Behavioral Synthesis of Area-Efficient Testable Designs Using Interaction Between Hardware Sharing and Partial Scan

Miodrag Potkonjak, *Member, IEEE,* Sujit Dey, *Member, IEEE,* and Rabindra K. Roy, *Member, IEEE*

Abstract—**We introduce *BETS*, a behavioral test synthesis system, for the synthesis of high-throughput, area-efficient testable designs. While hardware sharing is a powerful technique to achieve area efficiency, it may adversely affect the testability of the synthesized design by introducing new loops. Besides CDFG loops, hardware sharing introduces three other types of loops: assignment loops, sequential false loops, and register files cliques. We provide a comprehensive analysis and a formal grammar characterization of the sources of loops in the data path during behavioral synthesis. Partial scan is a cost-effective technique for sequential circuit testing. Hardware sharing of scan registers can be used to minimize the number of scan registers required to synthesize data paths with minimal number of loops. The scan registers can be shared amongst several variables of the CDFG, to break not only the loops in the CDFG, but also the very loops introduced in the data path by hardware sharing. A new random walk based algorithm is proposed to break all CDFG loops using a minimal number of scan registers. The subsequent scheduling and assignment phase avoids formation of loops in the data path by reusing the scan registers, while ensuring high resource utilization. The experimental results demonstrate the effectiveness of the new technique to synthesize easily testable data paths, with nominal hardware overhead, while maintaining the performance of the designs. The partial scan overhead incurred by the technique is significantly less than that of a gate-level partial scan approach.**

I. Introduction

HIGH-LEVEL synthesis encompasses a variety of synthesis tasks, such as partitioning, module selection, and transformations, each of which has the potential to influence a number of design parameters, e.g., area, speed, and power. However, only allocation, scheduling, and assignment (binding) are widely recognized as the mandatory backbone tasks in high-level synthesis [32], [43], [33]. Allocation is the step during which it is decided how many basic blocks of each type of hardware unit (like adder) will be used in the design; scheduling is the step during which the temporal location within boundary of available time for each operation is fixed; and assignment is the step which assigns instances of operations to hardware units in the design.

A great variety of scheduling, allocation, and assignment algorithms have been studied in the high-level synthesis literature [32], [43]. While the underlying algorithmic techniques vary from very simple heuristics to involved formal techniques, the goal of almost all the algorithms has been optimization of speed under resource constraints or its dual. More recently, the list of targeted goals was enchanced to include power [6], fault tolerance [25], and testability [20], [36], [3], [7], [28], [29], [27], [31], [39], the latter being the main topic of this paper.

Manuscript received September 23, 1993; revised December 5, 1994. This paper was recommended by Associate Editor K.-T. Cheng.

The authors are with C&C Research Laboratories, NEC USA Princeton, NJ 08540 USA.

IEEE Log Number 9412150.

A. High-Level Synthesis and Testability: The State of the Art

It is widely recognized that testability of a circuit is dependent on the testing methodology adapted. The high-level synthesis for testability approaches can be broadly classified into two groups—BIST-based (built-in-self-test) and ATPG-based (automatic test pattern generation). The BIST-based approaches assume the presence of a pseudo-random pattern generator for test vector generation and a MISR (multiple-input signature register) or other signature analyzers for response compression [1]. Almost all BIST approaches assume a full-scan design methodology since random testing is not well-suited for sequential circuits [20], [12], [36], [23], [3], [4]. Area overhead due to incorporation of BIST was reported in all of these studies; however, the quality of testing (fault coverage) obtained in the final design was not reported by any of them.

ATPG-based testing approaches assume that the test patterns would be generated by deterministic automatic test pattern generators. These techniques do not routinely assume full-scan; however, for ease of sequential circuit test generation, some techniques assume presence of partial scan [9]. ATPG-based approach for testing after high-level synthesis was taken by [8], [7], [28], [29], [31], [41], and [15]. Testability improvement using register assignment and scheduling were reported in [28] and [29], respectively. An approach to generate testable data paths, by minimizing the number of self-loops, was reported in [31]. More recently, Lee, Jha, and Wolf [27] developed a method to minimize formation of loops in the data path by partial scan and proper register assignment.

Several researchers have attempted to improve the testability of circuits by manipulating the RT-level description. Chickermane, Lee, and Patel [10] showed that the use of RT-level information to select scan flip-flops results in significantly better performance when compared to techniques limited to gate-level information only. Also, transformation and optimization techniques were proposed in [5], which

utilize RT-level information to generate optimized designs that are 100% testable under full scan. More recently, a design-for-testability technique was presented to make RT-level data paths easily testable without the use of scan [14].

B. Testability of Sequential Circuits

Since our goal is to synthesize designs which are easy-to-test by a gate-level sequential deterministic automatic test pattern generator, we briefly discuss the factors which influence the testability of a sequential circuit. The dependencies of the flip-flops (FF's) of a sequential circuit are captured by an S-graph [9], [26], where each node corresponds to a FF. There is a directed edge from node u to node v if there is a combinational path from FF u to FF v in the sequential circuit. It has been empirically found by Cheng and Agrawal [9], and confirmed later by others [26], that sequential test generation complexity might grow exponentially with the length of cycles in the S-graph. The sequential test generation complexity grows only linearly with the longest path (sequential depth) in the S-graph. These observations have been used by several researchers to guide the process of selecting scan flip-flops for partial scan [9], [26]. The scan flip-flops are selected so that the S-graph has no cycles except self-loops, and the sequential depth is minimal. It has been shown experimentally that a sequential circuit with no loops, other than self-loops, and having low sequential depth, is easily-testable by current deterministic sequential test pattern generators [9], [26].

Two particularly hard-to-test structures containing cycles in the S-graph are strongly connected components (SCC's) and cliques. A strongly connected component of a directed graph consists of a set of nodes V, and the edges between the nodes in V, such that there is a directed path from node x to node y, and node y to node x, for any pair of nodes $x, y \in V$. A clique is a strongly connected component such that for each pair of nodes u, v in the clique, there is a direct edge from u to v, and v to u. Chickermane and Patel [11] observed that most of the hard-to-detect faults in a sequential circuit are found in moderately sized and large strongly connected components (SCC's). They also observed that cliques should be broken with high priority, and that hard-to-detect faults seldom occur on self-loops. Based on these observations, they developed a highly efficient partial scan approach at the gate-level, which selects scan FF's using the knowledge of faults aborted by the test pattern generator.

C. New Approach to Synthesis of Data Paths for Easy Testability

Since we target computation-intensive application domains, e.g., DSP, communications, control theory applications, and graphics, only a few FF's are needed for the states of the controller. In our design for testability framework, we assume all the control signals to the data path to be made fully controllable by scanning the FF's of the controller. We target data paths with dedicated register files, explained in Section II.

In this paper, we concentrate on generating easily testable data paths from high-level specifications, achieving high resource utilization, and satisfying given performance constraints. We emphasize on achieving high resource utilization so that the testability of the final design is achieved with minimal area overhead, thus giving our approach a significant advantage over gate level design for testability schemes [9], [26], [11].

Hardware sharing is a widely used methodology to achieve high resource utilization, but it might affect the testability of a circuit negatively by introduction of new loops in the data path. However, when hardware sharing is exploited properly in conjunction with the partial scan methodology, improvements in testability can be achieved despite possible introduction of loops. The scan registers can be shared amongst several variables of the CDFG, to break not only the loops in the CDFG, but also the loops introduced in the data path by hardware sharing. We attempt to make the data path easily testable by ensuring that the synthesized data path has no loops, except self-loops.

A comprehensive analysis of the formation of loops in circuits synthesized by behavioral synthesis is presented. The analysis uses the data dependencies and the compatibilities of the operations of the Control Data Flow Graph (CDFG) specification to develop a regular expression-based representation to identify conditions under which loops will be created.

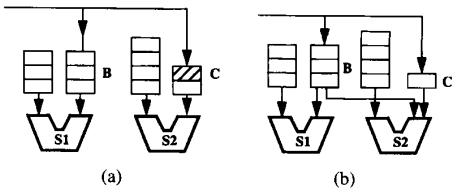
This paper uses the interaction between hardware sharing and partial scan to synthesize a minimal-loop data path. Based on the loop analysis, simultaneous scheduling and assignment algorithms are presented which avoid the formation of loops by sharing the scan registers. We applied the new algorithms to several industrial designs, including a 20-b, maximally parallel fifth order Elliptical Wave Digital Filter (EWF). The experimental results show that the new technique can synthesize testable designs, with very low hardware overhead, without compromising the performance of the designs.

D. Overview

The hardware model used in the sequel is presented in Section II. Section III motivates the advantages of incorporating testability in the design during the early phase of high-level synthesis. A comprehensive analysis of the sources of loops in the data path is presented in Section IV. In Section V, we present algorithms to synthesize area-efficient minimal-loop data paths, by exploring the interaction between hardware sharing and partial scan. The effectiveness of the proposed behavioral synthesis approach is demonstrated on a set of numeric-intensive benchmarks in Section VI. Section VII summarizes the main contributions of the paper.

II. Hardware Model: Importance, Description, and Justification

It has been widely recognized that the implementation area in hardware-shared architectures is most often dominated by interconnect requirements [32]. Unfortunately, addressing of interconnect issues during high-level synthesis is often postponed to the phase when all major design decisions have been already made. This situation can be attributed to the fact it is very difficult to accurately correlate interconnection requirements at behavioral and register transfer (RT) level to



(a) (b)

Fig. 1. Hardware model.

that at physical level. However, an effective measure can be easily achieved if the hardware model shown in Fig. 1(a) is accepted. In this model, called *register file* model, all registers are clustered in a number of registers files. While each execution unit can send data to any register file in the general case, each register file is connected to one input of a single functional unit.

Numerous variations of the hardware model, where registers are grouped in register files and access to a particular register is limited to only one functional unit, are widely used in industrial designs. A number of fully operational high-level synthesis systems, such as Cathedral-II and Hyper, are also based on the register file model [18], [38]. It is de facto standard for modern general purpose architecture [37], [30], [42]. There are several reasons for adapting the described hardware model. First of all, its intentional restrictions powerfully enforce a limitation on the number of interconnects and encourage heavy use of local interconnects (between register files and execution units). Also, interconnects can be effectively shared.

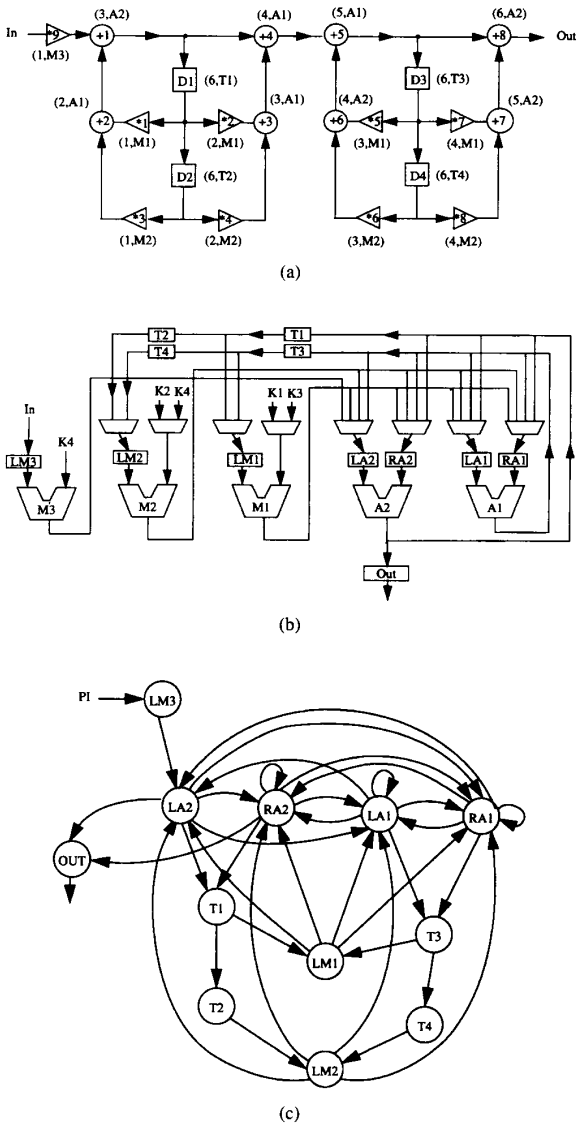
Fig. 1(a) and (b) show a simple analysis which directly illustrates the advantages and disadvantages of the adopted hardware model. Suppose that in a particular control step we have to store a variable which is needed to be executed on the subtractor S2. While all the registers in the register file C are already occupied, there is at least one free register in the register file B. We have two possibilities: 1) to increase the number of registers in the register file C by 1; or 2) introduce a new interconnect from the register file B to the right input of subtractor. The register file model enforces the use of the first alternative, which is most often superior to the second possibility, due to the reasons described above.

Although we will use the register file model exclusively in the rest of the paper, only very limited and straightforward modification of the design and test synthesis algorithm is required if the described interconnect restrictions are not considered.

III. Motivation

Consider the Control Data Flow Graph (CDFG) of the 4th order IIR cascade filter shown in Fig. 2(a). The example consists of eight additions represented by $+1,\cdots,+8$, nine multiplications represented by $*1,\cdots,*9$, and four delays represented by $D1,\cdots,D4$. A **delay** (state or boundary variable) represents a variable to be transferred from one iteration to a consecutive iteration.

Assume that each operation in the CDFG takes one control cycle. The critical path is 6 control cycles long. Fig. 2(a) shows



(a) (b) (c)

Fig. 2. Fourth order IIR cascade filter, satisfying performance constraint (6 control steps), and using minimum number of execution units (2A, 3M). (a) Scheduled and assigned CDFG. (b) Data path. (c) S-graph.

one possible feasible schedule and assignment, satisfying a performance constraint of 6 control cycles, and using minimal number of execution units. For instance, the operation $+_2$ is scheduled in control cycle 2, and assigned to be executed in adder $A1$, shown in Fig. 2(a) by the ordered pair $(2, A1)$. Similarly, the delay D1 is scheduled in control step 6, and assigned to transfer unit T1, as shown by the ordered pair $(6, T1)$ in Fig. 2(a). The resultant data path is shown in Fig. 2(b). It has three multipliers, two adders, four transfer units containing four registers and four multiplexers, and eight other registers. For example, LA2 and RA2 represent the left and right registers of the adder A2.

Similar to the S-graph of a gate-level sequential circuit, the S-graph of a data path identifies the dependencies between

the registers of the data path. The S-graph in Fig. 2(c), corresponding to the data path in Fig. 2(b), reveals the existence of several loops involving the registers of the data path. The longest loop in the S-graph has length 8. As can be expected, the data path is very hard to test, as indicated in Table II by the row *IIR.16 Orig.*

The testability of the data path can be improved using partial scan techniques at the gate-level [9], [26], [11], to break all the loops of the circuit. Breaking all the loops of the S-graph in Fig. 2(c) needs scanning at least 3 registers, namely LA1, LA2, and LM1, which translates to $3n$ FF's, where n is the wordsize. However, the associated area and performance overheads due to the large number of scan FF's can be very expensive.

The example in Fig. 2 illustrates that hard-to-test data paths can be generated if testability of the data path is not considered during high-level synthesis. Instead of postponing the task of making the design testable to the gate-level, it is possible to incorporate testability as one of the design goals, besides performance and resource utilization, during the various high-level synthesis tasks. Fig. 3(a) shows the same flow graph of the IIR filter, with a different assignment, satisfying the same performance constraints as before. Note that this solution is also optimal with respect of the number of execution units used. The corresponding data path and the S-graph, shown in Fig. 3(b) and (c) respectively, are significantly more amenable to sequential testability. Notice that the S-graph still has loops; however, scanning register RA2 will break all the loops. The resultant data path, with register RA2 scanned, has no loops, and is very easy-to-test. A test efficiency of 100% could be achieved on the resultant data path, as evidenced by the row *IIR.16 SFT* in Table II.

IV. Formation of Loops in the Data Path

The data path synthesized from a CDFG can contain several types of loops. In this section, we identify and formulate the formation of different types of loops in the data path. We begin by modeling the data dependencies and the compatibilities of the operations of the CDFG by a Data-Dependency and Compatibility Graph (DDCG). Each node in the graph represents an operation of the CDFG. There is a (undirected) compatibility edge between two operations if there is a nonzero probability that both the operations can be assigned to the same module. There is a (directed) data-dependency edge from operation v to operation w if operation w depends on data produced by operation v. We will denote a compatibility edge between v and w as $c(v, w)$, and a data-dependency edge from v to w as $d(v, w)$.

Fig. 4(a) shows segments of two paths in a CDFG, and Fig. 4(b) shows the corresponding data-dependency and compatibility graph. Assuming that each operation in the CDFG takes one control cycle, the longest path is 3 control steps long. For a schedule which requires 3 control steps, the As Soon As Possible (ASAP) and the As Late As Possible (ALAP) control steps in which each operation can be scheduled [43] is shown by the tuple [ASAP, ALAP] in Fig. 4(a). In Fig. 4(b), each compatibility edge $c(v, w)$, shown dotted, is weighted by an estimate of the compatibility between two nodes v and w, which is discussed below.

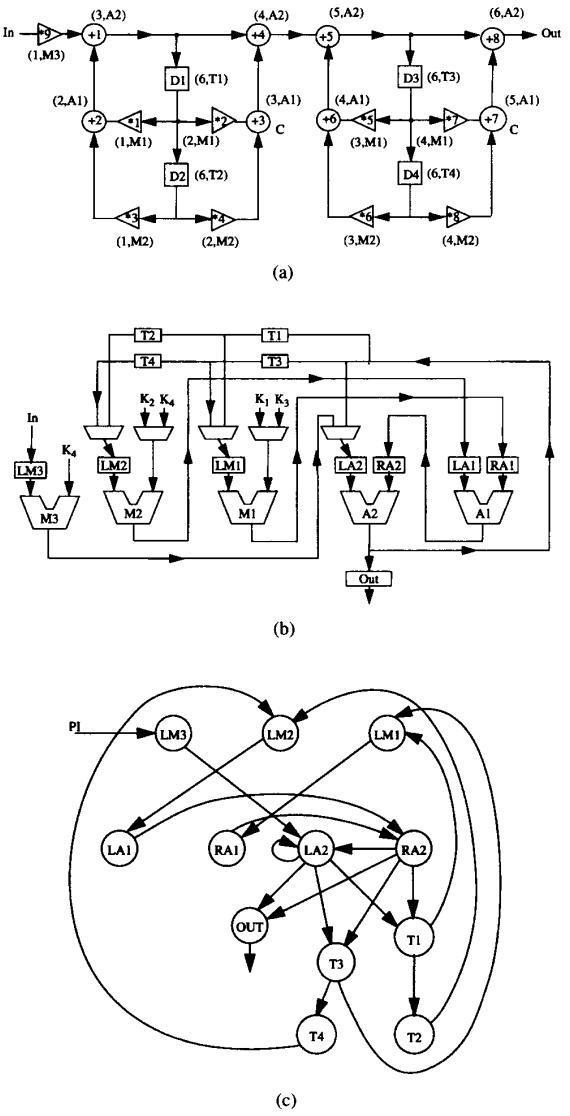


Fig. 3. Fourth order IIR cascade filter, synthesized for **testability and resource utilization**, satisfying performance constraint (6 control steps), and using minimum number of execution units (2A, 3M). (a) Scheduled and assigned CDFG. (b) Data path. (c) S-graph.

The mobility of an operation v, **mobility**(v), is the number of control steps in which it can be scheduled.

$$\text{mobility}(v) = \text{ALAP}(v) - \text{ASAP}(v) + 1;$$

The overlap between two operations, v and w, **overlap(u,w)** , is the number of control steps in which both v and w can be scheduled. Refering to Fig. 4(a), mobility$(+_1) = 1$, mobility $(+_3) = 2$, and overlap$(+_1, +_3) = 1$.

Let $p(v, w)$ represent a path, consisting of data dependency edges, from node v to node w in the DDCG. The estimate of the compatibility between two nodes v and w, **Comp(v,w)**, is a measure of the flexibility that v and w can be assigned to

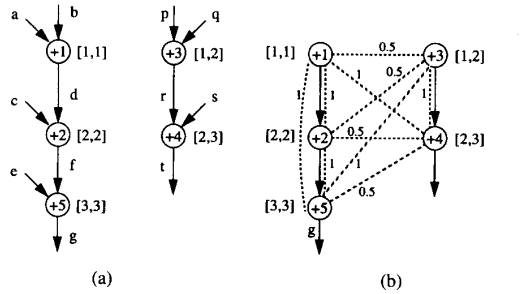


Fig. 4. Illustrating the formation of loops. (a) A CDFG. (b) The corresponding DDCG.

the same execution unit (module).

$$\text{Comp}(v,w) = \begin{cases} 1 & \exists p(v,w) \\ 1 - \dfrac{\text{overlap}(v,w)}{(\text{mobility}(v)*\text{mobility}(w))} & \text{otherwise.} \end{cases}$$

We assume that no data-chaining is allowed during scheduling, that is, no two data-dependent operations will be scheduled in the same control step. Consequently, whenever there exists a path $p(v,w)$, consisting of data-dependency edges, the operations v and w will be scheduled in two different control steps. Hence, $\text{Comp}(v,w) = 1$, indicating that the two operations can always share a resource, if desired during scheduling and assignment.

The compatibility between nodes $+_1$ and $+_3$, $\text{Comp}(+_1,+_3) = 1 - (1/2) = 0.5$. All the compatibility edges c(v,w) are weighted by $\text{Comp}(v,w)$, as shown in the DDCG in Fig. 4(b).

A. Representing Loops by Regular Expressions

A **path** p from node x to node y is a sequence of nodes and edges starting with x and ending with y. A path is *simple* if all nodes and all edges on the path, except the first and last node, are distinct. A **cycle** is a simple path which begins and ends with the same node. Throughout this paper, by a path, we refer to a simple path.

The paths of the DDCG can be represented using regular expressions [2]. d^+ represents a path consisting of a sequence of one or more data-dependency edges d. The concatenation of two paths p and q is represented by $p.q$, or simply, pq. For example, the regular expression cd^+ represents a path starting with a compatibility edge c, followed by one or more occurrence of data-dependency edges. In Fig. 4(b), the path $(+_5,+_1),(+_1,+_2),(+_2,+_5)$ can be represented by cd^+.

B. Detecting Formation of Loops Using DDCG

The first type of loops, data-dependency loops, are formed in the data path due to the presence of cyclic data dependencies in the CDFG. Assignment loops, sequential false loops and register file cliques are formed due to hardware sharing.

1) Data-Dependency Loop: A Data-Dependency Loop is formed in the data path if there exists a cycle of the form d^+ in the DDCG. In other words, if all the edges of a cycle in the DDCG, or the CDFG, are data-dependency edges, then a loop is formed in the data path, irrespective of the register and module assignment.

Consider the CDFG shown in Fig. 2(a). The cycle $(+_1, D1, *_1, +_2, +_1)$ consists of only data-dependency edges. Consequently, a data-dependency loop will be formed in the data path, irrespective of the register and module assignments used. If the assignment shown in Fig. 2(a) is used, the loop (RA1,T1,LM1,RA2,RA1) is produced in the corresponding data path. Similarly, the assignment shown in Fig. 3(a) produces the loop (RA2,T1,LM1,RA1,RA2).

2) Assignment Loop: During assignment of operations to modules (EXU's), we say that a compatibility edge is *used* if the two operations associated with the compatibility edge are assigned to the same module. An assignment loop is formed in the data path if there exists a cycle of the form cd^+ in the DDCG, and the compatibility edge c is used during module assignment. In other words, an assignment loop is formed whenever two or more operations in the path of a CDFG are assigned to the same module.

In the DDCG shown in Fig. 4(b), there is a cycle $\{(+_5,+_1),(+_1,+_2),(+_2,+_5)\}$ of the form cd^+. Using the compatibility edge $(+_5,+_1)$ to assign the compatible operations $+_5$ and $+_1$ to the same module, creates an assignment loop in the data path. Let the schedule and assignment of the operations be: $\{+_1 : (1,A1), +_2 : (2,A2), +_3 : (2,A1), +_4 : (3,A2), +_5 : (3,A1)\}$. It satisfies the constraint of three control steps, and uses the minimum number of EXU's (2 adders). The resultant data path, shown in Fig. 5(a), and its corresponding S-graph, has an assignment loop (RA1,RA2,RA1) shown in bold.

A self-loop is a special case: it can be formed in the data path either by the presence of a data-dependency loop of the form d, or an assignment loop of the form cd.

3) Sequential False Loop: A sequential loop in the data path is termed false when the loop cannot be sensitized. A false loop is a special case of a false path. Let the schedule and assignment of the operations of the CDFG in Fig. 4(a) be: $\{+_1 : (1,A1), +_2 : (2,A2), +_3 : (1,A2), +_4 : (2,A1), +_5 : (3,A2)\}$. It satisfies the constraint of three control steps, and uses the minimum number of EXU's (2 adders). The resultant data path, shown in Fig. 5(b), has two loops.

Consider the loop in Fig. 5(b) shown in bold. To sensitize the loop, the required control signals to the multiplexers M1 and M4, $c1$ and $c2$, should be $\{c1 = 1, c2 = 0\}$ (or, $\{c2 = 0, c1 = 1\}$) in any two consecutive control steps. However, this necessitates execution of operations $+_4$ followed by $+_2$ (or $+_2$ followed by $+_4$), which is clearly not possible. Consequently, the sequential loop can never be sensitized, and is a false loop. The S-graph corresponding to the data path has a sequential false loop (LA1,RA2,LA1). Note that the only other loop in the data path is the self-loop (RA2,RA2) which is an assignment loop.

Stok considered the formation of false loops through resource (module) sharing [40]. However, the treatment of false loops involving data paths was limited to combinational loops.

Combinational false loops, as analyzed in [40], are generated in the data path when data-chaining is allowed. That is, two or more operations are scheduled in the same control

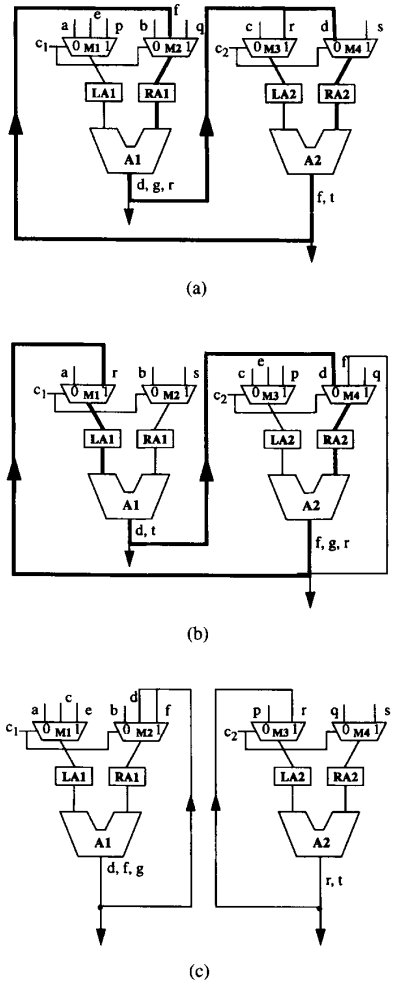


Fig. 5. Data path formed by different assignments of CDFG in Fig. 4(a). (a) Assignment loop. (b) Sequential loop. (c) No loops except self-loops.

step. Even when data-chaining is not allowed, resource sharing can still lead to *sequential* false loops in the data path (Fig. 5(b)). Since we make the control signals to the data path fully controllable by scanning the state flip-flops of the controller, sequential false loops contribute to the complexity of sequential test pattern generation in the same way as any other loops.

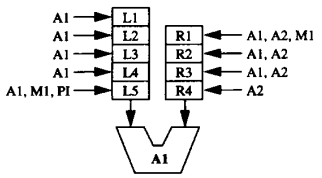
Let $c = (v, w)$ be a compatibility edge in the DDCG such that there does not exist any data-dependency edge from v to w. If there is a cycle of the form $(cd^+)(cd^+)^+$ in the DDCG, and the compatible edges c in the cycle are used during module assignment, a sequential false loop is formed in the data path.

In the DDCG shown in Fig. 4(b), there is a cycle $(+_4, +_1)$, $(+_1, +_2)$, $(+_2, +_3)(+_3, +_4)$ of the form $(cd^+)(cd^+)^+$. Assigning the compatible pairs $(+_4, +_1)$ to adder A1, and $(+_2, +_3)$ to adder A2, leads to a sequential false loop as illustrated in the data path in Fig. 5(b).

4) Register File Cliques: When a module M_i has a register file associated with each input, the registers in the register files may form a clique in the S-graph. Let a register belonging to

Variables	Lifetime	Source	Register Assignment
PI->0,12	1-14	PI	L5
0->1,9,10	1-11	A1	L1
1->2,6,7	2-7	A1	L2
2->4	3	A1	L3
6->19	7-8	A1	L3
12->13	15	A1	L5
19->22	9	A1	L3
32->33	16	M1	L5
33->31	17-14	A1	L4

(a)



(b)

Fig. 6. Register files of adder A1 in the EWF data path (Fig. 8(b)). (a) Lifetimes and register assignment of variables assigned to left register file. (b) Left and right register files of A1.

a register file of Module M_i be termed a *self-loop register* if it receives data from M_i, besides possibly other modules. Each self-loop register of module M_i has a path through module M_i to (from) itself and all other self-loop registers of M_i. In the corresponding S-graph, each self-loop register has an edge to (from) the other self-loop registers of module M_i. Consequently, if the left and right register files of module M_i have m and n self-loop registers respectively, then a clique of size $(m + n)$ is formed in the S-graph.

Fig. 6 shows the register files of module A1, one of the adders used in the data path of the elliptical wave filter shown in Fig. 8. The assignment of operations to modules is shown in Fig. 8. The left operand of each operation mapped to A1 has to be assigned to some register of the left register-file of A1. This assignment is also shown in Fig. 6(a). Column *variables* and *Lifetime* show the left operands of each operation that was assigned to A1, and the lifetimes of the variables. Column *Source* indicates the source of data to A1.

Multiple registers are required due to conflicts in the lifetimes of the variables. For instance, in Fig. 6(a), the variables (0,10), (1,7), (6,19), and (33,31) are all alive in the 7th control step and receive data from A1; hence they need to be assigned to four different self-loop registers. The left register file has 5 registers: $\{L1 \cdots L5\}$, and the right register file has 4 registers: $\{R1 \cdots R4\}$. The inputs of the registers are shown. For instance, register L1 has a single input coming from module A1, while register R1 has three inputs, A1, A2, and M1. Note that in Fig. 6(b), any register with input A1 is a self-loop register.

Since there are 5 self-loop registers in the left register file of A1, and 3 self-loop registers in the right register file, a clique involving the 8 self-loop registers is formed in the corresponding S-graph.

Since each register in a clique is completely connected with all the other registers of the clique, breaking all the loops of a clique of size k requires scanning $k-1$ registers. This means that formation of cliques not only makes test pattern generation very hard, it makes a partial scan solution very expensive.

C. An Example of Forming a Data Path Without Loops

Having analyzed the conditions which lead to the formation of loops in the data path, we show a sample scheduling and assignment which avoids the formation of loops in the data path. The basic idea is to avoid using any compatibility edge in the DDCG which is a part of a cycle of the form cd^+ or $(cd^+)(cd^+)^+$. Consider the following schedule and assignment which satisfies the performance constraints, and which uses the minimum number of execution units:

$$\{+_1 : (1, A1), +_2 : (2, A1), +_3 : (1, A2), \\ +_4 : (2, A2), +_5 : (3, A1)\}.$$

The resultant data path, shown in Fig. 5(c), does not contain any loops, except for two self-loops. While one register needs to be scanned to break the loops of the data paths in Fig. 5(a) and (b), no register needs to be scanned for the data path in Fig. 5(c).

V. Algorithms for Efficient Use of Partial Scan and Hardware Resources

A complex set of goals is imposed during scheduling and assignment which simultaneously addresses both hardware resource utilization and testability issues while satisfying the throughput constraints. As mentioned earlier, in order to design a hardware competitive solution, it is mandatory to consider all three components of implementation cost: execution units, registers, and in particular, interconnects. Testability imposes requirements on all four types of loops in the data path graph, and, to a lesser extent, sequential depth.

Our approach to the allocation, scheduling, and assignment problems has three phases, as shown in Fig. 7. We start with the initial allocation of the number of execution units, provided by any high-level synthesis system which targets exclusively resource utilization. Currently we are using the Hyper system from University of California at Berkeley [38] for the execution unit allocation.

In the second phase, all CDFG loops are broken by assigning a subset of variables (scan variables) to scan registers. Each operation which consumes at least one scan variable is assigned to an execution unit (module), and the scan variable is assigned to the associated register file.

In the third phase, we simultaneously schedule and assign each operation of the CDFG using global resource utilization and testability measures. Note that some of the variables and operations have already been assigned in the second phase.

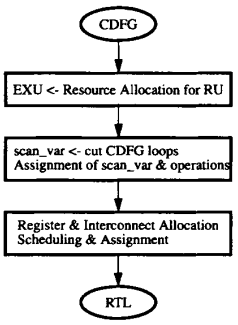


Fig. 7. Phases of allocation, scheduling, and assignment for testability and resource utilization.

A. Breaking CDFG Loops with Minimal Number of Scan Registers

In this section, we discuss the problem of breaking the CDFG loops using a minimal number of scan registers. A similar problem has been earlier addressed in the case of S-graphs of gate-level circuits [9], [26]. In the case of gate-level circuits, the minimum number of scan registers required is equal to the minimum number of vertices of the S-graph, which need to be deleted to break all the loops in the S-graph. Since the minimum feedback vertex set problem is NP-complete [19] (p. 191), several heuristics were successfully used [9], [26].

When hardware sharing is not used, a solution to the minimum feedback vertex set problem can be directly applied to break the CDFG loops. However, when hardware sharing is used, the minimum feedback vertex set is not necessarily a good solution. This is because, at the CDFG level, the variables selected to break the loops **(scan variables)** can share the scan registers.

The new dimension added by hardware sharing to the problem of breaking loops is illustrated by the CDFG of the IIR filter shown in Fig. 2. A possible solution to the minimum feedback vertex set problem is the edges: $\{(+_1, D1), (+_5, D3)\}$. However, since the variables $D1$ and $D3$ are simultaneously alive in the first control step, they cannot be shared, thus requiring 2 scan registers to break the CDFG loops. On the other hand, if we select as scan variables $(+_2, +_1)$ and $(+_6, +_5)$, all CDFG loops are broken. The scan variables can be now stored in the same scan register, since their lifetimes do not overlap, regardless of the schedule and assignment used.

Lee, Jha, and Wolf [27] proposed an approach to cut CDFG loops using a subset of boundary variables (variables which correspond to the edges with delays). Since all boundary variables are alive simultaneously, each selected variable has to be assigned to a separate scan register. To maximize likelihood of reuse of the scan registers, they select boundary variables with short lifetimes. Later, during register assignment, they share scan registers among intermediate variables to minimize the formation of assignment loops in the data path.

While [27] introduces the important idea of sharing scan registers, the technique does not exploit hardware sharing while selecting scan variables to break CDFG loops. Firstly, all boundary variables are simultaneously alive (in the first control step), and therefore cannot share scan registers. In

contrast, considering all variables in CDFG loops as possible candidates for scan variables, greatly improves chances of efficient sharing of scan registers. Consider the IIR filter shown in Fig. 2. Limiting the choice of scan variables to boundary variables results in the use of 2 scan registers $(+_1, D1)$ and $(+_5, D3)$. However, considering all variables in the CDFG loops results in a solution which uses only 1 scan register, as described earlier.

Secondly, the length of lifetimes of variables has only indirect and second order effect on hardware sharing. The necessary and sufficient conditions that two variables can share the same register are that they are not simultaneously alive and that proper interconnect for transferring the two variables is allocated.

1) The Minimum Hardware-Shared Cut (HSC) Problem: The goal of our approach to break CDFG loops is to select a set of scan variables such that the following criteria are simultaneously satisfied:

- **HSC1:** All CDFG loops, except self-loops, are broken;
- **HSC2:** The selected scan variables can be assigned to a minimum number of scan registers; and
- **HSC3:** Reusability of the scan registers, to break the other loops formed during the subsequent scheduling and assignment phase, is maximized.

We refer to the above problem as the minimum *hardware-shared cut* (HSC) problem. When no hardware sharing is allowed, the minimum HSC problem reduces to the minimum feedback vertex set problem. Consequently, an exact algorithm for the minimum HSC problem is at least as computationally intensive as the minimum feedback vertex set problem, which is known to be NP-complete [19].

We address the minimum HSC problem by using an approach which combines probabilistic and heuristic techniques. We introduce two measures to capture the effectiveness of a variable in satisfying the three criteria of the minimum HSC problem. Techniques to calculate the measures are also outlined.

The loop cutting effectiveness (LCE) measure helps to satisfy the first criteria, HSC1, of the minimum HSC problem. The LCE measure of a variable estimates the number of loops that will be broken by assigning the variable to a scan register. Since the number of loops can be exponential, and there is no known algorithm to count them efficiently, we use the random walk methodology, a probabilistic technique which has found wide range of applications recently [22].

The random walk starts by assigning a unit value to each node of the CDFG. In each successive iteration, the value at each node v is propagated along one randomly selected directed edge from v to one of its neighbors. Each time an edge is used to propagate the value, we increment the congestion factor of the edge. The number of iterations is user specified. It was proved in [24] that it is sufficient to conduct random walk for a quadratic number of steps in order to get statistical characterization of the edges. Hence, we choose the number of iterations to be a quadratic function of the number of nodes in the CDFG. The selection of the quadratic function was later substantiated by our experimentation. At the termination of the random walk, the congestion of each edge of the CDFG gives an estimate of the number of loops the edge belongs to, which is used as the LCE measure.

The hardware sharing effectiveness (HSE) measure is introduced to satisfy criteria HSC2 and HSC3 of the minimum HSC problem. The HSE of a variable v estimates the likelihood that v can share a scan register with other variables which may be needed to break all four types of loops in the data path. It consists of two components listed below.

The first component addresses the second criteria of selecting scan variables, which is, minimization of the number of scan registers required to cut the CDFG loops. The highest HSE value is assigned to variables which have least overlap in lifetimes with candidates for scan variables from other SCC's of the CDFG. Since the operations have not been scheduled yet, an estimate of the lifetime is calculated using the ASAP and ALAP timing information [43].

The second component measures the likelihood that the scan register SR1, to which the variable v will be assigned, can be effectively reused later to break the other three types of loops. Let f_v be the type of the operation to which variable v is an input. In the context of our hardware model, SR1 can be shared only by the set of variables $\{x\}$ which are inputs to operations of the same type f_v, and whose lifetimes do not overlap with the lifetime of v. As before, we use an estimate of the lifetimes to determine the variables whose lifetimes do not overlap with v. The second component is set to be the cardinality of the set $\backslash\{x\}$, thus favoring a variable whose assigned scan register can be reused later by the largest number of variables.

An algorithm to select scan variables to cut the CDFG loops is presented below.

select_scan_varables()

1) Identify nontrivial strongly connected components (SCC).
2) For each edge e which belongs to some SCC, calculate $eff_{HSC}(e) = \alpha * LCE(e) + \beta * HSE(e)$.
3) For each SCC, select as scan variable the edge e with highest $eff_{HSC}(e)$. Delete the selected edges from the CDFG.
4) If there exists any SCC, go to step 1.
5) Assign the selected scan variables to register files.

After the scan variables have been selected to cut the CDFG loops, at first, a minimum set of scan registers is identified to which all the scan variables can be assigned. This can be done optimally by assigning all scan variables with disjoint lifetimes to the same scan register. As a second step, the scan registers are selected from as many register files of as many execution units as possible. The second step increases the chances of reusing the scan registers to assign variables to avoid the formation of loops during scheduling and assignment.

We will illustrate the process of selecting scan variables using the IIR filter shown in Fig. 3(a). The edges $\{(+_2, +_1), (+_1, D1), (+_6, +_5), (+_5, D3)\}$ get maximally congested during random walk, and hence obtain the highest LCE values. Note that these variables cut the maximum number of loops. The variables $\{(+_2, +_1), (+_6, +_5)\}$ score high in both the components of the HSE measure, since their lifetimes do

not overlap with each other, and the cardinality of set $\{x\}$ is maximal. Hence, the algorithm **select_scan_variables** will select the variables $\{(+_2, +_1), (+_6, +_5)\}$, and assign them to the same scan register.

B. Scheduling, Assignment and Allocation for Testability and Resource Utilization

After the CDFG loops have been broken using a minimal set of scan registers, in the third and final phase, we simultaneously schedule and assign each operation of the CDFG using global testability and resource utilization measures. The aim is to produce a testable data path, avoiding the formation of the three types of loops mentioned before. However, priority is also given to use a schedule and assignment which satisfies the constraint on control steps and which maximizes resource utilization, so that the final design is not only testable, but also competitive in terms of hardware cost.

The simultaneous scheduling and assignment phase is conducted in the following way. At each iteration of the algorithm, from the operations that have not yet been scheduled and assigned, an operation op_i with least slack (ALAP−ASAP) is selected. The set of (module, control step) pairs, $\{(M_i, C_i)\}$, to which/in which the operation can be assigned/scheduled, are identified. For each pair, the cost in terms of testability, resource utilization and flexibility for scheduling and assignment of subsequent operations, is computed. Subsequently, a pair with the smallest cost is selected. The cost measures will be described in the subsequent sections. We outline below the algorithm for scheduling and assignment of operations.

schedule_and_assign()

1) while there exists a node which is not scheduled and assigned {
2) op_i = select_node();
3) $\{(M_i, C_i)\}$ = set of (module, control step) pairs to/in which op_i can be assigned/scheduled;
4) compute

$$\text{cost}(op_i, M_i, C_i) = \alpha * \text{cost}_{Test}(op_i, M_i, C_i) + \beta * \text{cost}_{RU}(op_i, M_i, C_i) + \gamma * \text{cost}_{Flex}(op_i, M_i, C_i);$$

5) select (M_i, C_i) with the **minimum** cost;
6) assign inputs(op_i) to scan registers/register files;
7) update clique graph and data-path graph;
8) update ASAP of all operations not yet scheduled;
9) }
10) assign all variables in register files to registers;

1) Testability Cost: The objective of associating a testability cost with a schedule and assignment for an operation is to measure the extent to which the schedule and assignment affects the testability of the data path, namely by forming loops in the data path. Since an assignment can introduce three types of loops, an assignment loop, a false loop or a register file clique, we derive a cost function which consists of the costs associated with each type of loops formed, and the scan registers that may have to be expended to break the loops.

A measure of the testability cost is given by (1). The first component of the cost measure, $(\text{size}_{assign\_loop} + \text{cost}_{assign\_scan})$, is the cost due to formation of assignment loops, where $\text{size}_{assign\_loop}$ is the length of loops formed, and $\text{cost}_{assign\_scan}$ is the cost of using some existing or new scan registers to break the loops. The second and third components deal with the other two types of loops, false loop and register file clique, while cost_{seq_depth} measures the increase in sequential depth due to the assignment.

$$\begin{aligned}\text{cost}_{testability} = {} & (\text{size}_{assign\_loop} + \text{cost}_{assign\_scan}) \\ & + (\text{size}_{false\_loop} + \text{cost}_{floop\_scan}) \\ & + (\text{size}_{clique} + \text{cost}_{clique\_scan}) \\ & + \text{cost}_{seq\_depth}. \end{aligned} \quad (1)$$

Before we discuss how to compute the costs due to forming the loops, we discuss how to efficiently detect the formation of such loops during module assignment.

Detecting Formation of False Loop

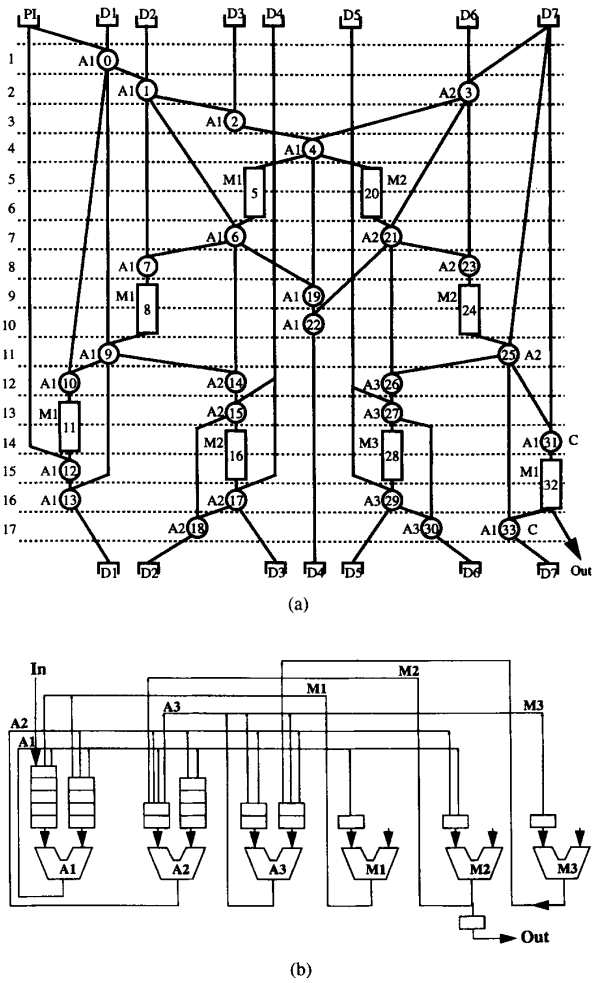
We briefly discuss how formation of a false loop by an assignment is detected. As module assignment progresses, a data path graph (DPG) is maintained, as indicated in the procedure **schedule_and_assign()** . Each node of the DPG represents a module. An edge (M_i, M_j) represents an interconnect from module M_i to module M_j in the data path, which is not broken by a scan register. Let M_i represent the module to which operation op_i is assigned. When an operation op_k is assigned to module M_k, for each operation op_i which sends data to operation op_k without going through a scan register, an edge will be added from module M_i to module M_k in the DPG.

For each edge (M_i, M_k) to be added to the DPG, if there exists a path in the DPG from M_k to M_i, a false loop will be introduced by assigning operation op_k to module M_k. Consequently, while assigning op_k to M_k , checking for paths from M_k to each module M_i that has been assigned to operation $op_i \in$ fanin(op_k), suffices to detect any false loop that may be introduced by the assignment.

Detecting Formation of Register File Clique

Let a CDFG edge (op_i, op_j) be termed a self-loop variable if both the operations op_i and op_j are assigned to the same module. The self-loop variables are associated with a register file (left or right) of a module. These variables are ultimately assigned to self-loop register(s) of the register file. If the lifetimes of the self-loop variables do not overlap, they are assigned to a single self-loop register. However, when the lifetimes overlap, they need to be assigned to multiple self-loop registers, leading to the formation of register file cliques in the S-graph.

Fig. 6 shows the variables that were assigned to the left register file of module $A1$, while assigning the operations of EWF shown in Fig. 8(b). The variables (op_i, op_j), their lifetimes, the source module (module assigned to op_i), and the register assignment are shown. Notice that the self-loop variables (0,10), (1,7), (6,19) and (33,31) are all alive in the 7th control step, and need to be assigned to four different self-loop registers. Since (PI,12) is also alive in the 7th control step, another register is required. Assigning the rest of the variables as shown in Fig. 6 produces the left register file shown, with 5 self-loop registers in the left register file of module $A1$.



(a)

(b)

Fig. 8. Fifth order wave digital elliptical filter (EWF), synthesized for resource utilization, satisfying performance constraint (17 control steps). (a) Scheduled and assigned CDFG. (b) Data path.

To detect formation of register file cliques during assignment, we maintain information of the lifetimes of the self-loop variables for each module M_i. When an assignment is made of operation op_i to module M_i, the input variables of op_i are checked for overlapping lifetimes with existing self-loop variables of M_i. A register file clique of size $(m + k)$ will be formed if a clique of size m existed for module M_i, and k input variables of operation op_i have lifetimes overlapping with the existing self-loop variables of M_i.

Cost Due to Formation of Assignment Loop

The cost due to formation of assignment loops is computed as follows. We first check whether assigning operation op_i to module M_j creates an assignment loop by traversing the paths in the transitive fanin of operation op_i in the CDFG. For each loop created, attempt is first made to break the loop using any available scan register. If successful, then $\text{cost}_{available\_scan}$ is added to the cost of scan registers, $\text{cost}_{assign\_scan}$, depending upon whether the used scan register could have been used by some other operation in the same control steps. If the loop cannot be broken by any available scan register, and adding a new scan register is allowable by the user-specified limit of *Max_scan_regs_allowed*, a new scan register is used to break the loop. The cost of a new scan register, cost_{new_scan}, is added to $\text{cost}_{assign\_scan}$. Note that if a loop is formed by assignment but is broken by using a scan register, then $\text{size}_{assign\_loop} = 0$.

In case neither an available scan register can break the loop nor a new scan register can be used, the assignment loop will be formed, and left unbroken, in the data path. To discourage assigning the operation op_i to module M_j which leaves a loop in the data path, the size of the loop formed is added to the cost function; since no scan register is used, $\text{cost}_{assign\_scan} = 0$.

The computation of the cost associated with the assignment of op_i to module M_j is given below.

cost_assignment_loop(op_i, M_j)

1) $\text{size}_{assign\_loop} = \text{cost}_{assign\_scan} = 0$;
2) if (*assignment_loop_introduced*(opi_i, Mi_j))
3) for each loop {
4) if loop can be broken by available scan register
5) $\text{cost}_{assign\_scan} = \text{cost}_{assign\_scan} + \text{cost}_{available\_scan}$;
6) else if (*#available_scan_regs* < *Max_scan_regs_allowed*) {
7) add new scan register to available scan registers;
8) $\text{cost}_{assign\_scan} = \text{cost}_{assign\_scan} + \text{cost}_{new\_scan}$;
9) } /(end if */
10) else /* cannot use any new scan registers; allow loop to be formed */
11) $\text{size}_{assign\_loop} = \text{size}_{assign\_loop}$+ size of loop introduced;
12) }/* end for */

The cost due to the formation of false loops and cliques is computed in a way similar to assignment loops. The increase in sequential depth due to an assignment can be computed by traversing the transitive fanins of the operation being assigned.

2) Resource Utilization Cost: While the primary goal addressed in this paper is generating testable designs, achieving high resource utilization should be simultaneously addressed. The area overhead for synthesizing the testable design should be minimal, so that our approach has a significant advantage over gate level design for testability schemes [9], [26], [11]. We briefly outline the criteria to achieve high resource utilization.

1) The most difficult operations for scheduling and assignment (the operations which are likely to require additional modules which may be underutilized) should be handled first, while the number of alternatives is still high.
2) Special attention should be paid to interconnect. Introduction of interconnects which can not be easily reused later should be avoided. Strong preference is given to local interconnect over global interconnect.
3) Registers are also an important part of the implementation cost. Any introduced register should have high likelihood to be effectively reused later.

We establish two simple, yet effective, criteria which are used to predict whether an interconnect will be local or global

after the physical synthesis of the design. An interconnect from an unit to itself will remain local after placement and routing. The second criteria is based on the observation that the greatest difficulty in routing often arises due to high congestion in some areas of the chip. To avoid congestion, during the interconnect assignment and allocation phase, we try to limit the number of interconnects which originate from or go to a particular register file.

For a particular asignment and schedule choice for an operation, we assign cost to the following resources that may be used by the choice, in an increasing order: 1) new register; 2) new local interconnect; and 3) new global interconnect.

When more than one scheduling and assignment decision have the same hardware cost, preference is given to a decision which introduces a new resource with higher likelihood for later reuse. Reusability of a new resource is calculated by counting how many unscheduled and nonassigned CDFG nodes can use the resource.

3) Flexibility Cost: The scheduling and assignment framework presented implies that scheduling and assignment decisions are, in some sense, made locally considering only nodes which are currently candidates for scheduling (i.e., has the smallest slack amongst nonscheduled nodes). A way to avoid local optima in favor of a globally optimum solution is to use the knowledge of how a particular scheduling and assignment decision influences subsequent scheduling and assignments of the remaining CDFG nodes.

To improve the chances of reaching a globally optimum solution, we use the flexibility cost as the third component of the scheduling and assignment cost. The flexibility cost component measures to what extent scheduling and assignment of an operation op_i to a particular control step and particular execution unit adversely affects the number of possibilities for scheduling and assignment of the unscheduled operations. Note, that due to data and control dependency edges, assignment and scheduling of one operation in the CDFG can have a number of consequences for assignment and scheduling of other operations. The DDCG graph provides convenient, fast and accurate information for characterizing the flexibility cost. The flexibility cost of the assignment of the operation op_i to the module M_i and control step C_i is proportional to the reduction of the weighted sum of compatibility edges in DDCG after the assignment of op_i to (M_i, C_i). The flexibility cost can be calculated in run time proportional to the square of the number of nodes in the DDCG.

VI. Experimental Results

The algorithms presented in this paper have been implemented in the the Behavioral Test Synthesis system, *BETS*, requiring approximately 5700 lines of C code. *BETS* has been integrated in the HYPER high-level synthesis system [38] from University of California, Berkeley. Hyper accepts the input behavioral specification in Silage, and converts it to a control-data flow-graph representation. All the synthesis routines, including the *BETS* algorithms, work on this flow-graph, and annotate it with the appropriate information. After the initial input specification has been accepted by HYPER, and module selection and estimation steps have been completed, *BETS* can be invoked to perform simultaneous scheduling and assignment to optimize both testability and resource utilization. The user is given options to perform appropriate trade-off between testability and resource utilization costs.

The following datapath-intensive benchmarks [21], [34], [43] were synthesized using conventional high-level synthesis techniques, and the behavioral test synthesis system, *BETS*, presented in this paper:

1) all-zero FIR wave digital filter (WDF);
2) 4th order cascade IIR Filter (4IIR);
3) MA Lattice Filter (MAL);
4) Speech Filter (Speech); and
5) 5th order elliptical wave digital filter (EWF).

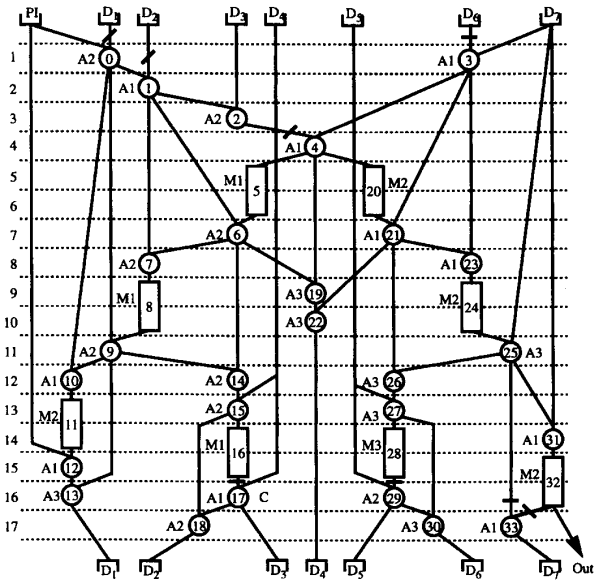
A. Synthesis of the Benchmarks: Performance and Hardware Characteristics

We illustrate the synthesis process of one of the benchmarks: the 5th order elliptical wave digital filter. We selected to implement its low pass version which is designed according to specification CCITT G712 PCM for use in telecommunication industry [13]. The example was simulated in bit-true mode, and verified that when 20 b for word-length and 6 b for coefficient are used fully conforms to required specification of having passband ripple from 0 Hz to 3 KHz of $\pm$ 0.125 dB and stopband attenuation of -14 dB at 4.0 KHz and -32 dB at 4.6 Mhz. Although the required sampling frequency of 16 KHz can be easily achieved with complete hardware sharing, we decided to schedule in critical path time of 17 control steps, assuming that an adder takes one and a multiplier takes two control cycles. The choice of time requirements for the adders and multipliers was influenced by the de facto standard benchmarking procedure in high-level synthesis literature.

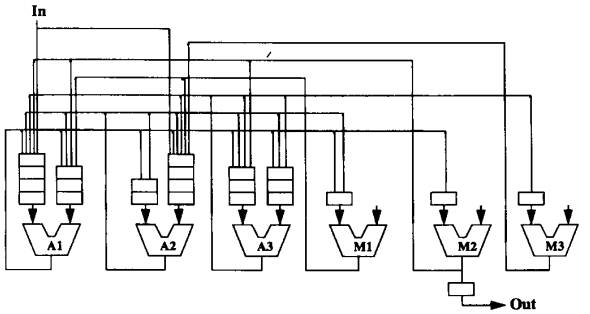
The schedule and assignment of EWF, and the corresponding data path, synthesized using the Hyper system [38], are shown in Fig. 8. The rectangles and the circles represent multiplication and addition operations, respectively. To facilitate sharing of registers, commutativity is applied to some addition operations, as indicated by "C" next to the operations in the CDFG's in Figs. 8 and 9. The detailed assignment of the register files of one of the modules, A1, is shown in Fig. 6. The schedule and assignment, and the corresponding data path, obtained by the new technique for simultaneous optimization of area and testability, are presented in Fig. 9. The 8 scan variables, selected by the *select_scan_variables()* procedure to break the CDFG loops, are indicated by cuts in Fig. 9(a). The scan variables needed only 3 scan registers, L1, R1 of module A1 (Fig. 6), and R1 of module A2, shown by the shaded registers in the data path in Fig. 9(b). A subsequent scheduling and assignment phase uses the 3 scan registers to produce a minimal loop data path, shown in Fig. 9(b).

In the sequel, **Orig** refers to the original implementation using *BETS*, when testability cost was not considered. **SFT** refers to the implementation using *BETS*, when testability was considered.

Table I shows various parameters of the original and SFT implementations after synthesis using *BETS*. The number of



(a)



(b)

Fig. 9. Fifth order wave digital elliptical filter (EWF), synthesized for **testability and resource utilization**, satisfying performance constraints (17 control steps). (a) Scheduled and assigned CDFG. (b) Data path.

TABLE I
CHARACTERISTICS OF DESIGNS SYNTHESIZED BY *BETS*

		CS	EXU	Reg	Mux	Inter	Area	CPU (sec)
Wave Digital Filter (WDF)	Orig	5	2A, 2M	10	8	11	4094	0.93
	SFT	5	2A, 2M	9	6	10	3872	0.95
4thIIR Filter	Orig	6	2A, 3M	12	12	20	5724	1.08
	SFT	6	2A,3M	12	9	9	4810	1.23
MAL Filter	Orig	7	2A, 2M	10	10	11	4422	1.03
	SFT	7	2A, 2M	9	8	10	4290	1.09
Speech Filter	Orig	17	2A, 3M	12	20	20	7486	2.12
	SFT	17	2A, 3M	12	9	9	6022	2.72
Elliptical Wave Digital Filter (EWF)	Orig	17	3A, 3M	23	29	20	8000	2.08
	SFT	17	3A, 3M	23	32	27	8784	5.25

execution units (**EXU**) and control steps (**CS**) needed are the same for both versions of the designs. Columns **Reg**, **Mux**, and **Inter** report the number of registers, multiplexers, and interconnects, respectively. In some cases, for example EWF, the SFT implementation needed a few more registers, multiplexers, and interconnects, indicating that testability improvement may result in a small increase in resource requirements. However, on the average, the area overhead is marginal.

Each RT-level implementation generated by *BETS* was processed by OASIS [17] to generate the gate-level circuits, using the *lib2.genlib* standard cell SCMOS 2.0 library [44]. Subsequently, OASIS [17] place and route tools were used to obtain the standard cell layout. The column **Area** reports the area of the gate-level implementations in terms of the number of cells (transistor pairs) of the *lib2.genlib* standard cell SCMOS 2.0 library [44].

Finally, the column **CPU (secs)** reports the CPU time (seconds) taken by *BETS* to synthesize the RT-level implementations from the behavioral specifications, on a Sun Sparcstation 2 workstation.

B. Test Generation and Partial Scan Results

The testability of the synthesized designs was evaluated using the gate-level sequential ATPG tool, HITEC [35]. Results are shown in Table II. The numerical suffix after the name in the first column corresponds to the wordsize of the implementation. For each design, besides the rows corresponding to the original and SFT implementations, the other two rows correspond to circuits obtained from the original design by a gate-level partial scan tool OPUS [11]. The row **GPS_c** refers to the circuit obtained from *Orig* by OPUS, after breaking all loops (except self-loops) using scan FF's. The row **GPS_n** indicates the circuit obtained from *Orig* when OPUS was constrained to use the same number of scan FF's as present in the SFT design. For each version of a design, the total number of FF's and the number of FF's scanned are reported. The results of running HITEC on the four versions of each design are shown in Table II. The total number of faults, the number of faults aborted by HITEC, the fault coverage and test efficiency achieved, and the ATPG time taken on a SUN Sparcstation 2 are reported.

Table II demonstrates that the SFT designs obtained by *BETS* were consistently more testable than the original designs obtained by traditional high-level synthesis techniques. To achieve the same level of testability, the gate-level tool OPUS needed to scan a significantly larger number of FF's *(GPS_c)* than required by the SFT designs. For instance, in the case of EWF.20, OPUS needed to scan 300 FF's to break all loops (except self-loops) and achieve the same level of testability as the SFT design, which had only 60 scan FF's.

Moreover, when OPUS was restricted to scan the same number of FF's *(GPS_n)* as in the SFT design, the testability achieved was significantly lower than that of the SFT design. In the case of EWF.20, while the SFT design achieved 100% test efficiency in only 233 CPU seconds, *GPS_n* could only achieve 40% in 14 895 CPU seconds.

VII. CONCLUSIONS

This paper presented a new behavioral test synthesis technique which exploits hardware sharing to minimize the number of scan registers needed to synthesize a minimal-loop data

TABLE II
EFFECT OF CONSIDERING TESTABILITY DURING HIGH-LEVEL SYNTHESIS

Circuit name	Circuit type	# of total FFs	# of scanned FFs	# of total faults	# of aborted faults	fault cov. (%)	test eff. (%)	ATPG CPU time (sec)
WDF.16	orig	160	0	5798	286	91	95	74346.8
	GPS_c	160	48	5798	2	96	100	525.5
	GPS_n	160	16	5798	6	96	100	1515.1
	SFT	144	16	5534	3	96	100	754.7
4IIR.16	orig	192	0	7976	7709	0	3	20095.5
	GPS_c	192	64	7976	3	97	100	820.3
	GPS_n	192	16	7976	6762	12	15	17205.9
	SFT	192	16	6776	10	96	100	377.7
MAL.16	orig	160	0	6192	191	93	97	1159.6
	GPS_c	160	48	6192	2	97	100	63.7
	GPS_n	160	16	6192	4	96	100	189.5
	SFT	144	16	5928	5	96	100	86.1
Speech.20	orig	240	0	10004	9677	0	3	23883.7
	GPS_c	240	60	10004	3	97	100	163.9
	GPS_n	240	20	10004	7621	20	24	19593.2
	SFT	240	20	8514	22	96	100	192.8
EWF.8	orig	184	0	3698	3515	2	5	79664.5
	GPS_c	184	121	3698	0	97	100	38.3
	GPS_n	184	24	3698	259	90	93	7166.8
	SFT	192	24	4108	5	97	100	64.9
EWF.16	orig	368	0	9088	8879	0.3	2	212599.1
	GPS_c	368	240	9088	0	98	100	209.1
	GPS_n	368	48	9088	4932	44	46	14086.1
	SFT	384	48	9791	0	97	100	183.3
EWF.20	orig	460	0	10364	10077	1	3	>72h
	GPS_c	460	300	10364	0	98	100	309.8
	GPS_n	460	60	10364	6216	38	40	14894.5
	SFT	480	60	10916	16	98	100	233.2

path. Novel algorithms have been proposed to select a minimal number of scan registers to break CDFG loops, and reuse the scan registers during scheduling and assignment to avoid the formation of further loops in the data path. The proposed technique has been implemented in the *BETS* behavioral test synthesis system. *BETS* achieves high testability, without compromising resource utilization or performance, for all the benchmark designs synthesized. Experimental results demonstrate the superiority of selecting partial scan registers during high-level synthesis over partial scan selection at the gate level.

ACKNOWLEDGMENT

The authors would like to acknowledge V. Gangaram for implementing *BETS*, the behavioral test sythesis system presented in this paper. They would like to thank Prof. J. Patel and Dr. V. Chickermane for providing them with HITEC and OPUS tools. They would also like to acknowledge S. Bhattacharya for helping them with OASIS.

REFERENCES

[1] M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital Systems Testing and Testable Design*. New York: Computer Science, 1989.
[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley, 1974.
[3] L. Avra, "Allocation and assignment in high-level synthesis for self-testable data paths," in *Proc. Int. Test Conf.*, 1991, pp. 463–472.
[4] L. Avra and E. McCluskey, "Synthesizing for scan dependence in built-in self-testable designs," in *Proc. Int. Test Conf.*, Oct. 1993, pp. 734–743.
[5] S. Bhattacharya, F. Brglez, and S. Dey, "Transformations and resynthesis for testability of RT-level control-data path specifications," *IEEE Trans. VLSI Syst.*, vol. 1, no. 3, pp. 304–318, Sept. 1993.
[6] A. Chandrakasan, M. Potkonjak, J. Rabaey, and R. W. Brodersen, "HYPER-LP: A system for power minimization using architectural transformations," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1992, pp. 300–303.
[7] C.-H. Chen and D. G. Saab, "Behavioral synthesis for testability," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1992, pp. 612–615.
[8] C.-H. Chen, C. Wu, and D. G. Saab, "BETA: Behavioral testability analysis," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1991, pp. 202–205.
[9] K.-T. Cheng and V. D. Agrawal, "A Partial Scan Method for Sequential Circuits with Feedback," *IEEE Trans. Comput.*, vol. 39, no. 4, pp. 544–548, Apr. 1990.
[10] V. Chickermane, J. Lee, and J. H. Patel, "Addressing design for testability at the architectural level," *IEEE Trans. Computer-Aided Design*, vol. 13, no. 7, pp. 920–934, July 1994.
[11] V. Chickermane and J. H. Patel, "A fault oriented partial scan design approach," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1991, pp. 400–403.
[12] S. S. K. Chiu and C. Papachristou, "A built-in self-testing approach for minimizing hardware overhead," in *Proc. Design Automation Conf.*, 1991, pp. 282–285.
[13] L. Claesen, F. Catthoor, D. Lanneer, G. Goossens, S. Note, J. Van Meerbergen, and H. De Man, "Automatic synthesis of signal processing benchmark using the CATHEDRAL silicon compilers," in *Proc. IEEE 1988 Custom Integrated Circuits Conf.*, 1988, pp. 14.17.1–14.7.4.
[14] S. Dey and M. Potkonjak, "Non-scan design-for-testability of RT-level data paths," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1994, pp. 640–645.
[15] ——, "Transforming behavioral specifications to facilitate synthesis of testable designs," in *Proc. Int. Test Conf.*, Oct. 1994.
[16] S. Dey, M. Potkonjak, and R. Roy, "Exploiting hardware sharing in high level synthesis for partial scan optimization," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1993, pp. 20–25.
[17] K. Kozminski, Ed., *OASIS Users Guide*. Research Triangle Park, NC: MCNC, 1991.
[18] H. De Man *et al.*, "Synthesis of DSP systems at Leuven," in *Proc. IEEE ICCD*, 1987, pp. 133–145.
[19] M. R. Garey and D. S. Johnson, *Computers and Intractibility*. San Francisco: Freeman, 1979.
[20] C. H. Gebotys and M. I. Elmasry, "Integration of algorithmic VLSI synthesis with testability incorporation," *IEEE J. Solid-State Circuits*, vol. 24, no. 2, pp. 458–462, Apr. 1989.
[21] R. A. Haddad and T. W. Parsons, *Digital Signal Processing: Theory, Applications and Hardware*. New York: Computer Science, 1991.
[22] L. Hagen and A. B. Kahng, "A new approach to effective circuits clustering," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1992, pp. 422–426.
[23] H. Harmanani and C. Papachristou, "An improved method for RTL synthesis with testability tradeoffs," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1993, pp. 30–35.
[24] J. D. Kahn, N. Linial, N. Nisan, and M.E. Saks, "On the cover time of random walks on graphs," *J. Theoretical Probability*, vol. 2, no. 1, pp. 121–128, 1989.
[25] R. Karri and A. Orailoglu, "Scheduling with rollback constraints in high-level synthesis of self-recovering ASICs," in *Proc. 22nd Int. Symp. Fault-Tolerant Computing*, 1992, pp. 519–526.
[26] D. H. Lee and S. M. Reddy, "On determining scan flip-flops in partial-scan designs," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1990, pp. 322–325.
[27] T. C. Lee, N. K. Jha, and W. H. Wolf, "Behavioral synthesis of highly testable data paths under non-scan and partial scan environments," in *Proc. Design Automation Conf.*, 1993, pp. 292–297.
[28] T. C. Lee, W. Wolf, N. K. Jha, and J. M. Acken, "Behavioral synthesis for easy testability in data path allocation," in *Proc. Int. Conf. Comput. Design*, 1992, pp. 29–32.
[29] T. C. Lee, W. H. Wolf, and N. K. Jha, "Behavioral synthesis for easy testability using data path scheduling," in *Proc. Int. Conf. Computer-Aided Design*, 1992, pp. 616–619.
[30] H. M. Levy and Jr. R. H. Eckhouse, *Computer Programming and Architecture: The VAX*, 2nd ed. Bedford, MA: Digital, 1989.
[31] A. Majumdar, K. Saluja, and R. Jain, "Incorporating testability considerations in high-level synthesis," in *Proc. Int. Symp. Fault-Tolerant Computing*, 1992, pp. 272–279.
[32] M. C. McFarland, A. C. Parker, and R. Camposano, "The high-level synthesis of digital systems," *Proc. IEEE*, vol. 78, no. 2, pp. 301–317, 1992.
[33] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994.
[34] S. K. Mitra and J. F. Kaiser, *Handbook for Digital Signal Processing*. Somerset, NJ: Wiley, 1993.
[35] T. M. Niermann and J. H. Patel, "HITEC: A Test Generation Package for Sequential Circuits," in *Proc. EDAC*, 1991, pp. 214–218.

[36] C. Papachristou, S. Chiu, and H. Harmanani, "SYNTEST: a method for high-level SYNthesis with self-TESTability," in *Proc. Int. Conf. Comput. Design*, Oct. 1991, pp. 458–462.
[37] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufman, 1989.
[38] J. Rabaey, C. Chu, P. Hoang, and M. Potkonjak, "Fast prototyping of data path intensive architectures," *IEEE Design and Test of Computers*, vol. 8, no. 3, pp. 40–51, 1991.
[39] J. Steensma, W. Geurts, F. Catthoor, and H. De Man, "Testability analysis in high level data path synthesis," *J. Electron. Testing: Theory Applicat.*, vol. 4, no. 1, pp. 43–56, Feb. 1993.
[40] L. Stok, "False loops through resource sharing," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1992, pp. 345–348.
[41] T. Thomas, P. Vishakantaiah, and J. A. Abraham, "Impact of behavioral modifications for testability," in *Proc. 12th IEEE VLSI Test Symp.*, Apr. 1994, pp. 427–432.
[42] S. G. Tucker, "Microprogram control for the system/360," *IBM Syst. J.*, vol. 6, no. 4, pp. 222–241, 1967.
[43] R. A. Walker and R. Camposano, *A Survey of high-level synthesis systems*. Boston, MA: Kluwer Academic, 1991.
[44] S. Yang, "Logic synthesis and optimization benchmarks, user guide version 3.0," presented at the Int. Workshop Logic Synthesis, MCNC, Research Triangle Park, NC, May 1991.

Miodrag Potkonjak (S'90–M'91), for a photograph and biography, see p. 30 of the January 1995 issue of this TRANSACTIONS.

Sujit Dey (S'90–M'91), for a photograph and biography, see p. 546 of the May 1995 issue of this TRANSACTIONS.

Rabindra K. Roy (S'84–M'92), for a photograph and biography, see p. 546 of the May 1995 issue of this TRANSACTIONS.