that every step of the plan is executed once every D clock period. Thus, one can consider D as being the *effective length* of the test plan.

Let $t_i(c)$ indicate the execution time of step i of iteration c, for $c = 1, 2, \ldots, T$. Assume $t_i(1) = i$, for $i = 1, 2, \ldots, S$. Hence $t_i(c) = i + (c - 1) \times D$, and the *total test time* is $t_S(T) = S + (T - 1) \times D$.

Since T is usually large compared to S, the total test time is much more sensitive to D than to S. Clearly, it might not be possible to realize an execution schedule with an arbitrary value of D due to resource sharing conflicts. Thus, to explore the feasibility of these schedules, it is necessary to identify those steps in the test plan that cannot run in parallel either because they

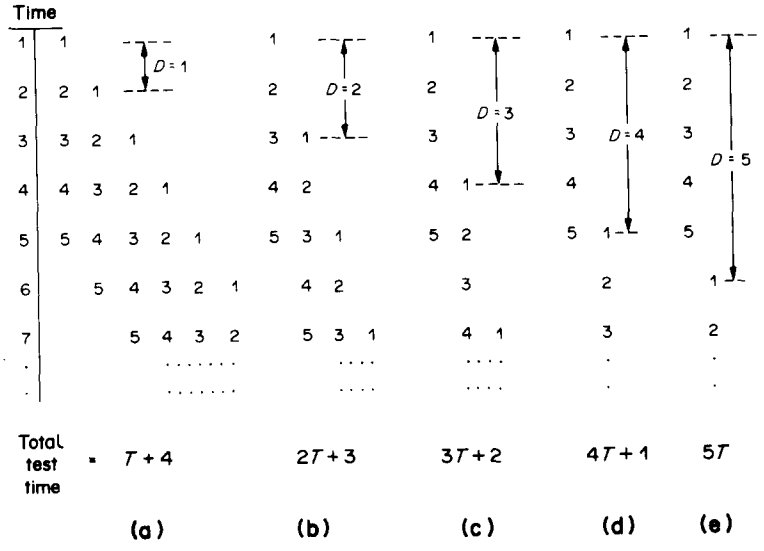

Fig. 6. Various execution schedules for a generic 5 step test plan.

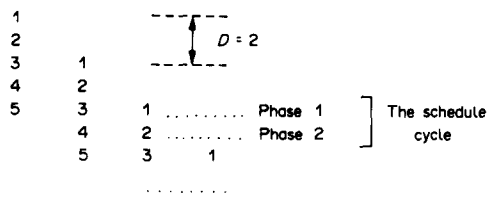

Fig. 7. An execution schedule with 2-phase schedule cycle.

use the same hardware or they require conflicting control signals. For instance, in the test plan of Fig. 5 both steps 2 and 4 utilize the bus, hence they cannot be executed concurrently. This eliminates the applicability of the schedules with $D = 1$ or $D = 2$.

4.2. *The conflict graph*

Clearly, knowledge about conflicts that exist between the steps of a test plan is essential in order to analyze the potential of different execution schedules. One way of representing such knowledge is to construct a conflict graph (CG).

The CG of a test plan consists of S nodes, one for every step in the plan. An edge between two nodes i and j in the CG, denoted by $c(i,j)$, exists if and only if steps i and j of the plan utilize a common resource, or they call for actions that require conflicting control signals. For example, the CG of the test plan of Fig. 5 has an edge $c(2,4)$ since steps 2 and 4 utilize the bus.

If a structure in the circuit is used in q steps of a test plan, then the CG of the test plan has a clique of size q between the nodes corresponding to those q steps. A simple mechanism to generate the CG of a test plan is to form a clique for every structure used in at least two steps of the plan. Now by superimposing all these cliques we will get the CG. Note that if a structure is only used once in a test plan, its corresponding clique is of size 1, which is a singular node.

4.3. *Finding feasible values for* D

Clearly it is of interest to find the minimum value of D that leads to a feasible execution schedule for a given test plan. The following lemma provides a simple rule that can quickly be used to solve this problem:

LEMMA 1

An execution schedule of a test plan with an initiation delay D is feasible if D does not divide $(j - i)$ for all $c(i,j)$ in the CG of the test plan, where $j > i$.

Proof. By contradiction. Assume an execution schedule with an initiation delay D is feasible, such that D divides $(j - i)$, where $c(i,j)$ is the CG of the test plan and $j > i$. Hence, $(j - i)/D = k$ where k is an integer. Thus, $j = i + D \times k$. Recall that the execution time of steps s of iteration c is $t_s(c) = s + (c - 1) \times D$. Hence, $t_j(1) = t_i(k + 1)$ which means that step j of iteration 1 is scheduled in parallel with step i of iteration $k + 1$. Thus, the schedule is infeasible which leads to a contradiction. Q.E.D.

COROLLARY

$D = 1$ is void if any conflict exists since 1 divides all integers. Moreover, $D = S$ is always feasible as S doesn't divide $(j - i)$ for all possible steps i and j $(1 \leqslant i, j \leqslant S)$.

Example 2. Consider a test plan whose CG is shown in Fig. 8. Both $D = 2$ and $D = 3$ are infeasible due to $c(6,4)$ and $c(4,1)$, respectively. Also both $D = 5$ and $D = 6$ are infeasible due to $c(7,2)$ and $c(8,2)$, respectively. $D = 4$, $D = 7$ or $D = 8$ are feasible.

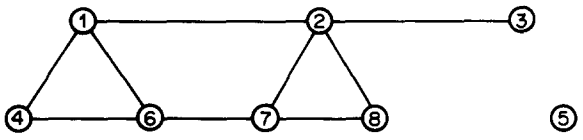

Fig. 8. The CG for example 2.

5. LOWER BOUNDS ON D

THEOREM 1

The size of the largest clique in the CG of a test plan is a lower bound on any feasible value for D.

Proof. Let Q be the size of the largest clique in the CG, and let the steps that form that clique be denoted by $x_1, x_2, \ldots, x_Q$, where $x_1 < x_2 < \ldots < x_Q$. Assume that an execution schedule with an initiation delay $D \leqslant Q - 1$ is feasible. Using the rule derived earlier for selecting feasible values for D, this implies that D does not divide $x_i - x_j$, for all $i, j = 1,2, \ldots, Q$ and $i > j$. Hence, $(x_i)_{\text{mod } D} \neq (x_j)_{\text{mod } D}$, for all $i, j = 1, 2, \ldots, Q$ and $i > j$. But since there are only D possible values for $x_{\text{mod } D}$, the above inequality cannot hold for Q different steps. Hence a contradiction. Q.E.D.

Even though finding the largest clique in a graph is a known NP-complete problem [8], the situation here is relatively simple. This is because of the one to one correspondence that exists between circuit structures and cliques in the CG. For example if register A is involved in actions in steps $x_1, x_2, \ldots, x_Q$ of the test plan, then there must be a clique in the CG between those steps. Thus, by identifying the structure used most often in a test plan, we can obtain a close estimate of the size of the largest clique in the CG. Clearly, this is a very fast way for obtaining a close lower bound on the optimal value of D. To find the largest clique, one has to check if the cliques that correspond to circuit structures combine producing larger cliques. Note that $Q + 1$ cliques of size Q are needed to form a clique of size $Q + 1$.

THEOREM 2

The chromatic number of a CG is a lower bound on the value of D of any feasible test plan execution schedule.

Proof. By contradiction. Assume the chromatic number of the CG is X. Now assume that an execution schedule with D less than X is feasible. This implies the existence of a schedule cycle with D phases, such that steps in the same phase are not connected in the CG. Hence the steps in each phase can be colored with one color. Thus, D colors are needed to color the CG, which leads to a contradiction. Q.E.D.

The lower bound of Theorem 2 is stronger that the one of Theorem 1 [8]. Coloring the CG with X colors is equivalent to partitioning the steps of a plan such that the members of any partition do not conflict. The existence of such an X-way partition does not necessary imply the existence of a feasible execution schedule with $D = X$. For example, in order for a $D = 2$ schedule to be feasible for any test plan, all the old steps should be executed in one phase, and all the even steps in the other phase. Thus, even though there might be many feasible 2-way partitions, only one leads to a feasable schedule without changing the structure of the original test plan. In the following section we will describe a technique to construct schedules that meet the lower bound of Theorem 2.

6. INSERTING No-Op STEPS

In Section 4.3 we presented a lemma for checking the feasibility of different values of D. This, in general, leads to a minimal value of D which is far from the theoretical lower bounds. It will now be shown that the gap between the feasible value of D and the theoretical lower bounds can be eliminated by introducing No-Op steps in the plan body.

Definition. A *No-Op* step inserted after step i of a test plan consists of actions that call for holding the state of all the register(s) used in step i for one clock period.

The next example will help illustrate this idea.

Example 3. Consider the test plan of Example 1. The smallest feasible value for D was 3, leading to a total test time of $3T + 1$. By introducing a No-Op step after step 2 that calls for holding the state of R3, the conflicting steps are now separated by 3 clock periods. Hence, $D = 2$ is possible, as illustrated in Fig. 9, leading to a total test time of $2T + 3$. Note that the new value of D equals both theoretical lower bounds, and hence is optimal.

Before investigating the amount of improvement that can be reached using No-Op steps, it should be noted that a No-Op step may conflict with other steps of a test plan. Such a case is rare and will be addressed in the next section.

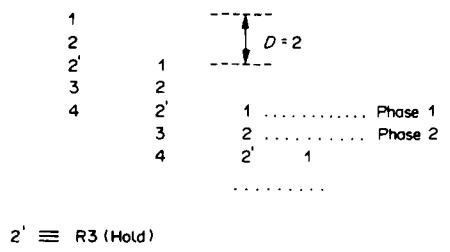

Fig. 9. An optimal execution schedule ($D = 2$) obtained using No-Ops.

THEOREM 3

Assuming that No-Ops do not conflict with other steps of a test plan, by adding No-Op steps it is always possible to find a schedule with D equal to the chromatic number of the CG.

Proof. Let X be the chromatic number of the CG associated with a test plan. It follows that one can partition the steps of the plan into X groups, $G_1, G_2, \ldots, G_X$, such that the steps in one group do not conflict with each other. It is always possible to insert No-Op steps in the test plan such that if steps i and j belong to the same group, then the difference between the execution times of i and j in one iteration is X or a multiple of X. Now by using an execution schedule with $D = X$, a one-to-one correspondence between the Gs and the schedule phases is created, i.e. the steps that belong to one group are all scheduled to execute in the same phase. This in turn implies that no conflicts will arise, and hence the execution schedule of the new test plan with D equal to X is feasible. Q.E.D.

Example 4. Consider the 8-step test plan of Example 2. Without adding No-Ops, $D = 4$ was the best we could achieve. However, the chromatic number of the CG is 3. Assume that the steps are partitioned as follows: $G_1 = \{1, 3, 7\}$, $G_2 = \{2, 4, 5\}$, and $G_3 = \{6, 8\}$. Following the argument used in the proof of Theorem 3, an optimal execution schedule with $D = 3$ is constructed as shown below. Four No-Op steps were added to the test plan. One can see the one-to-one correspondence between the phases of the schedule cycle and the partitions. Given the schedule cycle, one can find the test plan by traversing the columns of the cycle from right to left.

Phase 1:	7	No-Op	3	1
Phase 2:	No-Op	5	4	2
Phase 3:	8	6	No-Op	No-Op

The number of No-Op steps added depends on the distribution of the steps in the partitions supplied as input. Clearly these partitions are not unique, and some will result in adding more No-Ops than others. For instance, in the previous example a test plan with only two No-Op steps added between steps 3 and 4 also supports a schedule with $D = 3$. However, every extra No-Op step added to the test plan increases the total test time by only one clock period. Such a small increase can usually be ignored when compared to the reduction due to the use of an optimal value for D.

The problem of coloring a graph (to partition the nodes of the CG) is in general an NP-complete problem. [8] The following procedure avoids this expensive preprocessing step, and only assumes knowledge of a lower bound for D, denoted by B, such as the size of a large clique in the CG.

Procedure 1 (CG, B)

This procedure will attempt to construct an execution schedule with B phases. The procedure has a loop that cycles through the B phases. In each iteration either a step is added to the current phase if it does not create conflicts, or a No-Op is added. Adding B No-Ops in a row implies failure, and the procedure restarts using a larger value for B. Procedure 1 is a fast heuristic which in most cases generates optimal schedules.†

†The probability of not finding an optimal solution, when one exists, can be reduced significantly by employing a simple trick. Whenever the procedure fails to schedule the steps in B phases, it will try scheduling once more, but this time processing the steps in the reverse order. Only when that too fails, is the number of phases incremented.

$s = 1$, $p = 0$, No-Op-counter = 0, Test-Plan = empty list.

REPEAT WHILE $s \leqslant S$
  $p = p_{\text{mod } B} + 1$
  IF No-Op-counter = B
    THEN increment B by 1 and RESTART the procedure.
  IF $c(s, x)$ exists in the CG, where $x \in$ Phase p
    THEN append a No-Op to Test-Plan and to Phase p,
      increment No-Op-counter by 1.
    ELSE append step s to Test-Plan and to Phase p,
      $s = s + 1$,
      No-Op-counter = 0.
END REPEAT

Example 5. Consider the 8-step test plan of Example 4. Let us assume that procedure 1 starts with $B = 3$. It will succeed in constructing an optimal schedule with 3 phases as shown below.

Phase 1:	7	No-Op	No-Op	1
Phase 2:	No-Op	No-Op	4	2
Phase 3:	8	6	5	3

7. CONFLICTS CAUSED BY No-Ops

A No-Op step inserted in a test plan calls for holding the state of one or more registers one clock period. Such a hold mode can create a conflict with a step that calls for changing the state of a register. To illustrate this concept, consider the following example.

Example 6. Consider the circuit shown in Fig. 10a. Its BILBO test plan is given in the caption and its CG is shown in Fig. 10b.

Without adding No-Op steps, $D = 2$ is not feasible due to the conflict between steps 2 and 4. Clearly, both steps write data into R2. If a No-Op step is inserted after step 2, the new test plan cannot support $D = 2$ because step 4 will be scheduled in parallel with the No-Op step causing a conflict as shown below.

Phase 1:	4	No-Op	1 ⟵ Conflict
Phase 2:	5	3	2

That is, step 4 will latch data in R2 which is supposed to hold its state due to the No-Op.

7.1. *Modifying the conflict graph*

Clearly, the conflict graph has to be modified to reflect the extra conflicts caused by the No-Ops. For every conflict in the CG involving two steps i and j that write data into the same register we add two nodes to the CG, labeled i' and j', where node $i'(j')$ represents a No-Op following $i(j)$.

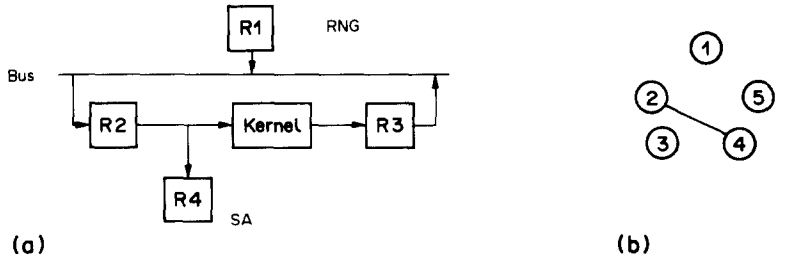

Fig. 10. An example with a potential for conflicting No-Ops. The BILBO test plan is as follows:
(1) R1(RNG).
(2) Bus(select R1), R2(Latch).
(3) Kernel(–), R3(Latch).
(4) Bus(select R3), R2(Latch).
(5) R4(SA).

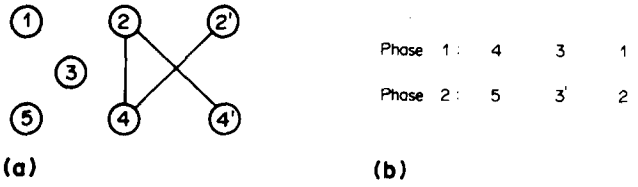

Fig. 11. (a) The extended CG; (b) a schedule with $D = 2$.

Then we connect node i' to node j, and node j' to node i. The resulting graph is called the extended CG.

A modified version of Procedure 1 can be used to generate solutions for cases with conflicting No-Ops. Before adding a No-Op to one phase, the procedure refers to the extended CG to check if that No-Op conflicts with any of the steps already scheduled in that phase. If it does, then the procedure backs up and restarts with more phases.

Example 7. Again consider the circuit of Example 6. The extended CG is shown in Fig. 11a. Using the modified version of Procedure 1, a new test plan has been found supporting $D = 2$ that involves adding a No-Op after step 3, as shown in Fig. 11b. Note that since a No-Op after step 3 does not conflict with other steps, no node labeled 3′ can be found in the extended CG.

Theorem 3 does not hold for test plans where adding No-Ops may create conflicts. However, such cases arise when a register is used in the I-path that transfers tests to the kernel, and also in the I-path that transfers responses of the kernel. This case is unusual. Thus it is fair to claim that for all practical tests plans one can always reach the lower bound given by Theorem 2.

8. TEST PLANS FOR SCAN-TYPE TDMs

Test schemas for scan-type TDMs (like LSSD and scan path) have unique characteristics. The serial shifting of data in scan-type TDMs limits the amount of possible overlapping.

Consider the template of the scan-path TDM shown in Fig. 12. An S/P I-path is required between the scan-in pin of the circuit and the input port of the kernel. This S/P I-path consists of an S/S I-path from the scan-in pin to a shift register that has an S/P I-mode, and then a P/P I-path from that shift register to the kernel input port. Another P/S I-path is also required between the kernel output port and the scan-out pin of the circuit as shown in Fig. 12.

Assume that the kernel has n inputs and m outputs. Any scan-type test plan for that kernel will contain n steps in which the input shift register is serially loaded with a test vector. Hence, there will always be a clique of size n in the CG of the test plan. Similarly, there will also be another clique of size m corresponding to the activation plan of the P/S I-mode of the output shift register.

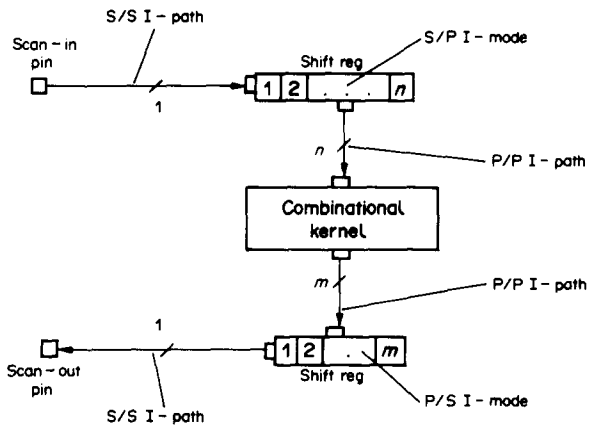

Fig. 12. The structural template of the scan/path TDM.

Theorem 1 implies that $X = \max(n,m)$ is lower bound on the value of D of any feasible execution schedule.

In most practical scan-type TDM embeddings the number of steps in the test plan corresponding to the parallel I-paths between the shift registers and the kernel is less than X. In all such cases it is easy to show that it is always possible to construct an execution schedule with $D = X$. Moreover, even if the extra steps due to the parallel I-paths is larger than X, but no cliques of size X or larger are generated by these steps (it is very unlikely otherwise), then again a schedule with $D = X$ is always feasible. Hence, it is fair to claim that for scan-type tests plans, test time is approx. $T \times X$, where T is the number of test vectors in the test set.

9. TEST PLANS FOR KERNELS WITH MULTIPLE PORTS

In Section 3 we described how to generate a test plan for a kernel with a single input and output port. In this section we will consider the general case of multiple I/O ports. Consider a kernel with n inputs ports and m output ports. There are $n(m)$ I-paths that drive data into (from) the kernel. Each one of these I-paths is completely specified by an activation plan. Some of these I-paths may share resources. Our goal is to generate a test plan for the circuit that will exercise those I-paths in a correct fashion without creating conflicts.

Next, we will describe a procedure that can be used to combine the activation plans of n I-paths into a single plan.

Problem statement

Given the activation plans of n I-paths $P_1, P_2, \ldots, P_n$, generate a feasible plan to activate all the I-paths without causing any conflicts. We will refer to step s of P_i by $i \bullet s$. A conflict graph CG is constructed to reflect resource sharing conflicts between steps of different I-paths. Thus an edge exists between $i \bullet s_1$ and $j \bullet s_2$ if the two steps utilize the same resource. Note that we are only interested in conflicts that exist between steps of different plans. In case conflicts occur due to sharing of registers, then as was done in a prevous section, nodes representing No-Ops are added to the CG. A No-Op step after step $i \bullet s$ is denoted by $i \bullet s'$, while a No-Op that precedes $i \bullet 1$ is denoted by $i \bullet O'$.

Procedure: *Combine-I-paths* $(P_1, P_2, \ldots, P_n)$

The procedure employs two dynamic lists. The first one, called Frontier-Steps(t), contains all the steps that are candidates for scheduling during time t. Every I-path is represented by two steps in this list, the one on the frontier of its activation plan (i.e. the step to be executed next), and a No-Op in case the first step cannot be scheduled due to conflicts.

Frontier-Steps(t): $\{i \bullet f_i, i \bullet f_i - 1'$, for $i = 1, 2, \ldots, n \mid f_i$ is the frontier step of P_i at time t, while $f_i - 1'$ is the No-Op step that precedes $f_i\}$.

The second list, called Scheduled-Steps(t), contains n steps, one for every I-path. These steps are scheduled during time slot t. Clearly, the steps of Scheduled-Steps(t) are selected from Frontier-Steps(t) subject to the condition that they do not conflict with each other.

Scheduled-Steps(t): $\{i \bullet x$ for $i = 1, 2, \ldots, n \mid i \bullet x \in$ Frontier-Steps(t), and no two steps in this set conflict with each other$\}$

```
BEGIN
  t = 0, Frontier-Steps(1) = {i•1, i•0' for i = 1,2, ..., n}.
  REPEAT UNTIL all steps are scheduled
    t = t + 1.
    Generate Scheduled-Steps(t) using Frontier-Steps(t).
    Generate Frontier-Steps (t + 1).
    END REPEAT
```

According to the definition given earlier, all the operations performed by the above procedure are staightforward except the one calling for selecting Scheduled-Steps(t) from Frontier-Steps(t). There are $2^n - 1$ possible selections; however, some of them may not be feasible due to conflicts.

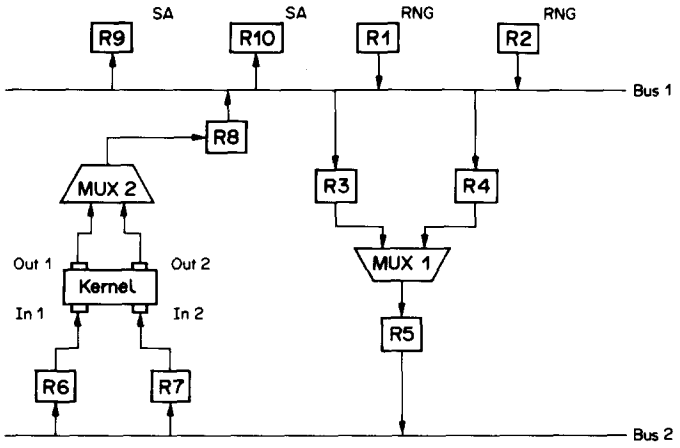

Fig. 13. BILBO imbedding with multiple I/O ports.

Also it is desirable to generate a solution with a minimal number of No-Ops so as to minimize the total length of the final plan. Even though the problem complexity is exponential, the value of n is often small (in the order of 2–4). Hence, an exhaustive search is feasible. One should start by trying the combinations with smaller number of No-Ops first. Moreover, the procedure can also be designed such that if two schedules are possible at a given point with the same number of No-Ops, it selects the one that advances data along the longest I-paths. The procedure is guaranteed to terminate, as deadlocks can only mean that some of the I-paths to be combined are forming a loop, which is not allowed.

Example 8. Consider the kernel structure in the circuit shown in Fig. 13, for which BILBO TDM is employed. Registers R1 and R2 assume the role of the test generator BILBO, while R9 and R10 assume the role of the signature BILBO.

There are two input I-paths to be combined. The activation plans of these two I-paths and their CG are given in Fig. 14 (isolated nodes are not shown).

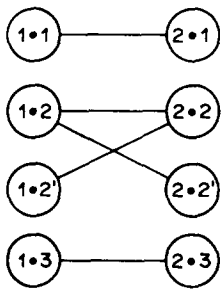

Fig. 14. The CG of Example 8. The plans of the input I-paths are as follows:

P_1(R1 → Kernel:in1) ≡
(1) Bus1(select R1), R3(Latch).
(2) MUX1(select R3), R5(Latch).
(3) Bus2(select R5), R6(Latch).

P_2(R2 → Kernel:in2) ≡
(1) Bus1(select R2), R4(Latch).
(2) MUX1(select R4), R5(Latch).
(3) Bus2(select R5), R7(Latch).

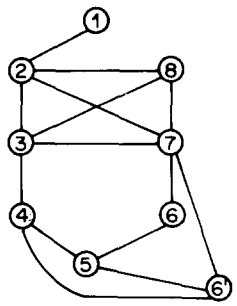

Fig. 15. Final CG of Example 8. The final test plan is as follows:

(1) R1(RNG), R2(RNG).
(2) Bus1(select R1), R3(Latch), R2(Hold).
(3) MUX1(select R3), R5(Latch), Bus(select R2), R4(Latch).
(4) Bus2(select R5), R6(Latch), MUX1(select R4), R5(Latch).
(5) R6(Hold), Bus2(select R5), R7(Latch).
(6) Kernel(–), MUX2(select out1), R8(Latch), R6(Hold), R7(Hold).
(7) Bus1(select R8), R9(SA), Kernel(–), MUX2(select out2), R8(Latch).
(8) R9(Hold), Bus1(select R8), R10(SA).

The 4-phase schedule cycle is:

Phase 1: 8 5 1
Phase 2: – 6 2
Phase 3: – 6′ 3
Phase 4: – 7 4.

A trace of the lists generated by Procedure "Combine-I-Paths" while combining the two input I-paths is shown below:

Time slot	*Frontier-Steps*				*Scheduled-Steps*	
1	1•1,	1•0′,	2•1,	2•0′	1•1,	2•0′
2	1•2,	1•1′,	2•1,	2•0′	1•2,	2•1
3	1•3,	1•2′,	2•2,	2•1′	1•3,	2•2
4	1•3′,	1•3′,	2•3,	2•2′	1•3′	2•3

Similarly, the two output I-paths can be combined. Now by substituting the combined activation plans for the data transfer actions of the test schema of the BILBO TDM, the test plan shown in the caption to Fig. 15 is generated. Clearly, there is a clique of size 4 due to the sharing of Bus1 by steps 2, 3, 7, and 8. Using Procedure 1, an execution schedule with $D = 4$ is found as described in Fig. 15. Only one No-Op has been added to the test plan after step 6. Note that 6′ does not conflict with step 3.

10. SCHEDULING TEST PLANS OF MULTIPLE KERNELS

In the previous sections we described how to generate a test plan for one kernel structure and how to construct an optimal test schedule for it. In this section we will consider the problem of multiple kernels. Kime and Saluja [9] considered the same problem. In their representation of the problem, every kernel is associated with a set of structures (called the resource set), which consists of the kernel itself and the structures used for test generation and for response evaluation. The problem then becomes that of scheduling the different kernel tests such that the overall test time is minimal. In such a schedule two tests can be scheduled in parallel only if their resource sets are disjoint. In the case where all tests have equal length, the problem is equivalent to the classical covering problem used in logic minimization. They also considered the case when tests have different lengths. No pipelining was considered in optimizing tests for single or multiple kernels.

Their work can be applied directly to tests generated using our techniques. Moreover, we will present two techniques that can further minimize the total test time.

10.1. Combining test schedules of different kernels

Assume that we have two kernels K1 and K2, and assume that we constructed optimal execution schedules for each kernel. Moreover assume that the initiation delay D of both schedules is the same. Let $Px(j)$ denote phase j of the schedule of Kx. We can form a conflict graph for modeling conflicts between phases of the two schedules.

THEOREM 4

The test schedules of two kernels that have the same initiation delay D can be executed in parallel using a schedule with initiation delay D if there exists a positive integer $k < D$ such that P1(i) does not conflict with P2[$(i + k)_{\text{mod } D} + 1$] for all $i = 1, 2, \ldots, D$.

Proof. A combined test schedule with D phases can be constructed as follow: Phase i of the new schedule consists of P1(i) and P2[$(i + k)_{\text{mod } D} + 1$], for all $i = 1, 2, \ldots, D$. Clearly, the schedule is feasible and both kernels will get tested by the repeated execution of the D phases of the new schedule. Q.E.D.

Note that even though the phases of two test schedules may have conflicts, it is quite possible to find a k that satisfies the above theorem, and hence execute the two schedules in parallel. Clearly, this was impossible to do using the problem formulation of [9].

10.2. Combining test plans of multiple kernels

The technique described in the previous subsection can only be applied if we have schedules with the same initiation delay. In cases where schedules have different delays or when we cannot combine schedules because of conflicts, we can try to combine the test plans of the different kernels. The same procedure used in the previous section for combining activation plans of different I-paths can be used here as well. Once a test plan for the combined kernels is produced, we can apply all the scheduling optimization techniques that we described earlier to generate a test schedule.

In using this approach we can explore ways of scheduling tests in parallel, for kernels that have nondisjoint resource sets.

11. REMARKS ABOUT PIPELINE OPTIMIZATION

As mentioned previously, the problem of generating optimal execution schedules for test plans is similar in concept to that of optimizing static pipelines. Using pipeline terminology [6], a *reservation table* can be used to model the hardware utilization of a test plan. Every row in the reservation table corresponds to a circuit structure (called stages), while every column corresponds to a step in the plan. A mark in row i and column j indicates that structure i is used during step j. Davidson [10] developed a procedure for constructing what is known as the *modified state diagram* of a reservation table. The diagram can then be traversed to find its cycles. Every cycle conveys a feasible value or a set of alternating values for D to assume. For example, a (2, 4, 5) cycle indicates that the delay between two consecutive iterations follows the sequence 2, 4, 5, 2, 4, 5, . . . , etc.

Even though reservation tables are widely accepted for optimizing pipelines, they do not model conflicts explicitly as do conflict graphs. To determine if two steps i and j conflict using a reservation table, one has to check if columns i and j have marks in the same row. On the other hand, the explicit representation of conflicts in the CG model has led to the derivation of lower bounds on D which are stronger than the one given by Shar [11], namely, the maximum number of marks in one row of a reservation table.

Moreover, the lemma presented in Section 4.3 for finding feasible constant schedule cycles is much simpler to program and requires much less computation than the state diagram method of Davidson [10]. However, our approach cannot generate alternating value cycles. Such a disadvantage is not very severe, because such schedules require complex controlling mechanisms which are not desirable in the context of built-in test.

Patel and Davidson [12] described a technique for inserting delays in pipelines to increase their throughput. The concept appears similar to the one of adding No-Op steps, yet there is a fundamental difference. To illustrate this, consider the reservation table shown in Fig. 16a. A delay, denoted by d, is inserted in the second row of the table as shown in Fig 16b.

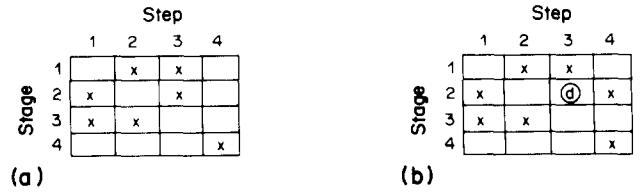

Fig. 16. Inserting a delay, d, in a reservation table.

The new table supports $D = 2$ while the first one does not. Note that the inserted delay has changed the structure of steps 3 and 4. Clearly, in the context of test plans such a transformation is not acceptable, without modifying the design, due to data dependencies. Recall that a No-Op step does not reorganize the actions performed in the steps of a test plan, it only introduces a step during which the data is held unchanged in some registers. Therefore, delay insertion techniques [12] are not generally applicable to test plans. Moreover, inserting delays in pipelines often requires the introduction of new registers. Such cost cannot be justified when attempting to optimize test plan schedules, and the delays may have some adverse effects on normal circuit operation. It is interesting to note that the use of special registers to implement delays alleviates the potential of creating conflicts as a result of inserting delays.

Thus, in summary, it is clear that even though one can apply classical pipeline optimization techniques to the problem of optimizing test plans, the approach presented in this paper is more practical and effective and has led to many useful new results.

12. CONCLUDING REMARKS

In this paper we introduced the new concept of test schema, I-modes, I-paths, activation plans, test plans, and test schedules. We developed a model for describing the conflicts that may arise due to the parallelism in these schedules. Lower bounds on the time required to execute a test plan were derived, and algorithms for constructing optimal test schedules were presented.

The results presented in this paper have been implemented as a part of a knowledge based system for designing testable VLSI chips [2]. The system is menu-driven with a user-friendly interface. The system explores all ways of embedding TDMs into a circuit for all its kernels as it guides the user in creating a modified design that meets testability goals and design constraints. Test plans for embedded TDMs are automatically produced together with their optimal execution schedules. The execution time of test schedules is used as one of the measures for evaluating embedding solutions. Other measures include area overhead and fault coverage. The system currently deals with three different TDMs, but others can be easily added.

REFERENCES

1. M. A. Breuer, A methodology for the design of testable VLSI chips. Report SD-TR-85-33, The Aerospace Corp., El Segundo, CA. 90245 (1985).
2. M. S. Abadir and M. A. Breuer, A knowledge based system for designing testable VLSI chips. *IEEE Design Test Comput.* pp. 56–68 (1985).
3. M. S. Abadir and M. A. Breuer, Constructing optimal test schedules for VLSI circuits having built-in test hardware. *Proc. 15th Int. Symp. Fault-Tolerant Computing*, pp. 165–170 (1985).
4. T. W. Williams and K. P. Parker, Design for testability—A survey. *IEEE Trans Comput.* **C31,** 2–15 (1982).
5. M. A. Breuer and X. Zhu, A knowledge based system for selecting a test methodology for a PLA. *Proc. 22nd Design Automation Conf.*, pp. 259–265 (1985).
6. P. M. Kogge, *The Architecture of Pipelined Computers.* McGraw-Hill, New York (1981).
7. B. Konemann *et al.*, Built-in logic block observation techniques. *Proc.* 1979 *Int. Test Conf.*, pp. 37–41 (1979).
8. J. A. Bondy and U. S. R. Murty, *Graph Theory with Applications.* North Holland, New York (1976).
9. C. R. Kime and K. Saluja, Test scheduling in testable VLSI circuits. *Proc. 12th Int. Sym. Fault-Tolerant Computing*, pp. 406–412 (1982).
10. E. S. Davidson, The design and control of pipelined function generators. *Proc. Int. IEEE Conf. Systems, Networks, and Computers*, pp. 19–21 (1971).
11. L. E. Shar, Design and scheduling of statically configured pipelines. Digital System Lab Report SU-SEL-72-042, Stanford (1972).
12. J. H. Patel and E. S. Davidson, Improving the throughput of a pipeline by insertion of delays. *3rd Ann. Symp. Computer Architecture*, pp. 163–169 (1976).