



ELSEVIER

INTEGRATION, the VLSI journal 28 (1999) 55–99

INTEGRATION
the VLSI journal

www.elsevier.com/locate/vlsi

Unified data path allocation and BIST intrusion[☆]

Katzalin Olcoz*, Francisco Tirado, Hortensia Mecha

Department of Computer Architecture and Automatic control, University Complutense, Madrid, Spain

Received 16 January 1998

Abstract

This paper deals with an approach to the automatic synthesis of self-testable data paths. A self-testable data path contains some test registers that increase its area. Classical approaches synthesizing minimum area data paths and then adding minimum number of test registers to it do not lead to data paths with minimum global area. Testability consideration during synthesis makes design search more efficient and hence can possibly find self-testable data paths with minimum area. We present a model to evaluate the testability of data paths that is used when data path allocation is being done. Moreover, we propose some heuristics that guide the design space search during allocation, to save exploration time. When each allocation decision has to be made, an implementation alternative is chosen according to the area and testability increments that the alternative produces, so that the area is only increased when the testability gain is worth it. © 1999 Elsevier Science B.V. All rights reserved.

Keywords: High-level synthesis for testability; Synthesis for BIST; Automatic synthesis of testable data paths; Allocation of testable data paths

1. Motivation and previous work

Recent advances in VLSI technology are motivating changes in the traditional methods of design and test, that can no longer be undertaken separately. From the design point of view, the growing complexity of the designs that can be included in a chip demands an increase in the level of abstraction of CAD tools. *High-level synthesis* (HLS) tools are suited to deal efficiently with such complex designs and they also speed up the design process (which, according to recent studies

[☆]This work has been partially supported by the EC project HCM CHRX-CT94-0459 and by the Spanish Government grant TIC 96/1071.

* Fax: + 34-91-394-44-69.

E-mail address: katzalin@dacya.ucm.es (K. Olcoz)

in [1], is at least as important as optimizing circuit area or speed). The high-level synthesis of a digital system can be stated as follows: given an algorithmic level specification of the behavior of a system and a set of constraints and goals to be satisfied, a register-transfer level (RTL) structure must be found that implements the specified behavior while meeting the goals and constraints [2]. The synthesis process involves navigation through the space of possible designs with the specified behavior (e.g. the *design space*), making appropriate tradeoffs, until the best solution satisfying the constraints is reached.

From the test point of view, traditional test methodologies are not able to cope with the improvements in IC design [3]. As a result, the cost of the test process, due to the test time and to the cost of the automatic test equipment (ATE), is growing so that it is between 30 and 40% of the total production cost [4]. *Design for testability* (DFT) methodologies emerge to reduce testing cost with no IC quality loss [5]. Testability is defined as the facility to generate and apply the test and design modifications are proposed that improve the testability. The DFT methodologies range from ad hoc techniques to the structured approaches such as the scan design and the *built-in self-test* (BIST) design. The later includes in the same chip the circuits that perform the test (*test circuits*) with the circuits that correspond to the design (*functional circuits*). Thus, the test process is carried out within the chip. There are several advantages to this approach, namely: ATE cost reduction, test time saving and taking advantage of test hierarchy (if the design is made of complex components such as cores their BIST capabilities can be re-used for the tests of the IC and the system) [6]. The main drawbacks are an increase on the area due to the test circuits added to the design (test area), the decrease on operating speed that can be caused by the test circuits and the need of automatic tools to insert the test circuits into the IC. Fortunately, the test area does not grow in proportion to the area of functional circuits (functional area) and BIST is becoming less expensive every day [7]. So there is a great effort in developing tools for the insertion of design for testability structures that improve the testability at modest area overhead and trying not to introduce delays.

The above-mentioned need for DFT insertion tools is put together with the need for synthesis tools, motivating the development of *synthesis for testability* tools that offer solutions for both problems of design and test. These tools provide best results when they are merged to perform a global optimization process.

1.1. Previous work

Gate-level synthesis for testability has been the subject of intense research, concurrent with research into synthesis to satisfy area, timing and, more recently, power constraints [8]. Testability insertion techniques have emerged, and testability analysis and insertion of these testability structures have been automated. The need for fast time-to-market and increased productivity are driving the trend towards high-level design. Several RTL synthesis approaches have been proposed to generate implementations that are easily testable either by means of sequential automatic test pattern generation (ATPG, required for circuits with partial scan, test point insertion or no test hardware), combinational ATPG (only for full scanned circuits) or BIST methodologies.

Most of them concentrate on improving the testability of the data path, assuming that the controller can be made testable independently, and that its outgoing control signals to the data path are fully controllable in test mode.

The approaches are classified according to the moment of the synthesis in which testability is analyzed and improved. Since HLS is split into scheduling and allocation [2], the insertion of testability structures can be performed *after* scheduling and allocation, *before* scheduling and allocation or *during* scheduling and allocation.

The scheduling of operations to control steps binds each operation of the behavioral description to a specific control step, determining the tradeoff between area and speed of the circuit. Allocation consists of generating a data path of minimum area starting from the scheduled description of the behavior. It involves allocation of operations to functional units (FUs or modules), results to registers and data transfers to connections. For a given operation there are usually FUs of different types in the library, so module allocation consists of selection of the type of FU and of the instance. When both tasks are split, the latter task is called binding, while the former is module allocation.

First we will present the systems that perform *testability analysis after allocation*. Once an RTL description has been synthesized, there are several alternatives to enhance its testability [9,10]. Mitra et al. [9] add BIST structures after synthesis. Since this approach does not have the flexibility to change interconnections, it may be of limited help to improve the testability of a design.

Bhattacharya et al. [10] perform transformations on an existing RTL design in order to remove all false paths and get a fully testable design. However, their approach is constrained by the quality of the initial RTL design.

A second group of systems *modify the original behavioral description* to make the resulting implementation more testable than the one generated from the original description [11–13]. Behavioral testability enhancement approaches targeting no particular testability technique (such as [11]) analyze the behavioral description to detect hard-to-test areas. Based on the testability analysis, test statements, which are executed only in test mode, are added to improve the controllability and observability of all the variables in the description. The modified behaviors produce circuits with higher fault coverage and efficiency than the original description, at modest area overhead.

A behavioral description can also be modified to make it more amenable to the synthesis for testability techniques. Dey and Potkonjak [12] transform the control-data flow graph (CDFG) by adding operations which do not change the original computation, but enable more sharing of scan registers (they eliminate resource sharing bottlenecks like overlapping lifetimes) so as to minimize the number of scan registers needed. Potkonjak et al. [13] apply more complex transformations to the behavior so that synthesizing a testable data path from the transformed specification requires fewer scan registers than the ones needed for the original specification.

Finally, some other systems have focused on *scheduling and allocation methods that avoid creating some hard-to-test structures*. The objective of behavioral synthesis for sequential ATPG is usually to avoid the creation of loops, since they contribute significantly to the difficulty of sequential ATPG [14–19].

High-level synthesis for testability approaches use the number of loops (except self-loops) and the sequential depth of registers as a guide to synthesize testable implementations from behavioral descriptions while preserving the performance and area constraints of the design. In [14–16] the variables of the CDFG are assigned to maximize the number of (I/O) registers connected to primary I/O. Also, the number of loops and the sequential depth from an input register to an output register are minimized during register binding.

Potkonjak et al. [17] break loops formed by data dependencies in the CDFG by selecting a set of scan variables in such a way that each CDFG loop has a scan variable and by assigning each scan variable to a scan register. The selected scan variables can share scan registers. Two measures (loop cutting effectiveness and hardware sharing effectiveness) are used to select a set of scan variables that can be maximally shared.

In [18], Mujumdar et al. perform binding using rules similar to those in [15] but also try to minimize the number of self-loops. In [19], they maximize the controllability and observability of registers during register binding to minimize the number of scan registers.

To make a design self-testable using the pseudo-random BIST methodology, it has to be reconfigured during test mode into a set of acyclic logic blocks. Each block has a pseudo-random test pattern generation register (TPG) at each of its inputs, and a signature analyzer register (SA) at each of its outputs [20]. BIST requires reconfiguration of some functional register as a TPG or SA. A third kind of register, called BILBO [21], can be configured alternatively as a TPG and an SA. But a register cannot be configured both as a TPG and an SA simultaneously, unless it is implemented as a concurrent BILBO (CBILBO) [22], which is very expensive in terms of area and delay penalties. Hence, a *self-adjacent* register, which serves both as input and as output of a logic block, poses a problem, since it may have to be implemented as a CBILBO.

The systems that perform *synthesis for BIST* try to generate self-testable data paths with low area overhead in different ways. The objective of some of them [23,24] is to minimize the formation of self-adjacent registers: Avra [23] assumes that every self-adjacent register will have to be implemented as CBILBO. Given the scheduling and the allocation-binding of operations to modules, an additional constraint is imposed to the register allocation that avoids the formation of self-adjacent registers.

Papachristou et al. [24] restrict the data path architecture to accomplish the same goal. The basic building block, TFB, used to map both an operation and the result generated consists of an ALU with a multiplexer at each input and a test register at the output. Allocation is performed to minimize the number of TFBs with the restriction that no self-adjacent register is formed (the output of a TFB cannot be connected to any of its inputs). In [25] the model is extended so that the extended TFB has multiple input as well as output registers. Only one of the output registers have to be configured as SA in test mode, so the rest of them can be self-adjacent and still no CBILBOs are required.

Our approach differs from theirs in that no constraints are imposed neither on the data path structure nor on the allocation problem. So, the design space is not limited.

BIST overhead can be reduced by not only minimizing the number of CBILBO registers that are to be used, but also the number of TPGs and SAs needed to test all the data path modules. After the scheduling and allocation of modules, register allocation is done in [26] to maximize the number of modules for which a register is an input register (it can act as TPG of the module) and the number of modules for which a register is an output register (it can act as SA). Exact conditions under which a self-adjacent register needs to be a CBILBO are given, and register allocation avoids them whenever possible.

In the most general BIST scheme, a *test path* through which test data can go from the TPGs to the SA at the output of a logic block may pass through several functional units. If two test paths share the same hardware a conflict is created, forcing the need for multiple test sessions. Scheduling and binding techniques which use test conflict estimates to generate data paths which require

minimal number of test sessions and hence have maximal test concurrency are proposed in [27].

1.2. Our approach

In this section we present a global perspective of our approach to the synthesis of testable data paths. First, we place our work among the different alternatives. So, it must be classified according to the moment of the synthesis in which testability structures are inserted and according to the DFT methodology chosen. Then, we will highlight the most relevant features of our algorithms in an intuitive way with the help of an example.

1.2.1. Main features

From the analysis of the different synthesis for testability systems we come to the conclusion that testability improvement will be constraint by the quality of the design unless testability is analyzed and improved before the RTL structure of the design is finished, that is, unless testability is improved either before or during synthesis.

The first choice consists of improving the testability of the behavior and then perform synthesis with no regard to testability. As a result, the design space is not explored globally and no tradeoffs between area and testability are possible. *Since we want testability to be taken into account when design choices are taken we will analyze and improve the testability during HLS*, which leads us to the need of a more precise definition of the HLS tasks.

HLS tasks are interdependent, so the more the relationships between them are taken into account, the better the designs synthesized. Not only the allocation sub-tasks are related but also the scheduling of operations to control steps is closely related to the type and number of FUs needed to implement the operations. To bear in mind this dependence, some systems perform scheduling together with module allocation. Thus, they determine simultaneously the control step and the type of module each operation will be assigned to. Then, they perform binding as the mapping of operation and variables to particular instances of the modules and registers.

The main drawback of this approach is that module allocation is performed without regard to module binding, that determines the RTL structure of the data path and the interconnect area, that is, the area of multiplexers and wires. As a result, data paths can be obtained with non-minimum number of multiplexers and wires. Measures of the area of real circuits have shown that interconnect area can take up substantial circuit area, so that performing allocation without regard to binding can lead to non-minimum data paths. For instance, the area of a multiplexer can be similar to or bigger than that of a register or even a simple module (e.g. a ripple carry adder). Thus, the smaller design is not always made up of the minimum set of modules and registers. Some extra modules and/or registers can be included if that decreases the number of multiplexers.

In our approach the relationships between module allocation and the other two subproblems (scheduling and binding) are both borne in mind. So, scheduling performs a tentative module allocation. Then, the final module allocation and binding are performed starting with this initial module allocation. During allocation, precise estimates of the data path area are used to make tradeoffs between module and interconnect area in order to obtain the data paths with minimum total area. *This makes allocation an appropriate step in the design process to analyze and increase the testability of designs.*

The DFT methodology chosen is *partial intrusion of BIST registers*. The concept of I-path [28] is also used to take maximum profit of each test register. An *I-path* is a path in the data path that allows data to be transferred unchanged. We will generalize this concept and define test paths that can be used for the transfer of test data such as test vector and results.

The algorithm simultaneously allocates functional hardware and test hardware so that it allows global search of the design space. To find the self-testable solutions with minimum total area some test registers are created *when* the data path is being developed and the structure of the data path is determined in such a way that those test resources are used intensively. So, test paths passing through FUs are exploited to maximize the use of TPGs and SAs.

Given a FU, its *configuration for the test* consists of the test paths from the TPGs to the inputs of the FU and from the output of the FU to the SA. Since those paths cannot share any element, elimination of self-adjacent registers alone is not enough to guarantee high testability, which requires a broader constraint. So, allocation is performed to avoid hardware sharing among the paths in the configuration for the test of any FU whenever possible. Thus, self-adjacent registers are allowed when they do not need to be CBILBO, that is, they are not in more than one path in the configuration for the test of any FU.

To summarize, module and register allocation and binding are performed simultaneously together with intrusion of BIST registers, so the design space of solutions is searched globally. The objective is to obtain circuits that: (i) are self-testable and (ii) have a minimum area (including the area of FUs, registers and connections and the test area). Therefore, the global area of the data path, rather than the number of BIST registers, is minimized, allowing tradeoffs between module, register, interconnect and BIST area.

1.2.2. Example

A simple example will be introduced to illustrate some of the advantages of our approach to the problem of data path allocation. It shows that assigning test resources together with allocation helps to maximize their use, so that the data paths synthesized are smaller compared to designs synthesized for minimum area and afterwards improved for testability. Two allocations have been performed for the CDFG in Fig. 1(a).

For this small example the only choices that can be taken during allocation are module allocation for the nodes +3 and +4 and register allocation for the variables $u3$ and $u4$.

The data path in Fig. 1(b) is obtained when allocation takes into account minimizing area and maximizing the use of existing test resources. For instance, module allocation for +3 can choose between adder +1 and adder +2. The latter choice leads to minimum area increment because $u2$ is reused as right operand, but no testability increment is obtained. On the contrary, allocating the node +3 to the adder +1 causes a bigger area increment (due to the addition of an extra 2-to-1 multiplexer and a wire) but helps in the creation of the test paths needed for test generation for the left input of adder +1 and test response compaction of the output of adder +2. As a result, +3 is allocated to adder +1. This allocation increments interconnect area but decrements test area, corresponding to a good tradeoff between area and testability.

The resulting circuit can be self-tested in two test sessions even though it only contains two TPG (grayed) and an SA register. The configurations for the test of both adders are shown to the right of the data path. Note that despite the fact that the register storing variable x cannot be used for any

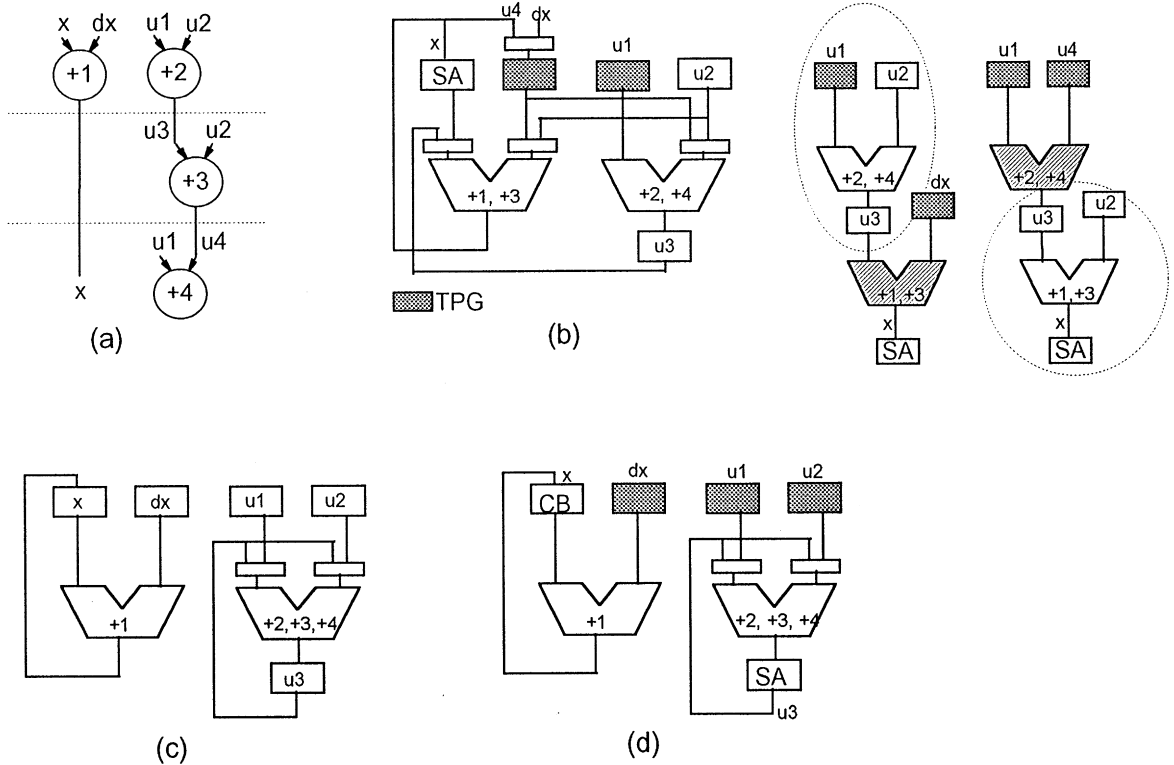


Fig. 1. (a) Scheduled CDFG, and two allocation approaches (b), (c).

other result, it is useful as SA for both adders due to the path created by the allocation of +3 to adder +1.

The register storing variable x is self-adjacent due to a data dependency in the CDFG and the one storing dx and $u4$ is self-adjacent because of hardware reuse. But no CBILBO register is required because no element (module or register) appears in more than one of the test paths of any test configuration.

On the other hand, in Fig. 1(c) we can see the data path reached by a minimum area allocation for the same CDFG. It has two 2-to-1 multiplexers and two wires less than the one in Fig. 1(b) and it contains no test registers. It also has two self-adjacent registers (the ones storing x and $u3$).

When the minimum number of BIST registers is added to this data path, we reach the self-testable data path in Fig. 1(d). It needs three TPG (grayed), an SA and a CBILBO register and can be tested in only one test session.

In order to compare the total area of both data paths we must require the test time to be the same for both, that is, one test session. The data path in Fig. 1(b) can be tested in only one test session if we add a TPG and an SA. As a result, the data path in Fig. 1(b) will have 3 TPG (x , $u1$, $u2$) and 2 SA ($u3$, dx) registers while the one in Fig. 1(c) needs 3 TPG, an SA and a CBILBO registers. Since the difference in interconnect area for both data paths is much smaller than the difference in test area, the latter data path is bigger than the first one presented. Besides, the data path in Fig. 1(b) succeeds in avoiding performance degradation, unlike data path in Fig. 1(c).

We introduced the main advantages of improving testability during synthesis. We saw that, although it initially seemed to be smaller than the data path in Fig. 1(b), the second design is bigger than the first one, once BIST registers are added to it.

The rest of the paper is organized as follows. First, the testability model used during allocation is described in Section 2. The general features of the algorithm for data path allocation are outlined in Section 3, and application of the testability model is also shown. The discussion in Sections 4 and 5 concentrate on the special features of the allocation of functional units and registers, respectively. The two most relevant features of the algorithm, the estimation of testability increment and the heuristics used for design space search guiding, are explained in each section. The paper concludes with experimental results and conclusions in Section 6.

2. Model of data path testability

Testability is usually defined as the ability to test a circuit. The data paths synthesized are to be tested using BIST methodologies. So, their testability depends on the facility to perform the different tasks involved in the test of a BIST circuit.

A model for the testability of data paths is formalized from this statement. The first step followed is characterization of these *test tasks* (model of the test). Then, the *ability* of the different data path elements to perform any test task is determined and stored in the library of modules (data path model). Finally, testability of any data path is modeled as the matching between these two features (test tasks and ability of the elements to perform them). The data path is testable if all the test tasks can be fulfilled successfully, that is, if there are elements with the ability to perform them all.

2.1. Test model

The test of a circuit consists of applying test vectors to every input of every component and observing the results obtained. BIST circuits are able to test themselves. This means that test vectors are generated inside the circuit and test results are also compacted inside the data path [5,20]. A common method of self-test generation is pseudo-random generation starting from an external test seed. It is also common to compact the results by signature analysis, producing a signature that is put out for external examination.

Therefore, BIST circuits have two operational modes: normal function mode and test mode. In test mode, some elements generate test vectors and some act as signature analyzers. A register configured, in test mode, as test pattern generator for an input of a module is called a *test source* of the module. The *test response destination* of the module is a register configured, in test mode, as test response compactor or signature analyzer of the output of the module.

Definition 1. Let i be a data path module with n input ports and one output port, the test of the module, $T(i)$, is made up of three sequential *test tasks*: initialization of test sources, application of test vectors to the module, and extraction of the test signature. The test application task is divided into $n + 1$ simultaneous sub-tasks: generation of test vectors for each input j of the module, $j = 1, \dots, n$; and compaction of test responses.

Table 1
Features of the data transfers that take part in the test

Type of task	I	G	C	O
Flow of data	Input	Input	Output	Output
Kind of data	Seed	Test vector	Test response	Signature
Amount of data	1	Many ^a	Many ^a	1
Potential (%)	100	Function of module ^a	Function of module ^a	100

^aThe amount of data to be transferred is the number of test vectors needed by the module or Complexity of the test of the module. It has been pre-computed (as will be explained below) and stored in the library.

We have also defined the following notation: $I(i)$ is initialization of the n test sources for module i . Every test source of module i must be initialized, but not necessarily at the same time. So, this task is split in a sequence of n non-simultaneous sub-tasks $I_1(i)$ to $I_n(i)$.

$G_1(i)$ to $G_n(i)$ are application of test vectors to the n inputs of the module, and $C(i)$ is compaction of test responses of module i . They must be done at the same time.

$O(i)$ is extraction of the signature for module i .

Thus, we define the test of a module as a sequence of tasks such as

$$T(i) = \{\{I_1(i)\}, \dots, \{I_n(i)\}, \{G_1(i), \dots, G_n(i), C(i)\}, \{O(i)\}\}.$$

Notice that the test tasks can be grouped into four different *types*: initialization of a test source (I), generation of test vectors (G), compaction of test responses (C) and extraction of the signature (O).

Each task is composed of a set of data transfers, that are clearly different for each type of task. Thus, a detailed analysis of those data transfers is needed to model their features, summarized in Table 1.

For instance, initialization of a test source consists of transmission of a test seed from an input port to the test source. On the other hand, generation of test vectors for one input of a module involves transmission of several test vectors from the test source to the module input. Notice that not only are the data involved of different *kind*, but also the *amount* of data to be transferred are quite different. Besides, transfers can be grouped into two sub-sets according to the data *flow direction*:

- *Input transfer*: during the transfer, data travel towards the module under test. Their prime feature is that test data are generated by the element that is the source of the transfer.
- *Output transfer*: during the transfer, data are carried away from the module. Test data are processed by the element that is the destination of the transfer (test responses are transformed into the test signature, and the signature is compared with the one of the correct circuit).

A deeper analysis of the test tasks results in the definition of an additional feature of test transfers. On the one hand, test vector initialization and test signature extraction consist of transmission of only one word of data. It is critical for test success that the exact data (seed or signature) arrives unchanged. On the other, test application consists of transmission of as many different

pseudo-random data as needed by the module. As long as such an amount of different pseudo-random data reach its destination, it is not so critical that they are unaltered.

Definition 2. The *transfer potential* needed by a data transfer t , $\text{pot}(t)$, is defined as the percentage of data that must arrive at the transfer destination so that the transmission of information is regarded as successful.

We apply this concept to the data transfers that make up the test tasks. The needs on transfer potential of a given task, t , of the test of a module i depend on the type of the task and the module, as displayed in Table 1. For initialization and extraction the data has to arrive unaltered at the destination, so they require 100% transfer potential. For test application, the number of different data that has to be transferred is the same as the number of test vectors needed. Thus, the potential depends on the complexity of test of the module that will be defined in the next section.

To summarize, the test of a module is a sequence of tasks of different types, which are characterized by four features. The test can be performed if the data transfers that make up the test tasks can be done successfully. In consequence, *the test can be represented by the set of data transfers with certain needs that have to be fulfilled.*

2.2. Data path model

The *data path* is a set of connected HW elements. The elements are instances of a given type taken from a library of modules. Thus, the data path is a hierarchical RTL structure.

The *library of modules* contains information about different features of its components. The first feature of any element, e_k , is the *class* of the element, C_k , defined according to its function in normal operation mode. There are four main classes of elements: storage elements, STO (registers, REG, and constants, CON), operational elements (functional units, FU), interconnecting elements, INT (multiplexers, wires and busses) and I/O elements (input ports, PI, and output ports, PO). That is:

$$C_k \in \{\text{STO}, \text{FU}, \text{INT}, \text{I/O}\}.$$

Registers are further classified into sub-classes according to their function in test mode. Functional units are arranged into sub-classes, usually called “types”, according to the set of operations they can perform.

Other features stored in the library are the design parameters of elements such as: area (expressed in gate-equivalent units, ge), delay (ns) and set of operations performed by each functional unit. In addition, elements have some test features, namely: test ability, test complexity and transparency.

Definition 3. The *ability for test* of an element is the function or functions that the element can perform in test mode. There is one ability for each different type of test task: initialization, generation, compaction and extraction.

Since initialization of a test source consists of transferring the test seed from the exterior, the data paths elements with initialization-ability are the input ports. Extraction of the test signature consists of transferring it to the exterior, so the elements with extraction-ability are the output

ports. The elements with generation-ability are the test sources (TPG or BILBO registers) and the elements with compaction-ability are the test destinations (SA or BILBO registers).

Definition 4. The *complexity of test* of a FU, i , $\text{Compl}(i)$, is the number of pseudo-random test vectors that must be applied to the inputs of the FU to have maximum fault coverage.

Its value has been computed by means of fault simulation of each element. A set of pseudo-random test vectors identical to the ones generated by the BIST registers in the library has been used as stimulus for the fault simulation.

It is needed to determine the amount of data to be transferred and the transfer potential of the test application task.

Definition 5. The *transparency* of the input i of an element e_k , $\text{trans}(e_k, i)$, is the percentage of data at the input that can be observed at the output once the rest of the inputs have been set to a given value. It is similar to the concept of transparency of functional modules introduced in [29].

Transparency is computed for the FUs in the library as explained in Appendix A.

In the library of modules the values of the parameters of every type of element are stored. On the other hand, the data path allows us to find out where each input of every instance of element originates and where the data at the output of every instance of element can be transmitted. The basic structure for data transmission in the data path is introduced next.

Definition 6. A *data transfer path*, p , is a directed non-cyclic sub-graph of the data path that is formed by a set of components interconnected so that data can be transferred from the first (*origin*) to the last one (*sink* of the path). This concept is a generalization of the I-path introduced in [28].

Distinguishing features of paths are:

1. *Elements*: set of elements (e_1, \dots, e_n) listed in the order they are traversed by data. To uniquely determine an element, e_k , we need to know its class (C_k), its number in the list of elements of that class (j) and the number of the input (l) that is fed by data from the previous element of the path (that is, the number of the input that *belongs* to the path).
2. *Sequential depth*, $\text{seq_d}(p)$: clock cycles that data take from the origin to the sink,
3. *Transparency*, $\text{trans}(p)$: percentage of the data at the origin that arrives at the sink.

Then, a path p is represented by $p = (e_1, \dots, e_n)/e_i \neq e_j \forall i, j \in (1, \dots, n)/i \neq j$ where $e_k = (C_k, j, l)$, and $C_k \in \{\text{STO}, \text{FU}, \text{INT}, \text{I/O}\}$.

To compute the sequential depth and transparency of the path, we define two sets: the set of storage elements belonging to the path (Sp) and the set of FUs in the path (Fp).

$$\text{Sp} = \{e_k \in p / C_k = \text{STO}\} \text{ and } \text{Fp} = \{e_k \in p / C_k = \text{FU}\}$$

Then $\text{trans}(p) = \prod \text{trans}(e_k), \forall e_k \in \text{Fp}$

And $\text{seq_d}(p) = \text{card}(\text{Sp})$, assuming no multicycle FUs in the path.

Definition 7. Two paths, p_1 and p_2 , are *compatible* if they are disjoint sub-graphs, that is, there are no elements shared by the paths. p_1 and p_2 compatible $\Leftrightarrow p_1 \cap p_2 = \{\emptyset\}$.

2.3. Testability model

The test of a design has been modeled as a set of tasks. Thus, the design is testable if all the tasks can be done. In a BIST design, it implies that there must be elements able to perform the tasks, that is, the data transfers that make up the tasks.

Let us say an “in-path” is the one corresponding to an input transfer and an “out-path” to an output transfer.

Definition 8. A path, p , is *suitable* for a data transfer, t , if

1. Its sequential depth and transparency allow transmission of the percentage of information required within the given time from the origin to the sink of the path.
2. The origin of the in-path or the sink of the out-path is an element with the ability for test that corresponds to the type of test task (able element).

According to this definition, a path suitable for test initialization is an I-path connecting an element with initialization-ability, that is an input port, to the test source. On the contrary, a path is suitable for test generation for a given input of an FU if a number of pseudo-random vectors equal to the complexity of the test of the FU reaches the input of the FU. Thus, for FUs with low complexity, such as ripple carry adders, the transparency required for the test generation paths is less than the transparency needed for complex multipliers.

Transfers that take place simultaneously require compatible paths, so that data can be transferred independently through them. That is the case with test generation for a two-input FU. Test vectors have to be transmitted independently and simultaneously to each input, so two compatible paths are needed.

To summarize, a *task can be done* if there are *suitable* paths for all the data transfers in the test task and paths corresponding to simultaneous transfers are *compatible*.

Once a suitable, compatible path is found, the test transfer is done. In order to guarantee that the test will be possible, such a path must exist for each transfer.

Definition 9. The *state in test mode* of the destination of an input transfer (the source of an output transfer) is a binary value that indicates whether or not the transfer can be done. Thus, the possible states are: *test needed* and *test fulfilled*.

If there is already a suitable path to carry out the transfer, it can be done, that is, the test transfer is fulfilled. On the contrary, if there is no path to perform the transfer, it cannot be done and it needs a path with the appropriate ability for the transfer. Four different abilities for test correspond to four different test needs, namely: initialization-need, generation-need, compaction-need and extraction-need.

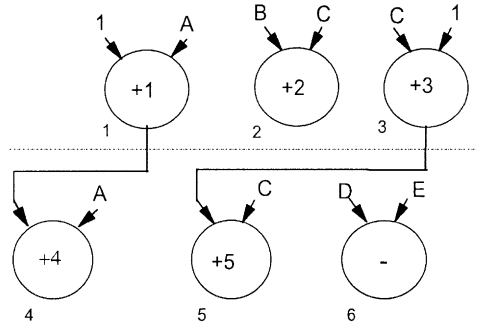


Fig. 2. Scheduled CDFG.

Table 2

List of FU types generated by the scheduler for the CDFG in Fig. 2

Node	Types of FU
1	1, 6, 2, 5
2	6, 1, 5, 2
3	1, 6, 2, 5
4	1, 6, 2, 5
5	1, 6, 2, 5
6	6, 3, 5, 4

Definition 10. A data path is *testable* if the state in test mode of all its elements is equal to fulfilled, i.e. there are suitable paths for all the test tasks of all the elements.

So, the problem of designing testable data paths becomes the one of creating suitable, compatible paths for all test tasks.

3. Data path allocation

As stated in Section 1.2.1, our algorithm for data path allocation generates self-testable data paths with minimum area. The inputs to the algorithm are a scheduled CDFG and the tentative module allocation for each operation (that is, the order in which the types of module have to be tried, and the number of modules of each type for the first solution) that are provided by the scheduler. Also, the designer can set some of the operation-to-module bindings and/or value-to-register bindings a priori. Finally, information of the available types of modules and registers is provided as a library of modules. It also holds low level (layout) information, needed for the precise estimation of module and interconnect area. The library of modules used throughout the paper has been generated using CADENCE-ES2 1 μm technology.

The inputs to the algorithm for a simple example, that will be used to explain how allocation is done, appear below. Fig. 2 is the scheduled CDFG and Tables 2 and 3 are the sorted list of FU types for the nodes and the library of modules respectively.

Table 3

Part from the library of modules used for the CDFG in Fig. 2

Type of FU	1	2	3	4	5	6
Description	Cheap adder	Fast adder	Cheap subtractor	Fast subtractor	Fast ALU	Cheap ALU
Operations	+	+	–	–	+ –	+ –
Area (ge)	127	763	161	790	886	245
Complexity of test	34	130	34	130	258	66
Delay (ns)	35	18	36	19	20	37

The scheduler sorts the list of FU types for each node as shown in Table 2 (explained in [30]). Notice that the last choices correspond to the use of fast FUs (types 2, 4 and 5). It means the cycle time allows operation of the slower FUs with the same set of operators than the more expensive ones and so we are sure that no fast FU will be part of the best designs.

Once the choice between the fast and the cheap implementation of the FUs is taken, the scheduler must decide which of the cheap types is most suited for each node. This second choice depends on the scheduling of operations to control steps. For the nodes in the first control step the types of FU advised are cheap adders for nodes 1 and 3 and a cheap ALU(+ –) for node 2. Then, the same number and types of FU are advised for the nodes in the second control step. If we follow this module allocation provided by the scheduler we will reach a design with minimum area of modules.

We can solve the allocation problem in three ways [2]:

- *Constructive* approaches, which progressively construct a design while traversing the CDFG. They add FUs, registers and interconnections to the data path as necessary. These algorithms are simple but the solutions they find are far from optimal.
- *Decomposition* approaches, which decompose the problem into a sequence of independent tasks and solve each of them separately. Because of dependencies among the tasks, no optimal solution is guaranteed even if all the tasks are solved optimally.
- *Iterative* methods, which try to combine and interleave the solution of the allocation sub-problems. Given a data path synthesized by the other methods, its quality is improved by reallocation. The strategy of reallocating a group of different types of entities can be a branch and bound search.

We use a branch and bound algorithm so that it searches the portion of the design space allowed by the bounding function and finds a number of solutions with different testability and area figures [31–33]. One of the solutions reached must be chosen as the final design.

In order to obtain each solution we split the allocation problem into a set of sub problems, one for each *atomic allocation* of an operation or a result. Then we solve each one of them sequentially beginning from the first control step.

Each atomic allocation is modeled as the mapping of an operation or a result to an instance of a module or a register that is picked up from a set of candidates. Both the definition of the set of

candidates and the selection of one of them have the objective of maximizing testability while minimizing area. While solving each sub-problem we use information of the previously solved sub-problems, namely the *partial design* (that is, the set of FUs, registers and connections that has been allocated up to the moment), to guide our algorithm.

If the bounding function works correctly, we are sure that branch and bound algorithms find the optimal solution when the search time is not limited. The search time is determined by the order in which candidates are explored. If the best options are taken in the first places, the best design is reached in a small amount of exploration time. Some heuristics try to predict in advance which partial design is presumably going to lead to the best design to speed up design search. These heuristics are used to select the order in which the candidates are tried, so that those producing maximum testability increment and minimum area increment are the first ones to be chosen.

In the remainder of this section, we will present the algorithm for allocation and we will highlight the different estimations of testability that it uses. In the following sections, we will explain in depth the two aspects that have more influence on the quality of the results obtained by the allocation algorithm: the estimation of the effect that each alternative has on the testability of the whole design and the development of heuristics that guide the search in the design space.

3.1. Algorithm for atomic allocation

CDFG elements (operations and variables) are allocated in a pre-defined order, marked by the function *Select element for next allocation* in Fig. 3. So allocation proceeds from the first control step to the last control step. Within a control step, operations scheduled to begin at the control step are chosen before variables generated in the control step. In this generic way, multicycle is supported in a natural fashion.

The algorithm for the atomic allocation, included in Fig. 3, starts from a partially allocated CDFG and a partial design. An operation is allocated to an FU while a variable is allocated to a register. Data transfers are assigned to interconnections after each allocation. Interconnections are reused or added if necessary.

There are usually several alternatives for the allocation of any operation or result. The testability and area added to the former partial design by selection of every candidate are estimated by the function *Evaluate CAND*. As explained in [32] multiplexer and interconnection area are included in the area estimations.

Then, testability and area estimations are compared to check values given by the bounding functions. Branches producing new partial designs with testability or area estimations not allowed are discarded to save exploration time.

Then, candidates are sorted out by means of heuristics based on the results of the estimations of area and testability. The first candidate that have not been tried to the moment is then allocated producing a new partial design and a new partially allocated CDFG.

Once a new partial design is selected, the algorithm goes on with allocation of the following element. Backtracking is done when either a data path is generated or there are no candidates allowed by the bounding functions.

Allocation is thus done as a recursive search that generates a set of data paths. Then the final design must be chosen from the set of designs obtained, as stated by their testability and area figures.

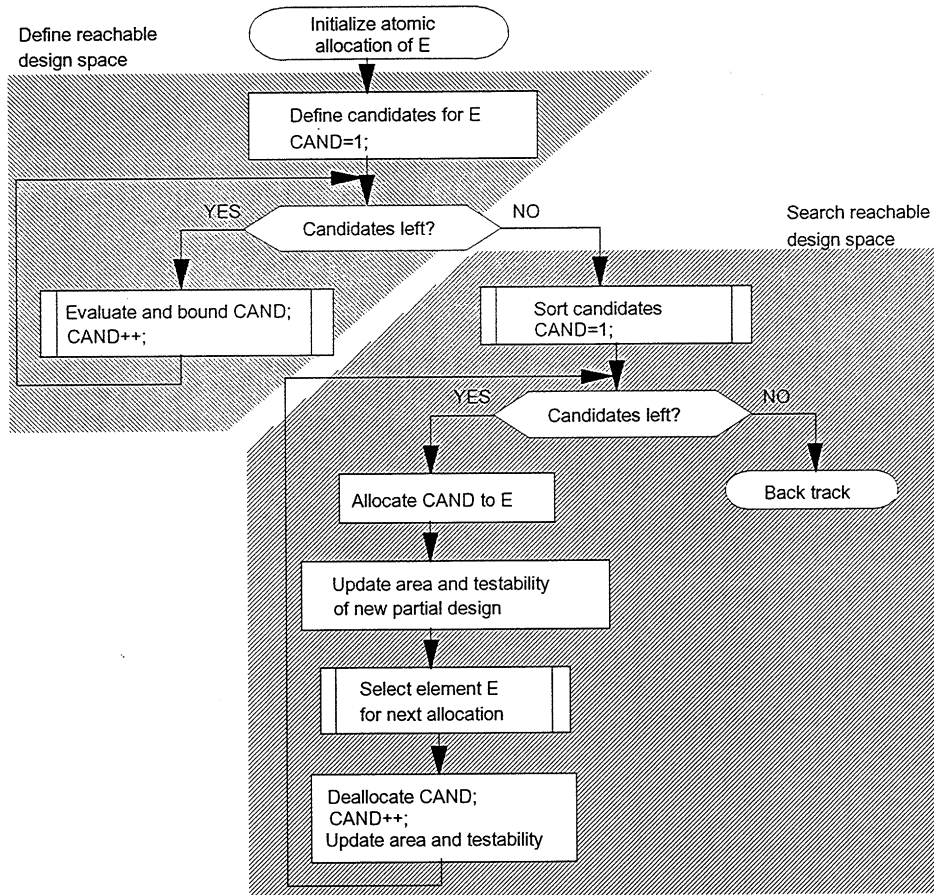


Fig. 3. Algorithm for atomic allocation of element E.

In line with this, the testability and area estimations are used in two different conditions: after a design is obtained, to compare it with other designs, and each time an FU must be chosen for allocation of an operation or a REG for storage of a result, to sort out allocation alternatives. Since the conditions and aims of both applications of the estimates are strongly diverse, they are studied separately.

3.2. Estimation of testability after allocation of a data path

Area and testability are used as quality measures to determine which design is the best among all the ones generated. Thus, when a new data path is allocated, its area and testability figures have to be determined so that the new design can be compared to other designs generated. Then, the best design among them will be selected.

According to Definition 10 in Section 2, we will mark a design as testable if all the test transfers of all the modules in it can be done. Since all the data paths synthesized are testable we will use the test time as estimate of the testability of the designs.

Therefore, we will group the solutions into classes according to their test time. The smallest data path of each group is the best solution in the group. Thus, there is a set of self-testable data paths, which represent the best solution for different tradeoffs between area and test time. The final solution among these best ones is chosen by the user.

3.3. Estimation of testability during allocation

The data path has not been designed yet and its total area and testability cannot be known, but the increment in area and testability corresponding to each allocation choice can be estimated. Area and testability increment estimates must be as reliable as possible in order to drive the allocator towards the best designs instead of misleading it. Area increment is estimated as explained in [32].

According to our model, testability is estimated as a function of the number of test tasks that can be done. Then, it is incremented each time an unsolved test transfer becomes solved. Following definitions in Section 2, *testability is incremented when there is a suitable, compatible path connecting two elements, one with a certain unsolved test task and the other with the counterpart test ability*.

As a result, elements with a given test need fulfilled can produce no testability increment. However, connecting them to elements with ability for that task results in waste of ability for that task. On the contrary, connecting them to elements with no ability for the task can result in *future testability increments*, because elements with test ability remain idle and can be connected to elements with test need, producing testability increment. Since we want to maximize the use of test resources, the heuristics will have the future testability increment into account.

To generalize, we can say that *testability estimation is based on the correspondence between the state in test mode of one element and the test ability of another one*. Each time there is a matching of states and abilities, testability is increased.

There are two different conditions that result in testability increments as shown in Fig. 4. The first, which leads to a real increase, is the *direct testability increment*, that is produced when an allocation choice involves creating a path for an unsolved test task.

The second, that we call *indirect testability increment*, considers the fact that no test abilities are wasted; therefore, it is produced when the agreement is found between an element with test need fulfilled and an element with no test ability.

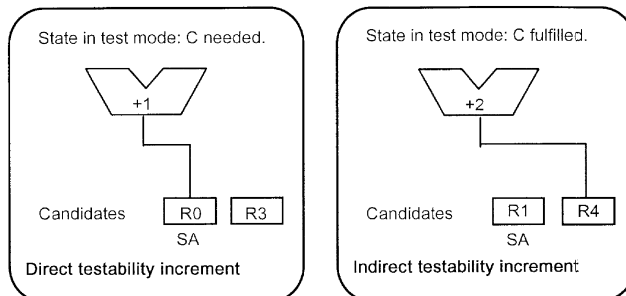


Fig. 4. Testability increments due to REG allocation.

We bear in mind that the direct increment has a present effect on the testability of the data path while the indirect one favors future effects, so the first one will have a bigger weight in the selection of candidates, that is, it will be bigger than the second one.

As an example, we present the testability increments that can be produced due to the allocation of a register for the output of adders +1 and +2 in Fig. 4. The output of a FU has either compaction-need or no test need (compaction-fulfilled).

In Fig. 4, adder +1 has no test destination for its output (compaction-need) and one of the candidates to store its results can act as test destination (register R0 has compaction-ability). So, if register R0 is selected for allocation, test compaction can be done and testability is increased.

On the contrary, if the state in test mode of an element is such that it has no need such as adder +2 in Fig. 4, that element corresponds to elements without test ability for that need like R4. If adder +2 is connected to register R1 no test task becomes solved. Besides, R1 is no longer idle to be connected to another functional unit with compaction-need. Therefore, its test ability is wasted. According to this criterion, register R4 should be selected for allocation because of the indirect testability increment.

4. Allocation of functional units

The problem of functional unit allocation consists of finding a functional unit that performs the operation of the node. Then, it is connected to the elements that store the input variables of the node, that have already been allocated.

For each operation the FUs available can be identical or have different implementations. The information about the implementation of a FU is called the type of the FU. So there can be modules of various types for some operations, e.g. the addition, such as ripple-carry adders, carry look-ahead adders, ALUs performing addition and subtraction and so on, as shown in Table 3. The features related to the implementation of all the different types of modules are stored in a library of available modules. It is used during scheduling to determine which module, that is, which type, is most suitable for each node. Further, for each node, a list of all the different implementations available for the operation of the node (e.g. all the types of adder if the operation is an addition) is provided by the scheduler to help during allocation. The types in the list are sorted according to their suitability to implement the node once the schedule and the cycle time have been fixed, as shown in Table 2 for the example in Fig. 2. When the node has to be allocated to a module this information is useful.

As introduced in the previous section, allocation of an operation involves definition of the set of alternatives for its implementation and selection of one among them. The candidates are, on the one hand, the functional units in the partial design that are able to perform the operation of the node and that are not used in the control step considered. Moreover, another alternative is adding a new functional unit to the partial design.

When an available and existing FU is selected, the area increment, if any, comes from the creation of connections between the elements storing the operands and the inputs of the FU and also from the growing in size of the multiplexers at the FU inputs.

Area increment caused by addition of a new FU to the partial design is the sum of the size of the functional unit and the creation of the connections to its inputs. Testability increment for every

candidate is explained below. Then, the heuristic for FU ordering based on the estimates is introduced.

4.1. Testability increment due to FU selection

The testability increment comes from the connection of the candidate FU to the elements storing the operands. Those elements can either be registers or constants. Thus, the test transfer involved in FU allocation is *generation of test vectors* for the inputs of the candidate FU. The testability increment for each input depends on:

- the *ability* as test source of the element storing the operand (whether or not it has generation-ability), and
- the *state in test mode* of the input of the candidate FU (whether it has generation-needed or generation-fulfilled).

Direct testability increase is obtained if a path for test generation is created, that is, a path suitable for test generation is created and it is compatible with the test generation path for the other input and the test compaction path of the output. Indirect increment is achieved if the storage element cannot perform test generation and the generation task for the input of the FU selected is already solved.

An example of FU allocation indicates how testability increment is estimated. Given the partial design in Fig. 5, one of the three adders has to be chosen for a new addition. The new addition has its left operand stored in register R2 and its right operand is a constant. In the figure, it can be noticed that register R2 can act as a test source. As a consequence, the abilities for the test of the elements storing the operands are: *generation-ability* for the left input (in 1) and *no generation-ability* for the right one (in 2). It means that for the left input we sort candidates according to the direct testability increment while for the right input we sort candidates according to the indirect one.

The states in test mode of the inputs of the candidate FUs are shown in Table 4, where 0 means generation-fulfilled (no need) and *G* means the opposite. When there is no generation-need, the register acting as test source is also included in brackets.

The testability increment for the three choices is:

The *left input* of adder +1 has generation-fulfilled, so connecting it to register R2 would waste generation-ability and no testability increment is produced. The left input of adder +2 has test need matching the ability of the storage element (it has generation-needed). Although it seems that

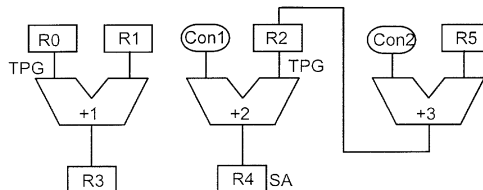


Fig. 5. Partial design before allocation.

Table 4
State in test mode of the FUs in Fig. 5

FU, port	+ 1, in1	+ 1, in2	+ 2, in1	+ 2, in2	+ 3, in1	+ 3, in2
State in test mode	0 (R0)	G-need	G-need	0 (R2)	G-need	G-need

direct testability increment should be obtained, no path is created for test generation of the left input. The reason is that register R2 has already been assigned as test source for the right input of the adder, so it is not compatible as test source for the left input. As a result, no suitable path is found and no increment is obtained. Finally, adder + 3 needs test source for the left input. Register R2 is suitable to perform test generation, producing direct testability increase.

On the other hand, the *right input* of adder + 1 has generation-needed that the constant is not able to satisfy, so no indirect testability increment is obtained. The input of adder + 2 has test need matching the ability of the storage element, because it has no need. As a result, indirect increment is obtained. Finally, adder + 3 needs test source for the right input and it produces no indirect testability increase.

The conclusion is that for the three choices, testability increments (ΔTest) are:

$$\Delta\text{Test}(+1) = 0$$

$$\Delta\text{Test}(+2) = \text{indirect (CON connected to the right input)}$$

$$\Delta\text{Test}(+3) = \text{direct (test generation for the left input is feasible)}$$

The greatest testability increment is achieved when adder + 3 is selected for the allocation of the new addition.

4.2. Heuristic for FU ordering

The goal of the FU ordering heuristic is to sort out the candidates for FU allocation so that the ones leading to the testable design with minimum area are tried first. If the functional units are much bigger than the rest of the data path components (such as multipliers or carry look-ahead adders), the smaller design belongs to the set of designs with minimum FU area. Even when smaller FUs are used, the number of them is not much bigger than the minimum number (the difference is at most 1 or 2 extra FUs).

Besides, there is a strong influence between the scheduling of nodes and the selection of the type of functional unit that implements the operation. When a node is scheduled at a control step, the type of FU in the library that seems to be most suitable for it is also annotated. This knowledge is used to find the optimal set of FUs quickly.

From the above statements it seems clear that the heuristic for FU ordering is mainly focused on reaching this set of FUs with minimum area. Testability is only considered as a second goal.

Sorting out the candidates for FU allocation is done according to four criteria, that are applied in decreasing importance order. The two first and most important ones try to select candidates that will lead to the set of FUs with minimum area, the third one is geared towards a maximum

testability increase and, finally, the last one is concerned with interconnect (multiplexer and connection) area minimization.

Criterion 1. All the FUs in the partial design that can perform the operation of the node and are idle in its control step are tried before adding a new FU to the partial design.

The first criterion applied is a greedy one because we want the first design synthesized to have the minimum area-set of FUs. After trying the existing alternatives the algorithm will explore the designs with non-minimum number of FUs that correspond to tradeoffs between module area and interconnect area.

Since there can be several types of FU in the data path that share operators, this greedy criterion is not enough to reach the minimum area-set of FUs and we need to take into account the dependence between scheduling and module allocation.

Criterion 2 (*Selection of FU type*). When there are candidates of different types they are sorted out according to the list of FU type preferences given by the scheduler. This criterion also holds for FU creation.

This second criterion is applied to select the FU type when there are FUs of different types among the candidates for a node and when a FU is added to the partial design. Since the scheduler assigned that node to its control step based on an assumption of which FU type would correspond to the node, candidates of that type are probably the best ones. In this way we take into account the dependence between scheduling and module allocation, because *the first alternative selected always corresponds to the module allocation advised by the scheduler*. But the remaining alternatives will be tried in time if testability increment or interconnect area decrement can be gained.

There are usually several candidates of the same type. Selection of one among them has no influence on the set of FUs. But it has some effect on the testability of the design and the interconnect area. As testability has not been considered so far, it will be given preeminence over interconnect area. Thus, the third criterion deals with testability increase and only after the interconnect area is taken into account.

Criterion 3. When there are several candidates of the same type, the FUs that produce the maximum testability increment (as computed in Section 4.1) are selected first.

Testability increment is either direct, when a suitable path for an unsolved test task is created; or indirect, when there is a matching between an element with no need and another one with no ability. Direct increment is always preferred to the indirect one.

Even when none of these matching happen, candidates are not equivalent regarding testability of the data path. A deeper analysis of the candidates divides them into two groups: the ones that are harmful for testability because they create connections that prevent testability increase, and the rest of them, that can lead to some testability increment (see the example in Section 4.3 for further details). As a consequence, the ones that prevent testability increase are put in the last places of the list of candidates.

Criterion 4. When there are candidates producing the same testability increment, the ones that lead to minimum interconnect area increment are selected first.

To summarize, the first two criteria sort the alternatives so that the ones leading to the design with minimum module area are tried first. The third criterion sorts alternatives with the same module area according to their effect on the testability of the data path and the last one sorts alternatives with the same testability increment according to the increment on interconnect area. As a result, candidates that are equivalent according to area are sorted according to testability while candidates with the same testability are sorted according to their area. This scheme produces good testability-area tradeoffs.

4.3. Example

As an example, allocation will be done for the CDFG in Fig. 2. The CDFG has 3 additions in the first control step and 2 add operations plus a subtract one in the second control step. So, the minimum area set of FUs is made of two adders and one ALU (+ −).

Nodes 1–3 are scheduled in the first control step. That means there were no functional units in the partial design before allocating them. Following criterion 2 and according to the list of FU type preferences in Table 2, two adders of type 1 and one ALU(+ −) of type 6 are included in the partial design.

If we followed the greedy criterion, three adders would be created for the nodes in the first control step. Then an additional FU (a subtractor) would be needed in the second control step. As a conclusion, the second criterion, that takes into account the features of the scheduled CDFG is needed whenever an FU must be added to the partial design.

Then, registers are allocated for the results generated in the control step. The resulting partial design that appears in Fig. 6 is made of three functional units (adders +1 and +2 and ALU + −) and it is the starting point for allocation of FUs in control step 2 (nodes 4–6). The state in test mode of the FUs is also provided in Table 5.

Application of the heuristics explained is exemplified for allocation of FU for *node 4*:

Following criterion 1 all of the existing and idle FUs have to be tried before adding a new FU to the partial design, so the two adders and the ALU are candidates for the new allocation.

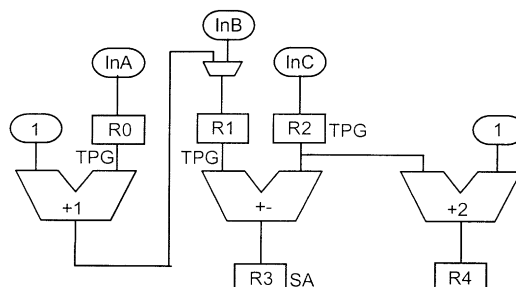


Fig. 6. Partial design before allocation of node 4 in Fig. 2.

Table 5
State in test mode of the FUs in Fig. 6

FU, port	+1, in1	+1, in2	+1, out	+−, in1	+−, in2	+−, out	+2, in1	+2, in2	+2, out
State in test mode	G-need	0 (R0)	0 (R1, +−, R3)	0 (R1)	0 (R2)	0 (R3)	0 (R2)	G-need	C-need

According to criterion 2 the adders should be tried before the ALU if we want the module area to be minimal. In fact, if the adder-subtractor is chosen, a new subtractor will be needed for node 6. Thus, FUs of type 1 (adders) should be tried first. If a new FU is created it will be an adder.

Selection of one candidate among the ones of the same type (that is, choosing between adders +1 and +2 for node 4) has no influence on the set of FUs. According to criterion 3 we have the following testability increments:

The *left operand* of node 4 is the result of adder +1, that is stored in register R1. So, its test ability is generation-ability (G) and candidates are sorted according to direct testability increment.

Adder +1 needs test generation for its left input but R1 belongs to the test compaction path, so it is useless for test generation. Since no compatible path is created, there is no increment. The state in test mode of adder +2 is generation-fulfilled, so it does not match the test ability of R1.

Since there is no testability increment for the left input, the effect that connecting R1 to both adders has on the data path structure is analyzed. R1 belongs to the test compaction path of +1. Since that task is simultaneous with test generation for the left input, no path for test generation can ever be found coming from R1. Thus, connecting R1 to +1 prevents any testability increase due to test generation for the left input. On the contrary, R1 does not belong to any of the test paths of +2. It can be connected to its left input with no testability damage.

As a summary, testability increment for the left input is equal to zero for both adders but attending to testability prevention adder +2 should be preferred.

The same analysis is done for the *right input*, which is stored in R0, and the results obtained are: no testability increase (non-matching state in test mode and test ability) for +1; and direct testability increase for +2 (the input has generation-need and R0 can be its test source).

According to the direct testability increment adder +2 is tried before adder +1.

The result of this reasoning is summarized below:

0. Since the three existing FUs are idle and can add, the alternatives are $\text{Candidates} = \{+1, +-, +2\} \cup \{\text{add new FU}\}$.
1. According to criterion 1 (greedy) the new FU is the last alternative, and the list remains: $\{+1, +-, +2, \text{add new FU}\}$.
2. According to criterion 2 (selection of FU type) adders are preferred to ALU(+−). The resulting list is $\{+1, +2, +-, \text{add} +3\}$.
3. According to criterion 3 (testability increment) the adder producing direct testability increment for one of the inputs is preferred. The list of alternatives is finally obtained: $\{+2, +1, +-, \text{add} +3\}$ and the last criterion is not used for this simple allocation example.

Allocation for node 5 only needs criteria 1 and 2 to reach the following list of alternatives: $\{+1, +-, \text{add} +3\}$.

In Fig. 7(b), only $+ -$ has no test need. Both adders have unsolved test tasks: $+1$ needs test generation for its left input and $+2$ needs test generation for its right input and also has compaction-need. Both R1 and R4 have to be CBIIBO to test this data path.

Note the output of $+2$ has compaction-need in both cases, but in Fig. 7(b) it is only connected to a self-adjacent register, that would have to be replaced by a CBILBO, while in Fig. 7(a) it is connected to register R4, that could be replaced by a much smaller SA register.

A comparison of both data paths leads us to conclude that due to the testability increment criterion the data path in Fig. 7(a) is more testable than the one in Fig. 7(b) and their areas are quite close (the differences are a 2-to-1 multiplexer and one interconnection).

5. Allocation of registers

A register needs to be allocated to store every result generated by the output of an element, that can be an input port or an FU. Solving a register allocation means mapping a result to a register that is picked from a set of candidates. Each candidate register has some test abilities that depend on its implementation. Then, the type of any register is defined according to its abilities for test as follows: TPG registers, SA registers, BILBO registers and normal registers (non-BIST). The physical features of the different types of registers are stored in the library of modules, where the delay and area corresponding to each type are also stored.

The algorithm for register allocation is similar to the one introduced in the previous section for functional units: all the candidates are determined, their testability and area increments are computed and according to the estimates, candidates are tried in as pointed out by the sorting heuristics.

The set of candidates for allocation of any result is made of all the registers in the partial design that are not used in the control step generating the result. The choice of adding a new register to the partial design is also included.

Computation of the testability increment for each candidate is more complex for registers than for functional units because each candidate can produce several effects on the data path testability. On the one hand, the testability increment caused by the allocation of one candidate to the result depends on the class of element originating the result and on the abilities for test of the candidate. Different classes of element (input port or FU) produce different testability increments.

On the other hand, selection of the candidate influences the future testability increments in the following way: since allocation goes on from the first to the last control step, when a result is allocated to a register the elements that are using that result in future control steps have not been allocated yet. However, the register is being selected in the present control step, that is, the source of those *future connections* is being determined and so are its test ability and state in test mode. The testability increment of those future allocations will depend on the candidates for them, but also on the test ability and state in test mode of the register selected for this present allocation. Both testability increments, the one that is produced by the present allocation and the one that will be produced by future allocations must be taken into account to select register for the present allocation. Thus, the successors of the CDFG node whose output link is being allocated are explored and the testability increment that will be produced by the future connections (*testability increment look-ahead*) is computed for all the candidates.

The area increment of any candidate is computed as follows. When an existing register is selected, the area increment, if any, comes from creation of a connection between the output that has the result and the input of the register, and also from the increase in size of the multiplexer at the input of the register.

Area increment caused by the addition of a new register is the sum of the size of the register and the creation of the connection to its input. The area of a register depends on its type, so that the area of TPG and SA registers is almost three times the area of a non-BIST one. Further, a BILBO register is five times bigger than a normal one. As a result, the area of a register depends mainly on the test abilities of the register, so we divide it into two parts: the register area, that is computed assuming the register is a normal one, and the BIST area, that is the increment on the size of the register due to its test abilities. It is important to exploit the use of test abilities in order to minimize the area.

Creation of a register involves selecting its type from the ones available in the library. Since test abilities of the instance are fixed from that moment on, type register selection is done with the help of an additional heuristic that tries to minimize BIST area.

Testability increment and testability increment look-ahead for every candidate are computed as explained next. Then, the heuristics for register creation and for register ordering based on the estimates are introduced.

5.1. Testability increment due to register selection

Two different conditions for the allocation of a register require two different heuristics for register selection, one for storage of *input port results* and the other for storage of *functional unit results*. So, computation of the testability increment is done separately for each kind of register allocation because the test task that can be solved by the atomic allocation is also different.

5.1.1. Testability increment for input port results

If the element holding data is an *input port*, its test ability is initialization of a test source. Thus, *testability is increased directly when the register selected is a test source with no initialization path* and the test task involved is initialization of a test source. No testability increment is obtained when a register that is already controllable from another input port is selected.

Finally, *testability is increased indirectly when a register that is not controllable from any input port and without generation-ability is selected*. This increment is due to the fact that the controllability of the selected register is improved. Increasing the controllability of a register connected to a FU can result in larger figures of transparency for the other input of the FU. Thus, a path traversing the FU can be improved so that it becomes suitable for a test task. But this increment is indirect because no initialization task is solved, and only future task solving is favored.

As an example, let us consider that a register is searched for connection to a port In3 in the partial design of Fig. 8. Assuming that all the registers are idle, testability increment will be explained for the four candidates. Registers R2 and R3 are test sources so they can produce direct testability increment while registers R1 and R4 can only produce indirect testability increment.

Registers R1 and R2 are directly connected to other input ports, so their state in test mode is equal to initialization-fulfilled. Thus, no testability can be gained from their connection to an element with initialization-ability.

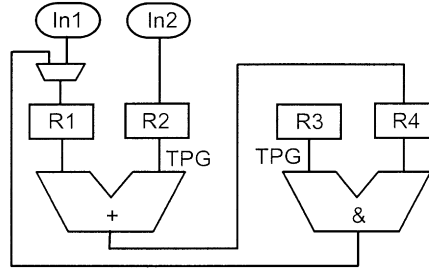


Fig. 8. Partial design before allocation of In3.

Register R3 is a test source that has no initialization path. Selecting it for storage of In3 results will solve the test source initialization task for the left input of &. Direct testability increment is obtained.

There can also be some smaller indirect testability increment if the register selected is not controllable even though it is not a test source. That is the case with register R4 in Fig. 8. If register R4 was fully controllable, the left input of & would increase its transparency from 0 to 100%. Since the output of & feeds the left input of + through register R1, this transparency increase would allow transmission of test vectors from register R3 to the left input of + by means of a suitable path. Indirectly, controllability increase for register R4 results in testability increase due to the creation of a path suitable for test generation for the left input of the adder. The resulting testability increments are

$$\begin{aligned}\Delta\text{Test}(R1) &= 0, \\ \Delta\text{Test}(R2) &= 0, \\ \Delta\text{Test}(R3) &= \text{direct (test initialization is feasible)}, \\ \Delta\text{Test}(R4) &= \text{indirect (REG controllable)}.\end{aligned}$$

5.1.2. Testability increment for FU results

On the contrary, when register allocation is done for storage of FU outputs the test task studied is test response compaction for the FU. *Testability increment is produced when the state in test mode of the FU output matches the ability as test response destination of the selected register.* The increase is either direct if the state in test mode of the output of the FU is compaction-needed and the candidate register can act as test destination, or indirect if the FU has compaction-fulfilled and the register has no compaction-ability.

As an example, register allocation is performed for the output of the subtractor in Fig. 9. Let us assume that all the registers shown in the figure are idle. The state in test mode of the functional units before allocation is shown in Table 6 (test paths are shown in parantheses).

Testability increment for each candidate is:

- Registers R1 and R2 have no compaction-ability and would produce no testability increment if they were chosen.
- Register R0 has compaction-ability but it cannot be used as test destination for the subtractor because it has been assigned as test source for one of its inputs. Since no compatible path is found, the test task remains unsolved and no testability increment is obtained.

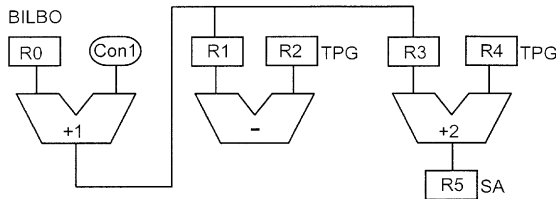


Fig. 9. Partial design before allocation.

Table 6

State in test mode of the FUs in Fig. 9

FU, port	+ 1, in1	+ 1, in2	+ 1, out	– , in1	– , in2	– , out	+ 2, in1	+ 2, in2	+ 2, out
State in	0 (R0)	G-need	0 (R3, +2, R5)	0 (R0, +1, R1)	0 (R2)	C-need	0 (R0, +1, R3)	0 (R4)	0 (R5)
test mode									

- On the contrary, selection of registers R3, R4 or R5 would increase the testability of the FU output by solving its test compaction task. Register R5 has compaction-ability and can be the test destination of the subtractor. The other ones cannot be test destinations but they are connected to a register (R5) with compaction-ability by means of a suitable path, which is compatible with the test generation paths. Since the test task is solved any way these three registers produce direct testability increment.

5.2. Testability increment look-ahead

The result stored in the register selected for the present allocation can either be transferred to an output port or be used as operand for future operations. If these circumstances are not taken into account when the register is chosen the testability of the data path can be small either when the result stored in the register will be transferred to an *output port* or when it is used as *operand for several or complex operations*.

As it will be explained below, the testability increment look-ahead for both circumstances are different in nature (for ports it is a statement but for FUs it is only an estimation) and test task (the task involved is test extraction for ports and test generation for FUs) and so will be the heuristics using them.

5.2.1. Testability increment look-ahead for output ports

Once a result that is going to be transferred to an output port is stored in a register, the selection of which register will be, in a future control step, connected to the port is taken. So, testability increment due to the matching of the test ability of the port (extraction-ability) and the state on test mode of the register is fixed from that moment.

If the register allocation heuristic wants the testability of the data path to reach its maximum, this selection of the register to store data for output ports should take into account the future testability increments that all the candidates will cause.

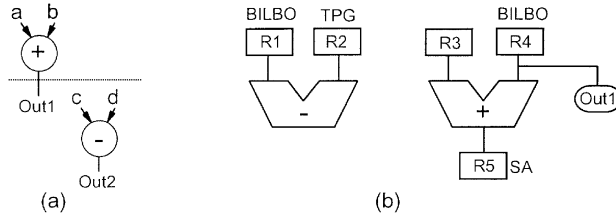


Fig. 10. (a) CDFG, (b) partial design before allocation of register for the subtractor.

Although the testability increment takes place when the register is connected to the port, in the future control step, *we are sure that an I-path to an output port will be created just exploring the successors of the node in the CDFG.*

There will be *direct testability increment for the candidate registers that are test destinations and have their test extraction task unsolved* because selection of that candidate solves a test task.

Besides, there will be *indirect testability increment for the candidates that are not observable from any output port even though they cannot act as test destinations*, like the indirect testability increment for input port results (Section 5.1.1). This second increment is due to the fact that no test task becomes solved, but it is possible that increasing their observability will result in creation of a path for test signature extraction.

Computation of the testability increment that will be obtained from connection to an output port (testability increment look-ahead) will be exemplified below. In Fig. 10 a register is to be selected for the result of the subtract node. Note that registers R1, R4 and R5 are test destinations.

Register R4 is already connected to Out1, so its state in test mode is equal to extraction-fulfilled and its selection produces no testability increase. On the contrary, if registers R1 or R5 are selected, a suitable path for test extraction will be created. So, registers R1 and R5 produce maximum (e.g. direct) testability increment look-ahead.

Besides, there will be indirect testability increment for the registers R2 and R3. For example, if register R3 were selected, a path would be found from register R1, through the subtractor to register R3 and consequently, to Out2. Then, the testability increment look-ahead produced by registers R2 and R3 is indirect, the one of registers R1 and R5 is direct and only register R4 produces no increment.

5.2.2. Testability increment look-ahead for node successors

Selection of a register to store a result that is going to be the input of future operations determines the generation-ability of the register without regard to the state in test mode of the FUs that are being allocated to the operation of the successors. Thus, the testability increment of the inputs of the FUs can be dramatically bounded by a wrong choice in register selection: *if the functional units that will be candidates for the future operation need test sources, the register chosen to store the operands should have generation-ability to get testability increment when the future connection is done.*

Just as for output ports, allocation of FUs for the future operations has not been done yet, but this time the testability increment cannot be established so certainly as for output ports. It will only be determined when FUs are allocated. There will be future testability increment (either direct or indirect) if the generation-need of these FUs matches the generation-ability of the candidate register, as explained in Section 4.1.

Some estimations of the state in test mode of the FUs that will be allocated to the successors are done based on the information already available in the CDFG and the partial design. From the CDFG we can find out the number of successors and for each successor we know the operation of the node, the list of FU type preferences for the successor node and the maximum parallelism of that operator along the CDFG.

According to this, estimate of the state in test mode of the successors is developed, so that the generation-need can be predicted and its matching with the generation-ability of the candidates for register allocation can be computed.

The *estimation of the generation-need of each successor* is done assuming that an FU of the type advised by the scheduler will be used for every node with that operation, that is, assuming the allocation of modules provided by the scheduler. Under this assumption, the number of FUs of that type that will exist in the CDFG at the moment of allocating FU for the successor is known because it is equal to the maximum parallelism of operations allocated to that type of FU in the CDFG along all the control steps before that of the successor. All the FU candidates for the successor can be separated into two groups: the ones that are part of the partial design and the ones that will be part of the partial design at the moment of allocation for the successor. From the former we can find out its generation-need. The latter have generation-need for sure.

So the estimated generation-need of the successor is computed as the arithmetic mean of the generation-need of all the candidates for the successor.

Once the state in test mode of the successors has been estimated, the testability increment produced by the matching of those states in test mode with the test ability of the register has to be computed. There is a testability increment look-ahead if a suitable and compatible path for test generation is created. The compatibility of the path is still unknown until the FU is allocated but it can again be estimated for all the FU candidates for the successor. If a FU is created for the successor, we already know the path is compatible. Otherwise, the compatibility with the test paths of the FU candidates that belong to the partial design has to be computed.

5.3. Heuristic for register type selection

Whenever a register has to be added to the partial design, its type has to be selected among the ones available in the library of modules. That is, the test ability of the new instance must be determined together with its area.

To reach the designs that, apart from being testable have a minimum area, *the area (cost) of selecting a BIST register should be spent only if the related testability increment is worth it*. Since testability is a matching between test ability and test task, the increment due to selection of a type with a given test ability depends on the state in test mode of the elements that are being connected to it.

Two test abilities of the registers require two separated choices to be made: whether or not select a type with generation-ability and select a type with compaction-ability.

On the one hand, the cost of *adding a test response destination* to the ones in the partial design is balanced with the testability increment due to creation of a path for test compaction of the FU output that is being connected to the register.

If the FU that generates the result to be stored in the new register has no compaction-need, a register with compaction-ability is useless. However, not every time a FU needs a test destination

one should be created because either FU reuse for future operations or creation of new paths in the partial design can provide a test destination. Thus, *a register with compaction-ability is only advised if the FU generating the results has large test complexity and there are no paths suitable for test compaction.*

On the other hand, *selecting a test source* depends on two testability increments: if the source of data is an input port, testability increment is due to initialization of the test source. Besides, if there are successors using the result as operand, there can be testability increment look-ahead due to test generation for the successors.

Since the initialization path should be 100% transparent and the input ports are not usually reused, there are only few opportunities to solve that test task. Besides, a datum that comes from an input port is needed for operations in the data path and storing it in a TPG register can improve the testability of the FUs selected for them. So, a TPG register is advised when the source of data is an input port.

On the contrary, when testability increments are due to test generation for successors, the balance of testability and area is taken more carefully. The reasons are that testability increment look-ahead is only an estimate and that the testability increment is not determined until the FU is selected, so if a register without generation-ability is chosen, an FU will be searched with generation-fulfilled.

To sum up, *a register with generation-ability is advised if data come from an input port or there are many complex operations with large estimated generation-need that will use them.*

5.4. Heuristic for register ordering

This heuristic sorts out the candidates for register allocation so that the ones leading to the minimum area testable design are tried first. Since all registers do the same function in normal operation mode, all of them are candidates for allocation of a result. So, the design space for register allocation is usually larger than for FUs.

Besides, registers are the data path elements with test abilities for the different test tasks. It means that their function in test mode is quite different. Consequently, their function in test mode should be taken into account for candidate sorting, to take maximum profit of test abilities.

Finally, FU allocation has been directed mainly to area minimization, so the testability improvement due to it is usually not enough. For all these reasons, register allocation is the major responsible of testability improvement of the data paths.

The goal of register selection heuristics is to maximize testability while keeping register, BIST and interconnect area as small as possible. In spite of this, the first criterion applied is not the testability increment one but a greedy criterion. The reason is that applying testability in the first place would result in the creation of a lot of registers to solve all test tasks with allocation of nodes in the first control steps. Since the number of registers in the data path is usually large, future hardware reuse can solve them when the data path is completed.

So, the first criterion is a greedy one, the second is driven to testability increment and the third one is concerned with minimization of interconnect area.

Criterion 1. All the registers in the partial design that are idle in the control step in which results are generated are tried before adding a new one to the partial design.

The goal of this criterion is to obtain in the first places the designs with minimum number of registers. Only after discarding all the existing registers a new one is added to the partial design. As a result the solutions that correspond to tradeoffs between register and interconnect area are explored after.

Criterion 2. When there are several candidates in the partial design, select the ones that lead to maximum testability increment and testability increment look-ahead in the following way:

If exploration of the CDFG points out that the register will be connected to an *output port*, candidates are sorted out according to the testability increment look-ahead for output ports.

Otherwise, if there is a test task that can be solved by the *present allocation* select candidates according to the testability increment. If there are still several candidates or there is no testability increment, a second sorting is needed. Then, if the *estimated generation-need of the successors* of the node is large, select according to the testability increment look-ahead for the successors of the node.

This criterion tries to maximize the testability of the data path while keeping its area as small as possible. So, the testability increment is used to select the candidates only when a test task can become solved.

As explained in Sections 5.1 and 5.2, the testability increment, present and look-ahead, can be diverse depending on the element generating the result to be stored and on the successors of the node in the CDFG, respectively. Thus, some kind of balance has to be done among them.

First, if there is an output port, testability increment look-ahead for extraction of test signature should be preferred. The reasoning is similar to the input ports one: there are not usually many chances to create I-paths from test destinations to the outputs, and the ports are not reused, unlike FUs. So, registers are sorted out according to their distance to any output port.

If no successor is an output port, the testability increment produced by the allocation is studied before the testability increment look-ahead is. This one is put in the last place because it is the only one that is not totally fixed by the present allocation and it can be improved to some extent by future FU allocations.

If there is a test task that can be solved by the present allocation (the source of data is either an input port or a FU with compaction-need) the candidates are sorted out by their testability increment. Since task solving is the major interest, direct testability increases are preferred over indirect ones. As it has been explained for FU allocation, candidates producing no increment are classified into two groups: the ones that will not lead to any testability increase (because they belong to any of the test generation paths of the FU that generates the result to be stored in the register) and the rest of them. Registers that would prevent testability increase are only tried after the rest of the existing ones.

If there are still several candidates unsorted and the estimated generation-need of the successors is big enough, registers are sorted out according to the testability increment look-ahead.

Criterion 3. When there are candidates that have not been classified by the application of the above criteria, select the ones that lead to minimum interconnect area increment.

When there are still several candidates producing the same the testability increment, present and look-ahead, or when no candidate produces testability increment of any kind, the third criterion,

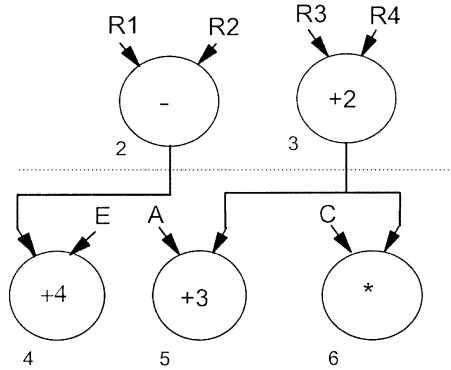


Fig. 11. Successors of nodes 2 and 3 in the CDFG.

directed towards interconnect area minimization is applied so that interconnect area is not wasted when no test task is becoming solved.

Although this criterion is applied in the last place, multiplexer area is not dramatically increased because testability is preferred only when it is worth it. Thus, the tradeoffs between testability and interconnect area are taken only when there is a large enough testability increment, otherwise, the area criterion is applied first. Both criteria are applied any way: when the testability increment is large, it goes first, when it is small, area goes first. As a result, tradeoffs taken produce either large testability increments or small area ones.

5.5. Example

The partial design in Fig. 9 will help to illustrate the register selection heuristic used for allocation of FU results. A portion of the CDFG, in Fig. 11, is also used to compute the testability increment look-ahead for node successors. First, a register is selected to store the output of the subtractor and then, register allocation is done for the output of adder + 2. Register allocation for the *output of the subtractor* starts from the situation explained in Section 5.1.2, where testability increment of the candidates are computed as well. Those results are briefly summarized below.

1. The alternatives are $\{R0, R1, R2, R3, R4, R5, \text{add new register}\}$.
2. The state in test mode (see Table 6) of the output of the subtractor is compaction-needed, so the test task involved is test compaction of the subtractor. According to Criterion 2, we must take into account that registers R0, R1 and R2 belong to test paths that require to be compatible with the test task involved, so those alternatives are the last ones to be considered before adding a new register, and the list is: $\{R3, R4, R5, R0, R1, R2, \text{add new}\}$.
 - 2.1. Following the testability increment, registers R3, R4 and R5 produce direct increment because the test-compaction task is solved if any of them is chosen.
 - 2.2. Estimate of testability look-ahead for the only successor of node 2 is done. In the partial design there are two FUs that can add (+ 1 and + 2). Assuming the maximum parallelism of adders in the CDFG is also 2, no new adder will be created. The estimated state in test

mode of the left input of the adders is obtained as the arithmetic mean of the state in test mode of $+1$ and $+2$: both have generation-fulfilled. The estimated generation-need of the left input of the successor of node 2 is 0.0 generation-need. Since the existing FUs for the only successor (node 4) have generation-fulfilled a register with generation-ability is useless.

Register R5 is preferred because direct connection to test registers is preferred to longer test paths. The list of alternatives is: {R5, R3, R4, R0, R1, R2, add new}

3. Selection between registers R4 and R3 is done to minimize interconnect area.

For subsequent allocation of register for the result of $+2$, let us assume register R5 has been selected for the subtraction. Then, the candidates for storage of the *output of* $+2$ are {R0, R1, R2, R3, R4, add new}.

First of all we take into account that registers R0, R3 and R4 belong to the test generation path for the inputs of $+2$, so they are the last of the existing registers to be tried. This results in the list: {R1, R2, R0, R3, R4, add new}

Since the state in test mode of the output of $+2$ is compaction-fulfilled, only indirect testability increment is possible. Both registers R1 and R2 have no compaction-ability and thus they produce indirect testability increment. Selection between them is based on the testability increment look-ahead. Node $+2$ has two successors: a multiplication (node 6) and an addition (node 5). The estimated generation-need of node 5 is the estimated generation-need for the adders, that is computed as it was for the successor of the subtractor (notice the input is not the same one) and the result is 0.5 generation-need.

The estimation on the need of the multiplier follows: the number of multipliers in the present control step is zero and the number of multiplication operations in the following control step is one, so one multiplier is created for the successor. This implies that the multiplier has no test task solved and that any register with generation-ability can be its test source (it will be compatible). To sum it up, the estimated generation-need of the multiplication is 1.0 generation-need and any register with generation-ability will do.

Since multipliers are complex operators, their generation-need is large and there is also another successor with some generation-need, selection of a register with generation-ability is advised. As a result, register R2, which has generation-ability is preferred over R1. The list of alternatives is transformed again: {R2, R1, R0, R3, R4, add new}.

Let us assume that register R2 is chosen to show how the testability increment look-ahead involves future testability increment. First, when a FU is allocated for node 5 the register storing the right operand has generation-ability. As a result, direct testability increment is possible if we select the FU with generation-need (adder $+1$). Second, FU allocation for node 6 implies adding a new multiplier to the partial design. Since the register storing its right operand is R2, the generation task is solved and direct testability increment is also obtained. In this case, selecting register R2 to store the output of node 3 results in solving the test generation task of two FU inputs.

6. Results

Some well-known CDFG have been used to demonstrate the main advantages of our algorithm for the allocation of self-testable data paths. Example 1 (Avra) is a CDFG borrowed from [23];

Table 7
Results for Example 1

	Testable allocation		Non-testable allocation
	Data path 1	Data path 2	
No. FUs	3	3	3
No. registers	5 (3 self-adj.)	5 (3 self-adj.)	5 (3 self-adj.)
No. muxes, no. wires	7 (1 mux4, 6 mux2), 22	6 (1 mux4, 5 mux2), 21	5 (2 mux4, 3 mux2), 19
BIST REGs	3 G, 2 S	2 G, 2 S	1 G, 2 B, 1 CB
Functional area (ge)	1818	1666	1528
Total area (ge)	2528	2246	3164

Example 2 (Ecudif) is the differential equation solver from [34]; Example 3 (Lee) is a CDFG borrowed from [14] and Example 4 (Filter) is the elliptic wave filter benchmark from [35].

Since our data paths are self-testable all the test tasks can be performed, and we will use test time, number of CBILBO registers and BIST area as measures of the testability of the different data paths. The data path area is measured in gate-equivalent units (ge).

6.1. Optimization of total area vs. optimization of functional area

Data paths from the examples were synthesized, using traditional allocation algorithms which only optimize area and using the allocation algorithms presented in this paper in order to verify that allocation performed to optimize area and testability (*testable allocation*) produces better data paths than allocation for minimum area (*non-testable allocation*) followed by intrusion of BIST registers.

The results in Table 7 were obtained by application of our algorithm to the *Avra* behavior, scheduled in four control steps as shown in [23]. The numbers in the table include the connections to the I/O ports of the data path (5 inputs and 2 outputs).

Our allocation algorithm finds two designs with different tradeoffs between area and testability, that the designer will have to examine to decide which is most suitable for him. Both are self-testable data paths with minimum module and register area.

Data path 1, which was the first solution explored, is bigger than data path 2 (the sixth one to be obtained). But, as shown in Fig. 12, the first data path can be self-tested in only two test sessions represented in Fig. 12(a), while the last one needs three test sessions (in Fig. 12(b)). Data path 1 can be chosen as the solution to save test time. On the contrary, if the area is the designer's priority, data path 2 will be selected.

Allocation that only minimizes the area of the data paths was also performed for comparison. The results in the third column of the table correspond to this minimum area solution once BIST registers have been added to it. The result of the non testable allocation can be tested in three test sessions and it needs a TPG, a CBILBO and two BILBO registers.

The different components of the total area for the three data paths are shown in Fig. 13. We notice that the functional area of data paths 2 and non-testable are very close while their BIST area

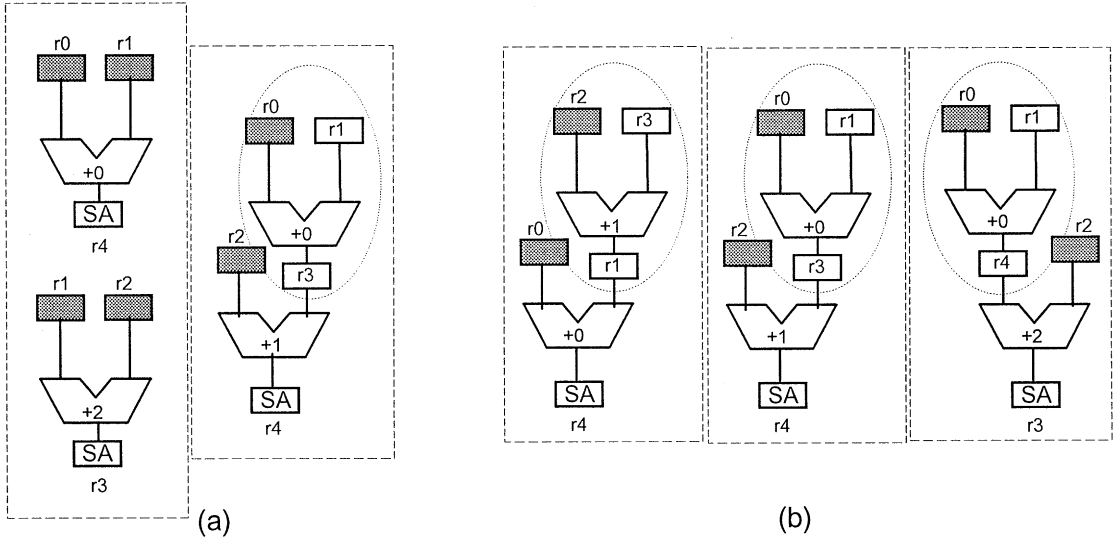


Fig. 12. Test sessions for data path 1 (a) and data path 2 (b) in Table 7.

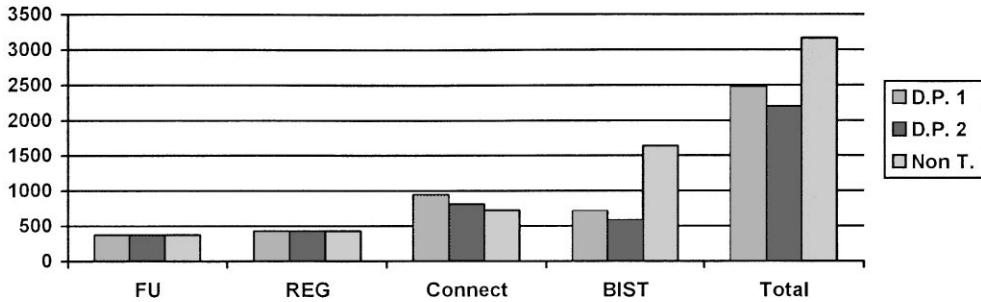


Fig. 13. Area figures of the data paths in Table 7.

figures are quite apart. Results in Fig. 13 show clearly that the increment on interconnect area of our data paths due to the testability-area tradeoffs reached is much smaller than the related decrement on BIST area. As a result, our data paths are smaller than the ones obtained by non testable allocation because *global optimization of area and testability during allocation takes maximum advantage of test resources and reaches the best tradeoffs between area and testability*.

Besides, the data paths obtained by our algorithms need no CBILBOs, while the minimum area data path needs one.

The results in Table 8 correspond to the *Ecudif behavior*, scheduled in four control steps as shown in [34]. The figures in the table include the connections to the I/O ports of the data path (4 inputs and 3 outputs). The CDFG in [34] has two data dependencies that cause two self-adjacencies in the data path.

Two data paths obtained by our algorithm are included. Both designs need the same number of test sessions – unlike the two data paths in Table 7 – so the smaller one of them is better than the other. Data path 1 is reached before data path 2, which is the best one.

Table 8
Results for Example 2

	Testable allocation		Non-testable allocation
	Data path 1	Data path 2	
No. FUs	4 (2*, +, –)	5 (2*, 2 +, –)	4 (2*, +, –)
No. registers	5 (4 self-adj.)	6 (2 self-adj.)	6 (4 self-adj.)
No. muxes, no. wires	11 (3 mux4), 29	9 (2 mux4), 30	8 (1 mux4), 26
BIST REGs	3 G, 1 S	2 G, 2 S	2 G, 2 S, 1 CB
Functional area (ge)	23432	23270	22901
Total area (ge)	23982	23850	24227

Table 9
Results for Example 3

	Testable allocation	Non-testable allocation
No. FUs	7 (3*, 2 +, 2 –)	8 (3*, 3 +, 2 –)
No. registers	6 (0 self-adj.)	5 (4 self-adj.)
No. muxes, no. wires	11 (2 mux4), 36 w	10 (4 mux4), 39 w
BIST REGs	3 G, 2 B	1 S, 3 CB
Functional area (ge)	33941	33631
Total area (ge)	35091	36029

Note in Table 8 that the functional area of the data path 2 is smaller than the one of data path 1, even though the data path 2 has an extra register and one extra adder. These additions of extra modules and registers produce a decrement on interconnect area that decreases the functional area. *This tradeoff between FUs, registers and interconnect area, that are favored by a global allocation and binding of modules, registers and connections, leads allocation to the best solutions.*

Comparison of solutions of our algorithms to the one synthesized with no testability consideration confirm that the scheme for simultaneous total area minimization and testability maximization produces better results, that is, smaller total area.

Once again, the non-testable data path has a CBILBO register. On the contrary, the testable data paths have no CBILBOs despite the data dependencies in the behavior.

The results in Table 9 correspond to the Lee CDFG as shown in [14]. It is scheduled in four control steps and the figures in the table include the connections to the I/O ports of the data path (4 inputs and 3 outputs).

Only the best solution obtained by our algorithms and the best one provided by non-testable allocation are presented in Table 9. None of them has minimum module and register area: the first one has an extra register and the second one has an extra adder.

Our data path has slightly bigger functional area but smaller total area than the non-testable one. Further, the testability of our data path is bigger: the data path reached by non-testable allocation has test problems for a FU, which feeds both inputs either from the same register or from constants. Thus, two compatible test generation paths cannot be found for the mentioned FU.

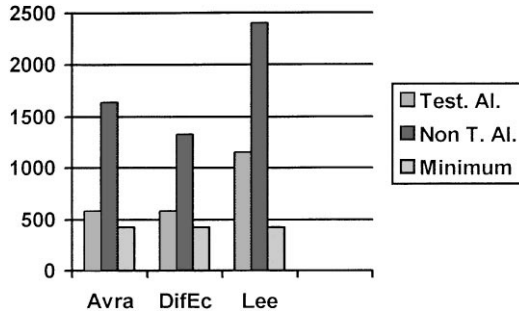


Fig. 14. BIST area in Tables 7–9.

Just in this third example, the solution reached by testable allocation contains no self-adjacent registers but no CBILBO is needed in any of them. This means that *our algorithm always succeeded in avoiding hardware sharing between the test paths of each FU*, and that *self-adjacent registers are included when they are not a problem for the self-test of the data paths*. CBILBO registers are avoided by correct hardware re-use even when there are data dependencies in the behavior.

BIST area of the smallest data path obtained by both our algorithms (testable) and the non-testable allocation algorithms for the three examples in Tables 7–9 are compared to the minimum test area (2 TPG and 1 SA) in Fig. 14. The number of BIST registers in our data paths is small and the BIST registers are never CBILBO and only once BILBO. As a result, *BIST area is always small, close to the minimum, because allocation takes profit of existing test resources*.

Note that not only the number of BIST registers needed for the self-test of the data paths synthesized by our algorithms is small but also it is almost independent of the size of the data path. This points out that the algorithms may produce good results for bigger circuits like the one presented below.

6.2. Search guiding heuristics

Example 4 (filter) was synthesized using our algorithms. Some relevant data paths picked from the set of solutions obtained are presented below, to show the way our heuristics work. The data paths in the tables are numbered according to the order in which they were obtained. All the solutions included have area figures smaller than the area constraint and can be self-tested with a small number of BIST registers. Some of them correspond to different tradeoffs between area and test time.

We have divided the solutions into two groups: the ones with minimum module area, in Table 10, and the ones that have an extra adder, in Table 11, to compare data paths with the same set of FUs. Due to the greedy behavior of the heuristics the first solution reached corresponds to the tentative allocation provided by the scheduler, and thus it always has minimum module and register area. Once data path 1 has been synthesized the following ones can have non minimum module and register area. That is the case with data path 2, which has an extra adder but is smaller than the previous solution.

Note that the solutions with the minimum number of registers are always reached before the ones with extra registers because there are usually many alternatives for each register allocation that have to be tried before adding extra registers to the data path.

Table 10
Results for Example 4 with minimum number of FUs

No. data path	No. REGs	BIST REGs	Test sessions	No. wires	No. muxes	Total area (ge)
1	9	3 G, 2 S	4	67	18 (3 mux8, 9 mux4)	42343
3	9	3 G, 2 S	4	66	18 (3 mux8, 8 mux4)	41745
8	9	3 G, 2 S	4	66	18 (3 mux8, 7 mux4)	41519
10	9	3 G, 2 S	4	64	18 (2 mux8, 8 mux4)	41063
13	10	3 G, 2 S	4	65	18 (2 mux8, 8 mux4)	40871
14	10	3 G, 2 S	4	64	17 (2 mux8, 8 mux4)	40775

Table 11
Results for Example 4 with non-minimum number of FUs

No. data path	No. REGs	BIST REGs	Test sessions	No. wires	No. muxes	Total area (ge)
2	9	3 G, 2 S	5	68	18 (2 mux8, 10 mux4)	41788
4	9	3 G, 2 S	5	67	18 (2 mux8, 9 mux4)	41538
5	9	2 G, 1 S	7	68	18 (3 mux8, 7 mux4)	41432
6	9	3 G, 1 S	7	69	18 (2 mux8, 9 mux4)	41408
7	9	2 G, 2 S	6	66	18 (2 mux8, 9 mux4)	41359
9	9	2 G, 2 S	6	66	18 (2 mux8, 8 mux4)	41139
11	10	2 G, 1 S	7	67	17 (2 mux8, 9 mux4)	41064
12	10	2 G, 3 S	6	67	17 (2 mux8, 8 mux4)	40949
15	10	3 G, 2 S	6	66	17 (2 mux8, 8 mux4)	41004

Solutions in Table 10 contain only 6 FUs (3 multipliers and 3 adders is the minimum set of FUs for the scheduled CDFG). These solutions, that were among the first to be reached, can be tested in 4 test sessions with 3 TPG and 2 SA registers. Since the test time is the same for all the solutions in Table 10 the smallest one is the best one. Note that the last solutions presented in the table are the smallest ones, even though they have an extra register. The best solution in the table is data path 14.

On the other hand, the solutions in Table 11 contain an extra adder, and the last ones have an extra register also. They correspond to different tradeoffs between area and test time.

The first ones can be tested in five sessions with almost minimum BIST area (namely, 2 TPG and 2 SA registers if the test time is smaller than 7 sessions). So they are designs with large testability and non minimum total area.

As the search goes on the algorithm obtains solutions with bigger test time and smaller total area than the former ones. For instance, data path 11 has minimum test area and maximum test time.

After the best solutions are explored, the algorithm finds the data paths that correspond to bad tradeoffs between area and testability. Solutions reached after data path 14 need more BIST registers and their total area increases.

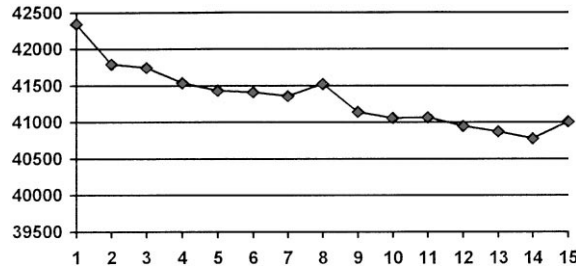


Fig. 15. Total area (ge) of the solutions for Example 4.

Table 12

A comparison of results for Example 1

	No. FUs	No. REGs	No. muxes, No. wires	BIST REGs
Ralloc	3	5 (2 self-adj.)	7 (2 mux4), 22w	2 G, 2 B
Ours	3	5 (3 self-adj.)	6 (1 mux4), 21w	2 G, 2 S

To sum it up, *the first solutions explored are the ones with shortest test time* but their total area is not minimal, *then the algorithm finds the data paths with minimum total area* and longer test time and finally the solutions that need more test registers and/or test sessions. Notice, for instance, that data path 4 and 15 have the same number of BIST registers but the former needs 5 test sessions instead of 6.

The total area of the solutions is displayed in Fig. 15. Designs obtained after data path 15 have bigger total area than the ones in the figure due to the test area increment. The smallest solutions is data path 14, that also has minimum number of test sessions. Thus, it is the best solutions synthesized.

The figures in Tables 10 and 11 show that the first designs reached are the most testable ones, and they need fewer test sessions and/or BIST registers. Then, the algorithm finds data paths with smaller interconnect area and more test sessions or BIST registers.

6.3. Our approach vs. other systems

The results obtained by our algorithms are also compared to those reached by other methods for testable allocation of BISTed data paths. The comparison is limited to the information available in the literature, that is, number of data path elements instead of area figures.

First, in Table 12, the cheapest testable solution reached for *Example 1* (data path 2) is compared to the one obtained by Ralloc, that is presented in [23].

Figures in Table 12 show that our algorithm generated a data path with the same number of FUs and registers and less multiplexers and wires than the one in [23]. In addition, our data path has smaller BIST area.

Table 13

A comparison of results for Example 2

	No. FUs	No. REGs	No. muxes, no. wires	BIST REGs
Ralloc [23]	4 (2*, +, –)	5	9 (3 mux4), 29w	4 B, 1 CB
Syntest [24]	3 (2* +, * – >)	5		4 G, 1 S
Parulkar et al. [26]	4 (2*, +, –)	4	6 muxes	2 G, 2 S, 1 CB
Ours	4 (2*, +, –)	5	11 (3 mux4), 29w	3 G, 1 S

A comparison of results obtained for *Example 2* to those obtained by Ralloc in [23], Syntest in [24] and the work of Parulkar et al. in [26] is presented below.

Our data path is two 2-to-1 multiplexers bigger than Ralloc's, so our functional area is slightly bigger than his. However, our data path only needs 3 TPG and 1 SA register while his design needs 4 BILBO and 1 CBILBO. Our solution has quite smaller total area than Ralloc's.

In [24] there is no reference to the number of multiplexers and wires in the data paths, so we can only compare BIST area, which is smaller in our solution than in theirs.

According to our scheduled CDFG and our model of input port, the minimum number of registers for this example is 5 and the minimum number of multiplexers is 8, if an extra register is added, or 9, if the minimum number of registers is used. The figures in the third row of Table 13 were the only information available in [26]. Thus, we cannot compare the complete data paths to find out the conditions that led to it. BIST area is smaller in our solution than in theirs.

The results in Tables 12 and 13 show that *our algorithm produces testable data paths that need fewer BIST registers and no CBILBOs*. So they have much smaller BIST area and their functional area is usually not much larger. It means that the global optimization of testability and total area (BIST and functional) takes advantage of test registers and generates designs that are testable at low cost.

6.4. Conclusions

We have shown that integration of the different synthesis tasks and the BIST intrusion task leads to global optimization of the area of BISTed data paths. Data paths synthesized by our algorithms have non-minimum interconnect area because allocation takes maximum advantage of test resources. As a result, their BIST area is so small that the total area of the data path is smaller than the one produced by minimum area allocation followed by BIST register intrusion.

This global approach favors area tradeoffs among the different components of the data path, so that the total area is minimized instead of the partial ones (namely module, register, connection and BIST area).

We also noticed that the data paths obtained never need CBILBO registers. Further, hardware sharing among the test paths of each FU is avoided by correct allocation, with no design space restriction (such as avoiding self-adjacency of registers).

Besides, the design space of solutions is explored efficiently due to the use of:

- (i) estimates of the effect each allocation alternative has on the testability of the data path, that not only considers present effects but also future ones;

- (ii) heuristics that reach good tradeoffs between testability and area, taking advantage of the module allocation provided by the scheduler, the estimates of testability and area.

Appendix A. computation of functional unit transparency

The transparency of input i of a functional unit depends on the operation or operations implemented by the element, the controllability of the control lines that select among the operations and the controllability of input j ($j \neq i$).

Under the common assumption that control lines of FUs are fully controllable, multi-operational units can have the transparency of their most transparent operator. If they are not controllable, the transparency of the least transparent operator is used for the FU.

The *transparency of each input of every operator* has been computed by functional simulation under the hypothesis that one value can be hold constant at input j during the whole simulation. Thus, computation of the transparency of input i is done as follows: input j ($j \neq i$) is held constant to a pre-defined value while all possible values are tried at input i . The number of different output values obtained from the simulation is divided by all possible values to get the transparency of the input.

For some operators, the transparency of one input depends strongly on the value at the other input. For example, an AND operator can be 100% transparent if the other input has all bits equal to one. On the contrary, the transparency is 0% if the other input holds a zero value. So, selection of the constant values used for transparency computation must be done carefully for each operator. Typical values that are assumed for input j are:

- *the best value*: the value at input j that allows maximum transparency of input i (e.g., a value of 1 for an AND). It is transparency when input j is controllable.
- *the worst value*: the value at input j that makes the transparency of input i reach its minimum (a 0 in one input of the multiplier makes the other input 0% transparent).

A transparency value will be chosen as transparency figure of input i of the instance of the FU depending on the value that can be assumed for input j (that is, on the controllability of input j). Table 14 shows the transparency for the inputs of the most common operators in a typical HLS library of modules. They are obtained assuming, firstly, that the other input is controllable (*best transparency*) and, then, that it is not (*worst transparency*). Both inputs of any operator except the divisor obtain the same value.

Table 14
Figures of the transparency (%) for the inputs of the operators in the library

Operator	+	−	And	Or	*	(left-in) /	/(right-in)
Best transparency	100	100	100	100	34.8 ^a	100	43.75
Worst transparency	100	100	0	0	0	0	0

^aTransparency when the complete output is observed (it has double length). If only the least significant half of the output is considered the best transparency reaches 100%.

References

- [1] S.Y.-L. Lin, Recent development in high level synthesis, *ACM Trans. Des. Automa. Electron. Systems* 2 (1) (1997).
- [2] D.D. Gasjki, N.D. Dutt, A.C.-H. Wu, S.Y.-L. Lin, *High-Level Synthesis Introduction to Chip and System Design*, Kluwer Academic Publishers, Boston, 1992.
- [3] C.F. Hawkins, H.T. Nagle, R.R. Fritzemeier, J.R. Guth, The VLSI circuit test problem—a tutorial, *IEEE Trans. Ind. Electron.* 36 (2) (1989) 111–116.
- [4] R. Chandramouli, S. Pateras, Testing systems on a chip, *IEEE Spectrum* (1996) 42–47.
- [5] M. Abramovici, M.A. Breuer, A.D. Friedman, *Digital System Testing and Testable Design*, Computer Science Press, Rockville, MD, 1990.
- [6] B.T. Murray, J.P. Hayes, Testing ICs: getting to the core of the problem, *IEEE Comput.* 29 (11) (1996) 32–38.
- [7] B. Koenemann, R.G. Bennets, N. Jarwala, B. Nadeau-Dostie, Built-in self-test: assuring system integrity, *IEEE Comput.* 29 (11) (1996) 39–45.
- [8] K.D. Wagner, S. Dey, High-level synthesis for testability: a survey and perspective, *Proceedings of 33rd ACM/IEEE DAC*, June 1996, pp. 131–136.
- [9] B. Mitra, P.P. Chaudhuri, Combined synthesis of easily testable datapath and control designs, *Proceedings of fifth International Conference on VLSI Design*, January 1992, pp. 187–192.
- [10] S. Bhattacharya, F. Brglez, S. Dey, Transformations and resynthesis for testability of RT-level control-data path specifications, *IEEE Trans. VLSI Systems* 1 (1993) 304–318.
- [11] C.-H. Chen, T. Karnik, D.G. Saab, Structural and behavioral synthesis for testability techniques, *IEEE Trans. CAD* 13 (1994) 777–785.
- [12] S. Dey, M. Potkonjak, Transforming behavioral specifications to facilitate synthesis of testable designs, *Proceedings of ITC*, October 1994, pp. 184–193.
- [13] M. Potkonjak, S. Dey, R. Roy, Considering testability at behavioral level: use of transformations for partial scan cost minimization under timing and area constraints, *IEEE Trans. CAD* 14 (1995) 531–546.
- [14] T.-C. Lee, N.K. Jha, W.H. Wolf, Behavioral synthesis of highly testable data paths under the non-scan and partial scan environments, *Proceedings of 31st ACM/IEEE DAC*, June 1993, pp. 292–297.
- [15] T.-C. Lee, N.K. Jha, W.H. Wolf, J.M. Acken, Behavioral synthesis for easy testability in data path allocation, *Proceedings of ICCD*, October 1992, pp. 29–32.
- [16] T.-C. Lee, N.K. Jha, W.H. Wolf, Behavioral synthesis for easy testability in data path scheduling, *Proceedings ICCAD*, October 1992, pp. 616–619.
- [17] M. Potkonjak, S. Dey, R. Roy, Behavioral synthesis of area-efficient testable designs using interaction between hardware sharing and partial scan, *IEEE Trans. CAD* 14 (1995) 1141–1154.
- [18] A. Mujumdar, R. Jain, K. Saluja, Incorporating testability considerations in high-level synthesis, *JETTA* 5 (1994) 43–55.
- [19] A. Mujumdar, R. Jain, K. Saluja, Incorporating performance and testability constraints during binding in high-level synthesis, *IEEE Trans. CAD* 15 (1996) 1212–1225.
- [20] P.H. Bardell, W.H. McAnney, J. Savir, *Built-In Test for VLSI: Pseudorandom Techniques*, Wiley, New York, 1987.
- [21] B. Koenemann, J. Mucha, G. Zwiehoff, Built-in logic block observation techniques, *Proceedings of ITC*, October 1979, pp. 37–41.
- [22] L.T. Wang, E.J. McCluskey, Concurrent built-in logic block observer (CBILBO), *Proceedings of the International Symposium on Circuits and Systems*, 1986, pp. 1054–1057.
- [23] L. Avra, Allocation and assignment in high-level synthesis for self-testable data paths, *Proceedings of ITC*, October 1991, pp. 463–472.
- [24] C.A. Papachristou, S. Chiu, H. Harmanani, A data path synthesis method for self-testable designs, *Proceedings of 29th ACM/IEEE DAC*, June 1991, pp. 378–384.
- [25] H. Harmanani, C.A. Papachristou, An improved method for RLT synthesis with testability tradeoffs, *Proceedings of ICCAD*, November 1993, pp. 30–35.
- [26] I. Parulkar, S. Gupta, M.A. Breuer, Data path allocation for synthesizing RTL designs with low BIST area overhead, *Proceedings of 33rd ACM/IEEE DAC*, June 1995, pp. 395–401.

- [27] I.G. Harris, A. Orailoglu, Microarchitectural synthesis of VLSI designs with high test concurrency, Proceedings of 32nd ACM/IEEE DAC, June 1994, pp. 206–211.
- [28] M.S. Abadir, M.A. Breuer, A knowledge-based system for designing testable VLSI chips, IEEE Des. Test 2 (1985) 56–68.
- [29] S. Chiu, C.A. Papachristou, H. Harmanani, A design for testability scheme with applications to data path synthesis, Proceedings of 29th ACM/IEEE DAC, June 1991, pp. 271–277.
- [30] R. Moreno, R. Hermida, M. Fernández, A unified approach for scheduling and allocation, *INTEGRATION, VLSI J.* 23 (1997) 1–35.
- [31] J. Septién, D. Mozos, J.F. Tirado, R. Hermida, M. Fernández, H. Mecha, FIDIAS: an integral approach to high-level synthesis, *IEE Proc. Circuits Dev. Systems* 142 (1995) 227–235.
- [32] H. Mecha, M. Fernández, J.F. Tirado, J. Septién, D. Mozos, K. Olcoz, A Method for Area Estimation of Data-Paths in High Level Synthesis, *IEEE Trans. CAD* 15 (1996) 258–265.
- [33] K. Olcoz, J.F. Tirado, D. Mozos, J. Septién, R. Moreno, Data path structures and heuristics for testable allocation in high level synthesis, *Microprocess. Microprogramm.* 39 (1993) 263–266.
- [34] P.G. Paulin, J.P. Knight, E.F. Girczyc, HAL: a multi-paradigm approach to automatic data path synthesis, Proceedings of 24th ACM/IEEE DAC, June 1986, pp. 263–270.
- [35] VHDL High Level Synthesis Benchmarks, 1991.



Katzalin Olcoz was born in 1968. She received the Applied Physics degree from Universidad Complutense de Madrid (UCM) in 1991, and the Ph.D. in Physics from UCM in 1997. She has held several positions with the Computer Architecture and Automatic Control Department of the UCM. Since 1991 she has been Assistant Professor of Computer Architecture and Technology. Her research interests include several aspects of the computer-aided design of integrated circuits with particular emphasis on high-level synthesis, and synthesis for testability of integrated circuits.



Francisco Tirado was born in 1951. He received the Applied Physics degree from Universidad Complutense de Madrid (UCM) in 1973, and the Ph.D. in Physics from UCM in 1977. He has held several positions with the Computer Architecture and Automatic Control Department of the UCM. Since 1978 he has been Associate Professor, and since 1986 Professor of Computer Architecture and Technology.

He has worked on different fields within Computer Architecture, Parallel Processing and Design Automation. His current research areas are design for testability, parallel architectures and processor design. He has published over 30 papers in technical journals and over 60 papers in international conferences.

Prof. Tirado is currently the Dean of the Physics Science and Electronic Engineering Faculty. He is member of the Informatics Advisory Board of the UCM, and he has also been Vice-Dean of the Physics Science Faculty and Head of the Computer Science and Automatic Control Department. During five years (1988–1992) he has been serving as General Manager of the Spanish National Program for Robotics and Advanced Automation. He is adviser of the National Agency for Research and Development (CICYT). He holds also the representation of CICYT in several national and international committees on Information Technology.

Prof. Tirado is a member of IEEE, of ACM, and of several European institutions and committees.



Hortensia Mecha was born in 1967. She received the Applied Physics degree from Universidad Complutense de Madrid (UCM) in 1990, and the Ph.D. in Physics from UCM in 1996.

She has held several positions with the Computer Architecture and Automatic Control Department of the UCM. Since 1990 she has been Assistant Professor, and since 1998 Associate Professor of Computer Architecture and Technology.

Her research interests include several aspects of the computer-aided design of integrated circuits with particular emphasis on automated synthesis, and optimization and test of integrated circuits.