Optimization of BIST Resources During High-Level Synthesis

Ishwar Parulkar

CENG 98-31

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213-740-4469)
May 1998

OPTIMIZATION OF BIST RESOURCES DURING HIGH-LEVEL SYNTHESIS

by

Ishwar Parulkar

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(Electrical Engineering)

May 1998

Dedication

To

Aai and *Dada*

Acknowledgements

An undertaking such as a Ph.D. is not possible without the guidance, love, support and companionship of many people...

Prof. Melvin Breuer and Prof. Sandeep Gupta for guiding and directing my doctoral research. Prof. Breuer for teaching me the value of perfection and meticulosity in scientific research. Prof. Gupta for teaching me that any problem can be looked at in n different ways.

Professors Alice Parker, Peter Beerel, Doug Ierardi and Michael Arbib for serving on my guidance and dissertation committees.

Prof. Edward McCluskey and Dr. LaNae Avra from Stanford University for providing the TOPS software which formed the experimental setup for the ideas in this dissertation.

Many other professors for honing my research skills through courses and discussions.

Charlie, Amit, Sridhar, Deb and Mody for countless discussions on topics ranging from Combinatorics to the 49ers.

Gopal and Henri for tolerating my numerous phone calls while sharing an office with me. Gopal for his level-headed advice on any academic matter. Henri for never failing to accompany me to a film screening... even the most obscure of obscure films.

Lana and Natalie for being my co-conspirators in exploring la-la-land in the early years. Lana for showing me the subtle difference between a *lutz* and a *salchow* and Natalie for showing me the difference between crème brûlée and crème caramel and other subtleties of French cuisine.

Jean-Luc for helping me fulfill my dream of experiencing what Jonathan Livingstone Seagull did - the beauty of flight...

Michael for putting up with me as a room-mate and for refreshing my German.

Numerous other friends (whom I shall refrain from listing for fear of doubling the pages in this dissertation) for lively conversations, sweaty tennis games, sunny beach-bumming days, bumpy Cessna flights, starry movie nights...

The nameless many working at the USC càfe for making my cappuccino just the way I like it, everyday, for the last six years.

Soma, Ema, Paro and Kirti for being there for me through the years. For creating a home away from home. Soma and Ema for keeping the child in me alive. Paro and Kirti for making sure that the adult did not die.

Athena and Nitin for their trust, support and comfortable companionship. For being the kind of friends only they can be.

And finally, my parents and my sister Padmaja for loving me unconditionally. For believing in me and for teaching me to believe in myself.

Thank you.

Contents

List Of Figures

List Of Tables

Abstract

Built-in Self-test (BIST) techniques modify functional hardware to give a design the capability of testing itself. The modification of functional registers into BIST resources that can generate pseudo-random test patterns and/or compress test responses, incurs an area overhead. Insertion of BIST resources after the synthesis stage in the VLSI design cycle, often leads to problems such as increased chip area, reduced performance, inability to achieve good fault coverage and restricted applicability of certain test methodologies. Hence there is a need for considering BIST requirements and their effects on the design, in the earlier stages of the design cycle. This thesis proposes optimization techniques that can be incorporated into the high-level synthesis stage of a design cycle such that, minimal area penalty is incurred in making a synthesized data path self-testable, while achieving all functional and test objectives. The testability model targeted by the optimizations is minimal intrusion BIST in which only a subset of the registers in the data path are modified for test. Techniques for estimation of BIST resources during high-level synthesis have been developed. Incorporation of testability optimization in all three phases of high-level synthesis is studied: 1) allocation and assignment (spatial domain), 2) scheduling (temporal domain), and 3) behavioral transformations (behavioral description domain). Each domain is shown to effect BIST resource requirement of a data path in a different way. Properties of each domain with regards to effect on BIST resources have been studied and techniques that exploit these properties to reduce BIST area overhead have been developed. Depending on the actual behavior, the degree of BIST resource optimization in a specific domain varies. Some behaviors lend themselves easily to BIST resource optimization in the assignment domain, some in the scheduling domain, while others require transformation of the behavior. Together, these techniques provide an efficient way of cutting down design cycle time and minimizing BIST resource overhead and total area of synthesized data paths.

Chapter 1

Introduction

Test is required to verify that a fabricated VLSI chip or a multichip system is fault-free or operates satisfactorily. With increase in the number of transistors on a chip and limited I/O, built-in self-test (BIST) techniques have gained importance as cost-effective means of testing digital circuits. BIST techniques involve modification of the hardware on a chip such that the chip has the capability of testing itself. One consideration in evaluating a BIST technique is the extra area needed for the test circuitry to achieve a certain level of testing.

The testability insertion phase normally occurs after the logic synthesis and verification phase in the VLSI design cycle. Considering testability at such a late stage in the design flow can often lead to problems such as exceeding chip area, inability to achieve the required throughput, degraded performance and inability to apply certain test methodologies effectively. The objective of this research is to incorporate BIST considerations in the high-level synthesis stage of the design cycle. Optimizing for BIST hardware resources during high-level synthesis reduces the number of design cycle iterations and the BIST area overhead of the synthesized data paths. In the following sections we give a brief overview of the VLSI design cycle and how testing and high-level synthesis (HLS) fit into this cycle.

1.1 Testing of Digital Circuits

Testing of digital circuits has become a formidable task with chip densities currently at 5 million transistors per chip and projected to grow up to a billion transistors per chip in 10 years [1]. Vast amounts of time, money and effort is invested by the semiconductor industry just to ensure high testability of products. A number of semiconductor companies estimate that about 50% of the cost is spent on enhancing testability of a design and on actual testing of a design.

The testability of digital circuits can be addressed using different methodologies. In *external testing* deterministic test generation is performed on a circuit. A test for a fault is an input (a vector or a sequence of vectors) that will produce different outputs in the presence and absence of the fault, thus making the fault effect observable. Test generation is a NP-hard problem and numerous heuristics and cost functions have been developed to prune the search space and speed-up test generation [2]. The complexity of test generation, especially on todays designs, gave rise to *design-for-testability* (DFT) and *built-in self-test* (BIST) techniques. Design-for-testability techniques deal with reducing the test generation effort for sequential circuits. These techniques improve the *controllability* and *observability* of the circuit that contribute to the complexity of test generation. Examples of DFT include full and partial scan. Full scan techniques modify all storage elements (latches and flip-flops) to have shift capabilities while partial scan modify only some of them. Test vectors are scanned in serially from circuit I/Os and the responses are scanned out through the scan chains. Though scan techniques reduce the test generation effort for sequential circuits, they result in an increase in gate count, pin count, routing overhead and test time [3],[4],[5],[6].

Built-in self-test involves the modification of a design to give it the capability to test itself. In BIST two basic test functions are required on chip: 1) capability of generating and supplying test patterns to the logic under test, and 2) capability of collecting and compressing test responses of logic under test. A wide variety of such BIST techniques, ranging from partial intrusion BIST to full BIST have been studied [6]. Depending on the test function performed, different types of test registers (called BIST resources) can be designed for performing the two test functionalities mentioned above. A BIST design is tested by first initializing (seeding) BIST registers,running the circuit in the test mode for thousands of clock cycles, and finally comparing compacted signature(s) with the *golden* signature(s) to determine whether the circuit is good or bad. The important issues in BIST are selection of appropriate registers to be modified as BIST registers, design of efficient BIST registers that produce test vectors that achieve high fault coverage in a short time and with low aliasing probabilities. It is desirable to attain all these goals with minimal area overhead [6],[7]. One way of minimizing BIST area overhead at the RT-level is using *minimal intrusion* BIST which is described in detail in Chapter 3.

1.2 VLSI Design Flow

The state-of-the-art overall VLSI design flow is shown in Fig. 1.1. It has the following three synthesis tasks.

1. *High-level synthesis* starts with a behavioral description of a digital system and synthesizes a register-transfer level (RTL) macroscopic structure consisting of functional units, registers and multiplexers (or buses).
2. *RTL/logic synthesis* starts with an RTL description and synthesizes a logic or gate level netlist using a standard cell library of cells.
3. *Physical synthesis* deals with placement, layout and routing of the gate level netlist.

Each synthesis task goes through a number of iterations until the design constraints are met. The test tasks are shown in bold in Fig. 1.1. In the current synthesis flow, testability is considered after the RTL/logic synthesis stage. Test pattern generation or modification of design for scan or BIST is done after the logic level netlist has been synthesized. There is a trend towards incorporating more and more of testability features in an RTL design before the logic synthesis stage. To ensure that testability insertion at RT-level meets the testability goals of area overhead and test quality, testability considerations need to be incorporated into the

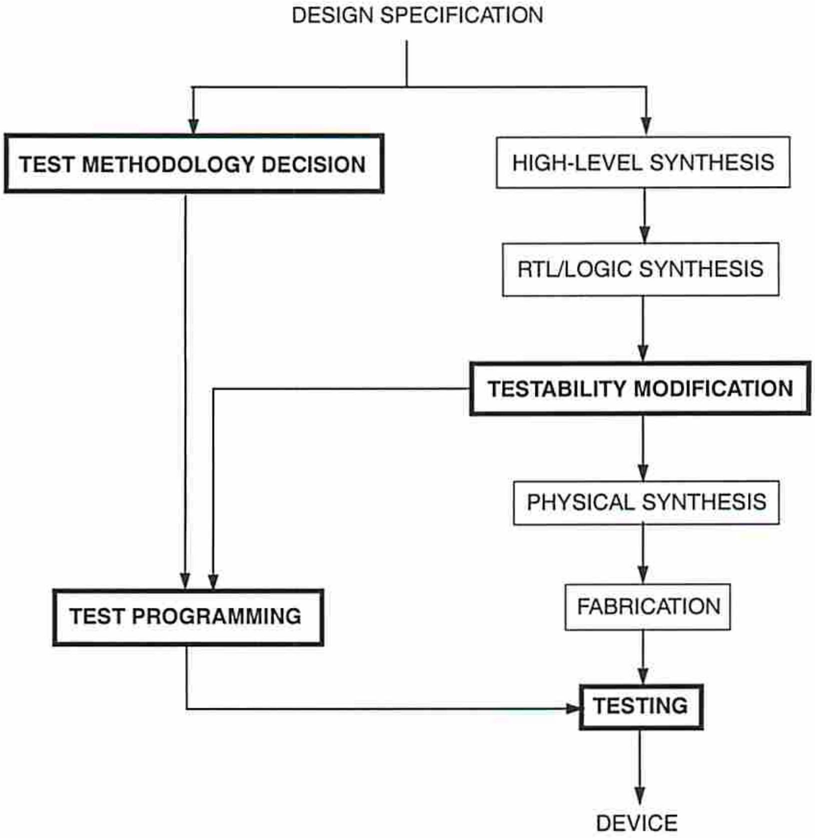


Figure 1.1: VLSI design flow

preceding stage in the synthesis flow, namely, high-level synthesis. A brief overview of high-level synthesis is presented next.

1.3 High-level Synthesis

High-level synthesis consists of the construction of a macroscopic RTL structure of a digital circuit that implements a given behavior and satisfies a set of design constraints. The behavior is modeled by data-flow graphs (DFGs) or sequencing graphs [8]. The synthesis problem addressed here is for *synchronous mono-phase* digital circuits. The target applications are computation-intensive and data path dominated digital architectures such as DSP, video and graphics applications. The outcome of high-level synthesis is a data path comprising of *functional modules* such as adders and multipliers, *registers* that store data values and *steering logic circuits* (multiplexers or buses) that transports data to the appropriate destination at the appropriate time.

1.3.1 Constraints in High-level Synthesis

Constraints in high-level synthesis can be classified into two groups: *interface constraints* and *implementation constraints.* Interface constraints are additional specifications to ensure that a circuit can be embedded in a given environment. They relate to the format and timing of I/O data transfers. For example, the timing separation of I/O operations can be expressed as timing constraints that ensure that a synchronous I/O operation follows/precedes another one by a prescribed number of clock cycles.

Implementation constraints reflect the desire of the designer to achieve a structure with certain properties. Area constraints and performance constraints are examples of implementation constraints. Circuit implementations are evaluated on the basis of area, cycle-time (i.e. the clock period) and latency (i.e. the number of cycles required to perform all operations). The synthesis process explores the design space to search for a design satisfying the all the given constraints and optimizing some (or all) of them. The goal of this thesis is to incorporate *testability constraints* of *BIST area overhead* and *fault coverage* of a design.

1.3.2 Sub-tasks in High-level Synthesis

The fundamental high-level synthesis and optimization problem can be stated as follows.

Given 1) a behavior in the form of a data-flow graph or a sequencing graph, 2) a set of functional resources, fully characterized in terms of area and execution delays, and 3) a set of constraints, synthesize a design that satisfies all the constraints and is optimum with respect to some (or all) design parameters such as area, performance and power consumption.

The main steps involved in high-level synthesis are stated below.

- **Behavioral Transformations:** These transformations aim at optimizing the behavior of the design. Obvious transformations are compiler optimizations such as constant propagation, dead code elimination, common subexpression elimination and loop unrolling [9]. Other transformations are more specific to high-level synthesis such as substituting multiplication by a power of two with selection of appropriate bits, taking into account commutativity of operators, increasing operator-level parallelism and reducing the number of levels in the data flow graph. Transformations have been studied for variety of goals in high-level synthesis, including area, performance, fault tolerance and partial scan overhead [10],[11],[12],[13],[14].

- **Scheduling (The temporal domain):** In this stage each operation in the behavior is assigned to a point in time. In synchronous systems, time is measured in *control steps.* The two fundamental scheduling approaches in relation to high-level synthesis are: *resource constrained scheduling* which aims at optimizing the number of control steps given a constraint on the number of resources, and *time constrained scheduling* which aims at optimizing the number of resources required given a constraint on the number of control steps. Other approaches that try to optimize both have also been studied. Scheduling is high-level synthesis is an NP-hard problem [15] and several heuristic scheduling techniques have been proposed such as As-Soon-As-Possible (ASAP) scheduling, As-Late-As-Possible (ALAP) scheduling, list scheduling, force-directed scheduling and path-based scheduling [16].

- **Allocation or Assignment (The spatial domain):** In this phase each operation, variable and data transfer is assigned to a piece of hardware. Allocation involves determining the number of resources whereas assignment (or binding) refers to the actual mapping to the resource instances. We will use the term assignment to refer to the combined tasks of allocation and binding, unless allocation is being referred to specifically. Assignment naturally falls into three parts: *functional module assignment, register assignment* and *interconnect assignment.* The goal of assignment is to minimize the overall hardware by sharing hardware, i.e. operations can share functional units, variables can be mapped onto common registers and data transfers can share buses and multiplexers. The three parts are interdependent and various orders of assignment have been studied in addition to concurrent assignment of modules, registers and interconnect. The types of assignment algorithms studied can be classified into heuristic assignment, linear programming approaches and graph-based approaches [16].

The scheduling and assignment tasks are interdependent. A schedule dictates the minimum number of resources required whereas a constraint on the assignment of resources may impose constraints on the schedule. Concurrent optimization of scheduling and binding is intractable and some linear programming approaches have been investigated [17],[18].

Many high-level synthesis systems have been developed in universities. Research in academia has helped create an algorithmic basis for the field of high-level synthesis [19],[20],[21],[22]. More recently, the acceptance of high-level modeling and functional simulation has spurred the development of high-level synthesis systems in the industry. Efforts in the industry by IBM [23], SIEMENS [24], IMEC [25], AT&T [26] and GM [27],[28] have contributed significantly to making high-level synthesis practical and bringing it closer to production use. The transition of high-level synthesis systems to the marketplace has been a slow one, but recently they have come of age and are being used by ASIC designers to significantly cut design cycle times [29],[30],[31].

1.4 Motivation and Goals

To reduce design cycle time and keep test costs under control, testability considerations need to be incorporated at early stages of synthesis flow. For BIST of a data path, functional registers in the data path are modified to give them the test functionalities of generating and supplying test patterns and collecting and compressing test responses from different portions of logic under test. Depending on the test function, four different types of test registers (called BIST resources) can be designed: 1) test pattern generation capability only (TPG), 2) test response compression or signature analysis capability only (SA), 3) test pattern generation and response compression capability at different times (BILBO), and 4) simultaneous test pattern generation and response compression capability (CBILBO). Different mappings of test register type to functional registers exist so that all functional modules in the data path are tested. The important issues in BIST are selection of appropriate registers for modification and design of efficient BIST registers that generate test patterns with high fault coverage and compress test responses with low aliasing probability [32]. Issues such as fault coverage and aliasing probabilities require detailed structural information of logic blocks which is not available before the RTL/logic synthesis stage. Hence these issues are best dealt with during later stages of synthesis. One way of reducing BIST cost at RT-level is by selecting a small number of registers for BIST modification.

Once the RTL design has been synthesized by high-level synthesis, there is limited flexibility in choosing registers for BIST modification. The choice of a BIST solution for a synthesized data path is restricted by the given RTL architecture. To test all modules, a considerable amount of area penalty can be incurred. The area penalty comes from 1) requirement of a large number of BIST resources, and 2) requirement of *expensive* BIST resources to test a data path. Ignoring BIST area during high-level synthesis has two consequences. Firstly, a significantly larger BIST area overhead might be incurred than the minimum possible. Secondly, the design cycle time is lengthened. Fig. 1.2 shows a three-dimensional design space. The testability axis corresponds to *test cost* which is reflected by parameters such as test generation time, fault coverage, test area overhead and test application time. In our case, the testability axis corresponds to impact on area because of testability. Design point A might have a low area and delay after synthesis but adding testability to this design might result in an inferior design point, B. On the other hand if testability is considered during synthesis a design point such as C, even though inferior to A, will result in a superior design point D after testability has been added. Fig. 1.3

shows the reduction in design cycle iterations as a result of considering testability during high-level synthesis. If the total area of the design after BIST modification exceeds the allowable chip area, many synthesis iterations might be required before finding an RTL design with BIST area overhead such that the total area is within constraints Fig. 1.3(a).

The goal of this thesis is to incorporate BIST resource optimization techniques in high-level synthesis. Many different RTL implementations can be synthesized from a given behavior. Of those some are optimum in terms of functional area and performance. Among the area-performance competitive implementations, some require fewer BIST resources than others. This thesis proposes optimization techniques that can be used in high-level synthesis to target good testability solutions in the optimum area-performance solution space. Optimization techniques in all the domains of high-level synthesis, namely assignment, scheduling and behavioral transformation, are considered under the purview of this thesis. Each domain affects the solution in a different way. Properties of each domain with regards to effect on BIST resources required to test a synthesized data path are studied. Techniques that exploit the properties in each domain have been proposed. Depending on the actual behavior, the degree of BIST resource optimization in a specific domain varies. Some behaviors lend themselves easily to BIST resource optimization in the assignment domain, while others in the scheduling domain. Together, these techniques provide an efficient way of cutting down design cycle time and minimizing BIST resource overhead and total area of synthesized data paths.

1.5 Outline of Thesis

In the next chapter an overview of previous work in high-level synthesis for testability is presented. The work described deals with different testability objectives such as test generation time, partial scan overhead and BIST overhead. In Chapter 3, the high-level synthesis model used in this research is presented. The concepts of BIST resources and BIST embeddings in a data path are presented. A low area overhead BIST methodology based on these concepts, called *minimal intrusion BIST*,

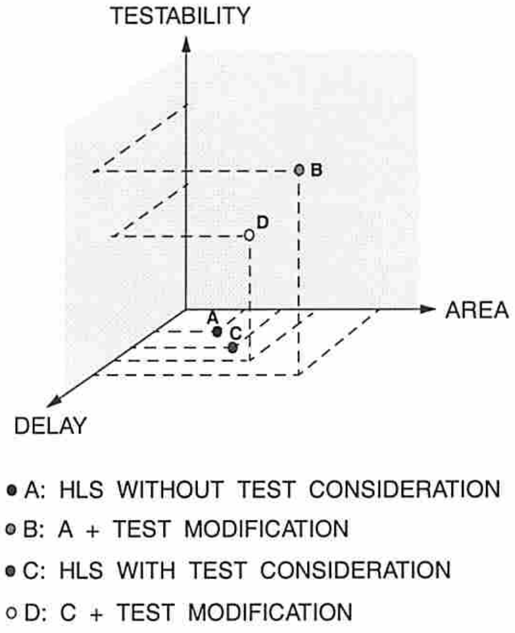

TESTABILITY
B
D
AREA
A
C
DELAY
● A: HLS WITHOUT TEST CONSIDERATION
● B: A + TEST MODIFICATION
● C: HLS WITH TEST CONSIDERATION
○ D: C + TEST MODIFICATION

Figure 1.2: Three-dimensional design space

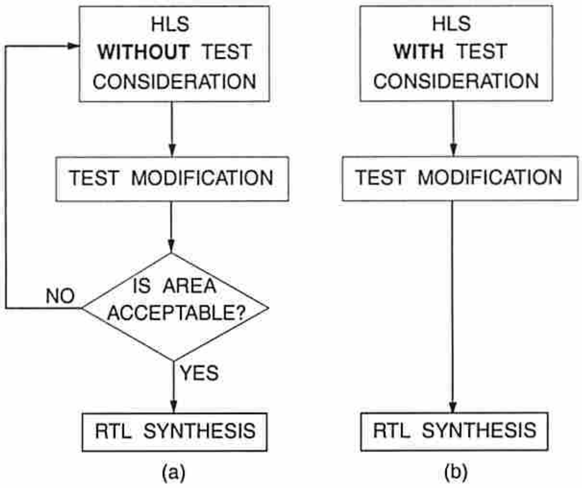


Figure 1.3: Reduction in design cycle time

is described. Structural properties of RTL data paths that contribute to optimum BIST solutions are discussed. Chapter 4 deals with estimation of BIST resources during high-level synthesis. Estimation is crucial for efficient design space exploration in any high-level synthesis system. We develop concepts that are useful in deriving lower bounds on the number of BIST resources required for a data path. Efficient algorithms to compute these bounds are described. The use of lower bound estimation techniques at various stages of synthesis is also discussed.

The next three chapters deal with each domain of high-level synthesis. In Chapter 5 we describe register and interconnect assignment techniques for reducing BIST area overhead. We develop the concepts of *test variables* and *BIST function* of a register that are used in constructing a register and interconnect assignment with low BIST area overhead. In Chapter 6, we present properties of the scheduling and module assignment domain that can be exploited for reducing BIST resources. An algorithm which performs scheduling and module assignment simultaneously is presented. In Chapter 7 we discuss how transforming a behavior can be useful in reducing BIST resources. The concept of redundancy and *identity-nodes* is introduced and a technique for modifying a behavior by adding redundant computations is described. The introduction of redundancy enables subsequent synthesis stages to search a design space with lower BIST resource requirements. We conclude in Chapter 8 with a summary of the contributions made by this dissertation.

Chapter 2

Background

In this chapter we present a brief overview of prior work in high-level synthesis for testability. Different testability objectives that can be used at the RT-level, such as increasing BIST concurrency and decreasing difficulty of test generation, partial scan overhead and BIST area overhead, are discussed. A summary of previous research targeting these different objectives is presented.

2.1 Testability at the RT-level

The success of high-level synthesis in optimizing area and performance can be attributed to ease of abstraction of these design metrics to higher levels. Testability is a broad term that refers to the ease of testing manufactured circuits. It manifests itself in quantities such as test generation time, fault coverage, test application time and test control complexity. Testability has been well characterized at the gate-level. For high-level modeling of testability, identification of testability-determining properties that are *independent* of the gate-level details is necessary since gate-level circuit information is not available during high-level synthesis. Various notions of a *testable* data path can be used as objectives in high-level synthesis algorithms.

- **Test pattern generation:** RTL data paths can be synthesized for which a sequential test generator can generate tests in a short time for a high fault coverage. RTL structural properties such as number of lines difficult to control or observe, sequential depth and number of loops are indicators of ease of test generation and high fault coverage. The concept of hierarchical testability in which high-level module information is used to *justify* and *propagate* pre-computed test sets also lends itself very well to RTL data paths. However, sequential test generation is inherently a very complex process and might not be realistic for very large data paths in spite of this improvement. Besides, this approach does not assume any pre-defined test methodology which is usually the case in practical designs.

- **Test application time:** Most sequential circuits use some form of DFT technique to alleviate the test generation problem or some form of BIST technique to avoid it completely. DFT techniques such as full scan and partial scan have large test application times because the patterns have to be shifted in and out of the scan paths. In circuits using BIST, test schedules can result in a large number of test sessions which in turn results in high test application time. Reducing the number of scan registers in the case of DFT and test sessions in the case of BIST, reduces the total time required to test a data path.

- **Test area overhead:** DFT and BIST techniques involve the modification of the hardware on the chip for test purposes. In partial scan and full scan, the storage elements selected to be scan elements have to be modified to possess the shift capability and have to be connected in a chain to form scan paths. In partial scan the objective is to select a subset of the storage elements such that it eases the test generation effort but keeps the area overhead to a minimum. Data paths that use this definition of testability synthesize data paths which which require a small number of scan elements to achieve the desired fault coverage with a reasonable amount of test generation effort. In BIST, registers are reconfigured to support test pattern generation and signature analysis during the test mode. The area overhead is proportional to the number of registers modified for this purpose. At the RT-level, the number of scan or BIST resources required to achieve a desired level of fault coverage can be used as a measure of testability.

2.2 Previous Approaches

2.2.1 Test Generation Efficiency

At the gate-level, difficulty of test generation is characterized by lines that are hard to control and/or hard to observe. It has also been shown that sequential depth and cycles with many flip-flops increase test generation effort significantly. During high-level synthesis there is no control over the gate-level structure of modules. However these structural properties can be also be identified at the RT-level and hence used during high-level synthesis [33],[34],[35]. In [33] testability optimization is achieved during the register assignment phase. The register binding uses the following testability goals: 1) minimize sequential depth between input and output registers, 2) minimize the number of cycles and 3) maximize the number of input and output registers. The scheduling algorithm [34] uses these same testability goals to generate a schedule such that the possible module assignments will not affect the optimization to be achieved in the subsequent register assignment phase. Partial scan implementations of the synthesized designs were studied in [35]. Genesis is a high-level test synthesis system that uses the notion of hierarchical testability to reduce test generation effort [36],[37]. Given a gate-level test set for each module, Genesis guarantees that a sensitizable path through the module embedded in the data path is established during synthesis so that the module can receive the test set from the primary inputs and the module outputs can be observed at the primary outputs. The complete system level test set is therefore obtained as a byproduct of high-level synthesis in much shorter time when compared with gate-level test pattern generation.

2.2.2 Partial Scan Overhead

Partial scan is used to reduce test generation complexity by modifying some storage elements in a data path into scan elements. An area overhead is incurred as a result of the modification. Storage elements that cut loops have been shown to be good candidates for scan elements [5].

A module and register binding algorithm with the objective of minimizing registers in loops was proposed by Mujumdar et al. [38]. The binding algorithm attempts to minimize cost that is a function of the number of loops and the number of input and output registers in the generated data path logic. If breaking all sequential loops is the goal of partial scan then a better objective for minimizing partial scan overhead is to have a small set of registers which, when made scannable, break all loops. Dey et al. proposed an assignment method that uses low cardinality minimal feedback vertex set of registers as the objective [39]. An extension of this work deals with behavioral transformations and techniques to achieve low partial scan overhead [13].

2.2.3 BIST Concurrency

One of the costs associated with BIST is the test time required for high fault coverage. Increasing *test concurrency* of a data path, increases the number of parts of a design that can be tested simultaneously thus reducing total test time. The data path characteristic that limits test concurrency is the existence of hardware sharing conflicts between different tests of the design [40]. A test path is defined as a subgraph of the data path through which test patterns must propagate in order for a component to be controlled or observed. At the high-level synthesis stage, test paths are not known, so test path conflicts cannot be directly avoided. In [41] and [42], metrics are defined that evaluate a DFG and identify test concurrency problems during synthesis. The effect of scheduling and binding on test concurrency is characterized. Test concurrency problems that remain after high-level synthesis are dealt with during definition of test paths and test plans by insertion of additional test registers and pipelining of test data through non-test registers [43].

2.2.4 BIST Area Overhead

BIST requires the modification of registers in a data path. For designs that support parallel BIST, Avra proposed a register assignment technique that aimed at minimizing the number of self-adjacent registers in the design [44]. The assumption in this work is that every self-adjacent register has to be modified into CBILBO [45] in order to test the data path and hence the overhead is high. Avra also showed that using *orthogonal scan paths* in data path logic allows for greater sharing of functional and test logic when testability techniques such as scan and circular BIST are implemented [46]. A synthesis procedure that assumes that the data path logic has an orthogonal scan path structure, then biases the register binding algorithm such that the occurrence of the types of functions that allow for logic sharing is maximized, is presented in [47].

Papachristou et al. presented a combined register and module assignment method that generated self-testable designs that did not have any self-loops [48]. The approach is based on constraining the assignment to generate a self-testable template and hence results in exploring a small subspace of the testable design space. Later they presented an improved method of generating self-testable designs by extending the self-testable template to include self-loops only in *one* specific configuration [49]. Though more general than the previous approach in terms of allowable templates, this approach is still restrictive because it merges modules, registers and interconnect simultaneously thus not utilizing the flexibility provided by the separate optimization of these sub-problems.

None of the previous approaches address the *minimal intrusion* BIST methodology which involves testing as much of the design as possible using a minimal number of resources for the purpose of test. All the previous approaches are based on a very rigid BIST template. Moreover, they deal only with the assignment phase of high-level synthesis [44],[50],[49]. The scheduling and behavioral transformation phases have not been explored in previous work. The subsequent chapters of this dissertation will address these issues.

Chapter 3

Synthesis and Testability Models

In this chapter we present the high-level synthesis model used in the research. We also introduce the BIST model. Different types of BIST resources and a minimal intrusion BIST methodology for data paths using these resources is described.

3.1 High-level Synthesis Model

3.1.1 Data Flow Graph Model

The input to high-level synthesis is a behavioral specification that consists of high-level language constructs such as conditional statements, assignment statements and loops, and the output is a RTL structural description of the design. The behavioral specification is stored internally as data flow graph(DFG).

Definition 1 (Data Flow Graph) *A data flow graph (DFG) is a graph* $G = (V, E)$*, where:*

1. $V = \{v_1, v_2, ..., v_n\}$ *is a finite set whose elements are nodes, and*

2. $E \subset V \times V$ *is an asymmetric data flow relation whose elements are directed edges.*

The nodes in a DFG represent operations. A directed edge (v_i, v_j) from $v_i \in V$ to $v_j \in V$ exists if the data produced by operation o_i (represented by v_i) is consumed by operation o_j (represented by v_j). A data flow graph example is shown in Fig. 3.1, representing the computation $e := (a*c)+(b+d)$. The data flow graph $G = (V, E)$ is composed of three nodes, v_1 representing operation $*$, and v_2 and v_3 representing a + operation each. Both data produced by v_1 and v_2 are consumed by v_3. An alternative representation of a DFG $G_{s,t} = (V, E, s, t)$ consists of two additional pseudo-nodes. A source node s with no incoming edge and primary input variables as outgoing edges and a sink node t with primary output variables as incoming edges and no outgoing edges. (The representation of G will be used for all discussions and $G_{s,t}$ will be used wherever required.) In the synchronous data flow model we assume that an edge may hold at most one value. Then, we assume that all operators consume their input values before new input values are produced on their input edges.

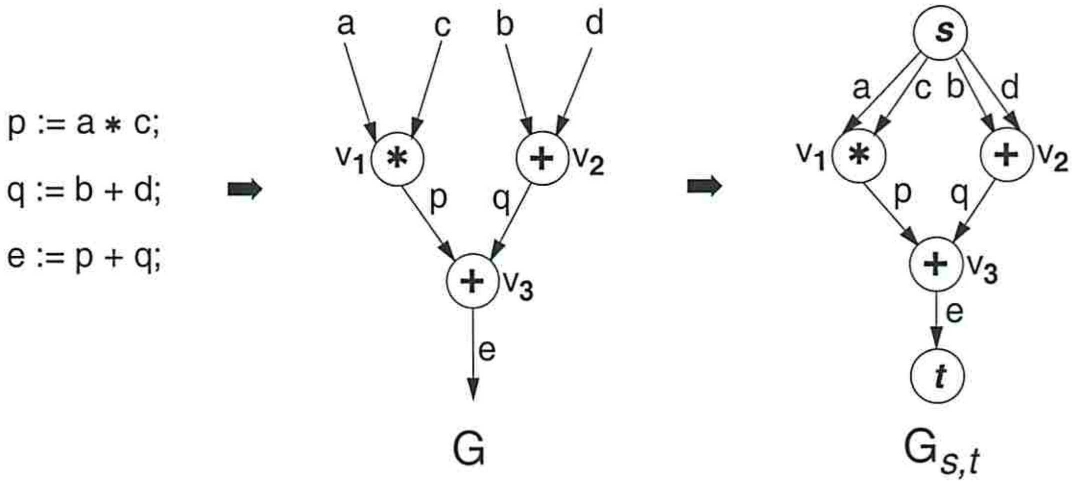


Figure 3.1: DFG representation

3.1.2 Data Path Hardware Model

The output of high-level synthesis is a RTL structural description of the design. It has two parts: an *operative* part, called the data path and a *control* part called the controller. The two parts are different, structurally and functionally. Hence the synthesis algorithms used for these two parts are different. Also the testability solutions tend to be different. In this research we focus on the data path portion of a design. A data path is composed of three types of components:

1. **Functional units (Modules):** These execute operations specified in the behavioral description. The modules could be simple units such as adders and multipliers or more complex units such as co-processors.

2. **Storage units:** These hold values of variables and constants generated and consumed during the execution of the behavior. Registers, register files, RAMs and ROMs are examples of these types of components.

3. **Communication units:** These construct the communication network for data transfers between the functional modules, the storage units and external ports. Buses, multiplexers and switches are examples of communication units.

We assume a data path model consisting of single operation non-pipelined functional modules. Each functional module executes in one clock cycle and can perform only one type of an operation such as addition or multiplication. A storage model with individual registers and a communication network with point-to-point multiplexers is assumed. Fig. 3.2 shows a structural view of the output of high-level synthesis.

3.1.3 Data Path Execution Model

The data path is synchronous with respect to a single-phase clock. In each clock cycle the following actions take place: 1) the controller enters a new state, computes the control signals and the next state; 2) the controller sends the control signals to the data path; 3) the data path executes the operations corresponding to the control signals; and 4) the results of the operation executions are stored in registers. The operation execution model assumes that chaining of operations or multi-cycling is not allowed. A result of an operation execution is stored in only one register even if the result is used by different functional modules. Storage value forwarding is not allowed. If a result is not used for consecutive clock cycles, it is stored in one register with a **hold** mode that enables the register to hold the data until it is used. The

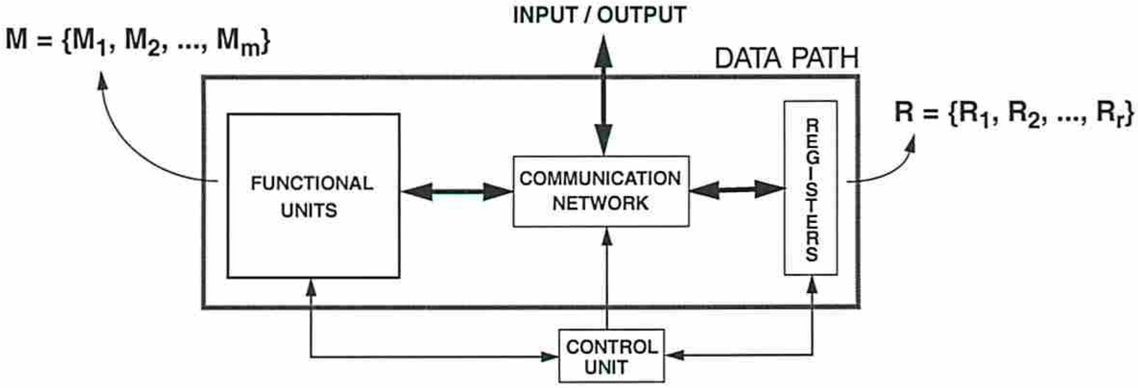


Figure 3.2: Structural view of RTL design

complete computation is done in a non-pipelined manner. Computation on input data is started only after computation on previous input data is completed.

The emphasis of this thesis is on studying synthesis and establishing the properties of various synthesis stages relevant to optimizing BIST resources. For the purpose of demonstrating the validity and feasibility of the techniques proposed in this research a complete current generation high-level synthesis system developed at the Stanford University called TOPS (**T**estability **OP**timized **S**ystem) was used. The input to TOPS is behavioral VHDL and the output is structural RTL VHDL. The synthesis flow of TOPS and the subset of VHDL that TOPS is able to handle is described in detail in [51].

3.2 Testability Model

The high-level synthesis process assumed in this research is directed towards synthesizing data paths that are to be tested using a pseudo-random BIST methodology. In pseudo-random BIST, two test functionalities are necessary on the chip: 1) capability of **generating** pseudo-random test patterns, and 2) capability of **compressing** test responses into a signature. Functional registers in the data path can be modified by adding extra hardware to provide these test functionalities in the test mode. One way of making a data path self-testable is to modify *all* functional registers and give them test capabilities. Such an approach is an overkill in terms of the area overhead. The self-test methodology adopted in our approach employs minimal intrusion BIST, in which only a subset of the functional registers are modified for self-test of the data path. Usually portions of a data path, such as multiplexers, interconnect and functional registers can be easily tested using functional patterns. The components that are hard to test using functional patterns are the modules such as ALUs, adders and multipliers. Hence we modify a subset of the registers and give them the capability of generating pseudo-random test patterns for the modules in the data path and for compressing the responses from these modules into signatures. A partial intrusion BIST version of an RTL data path is shown in Fig. 3.3. The goal of partial intrusion BIST is to test all the functional modules using a subset of registers in the test mode. A part of the communication network (multiplexers and interconnect) are tested for free in the process as depicted in Fig. 3.3. Any portion of the data path not tested by this self-test methodology is tested using functional patterns. *Note that in this partial intrusion BIST methodology the test resources and paths used to generate, transport and collect test data are a subset of the functional data path. No additional data path components such as registers, multiplexers or interconnect are added for the purpose of testing.*

3.2.1 BIST Resources

In minimal intrusion BIST, while in the test mode some registers in the data path can be configured to support test pattern generation, some to support test response compression (or signature analysis), and some to perform both of these test functionalities. When both the functionalities are performed by the same register, the issue of concurrency of these functionalities arises. The functionalities of generation and compression can be performed at different times (non-simultaneously) or they can be performed simultaneously. Depending on the test functionalities performed

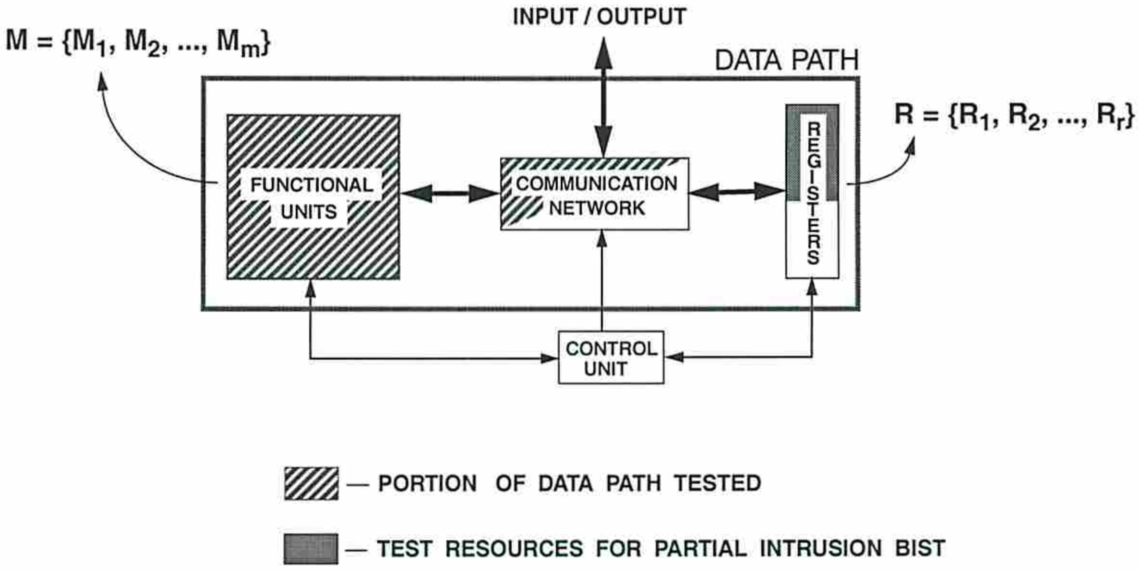


Figure 3.3: Partial intrusion BIST in RTL design

and their concurrency, four different types of BIST registers are possible. Table 3.1 summarizes the four types of BIST registers.

3.2.2 Minimal Intrusion BIST

Different mappings of functional registers to BIST register types are possible in partial intrusion BIST. Minimal intrusion BIST is a mapping that requires a minimal number of BIST resources such that the area overhead for modification is minimum. The various possible mappings can be explored using the concept of a **BIST embedding**. Consider a typical data path as shown in Fig. 3.4(a). Any register that is connected to an input port of a functional module through only multiplexers can supply test patterns to that input port. Such a register is a possible choice for modification as a BIST register that can generate patterns. Let IR_j^L denote the

Table 3.1: Types of Test Registers

Test Register	Test Activity performed	Concurrency of Test Activities	Type of BIST Register	Area[†] (16-bit)
1	Generate patterns	-	TPG	528
2	Compress responses	-	SA	528
3	Generate patterns, Compress responses	Non-Simultaneous	BILBO	688
4	Generate patterns, Compress responses	Simultaneous	CBILBO	960

[†] Cell units using a macro-cell library from LSI Logic Corp. [52]

set of registers connected to the *left* input port of module M_j through only multiplexers and IR_j^R denote the set of registers connected to the *right* input port of M_j through only multiplexers. The registers belonging to IR_j^L and IR_j^R are called **input registers** of M_j. Similarly any register that collects data from the output port of a module through only multiplexers is a possible choice for modification as a BIST register to compress test responses. Let OR_j denote the set of these **output registers** of module M_j.

Definition 2 *In a RTL data path, a* **BIST embedding** *of a module M_j is a selection of registers from $IR_j^L \cup IR_j^R \cup OR_j$ such that each input port of M_j receives test patterns from a* distinct *register and the output port transfers test responses to a register.*

Consider the RTL data path shown in Fig. 3.4(a). Fig. 3.4(b) and (c) show BIST embeddings for the multiplier and the adder module, respectively. The highlighted arrows indicate the registers selected to generate test patterns for each input port and the register selected to compress test responses for the output port. The type of the test register (TPG, SA, BILBO or CBILBO) is determined by the test function that the register performs in the chosen embeddings. If the embeddings for different modules are chosen such that the same register generates test patterns for one module and compresses test responses for a different module, then the register has to be modified to be a BILBO, since it needs to perform both test functionalities of generation and compression, however, not simultaneously. For example, in the embeddings chosen in Fig. 3.4(b) and (c), R_2 generates test patterns for the multiplier and compresses test responses for the adder. Hence R_2 will have to be modified to a BILBO. Note that the two modules in such a case would be tested in different test sessions. First, the adder would be tested completely and then the multiplier would be tested (or vice-versa). If an embedding is chosen such that the a register generates test patterns for a module and compresses test responses for the same module, the register has to be converted to CBILBO, since it would have to perform functionalities of generation and compression, simultaneously. For example, if the embedding shown in Fig. 3.4(d) was chosen for the adder, R_2 would have to be modified to a CBILBO. An embedding such as the one chosen in Fig. 3.4(e) is not valid even though both the input ports of the adder receive test patterns. The correlation between the test patterns at the left and right input port would decrease the number of unique patterns that can be applied to the input of the module by R_4, resulting in poor test quality. Hence such embeddings are forbidden in our BIST methodology.

3.2.3 ILP Formulation for Minimum Area BIST

We now present an ILP formulation to find BIST embeddings for all modules in a data path such that the cost of modification is minimum. Consider a synthesized data path comprising m modules $M_1, M_2, ..., M_m$ and r registers $R_1, R_2, ..., R_r$. We associate with each register R_i, four integer variables, namely, R_i^T, R_i^S, R_i^B and R_i^C. Each of them has a value of 0 or 1. The superscripts T, S, B and C denote modification to **T**PG, **S**A, **B**ILBO and **C**BILBO, respectively. If R_i is modified into a BIST register, the value of the variable corresponding to the type of BIST register is 1. Otherwise the value of that variable is 0. In a BIST solution each register R_i is either a normal register (unmodified for BIST) or one type of a BIST register, i.e. TPG, SA, BILBO or CBILBO. This gives us the following constraint for every R_i.

$$R_i^T + R_i^S + R_i^B + R_i^C \leq 1 \tag{3.1}$$

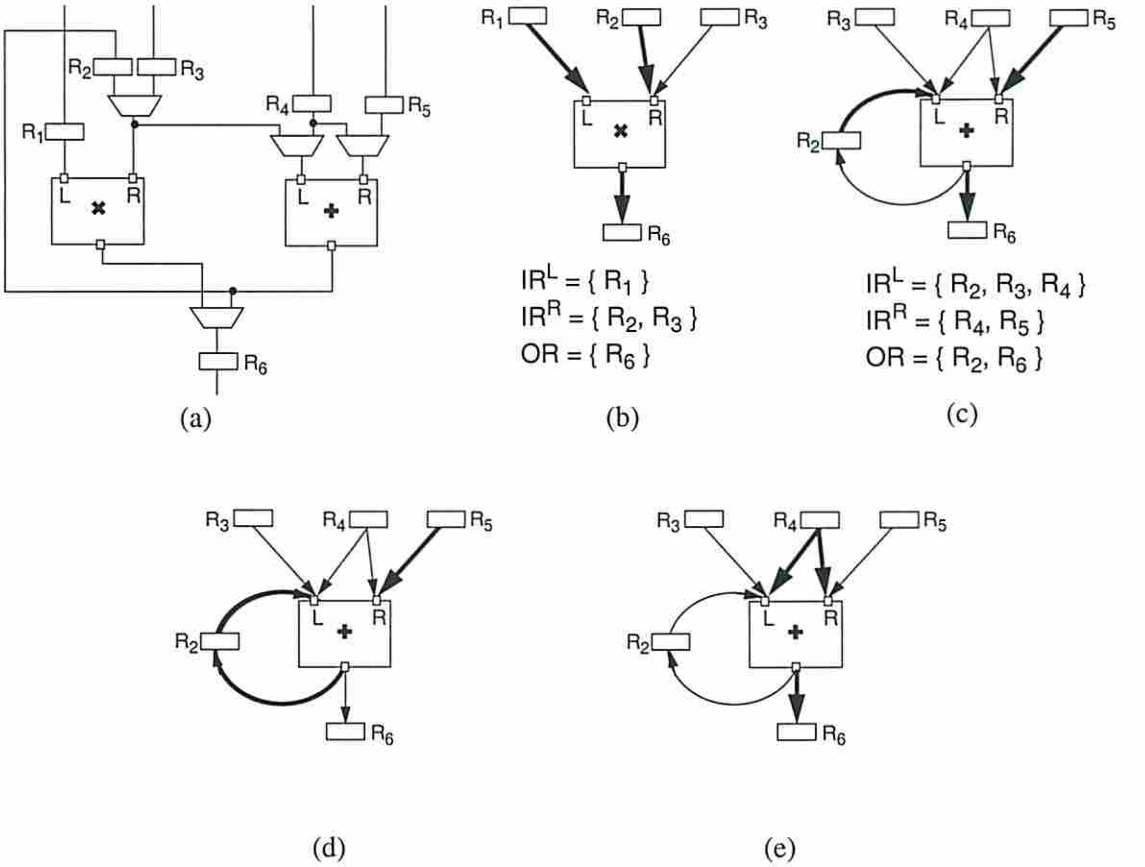


Figure 3.4: Selection of test registers for BIST

A BIST solution should have an embedding for every module in the data path. An embedding for a module M_j has at least one input register that can generate patterns for the left input port and at least one input register that can generate patterns for the right input port. It also has at least one output register that can compress test responses from the module. A TPG, BILBO or a CBILBO can be used to generate test patterns. An SA, BILBO or CBILBO can be used to compress test responses. This gives us the constraints for covering every port of M_j with a BIST register.

$$\sum_{i \in IR_j^L} (R_i^T + R_i^B + R_i^C) \geq 1 \tag{3.2}$$

$$\sum_{i \in IR_j^R} (R_i^T + R_i^B + R_i^C) \geq 1 \tag{3.3}$$

$$\sum_{i \in OR_j} (R_i^S + R_i^B + R_i^C) \geq 1 \tag{3.4}$$

Suppose there exists register R_i that is an input as well as an output register of M_j. The variables R_i^C and R_i^B would appear in an input port constraint (3.2 or 3.3) as well as in the output port constraint (3.4). The above constraints would then be satisfied with $R_i^C = 1$ (or $R_i^B = 1$) and the rest of the variables in the equations equal to 0. This implies that R_i is the only register that generates test patterns for an input port and compress responses for M_j. This is possible if R_i is a CBILBO and $R_i^C = 1$ corresponds to a valid embedding. However the situation where $R_i^B = 1$ corresponds to an embedding where R_i is a BILBO and it generates test patterns and compresses responses for the same module. This corresponds to an illegal embedding. To avoid embeddings where a register selected to be a BILBO is the only one to generate patterns for an input port and compress responses for an output port of the same module, the following constraints are required for M_j. These constraints ensure that for each input-output port pair, there exists one CBILBO or two non-CBILBO registers, precluding the one BILBO scenario.

$$\sum_{i \in IR_j^L \cup OR_j} (R_i^T + R_i^S + R_i^B + 2 \cdot R_i^C) \geq 2 \tag{3.5}$$

$$\sum_{i \in IR_j^R \cup OR_j} (R_i^T + R_i^S + R_i^B + 2 \cdot R_i^C) \geq 2 \qquad (3.6)$$

An invalid embedding is still possible with the above constraints. Suppose there exists a register R_i that is an input register connected to *both* input ports of M_j, i.e. $R_i \in IR_j^L, IR_j^R$. $R_i^T = 1$ would satisfy constraints 3.2 and 3.3. If that is the only variable satisfying the constraints, it would imply that R_i is the only register that supplies test patterns to the left and the right input port. This corresponds to the case in Fig. 3.4(e) and is forbidden in our methodology. To avoid this embedding the following constraint which forces each input port to have a *distinct* register to supply test patterns is required.

$$\sum_{i \in IR_j^L \cup IR_j^R} (R_i^T + R_i^B + R_i^C) \geq 2 \qquad (3.7)$$

The optimization objective is to minimize the total area of the test registers.

$$\min(\sum_{i=1}^{r}(c^T \cdot R_i^T) + \sum_{i=1}^{r}(c^S \cdot R_i^S) + \sum_{i=1}^{r}(c^B \cdot R_i^B) + \sum_{i=1}^{r}(c^C \cdot R_i^C))$$

Constants c^T, c^S, c^B and c^C are cost (area overhead) of a TPG, SA, BILBO and a CBILBO respectively. The size of the ILP is linear with respect to the number of registers and modules in the data path. There are four variables per register. Hence a total of $(4 \cdot r)$ variables are required. Constraint 3.1 is defined for each register $R_i, i = 1, 2, ..., r$. Constraints 3.2 through 3.7 are defined for each module $M_j, j = 1, 2, ..., m$. Hence a total of $(6 \cdot m + r)$ constraints are required for the ILP. The software package LINDO was used for solving the ILP for a minimum area overhead BIST solution [53].

3.2.4 Case Study

The effect of different costs of BIST registers on a minimum area overhead BIST solution is demonstrated on a data path synthesized from the *AR_filter* benchmark [54]. The data path consists of 12 modules (8 multipliers and 4 adders) and

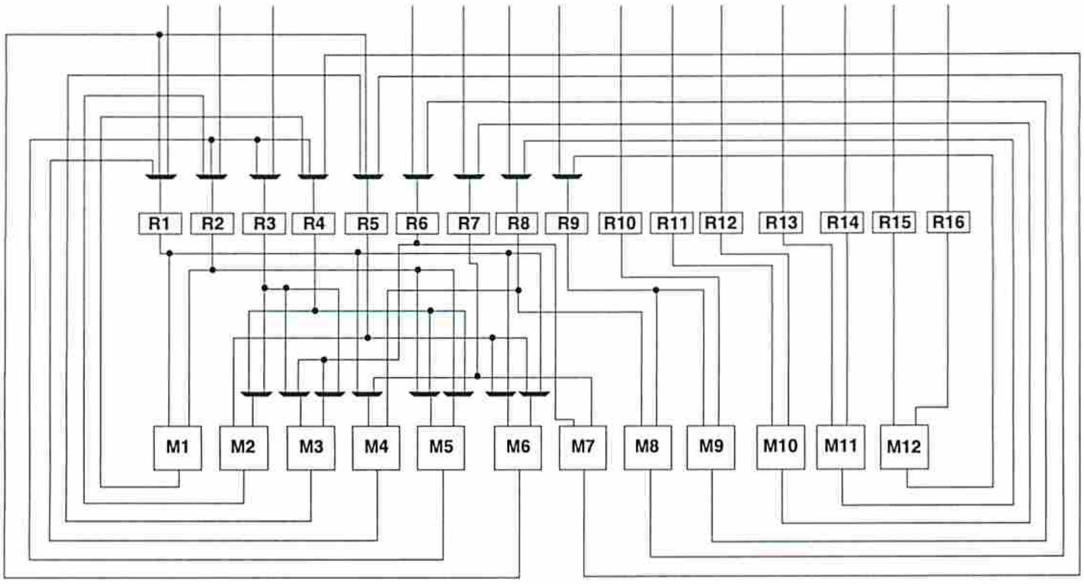


Figure 3.5: A data path synthesized from *AR_filter*

16 registers as shown in Fig. 3.5. The ILP constraints for the data path were determined as explained in the previous section. Table 3.2 shows four different sets of BIST register costs (c^T, c^S, c^B and c^C) and the optimum area overhead BIST solution generated using each set of costs. In Experiment 1, all costs are equal to 100 and the optimum BIST solution has 7 TPGs and 8 CBILBOs. The test functionality of a CBILBO subsumes that of the other three type of BIST registers and a large number of CBILBOs are selected since they are relatively inexpensive. In Experiment 2, we increased the cost of a BILBO to 200 and that of a CBILBO to 400. Now since CBILBOs are very expensive, the optimum solution does not select any register to be a CBILBO. Next to CBILBO, a BILBO register has the most test functionality and it can be seen that 7 BILBOs are selected in this case. However if the cost of BILBO is made equal to the cost of a CBILBO, as in Experiment 3, 1 CBILBO is selected. It can be seen from the first three experiments that even though the cost of a TPG is small, the optimum solution selects at least 7 TPGs. In Experiment 4, a very high cost was used for a TPG to bias the optimum solution towards selecting fewer TPGs. In spite of the high TPG cost, 7 TPGs were

Table 3.2: Effect of BIST register cost on ILP solution

Experiment	BIST register cost				Optimum solution				Total
No.	c^T	c^S	c^B	c^C	#T	#S	#B	#C	cost
1	100	100	100	100	7	0	0	8	1500
2	100	100	200	400	8	1	7	0	2300
3	100	100	400	400	8	1	6	1	3700
4	400	100	100	100	7	0	1	7	3600

selected resulting in a very high total area overhead. It can be seen from Fig. 3.5 that $R10, R11, R12, R13, R14, R15$ and $R16$ have to be made TPGs in *any* chosen embedding. One of the causes of high BIST area overhead in a data path is the presence of such BIST registers that are *essential* and expensive. In the subsequent chapters we will show how sharing test functionality of BIST registers between modules and minimizing expensive and essential BIST registers leads to data paths with low BIST area overheads.

3.3 Summary

In this chapter we have introduced the high-level synthesis model and the testability model used in this research. The minimal intrusion BIST methodology guarantees that all functional modules in a data path are tested with a minimum area overhead. Testing of the functional modules using BIST and the rest of the data path using functional tests, assures a high quality of test for the complete data path at very low cost. We have developed a 0-1 ILP that determines a minimum BIST area overhead solution for a RTL data path. In the rest of the chapters we discuss how the area overhead can be further reduced by considering BIST resource requirements early during the synthesis process.

Chapter 4

Estimation of BIST Resources

4.1 Introduction

Estimation of data path resources during high-level synthesis is essential for two reasons. Firstly, it enables the designer to evaluate the design quality by comparing the estimates of any design metric with the constraints specified for that metric. For example, if the estimated number of functional resources in the design corresponds to an area that is greater than the area constraint, then the schedule may have to be modified to allow a functional resource allocation that is within the area constraint. Secondly, estimates enable the designer to explore design alternatives by providing quick feedback for any design decision. This way a designer can explore a greater number of alternatives instead of synthesizing a complete implementation and measuring the particular design metric for each design alternative. Lower bounds on resources not only greatly reduce the size of the solution space but also provide a means to measure the proximity of the final solution to the optimal one.

There is some prior work for estimating lower bounds on functional resources [55], [56], [57], [58], [59], [60]. These works are concerned with *functional* resources such as registers, adders and multipliers. None of the approaches have a mechanism for estimating the effect of a decision on the number of BIST resources required for a data path.

In this chapter we derive lower bounds on BIST resources assuming that the schedule and module assignment is known. The lower bounds can be used as an estimate to determine the quality of a schedule, a module assignment or a register assignment in terms of BIST resources required for the synthesized data path. The bounds also serve as an independent measure of comparing the performance of different high-level synthesis systems and algorithms that perform test resource optimization.

4.2 Storage Concurrency of Variables

In the scheduling stage of high-level synthesis, a temporal sequence for execution of operations in a DFG is determined. The time of execution of various operations as determined by a schedule in turn determines the time when operands of operations are read and results of operations are written.

Definition 3 *A* **schedule** *of a DFG $G = (V, E)$ is a mapping $S : V \rightarrow \{1, 2, ..., L\}$ where for operations $o_i, o_j \in V$, $S(o_i) < S(o_j)$ if $(o_i, o_j) \in E$. $\{1, 2, 3, ..., L\}$ correspond to control steps. L is called the* **latency** *of the schedule and L_{min} denotes the minimum latency for which the DFG can be scheduled.*

Definition 4 *The* **birth time** *of a variable v, denoted by $v.birth$, is the control step in which v is first defined. The* **death time** *of a variable v, denoted by $v.death$, is the control step in which v is last used.*

Definition 5 *The* **lifetime** *of variable v, denoted by $\mathcal{L}(v)$, is the interval $[v.birth, v.death]$. A variable is said to* **alive** *during a control step c if $c \in [v.birth, v.death]$.*

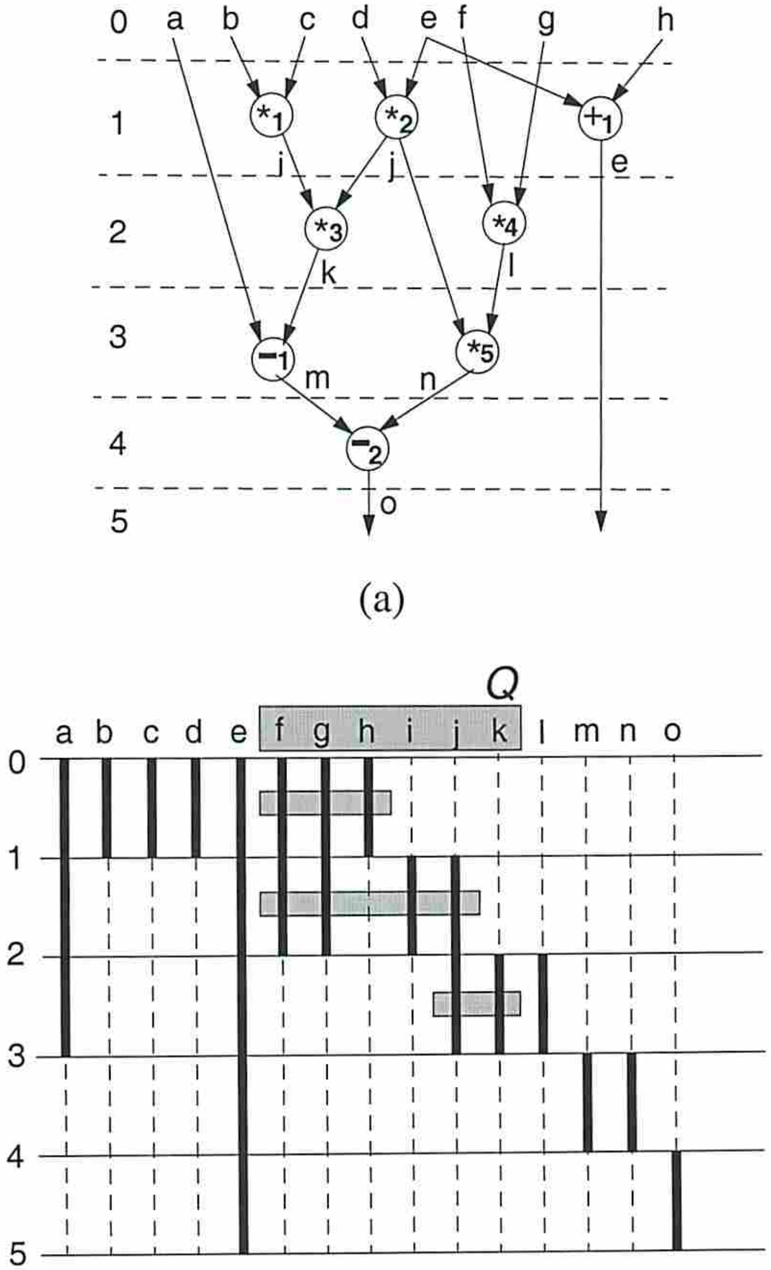
The lifetime of a variable denotes the control steps during which the variable is alive. A lifetime table of variables can be derived by performing lifetime analysis on the scheduled DFG [9]. Consider the scheduled DFG shown in Fig. 4.1(a). The lifetime table of the variables in the DFG is shown in Fig. 4.1(b). In a lifetime table, the control steps are shown on the vertical axis and the variables on the horizontal axis. The lifetime of each variable v is denoted by a bold line segment whose top endpoint corresponds to the control step in which the variable is defined ($v.birth$) and whose bottom endpoint corresponds to the last control step in which the variable is consumed ($v.death$). Any two variables with overlapping line segments cannot be assigned to the same register. The register assignment problem can be formally defined as follows.

Definition 6 *Given a scheduled DFG $G = (V, E)$, a* **register assignment** *is defined as a partition $\Pi_R = \{R_1, R_2, ...R_r\}$ of the set of variables V such that for any two variables v_i and v_j in R_k, $1 \leq k \leq r$, their lifetimes do not overlap.*

Note: In the rest of this dissertation, symbol R_k will be used to refer to the name of a register as well as the set of variables assigned to that register. The correct meaning of the symbol should be inferred from the context.

Definition 7 *The* **storage concurrency** *of a set of variables Q, $SC(Q)$ is the maximum number of variables in Q having overlapping lifetimes.*

Consider the set of variables $Q = \{f, g, h, i, j, k\}$ from the DFG in Fig. 4.1(a). It can be seen from the lifetime table in Fig. 4.1(b) that variables f, g and h have overlapping lifetimes during control steps 0-1, variables f, g, i and j have overlapping lifetimes during control steps 1-2 and variables j and k have overlapping lifetimes during control steps 2-3. The maximum number of variables from Q alive at the same time is 4 and hence $SC(Q) = 4$. The storage concurrency indicates the minimum number of registers that need to be synthesized to store variables in Q. It is equal to the size of the largest maximal subset of Q such that all variables are pairwise alive at the same time. In any valid data path implementation, the number of registers that store the variables in Q is greater than or equal to $SC(Q)$. The storage concurrency of E, the set of all variables in a DFG gives the minimum number of registers required in implementing any data path from the DFG. For the DFG in Fig. 4.1(a), $SC(E) = 8$.



(a)

(b)

Figure 4.1: Lifetime of variables in a DFG

4.3 Mutually Independent Operations

Associated with each operation o_i (and module M_i) is the *type* of the operation such as addition or multiplication, denoted by $type(o_i)$ ($type(M_i)$). Two operations scheduled in the same control step execute concurrently. Concurrent operations cannot be assigned to the same functional module. Two operations that execute in *different* control steps execute sequentially - one after the other. Sequential operations of different *types* cannot be assigned to the same module. Sequential operations of the same type can be assigned to the same module.

Definition 8 *Given a scheduled DFG $G = (V, E)$, a* **module assignment** *is defined as a partition $\Pi_M = \{M_1, M_2, ..., M_m\}$ of the set of operations V such that for any two operations o_i and o_j in M_k, $1 \le k \le m$, $S(o_i) \neq S(o_j)$ and $type(o_i) = type(o_j) = type(M_k)$.*

Note: In the rest of this dissertation, symbol M_k will be used to refer to the name of a module as well as the set of operations assigned to that module. The correct meaning of the symbol should be inferred from the context.

Numerous module assignments are possible based on the constraints in Definition 8. Depending on a chosen module assignment Π_M, we define the concept of mutually independent operations. The mutual independence is with regards to the hardware exclusivity of the modules that execute the operations.

Definition 9 *Given a scheduled DFG and a module assignment, a pair of operations o_i and o_j are* **mutually independent operations** *if they are assigned to distinct functional modules.*

Definition 10 *Given a scheduled DFG, $G = (V, E)$ and a module assignment, a set of operations $V^i_{I_{max}} \subseteq V$ is called a* **maximal independent operation set** *if it is a maximal set of mutually independent operations.*

A maximal independent operation set consists of one operation from each functional module. If there are m functional modules assigned, then $\mid V^i_{I_{max}} \mid = m$. A

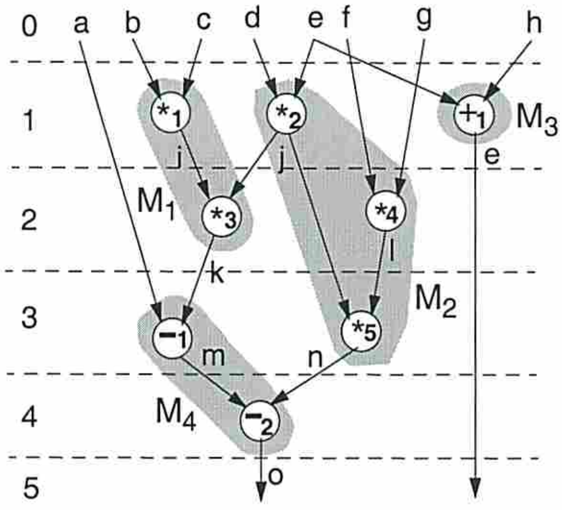


Figure 4.2: Mutually independent operations

module assignment of the scheduled DFG from Fig. 4.1(a) is shown in Fig. 4.2 where $M_1 = \{*_1, *_3\}$, $M_2 = \{*_2, *_4, *_5\}$, $M_3 = \{+_1\}$ and $M_4 = \{-_1, -_2\}$. For the given module assignment, concurrent operations $*_1$ and $*_2$ are mutually independent since they are assigned to different modules, namely, M_1 and M_2, respectively. Operations $*_3$ and $-_1$ are examples of sequential operations that are also mutually independent. $V^1_{I_{max}} = \{*_1, *_2, +_1, -_1\}$ is an example of a maximal independent operation set. It has exactly one operation from each functional module.

Lemma 1 *If the operations of a scheduled DFG $G = (V, E)$ are assigned to m functional modules, then $(\lceil\frac{|V|}{m}\rceil)^m$ is an upper bound on the number of maximal independent operation sets if each module is assumed to implement all operation types.*

Proof: Let $\mid V \mid = k$ and k_i be the number of operations assigned to module $M_i, 1 \leq i \leq m$. We have

$$k_1 + k_2 + ... + k_i + ... + k_m = k.$$

The number of maximal independent operation sets

$$I_m = k_1 \cdot k_2 \cdot \ldots \cdot k_i \cdot \ldots \cdot k_m.$$

To find the value of all k_i such that I_m is maximum we differentiate I_m w.r.t. each k_i and equate to 0. Differentiating I_m w.r.t. k_1,

$$\frac{dI_m}{dk_1} = 0$$

$$\Rightarrow \frac{d}{dk_1}(k_1 \cdot k_2 \cdot \ldots \cdot k_i \cdot \ldots \cdot k_m) = 0$$

Substituting $k_m = k - \sum_{i=1}^{m-1} k_i$

$$\Rightarrow \frac{d}{dk_1}(k_1 \cdot k_2 \cdot \ldots \cdot k_i \cdot \ldots \cdot (k - \sum_{i=1}^{m-1} k_i)) = 0$$

$$\Rightarrow (\prod_{i=2}^{m-1} k_i \cdot k) - (2 \cdot k_1 \cdot \prod_{i=2}^{m-1} k_i) - (\prod_{i=2}^{m-1} k_i \cdot \sum_{i=2}^{m-1} k_i) = 0$$

$$\Rightarrow k - 2 \cdot k_1 - \sum_{i=2}^{m-1} k_i = 0$$

$$\Rightarrow 2 \cdot k_1 = k - \sum_{i=2}^{m-1} k_i$$

$$\Rightarrow 2 \cdot k_1 = k_1 + k_m$$

$$\Rightarrow k_1 = k_m.$$

Similarly differentiating w.r.t. each k_i and substituting each variable $k_j, j \neq i$ we have the condition for I_m to be maximum which is

$$k_1 = k_2 = \ldots = k_i = \ldots = k_m.$$

This is possible only when $k_i = (k/m)$ for all k_i and the maximum value of I_m,

$$I_m^{max} = (\frac{k}{m}) \cdot (\frac{k}{m}) \cdot \ldots \ m \ times.$$

Since k_i is number of operations in a module, we are concerned with only integer values for each k_i. Each term in the above expression is upper-bounded by $\lceil \frac{k}{m} \rceil$. Hence

$$I_m^{max} = (\lceil \frac{k}{m} \rceil) \cdot (\lceil \frac{k}{m} \rceil) \cdot ... \ m \ times = (\lceil \frac{\mid V \mid}{m} \rceil)^m .$$

□

Corollary 1 *If the operations of a scheduled DFG $G = (V, E)$ are assigned to a total of m modules such that m_i modules are of type i $(1 \leq i \leq t)$, then an upper bound on the number of maximal independent operation sets is*

$$\prod_{i=1}^{t} (\lceil \frac{\mid V_i \mid}{m_i} \rceil)^{m_i},$$

where $V_i \subseteq V$ is the set of all operations in V of type i.

Proof: The number of maximal independent operation sets

$$I_m = k_1 \cdot k_2 \cdot ... \cdot k_i \cdot ... \cdot k_m,$$

where k_i is the number of operations assigned to module i. Considering the *type* of an operation and module,

$$I_m = (k_1^1 \cdot k_2^1 \cdot ... k_{m_1}^1) \cdot (k_1^2 \cdot k_2^2 \cdot ... k_{m_2}^2) \cdot ... (k_1^t \cdot k_2^t \cdot ... k_{m_t}^t),$$

where k_k^i is the number of operations of *type* i assigned to the kth module of *type* i. Since an operation of *type* i cannot be assigned to a module of *type* j $(i \neq j)$,

$$I_m = I_{m_1} \cdot I_{m_2} \cdot ... \cdot I_{m_t},$$

where I_{m_i} is the number of maximal independent operation sets considering operations in V_i assigned to the m_i modules of *type* i. Therefore, the maximum value of I_m

$$I_m^{max} = I_{m_1}^{max} \cdot I_{m_2}^{max} \cdot ... \cdot I_{m_t}^{max}.$$

Table 4.1: Typical values of I_m^{max} for benchmarks

DFG	# Operations			# Modules			I_m^{max}
	+	*	-	+	*	-	
FIR_filter	4	4	-	1	4	-	4
				2	4	-	4
Diffeqn	2	6	2	1	2	1	144
				2	3	1	16
AR_filter	12	16	-	4	4	-	20736
				6	8	-	16384
EW_filter	26	8	-	5	3	-	209952
				13	4	-	131072

From Lemma 1

$$I_m^{max} = (\lceil\frac{\mid V_1 \mid}{m_1}\rceil)^{m_1} \cdot (\lceil\frac{\mid V_2 \mid}{m_2}\rceil)^{m_2} \cdot \ldots \cdot (\lceil\frac{\mid V_t \mid}{m_t}\rceil)^{m_t}.$$

□

Table 4.1 shows typical values of I_m^{max} for some benchmarks. It can be seen that even though the number of maximal independent operations sets *grows* exponentially with respect to the number of operations, the *actual* number is not significantly large for typical cases.

4.4 Lower Bounds on BIST Resources

Definition 11 *A BIST register that can* generate *pseudo-random test patterns is called a $\mathcal{G}$*-**resource** *and a BIST register that can* compress *test responses is called a $\mathcal{C}$*-**resource**.

As described in Chapter 3, the TPG type of BIST register is a $\mathcal{G}$-resource and the SA type of BIST register is a $\mathcal{C}$-resource. BILBO and CBILBO types are $\mathcal{G}$-resources as well as $\mathcal{C}$-resources. The input and output variables of operations in a maximal independent operation set are significant in terms of BIST resources of the data path. The testing of a functional module using pseudo-random patterns corresponds to the execution of an operation assigned to that functional module in the test mode. The registers to which the input and output variables of an operation are assigned are candidates for BIST resources for the module to which the operation is assigned. The set of input variables and the set of output variables of all operations assigned to a module define all BIST resource candidates for that module.

Definition 12 *The* **input variable set** *$IVar(P)$ of a set of operations P is the union of input variables of all operations in P. The* **output variable set** *$OVar(P)$ of set of operations P is the union of output variables of all operations in P.*

The notions of input and output variable sets can be used for a set of operations such as a maximal independent operation set $V^i_{I_{max}}$. It can also be used for a module M_i where M_i is a set of operations assigned to the module. In Fig. 4.2, $V^1_{I_{max}} = \{*_1, *_2, +_1, -_1\}$ is a maximal independent operation set and $OVar(V^1_{I_{max}}) = \{i, j, e, m\}$. Similarly, $OVar(M_2) = \{j, l, n\}$.

4.4.1 $\mathcal{C}$-resources

A register with $\mathcal{C}$ functionality is required to compress the test responses of each module. In this section we derive the minimum number of registers that can be synthesized to provide $\mathcal{C}$ functionality for all modules in a data path.

Example 1 *Consider two different module assignments of the scheduled DFG shown in Fig. 4.3(a).*

- *Assignment I:(Fig. 4.3(b)) Operations $+_1$ and $+_3$ are assigned to one module and operation $+_2$ is assigned to a second module. $M_1 = \{+_1, +_3\}$ and $M_2 = \{+_2\}$.*
- *Assignment II:(Fig. 4.3(c)) Operations $+_1$ and $+_2$ are assigned to one module and operation $+_3$ is assigned to a second module. $M_1 = \{+_1, +_2\}$ and $M_2 = \{+_3\}$.*

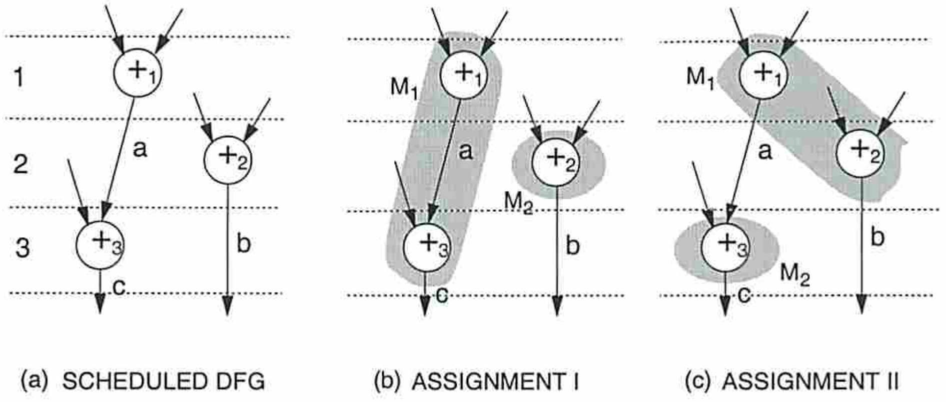


Figure 4.3: Lower bound on $\mathcal{C}$-resources ($LB_{\#\mathcal{C}}$)

Consider all possible register and interconnect assignments for the above two module assignments. Any register which is assigned at least one variable from the output variable set of a module can be used as a $\mathcal{C}$-resource for that module. For Assignment II, variables a and c can be assigned to the same register since their lifetimes do not overlap and this register can be used as a $\mathcal{C}$ resource to test both M_1 and M_2 since $a \in OVar(M_1)$ and $c \in OVar(M_2)$. Hence the lower bound on the number of $\mathcal{C}$-resources in this case is $LB_{\#\mathcal{C}} = 1$. For Assignment I it is not possible to find such a register assignment. In this case $OVar(M_1) = \{a, c\}$ and $OVar(M_2) = \{b\}$. Since the lifetime of b overlaps with the lifetimes of both a and c it cannot be assigned the same register as a or c. Hence for any register assignment the minimum number of $\mathcal{C}$-resources required to test M_1 and M_2 is $LB_{\#\mathcal{C}} = 2$.

Theorem 1 *For a given scheduled DFG, $G = (V, E)$ and a module assignment, a lower bound on the number of $\mathcal{C}$-resources required to test all the modules in a synthesized data path is given by*

$$LB_{\#\mathcal{C}} = \min_i \; SC(OVar(V^i_{I_{max}})),$$

where the minimum is over all maximal independent operation sets.

Proof: Let $\min_i SC(OVar(V^i_{I_{max}})) = n$ and the total number of modules assigned be m. Assume that there exists a way of assigning variables to registers such that the $\mathcal{C}$ functionality for m modules can be provided by n' registers and $n' < n$.

For a register to be a $\mathcal{C}$-resource for a module M_i, at least one variable from output variable set $OVar(M_i)$ must be assigned to that register. Since m modules can be tested using n' $\mathcal{C}$-resources, there exist m variables each belonging to a distinct output variable set $OVar(M_i)$ $(i = 1, 2, ..., m)$ such that they can be assigned to n' registers. Let the set of operations that have a variable from these m variables as an output variable be denoted by V_c. Each of the operations in V_c is mapped to a distinct module and $\mid V_c \mid = m$. Hence V_c is a maximal independent operation set from Definition 10. Since the output variables of operations in V_c can be assigned to n' registers, the storage concurrency $SC(OVar(V_c)) \leq n'$. Since V_c is a maximal independent operation set of the given module assignment, this contradicts the fact that

$$\min_i SC(OVar(V^i_{I_{max}})) = n.$$

Hence the lower bound on the number of $\mathcal{C}$-resources required to test the m modules is n. □

For Assignment I in Example 1, the maximal independent operation sets are $V^1_{I_{max}} = \{+_1, +_2\}$ and $V^2_{I_{max}} = \{+_3, +_2\}$. Hence the lower bound on the number of $\mathcal{C}$-resources, $LB_{\#\mathcal{C}} = \min \{SC(OVar(V^1_{I_{max}})), SC(OVar(V^2_{I_{max}}))\} = \min \{2, 2\} = 2$. For Assignment II, the maximal independent operation sets are $V^1_{I_{max}} = \{+_1, +_3\}$ and $V^2_{I_{max}} = \{+_2, +_3\}$. Hence $LB_{\#\mathcal{C}} = \min \{SC(OVar(V^1_{I_{max}})), SC(OVar(V^2_{I_{max}}))\}$ $= \min\{2, 1\} = 1$.

4.4.2 $\mathcal{G}$-resources

The lower bound computation for $\mathcal{G}$-resources is similar to that for $\mathcal{C}$-resources with the exception that we have to consider multiple input ports of modules as opposed to a single output port in the previous case. Almost all operations in a DFG are binary operations, hence we shall restrict the discussion to two input ports denoted by L and R for left and right, respectively. Given a scheduled DFG $G = (V, E)$ and a module assignment, we create a new set of operations $\widehat{V}$. $\widehat{V}$ has two instances for each operation $o_i \in V$ - a *left* instance denoted by o_i^L and a *right* instance denoted by o_i^R. Each instance has only one input variable feeding it. The left input variable of o_i is the only input variable of o_i^L and the right input variable of o_i is the only input variable of o_i^R. Note that operation instances o_i^L, $o_i^R \in \widehat{V}$ are *not* mutually independent since they are instances of the same operation $o_i \in V$ and are associated with the same functional module.

Definition 13 *A pair of maximal independent operation sets $\widehat{V}^i_{I_{max}}$ and $\widehat{V}^j_{I_{max}}$ form an* **input cover** *if for every module M_k, $IVar(\widehat{V}^i_{I_{max}} \cup \widehat{V}^j_{I_{max}})$ contains two distinct variables from $IVar(M_k)$.*

$IVar(\widehat{V}^i_{I_{max}})$ is a set of variables with one input variable for each operation. Since the set of operations is a maximal independent set, the variables cover at least one input port of each module. For testing all modules, a distinct $\mathcal{G}$-resource is required for the input ports of each module. Another maximal independent operation set $\widehat{V}^j_{I_{max}}$ can be chosen such that $IVar(\widehat{V}^j_{I_{max}})$ contains input variables that cover the other input port of modules not covered by $IVar(\widehat{V}^i_{I_{max}})$. For that to be guaranteed $IVar(\widehat{V}^i_{I_{max}} \cup \widehat{V}^j_{I_{max}})$ should be such that it contains two distinct input variables for each module. The registers formed by *any* valid assignment of variables in $IVar(\widehat{V}^i_{I_{max}} \cup \widehat{V}^j_{I_{max}})$ can be connected to the modules such that each input port of each module has a register connected to it. However, it is possible that the assignment is such that one register gets connected to both the input ports of a module. Since there are distinct input variables for the input ports of each module, an assignment that ensures a distinct register for each input port can be found. These registers are then candidates for $\mathcal{G}$-resources in valid BIST embeddings of modules.

Lemma 2 *Consider assignments of variables in $IVar(\widehat{V}^i_{I_{max}} \cup \widehat{V}^j_{I_{max}})$ to registers with the constraint that each input port of a module has a distinct input register. The minimum number of registers required for an assignment with this constraint is the same as the minimum number of registers required for any register assignment of variables in $IVar(\widehat{V}^i_{I_{max}} \cup \widehat{V}^j_{I_{max}})$.*

Proof: Given a schedule with latency L, any set of variables Q can be divided into sets Q_1, Q_2,..., Q_L such that Q_i is the set of *all* variables in Q that are alive during control step i. Considered the sets Q_1, Q_2,..., Q_L sorted in decreasing order of their cardinality. Without loss of generality we can assume that the order is the same as indicated by the subscript, namely, Q_1, Q_2,..., Q_L. Hence $\mid Q_1 \mid \geq \mid Q_2 \mid \geq ... \mid Q_L \mid$. Let $\mid Q_1 \mid = n$. From Definition 7, $SC(Q) = SC(Q_1) = n$. One way of constructing a minimum register assignment of variables in Q is to assign the variables of $Q_i, 1 \leq i \leq L$, to distinct registers in increasing order of i. There are two properties of sets Q_1, Q_2,..., Q_L that ensure a minimum register assignment. 1) All variables belonging to Q_i are alive at the same time, hence they have to be assigned to *distinct* registers; and 2) any *distinct* variables a, b, such that $a \in Q_i$ and $b \in Q_j$, $i \neq j$, are not alive at the same time and can be assigned to the same register.

The minimum register assignment can be constructed as follows. The n variables of Q_1 are first assigned to n registers. The variables of any set Q_i require $\mid Q_i \mid$ *distinct* registers. Since $\mid Q_i \mid \leq n$ the variables can be assigned to $\mid Q_i \mid$ distinct registers from the n available registers such that there is no conflict of variables in a register.

To prove the lemma we need to show that such a minimum register assignment can be achieved even with the constraint that each input port of a module has distinct input register. Let $Q = IVar(\widehat{V}^i_{I_{max}} \cup \widehat{V}^j_{I_{max}})$. Since Q forms an input cover it has two distinct input variables for every module. If any of the sets Q_i contains two input variables of the same module they get assigned to two different registers, satisfying the constraint. Now consider two input variables of the same module, say a and b, belonging to Q_i and Q_j ($i \neq j$), respectively. If variables in Q_i are assigned first, a is assigned to one of the n registers, say R_p. When variables in Q_j are assigned, b can be assigned to a register other than R_p, say R_q, with which it does not conflict from the n registers. The rest of the variables in Q_j can now be assigned to any of the $(n-1)$ registers other than R_q without violating the constraint for other modules, since $\mid Q_j \mid \leq n$. □

Theorem 2 *For a given scheduled DFG, $G = (V, E)$ and a module assignment, a lower bound on the number of $\mathcal{G}$-resources required to test all the modules in the data path is given by*

$$LB_{\#\mathcal{G}} = \min_{i,j} SC(IVar(\widehat{V}^i_{I_{max}} \cup \widehat{V}^j_{I_{max}})),$$

where the minimum is over all i, j such that $\widehat{V}^i_{I_{max}}$ and $\widehat{V}^j_{I_{max}}$ form an input cover.

Proof: Let $\min_{i,j} SC(IVar(\widehat{V}^i_{I_{max}} \cup \widehat{V}^j_{I_{max}})) = n$. Since $\widehat{V}^i_{I_{max}}$ and $\widehat{V}^j_{I_{max}}$ form an input cover, $IVar(\widehat{V}^i_{I_{max}} \cup \widehat{V}^j_{I_{max}})$ has two distinct input variables for each module. Using reasoning similar to the proof of Theorem 1, we can show that n is the minimum number of registers that can be synthesized such that they cover both input ports of all modules. From Lemma 2, even with the constraint on the assignment that each input port of a module have a distinct register, the minimum number of registers required is n. □

4.4.3 CBILBO Resources

Registers that have the test functionality of generating test responses and compressing test responses simultaneously are candidates for CBILBO. The CBILBO is a very expensive BIST register and hence the goal of BIST techniques is to minimize the usage of these expensive registers. In this section we derive a lower bound on the number of such BIST registers that are necessary to test a data path synthesized from a given schedule and a module assignment. It is essential to modify a register to CBILBO in a data path only if it is the only register with $\mathcal{G}$ functionality for an input port and the only register for $\mathcal{C}$ functionality for the output port of a module.

Theorem 3 *Given a scheduled DFG $G = (V, E)$ and a module assignment, a CBILBO is essential to test module M_i for all register assignments iff*

1. *M_i has only one operation assigned to it, and*

2. *one variable is an input as well as an output variable of M_i.*

Proof: If ($\Rightarrow$): Let the common input and output variable of M_i be assigned to register R_k. Since M_i has only one operation assigned to it, it has only one output variable and R_k is the only register to provide the $\mathcal{C}$ function. Also M_i has one input variable for each input port and R_k is the only register to provide the $\mathcal{G}$ function for one input port. Hence R_k is an essential CBILBO.

Only if ($\Leftarrow$): Let module M_i require an essential CBILBO register in any register assignment. The essential CBILBO register is the only one that can provide $\mathcal{C}$ function to M_i and the only one that can provide $\mathcal{G}$ function to an input port of M_i. This implies that in all possible register assignments the input variables corresponding to one input port and the output variables of M_i are assigned to only one register. This is possible only if M_i has one operation assigned to it with a common input and output variable. □

Definition 14 *An operation o_i is an* **essential independent operation** *if it exists in all maximal independent operation sets of a given module assignment. The set of all essential independent operations of a DFG for a given module assignment will be denoted by V_{ess}.*

From the definition of a maximal independent operation set, the module to which an essential independent operation is assigned has only *one* operation assigned to it. In Fig. 4.2, operation $+_1$ is an essential independent operation. If a module assignment has the property stated in Theorem 3, any register assignment and interconnect assignment that follows cannot avoid a CBILBO. If the register and interconnect assignment is performed without regard for functional area but with the sole objective of minimizing CBILBOs, the number of CBILBOs would depend on the number of essential independent operations with the same input and output variable.

Theorem 4 *For a given scheduled DFG $G = (V, E)$ and module assignment, a lower bound on the number of CBILBOs required to test all the modules in the data path is given by*

$$LB_{\#CBILBO} = SC\left(\bigcup_{\forall o_i \in V_{ess}} (IVar(o_i) \cap OVar(o_i))\right).$$

Proof: Given a schedule and a module assignment, the condition for essential CBILBO is given by Theorem 3. Only a register to which the common input and output variable of an essential independent operations is assigned forms a CBILBO in any data path. The minimum number of registers to which such variables can be assigned gives the lower bound on the number of CBILBOs. Hence $SC(\bigcup_{\forall o_i \in V_{ess}}(IVar(o_i) \cap OVar(o_i)))$ is the lower bound on CBILBOs. □

Example 2 *Consider the scheduled DFG shown in Fig. 4.4(a). Note that the output and one input variable of operations $+_3$ and $+_2$ is the same, namely, a and d. This occurs in the case of iterative computations in which the loop is broken for scheduling purposes. Consider the following three module assignments.*

- *Assignment I: (Fig. 4.4(b)) Operations $+_1$, $+_2$ and $+_3$ are all assigned to a different module. i.e. $M_1 = \{+_1\}$, $M_2 = \{+_2\}$ and $M_3 = \{+_3\}$. Since $IVar(+_2) \cap OVar(+_2) = \{d\} \neq \phi$ and $IVar(+_3) \cap OVar(+_3) = \{a\} \neq \phi$, registers to which variables a and d will be assigned will be self-adjacent registers. There is only one maximal independent operation set $V^1_{I_{max}} = \{+_1,+_2,+_3\}$ and each of the operation is an essential independent operation. However the intersection of the input and output variable sets is non-empty only for two of the three operations, namely $+_2$ and $+_3$. The lower bound on the number of CBILBOs according to Theorem 4 is $SC(\{a,d\}) = 2$ since both variables are alive at the same time.*

- *Assignment II: (Fig. 4.4(c)) In this assignment the essential independent operation $+_3$ in Assignment I is assigned to the same module as operation $+_1$. Now we have $M_1 = \{+_1,+_3\}$ and $M_2 = \{+_2\}$. This module assignment has only one essential independent operation, namely, operation $+_2$. Since $IVar(+_2) \cap OVar(+_2) = \{d\}$, therefore according to Theorem 4 the lower bound on the number of CBILBOs is $SC(\{d\}) = 1$. Note that even though $IVar(M_1) \cap OVar(M_1) = \{a, f\}$ it does not require a CBILBO since two operations are assigned to it and a register assignment can be found that does not necessitate a CBILBO.*

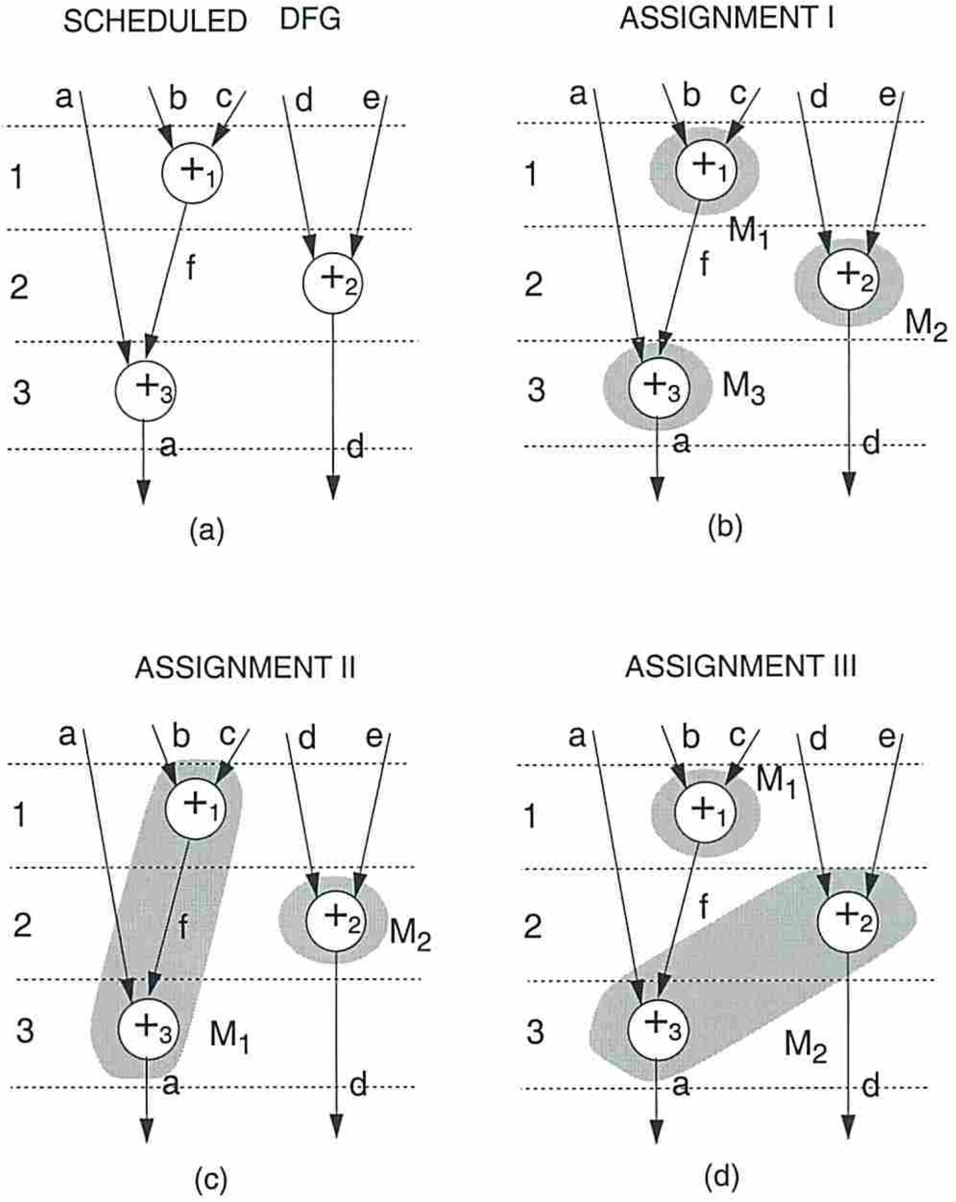


Figure 4.4: Lower bound on CBILBOs ($LB_{\#CBILBO}$)

- *Assignment III:* *(Fig. 4.4(d)) In this assignment we have $M_1 = \{+_1\}$ and $M_2 = \{+_2,+_3\}$. This module assignment also has only one essential independent operation like in Assignment II, namely, operation $+_1$. However, $IVar(+_1) \cap OVar(+_1) = \phi$. Hence the lower bound on the number of CBILBOs is 0. Note that each of the operations assigned to M_2 have a common input and output variable. However, since there are two operations assigned to M_2 a register assignment can be found such that M_2 does not require a CBILBO.*

4.5 Tightness of Bounds

The lower bounds $LB_{\#\mathcal{C}}$, $LB_{\#\mathcal{G}}$ and $LB_{\#CBILBO}$ have been derived assuming that a schedule S and a module assignment Π_M was known. *Any* register assignment, given S and Π_M will result in a data path that requires $LB_{\#\mathcal{C}}$ or more $\mathcal{C}$-resources, $LB_{\#\mathcal{G}}$ or more $\mathcal{G}$-resources and $LB_{\#CBILBO}$ or more CBILBOs. Next we demonstrate the tightness of the bounds.

Theorem 5 *Given a schedule S and a module assignment Π_M, the lower bound $LB_{\#\mathcal{C}}$ is the tightest possible bound on the number of $\mathcal{C}$-resources.*

Proof: To prove tightness we need to show that there exists at least one valid register assignment that results in a data path that can be tested using *exactly* $LB_{\#\mathcal{C}}$ $\mathcal{C}$-resources. Let $LB_{\#\mathcal{C}} = n$. From Theorem 1, there exists a maximal independent operation set $V^i_{I_{max}}$ such that $SC(OVar(V^i_{I_{max}})) = n$. Let Q be the set of variables $OVar(V^i_{I_{max}})$. Since the maximum number of variables in Q alive at the same time is n, they can be assigned to a minimum number of n registers. Let $\Pi^Q_R = \{R_1, R_2, ..., R_n\}$ correspond to one such partial register assignment. The remaining variables of the DFG from $E - Q$ can be assigned *independent* of Π^Q_R to a new set of registers, namely, Π^{E-Q}_R. The register assignment $\Pi_R = \Pi^Q_R \cup \Pi^{E-Q}_R$ is a valid register assignment for the variable lifetimes as determined by schedule S. In the data path synthesized using S, Π_M and Π_R, the n registers belonging to Π^Q_R have at least one output variable from each module in the data path and hence these n registers are sufficient as $\mathcal{C}$-resources for all modules in this data path. □

Theorem 6 *Given a schedule S and a module assignment Π_M, the lower bound $LB_{\#\mathcal{G}}$ is the tightest possible bound on the number of $\mathcal{G}$-resources.*

Proof: The proof is similar to that of Theorem 5. Here $Q = IVar(\widehat{V}^i_{I_{max}} \cup \widehat{V}^j_{I_{max}})$ where i and j are such that $SC(IVar(\widehat{V}^i_{I_{max}} \cup \widehat{V}^j_{I_{max}})) = LB_{\mathcal{G}}$ and, $\widehat{V}^i_{I_{max}}$ and $\widehat{V}^j_{I_{max}}$ form an input cover. □

Theorem 7 *Given a schedule S and a module assignment Π_M, the lower bound $LB_{\#CBILBO}$ is the tightest possible bound on the number of CBILBOs.*

Proof: The proof is similar to that of Theorem 5 with

$$Q = \bigcup_{\forall o_i \in V_{ess}} (IVar(o_i) \cap OVar(o_i)).$$

Here the n registers corresponding to Π^Q_R will be essential CBILBOs according to Theorem 3. The variables in $E - Q$ can always be assigned to registers such that none of them are essential CBILBOs. One such simple register assignment, Π^{E-Q}_R, would be where each variable in $E - Q$ is assigned to a *distinct* register. The register assignment $\Pi_R = \Pi^Q_R \cup \Pi^{E-Q}_R$ is a valid register assignment for the variable lifetimes as determined by schedule S. In the data path synthesized using S, Π_M and Π_R, the n registers belonging to Π^Q_R are essential CBILBOs and the rest of the registers are not. □

In demonstrating the tightness of bounds we have imposed certain constraints on the register assignment. The constraints necessary for achieving a particular lower bound might possibly disallow the constraints necessary for achieving another lower bound. Hence all three lower bounds are *individually* tight and achievable but might not be achievable simultaneously in the same register assignment. Next we describe how each of the lower bounds can be determined subject to a register assignment constraint in the form of a *partial* register assignment.

Assume P is a subset of variables from the complete set of variables E that is already assigned to p registers. Let $\Pi^P_R = \{R_1, R_2, ..., R_p\}$ denote the partial register assignment. Each lower bound, $LB_{\#\mathcal{C}}, LB_{\#\mathcal{G}}$ and $LB_{\#CBILBO}$, depends on computing the storage concurrency of a specific set of variables. Let Q denote the set of such variables for which the storage concurrency needs to be determined. The concept of storage concurrency was introduced in Section 4.2 to determine the minimum number of registers required to store a set of variables. From Definition 7 it can be seen that the storage concurrency of a set of variables Q is the size of the *largest* maximal subset of Q such that all variables of the subset are pairwise alive at the same time. With a partial register assignment, such as Π_R^P, storage (registers) is already allocated for some of the variables. We introduce the concept of *pairwise concurrence* of variables and registers to extend the notion of storage concurrency to partial register assignments.

Definition 15 1. *Variables v and w are* **pairwise concurrent** *if they are alive at the same time.*

2. *A variable v and a register R_i are* **pairwise concurrent** *if there exists at least one variable $w \in R_i$ such that v and w are alive at the same time.*

3. *Registers R_i and R_j are always* **pairwise concurrent**.

Pairwise concurrency of variables is the same notion as two variables being alive at the same time. Pairwise concurrency of a variable v and a register R_i indicates whether v can be assigned to R_i. If v cannot be assigned to R_i then the pair is said to be concurrent since they need to be stored separately. Two registers are always pairwise concurrent because they are already allocated as two separate registers. Now given a set of variables Q and a partial register assignment Π_R^P, we can determine its storage concurrency as follows. We first construct a set Q' of variables *and* registers such that if $v \in Q$ is an unassigned variable, then $v \in Q'$ and if $v \in Q$ is assigned to a register R_i, then $R_i \in Q'$. The storage concurrency of Q' is the size of the largest maximal subset of Q' such that all elements of the subset are pairwise concurrent according to Definition 15. Note that for $\Pi_R^P = \phi$, i.e. without assuming any register assignment, the computation of storage concurrency is the same as in Section 4.2. Using the above described method of determining storage concurrency, the lower bounds on $\mathcal{C}$-resources, $\mathcal{G}$-resources and CBILBOs can thus be determined given S, Π_M and Π_R^P.

4.6 Efficient Computation of Bounds

To use lower bounds as estimators or cost functions in synthesis algorithms, an efficient way of computing them is required. The lower bound computation for a DFG is essentially determining the storage concurrency of a few intelligently chosen subsets of its variables. A simple way of computing the lower bounds is to enumerate all the maximal independent operation sets and find the minimum storage concurrencies of their input variable sets and the output variable sets.

Theorem 8 *The worst case complexity of computing $LB_{\#\mathcal{C}}$ and $LB_{\#\mathcal{G}}$ of a scheduled DFG $G = (V, E)$ with latency L and requiring m modules is $O(L \cdot (\frac{|V|}{m})^m)$.*

Proof: The storage concurrency of a set of variables Q can be determined by finding the number of variables in Q that are alive at every control step boundary of the scheduled DFG. The control step boundary that has the maximum number of variables alive gives the storage concurrency of Q. Hence the storage concurrency of Q, $SC(Q)$ can be determined in $O(L\cdot \mid Q \mid)$ time. For determining input (or output) storage concurrencies of a maximal independent operation set, $\mid Q \mid = m$.

The worst case complexity of computing $LB_\mathcal{C}$ ($LB_\mathcal{G}$) is the complexity of computing the output (input) storage concurrencies of all maximal independent operation sets for a given module assignment. From Lemma 1, an upper bound on the number of maximal independent operation sets is $(\lceil \frac{|V|}{m} \rceil)^m$. Therefore, the worst case complexity is $O(L \cdot m \cdot (\frac{|V|}{m})^m) = O(L \cdot (\frac{|V|}{m})^m)$. □

The complexity is exponential with respect to $\mid V \mid$, the number of operations in the DFG. The algorithm is efficient in spite of the exponential complexity because the number of modules m is typically very small. Also, typically, as $\mid V \mid$ increases, so does m and $\frac{|V|}{m}$ does not increase significantly. Furthermore, properties of module assignments and maximal independent operation sets can be used to reduce the size of the exponential space. For example, consider two maximal independent operation sets $V^1_{I_{max}}$ and $V^2_{I_{max}}$. If all the operations in $V^1_{I_{max}}$ are scheduled in the same control step, then it can be shown that $SC(OVar(V^2_{I_{max}})) \leq SC(OVar(V^1_{I_{max}}))$. Hence maximal independent operation sets such as $V^1_{I_{max}}$ can be ignored during the lower bound computation.

Definition 16 *Given a scheduled DFG $G = (V, E)$ the* **width** *of a set of operations $V_s \subseteq V$, denoted by $width(V_s)$, is the maximum number of operations in V_s scheduled in the same control step.*

Lemma 3 *For any maximal independent operation set $V^i_{I_{max}}$, $SC(OVar(V^i_{I_{max}})) \geq width(V^i_{I_{max}})$. For any maximal independent operation set $\widehat{V}^i_{I_{max}}$, $SC(IVar(\widehat{V}^i_{I_{max}})) \geq width(\widehat{V}^i_{I_{max}})$ if input variables of concurrent operations in $\widehat{V}^i_{I_{max}}$ are not multiple-edged.*

Proof: Consider a maximal independent operation set $V^i_{I_{max}}$ of width w. According to Definition 16 this implies that w operations from $V^i_{I_{max}}$ are scheduled in the same control step. Therefore they define w distinct output variables in the same control step. The storage concurrency of these w output variables is w. Hence $SC(OVar(V^i_{I_{max}}))$ is at least w.

In the case of input variables, a single variable could be an input variable for more than one operation. Such a variable is called a multiple-edged variable (e.g. variable e in Fig. 4.1(a)). If the w concurrent operations have only single-edged input variables, an argument similar to the one made for output variables can be made for input variables and $SC(IVar(V^i_{I_{max}})) \geq w$. □

Lemma 3 can be used to prune the search space during lower bound computation of BIST resources. From Theorem 1 the lower bound $LB_{\#\mathcal{C}}$ is the *minimum* output storage concurrency of all maximal independent operation sets. At every point during the computation of the lower bound, the minimum output storage concurrency of the maximal independent operation sets considered up to that point is known. If the width of the next maximal independent operation set is greater than or equal to the minimum output storage concurrency at that point, that maximal independent operation set is ignored, thus pruning the search space. A similar argument holds for computation of $LB_{\#\mathcal{G}}$.

Theorem 9 *The worst case complexity of computing $LB_{\#CBILBO}$ of a scheduled DFG $G = (V, E)$ with latency L and requiring m modules is $O(L \cdot m)$.*

Proof: Finding the set of essential independent operations V_{ess} can be done in $O(m)$ time. There are at most m essential independent operations with common input and output variables. The complexity of computing the lower bound is the same as determining the storage concurrency of these variables which can be done in $O(L \cdot m)$. □

4.7 Use of Lower Bound Estimation in Synthesis

The ability to predict area-performance characteristics of designs without actually synthesizing them is vital to produce quality designs in a reasonable time. Using a high-level synthesis system, a designer often needs to repeat the synthesis process several times while searching for a satisfactory design. Comparison of synthesis results using different module sets, module assignments and schedules are made to locate the desired design. Computation of lower bounds on *BIST resources* provides a synthesis system with a quick way of evaluating the testability overhead of the design. More specifically, the proposed lower bounds can be used for the following applications.

1. To select schedules from a set of schedules with the same latency and resource requirement. For a given behavior different schedules are possible such that they have a desired latency and satisfy a constraint on the number of modules. However the minimum number of BIST resources required to test the designs synthesized using these schedules could vary significantly. The lower bounds on BIST resources can be used to select an appropriate schedule.

2. Given a schedule, to find a module assignment that requires few BIST resources. For a given schedule, different module assignments have different lower bounds on BIST resources. The lower bound estimation technique can be used to compare different module assignments for the same schedule. Alternatively, the lower bound estimation can be done incrementally *during* module

assignment using information from partial module assignments. For example, if at some stage in the module assignment process there is a choice between many assignments, then an assignment that results in a maximal independent operation set with a lower output storage concurrency should be chosen.

3. To trade-off latency for area. Sometimes latency is increased if a reduction in the number of modules is desired. In some cases, increasing the latency by a few clock steps does not reduce the module requirement. However it does increase the number of different possible schedules, and thus a schedule that has a small lower bound on the number of BIST resources may be identified. Such a schedule could lead to savings in area.

4. To prune the search space and direct the search during register and interconnect assignment towards low testability overhead designs. The information used in the estimation of the lower bounds (e.g. maximal independent operation sets and storage concurrency) can be used to prune the search space and select assignments that will achieve those bounds or will be close to the bounds.

5. To provide an independent measure of comparing the performance of different high-level synthesis systems and algorithms that perform testability optimization. High-level synthesis systems that have testability overhead as an optimization criterion use a variety of heuristics and cost functions in their algorithms. The bounds provide a common base to evaluate the quality of the various synthesis algorithms.

4.8 Experimental Results

To demonstrate the use of the proposed lower bound computation in evaluating the testability qualities of schedules and module assignments, we applied it to some well-known high level synthesis benchmarks: 1) the 2nd order differential equation - *Diffeqn* [61], 2) the Tseng data flow graph - *Tseng* [62], 3) the auto regression filter element - *AR_Filter* [63], and 4) the 5th order elliptic wave filter - *EW_filter* [64].

Table 4.2: Lower bounds for minimum latency schedules of *Diffeqn*

Schedule	Type of schedule	Latency (L)	Number of modules (m)	$LB_{\#\mathcal{C}}$	$LB_{\#\mathcal{G}}$	$LB_{\#CBILBO}$
S_1	ALAP	4	5	3	5	1
S_2	ASAP	4	5	2	4	0
S_3	Intermediate	4	5	3	4	0

Table 4.3: Lower bounds for *Diffeqn*

Schedule	Latency (L)	Number of modules (m)	$LB_{\#\mathcal{C}}$	$LB_{\#\mathcal{G}}$	$LB_{\#CBILBO}$
S_1	6	4	2	4	1
S_2	6	4	2	4	0
S_3	6	4	1	4	0

Table 4.4: Lower bounds for *AR_Filter*

Schedule	Latency (L)	Module assignment	Number of modules (m)	$LB_{\#\mathcal{C}}$	$LB_{\#\mathcal{G}}$	$LB_{\#CBILBO}$
S_1 ALAP	8	M_1	12	5	16	0
		M_2	12	4	12	0
		M_3	12	2	8	0
S_2 ASAP	8	M_1	12	5	12	0
		M_4	12	3	8	0
		M_3	12	2	8	0

Table 4.5: Lower bounds for *EW_filter* ($L = 19$)

Module assignment	Number of modules (m)	$LB_{\#\mathcal{C}}$	$LB_{\#\mathcal{G}}$	$LB_{\#CBILBO}$
M_1	8	3	6	2
M_2	8	2	4	1
M_3	8	1	3	0

Table 4.2 depicts bounds for three different schedules of *Diffeqn*. The minimum latency for this benchmark is 4. As-late-as-possible (ALAP) scheduling and as-soon-as-possible (ASAP) scheduling are two popular scheduling techniques for achieving minimum latency schedules. Both ASAP and ALAP schedules of the *Diffeqn* require 5 modules. In Table 4.2 schedule S_1 is the ALAP schedule and schedule S_2 is the ASAP schedule. It can be seen that the lower bound on $\mathcal{C}$-resources, $\mathcal{G}$-resources and CBILBOs required if schedule S_2 is used is lower than the lower bounds of schedule S_1. The bounds for an intermediate schedule, S_3, with the same latency are also shown. Three more schedules each with a latency of 6 and using 4 modules are shown in Table 4.3. These results demonstrate that schedules that are equally attractive from the functional resources and latency point of view can differ greatly in the minimum BIST resource requirement.

Table 4.4 shows the bounds for different module assignments of the ASAP and ALAP schedules for the *AR Filter* benchmark. The latency of both schedules is 8 and the minimum number of modules for this latency is 12. All module assignments in Table 4.4 use 12 modules. Table 4.5 shows the bounds for different module assignments for a schedule of the *EW_Filter* benchmark. The latency of the schedule used is 19. These results show that a significant variation in test resources exist for different module assignments of the same schedule as well as different schedules of the same latency.

The *Tseng* benchmark does not have any variable that is an input as well as an output variable of the same operation. Hence according to Theorem 4, the lower bound on CBILBOs is zero. We used the *Tseng* benchmark as a case study to investigate the lower bounds on $\mathcal{C}$-resources and $\mathcal{G}$-resources for *all possible* schedules and *all possible* module assignments for this benchmark. For this particular case it was assumed that a module could perform any type of operation. Table 4.7 shows the lower bounds on $\mathcal{C}$-resources and Table 4.8 shows the lower bounds on $\mathcal{G}$-resources. Each entry in the tables corresponds to the minimum lower bound among all possible schedules and module assignments for that particular latency and number of modules. For example, among all module assignments using 6 modules that were possible for different schedules of latency 5, the minimum lower bound on the number of $\mathcal{C}$-resources was 3. A '$-$' entry indicates that no schedule and

Table 4.6: Lower bounds for *FIR filter*

Schedule	Latency (L)	Number of modules (m)	$LB_{\#\mathcal{C}}$	$LB_{\#\mathcal{G}}$	$LB_{\#CBILBO}$
S_1	5	8	4	10	0
S_2	5	4	1	4	0
S_3	5	4	1	4	0

Table 4.7: Variation in $\mathcal{C}$-resource lower bounds for *Tseng*

	Number of modules (m)							
Latency (L)	1	2	3	4	5	6	7	8
4	–	–	1	2	2	3	3	4
5	–	1	1	1	2	3	3	4
6	–	1	1	1	2	2	2	4
7	–	1	1	1	2	2	2	4
8	1	1	1	1	2	2	2	4

Table 4.8: Variation in $\mathcal{G}$-resource lower bounds for *Tseng*

	Number of modules (m)							
Latency (L)	1	2	3	4	5	6	7	8
4	–	–	2	3	3	4	5	5
5	–	2	2	3	3	4	5	5
6	–	2	2	3	3	4	4	5
7	–	2	2	3	3	4	4	5
8	2	2	2	3	3	4	4	5

module assignment solution is possible for that (L, m) value of the DFG. It can be observed that the bounds increase as the number of modules increases. Note that the *actual* functional area corresponding to the modules is not being considered here. The actual functional area depends on the particular module assignment and a higher number of modules does not necessarily imply a larger functional area [65]. The lower bounds have a strong corelation to the *number* of modules. The lower bounds on the BIST resources increase with number of modules because there are more modules to test which results in a larger number of input and output variable sets, while the storage concurrency of the variables remains the same. Tables 4.7 and 4.8 demonstrate that among two module assignments Π_M^1 and Π_M^2 such that $\mid \Pi_M^2 \mid < \mid \Pi_M^1 \mid$, assignment Π_M^2 might be desirable from the BIST resources point of view even if $Area(\Pi_M^2) > Area(\Pi_M^1)$.

4.9 Summary

In this chapter we have derived lower bounds on BIST resources that would be required to test a synthesized data path. The lower bound estimation techniques are performed on scheduled data flow graphs with a module assignment. The lower bounds are shown to be tight and given complete flexibility in the assignment of registers, each of the bounds can be individually achieved. The bounds give a mechanism for comparing the quality of area-performance competitive schedules and module assignments with respect to BIST resource requirement. The bounds along with a library of BIST registers can give an estimate of the actual BIST area overhead. The theory developed in this chapter on *BIST resource* bounds can be used in conjunction with that for estimating *functional resources* so that the *total* area of the synthesized design can be accurately estimated.

Chapter 5

Assignment for Reducing BIST Resources

In this chapter we explore the contribution of register and interconnect assignment to BIST resource cost of data paths and develop techniques to synthesize data paths with low BIST area overhead. It is assumed that a schedule and a module assignment has already been determined. BIST resource optimization during those stages of high-level synthesis is the subject of discussion in the next chapter.

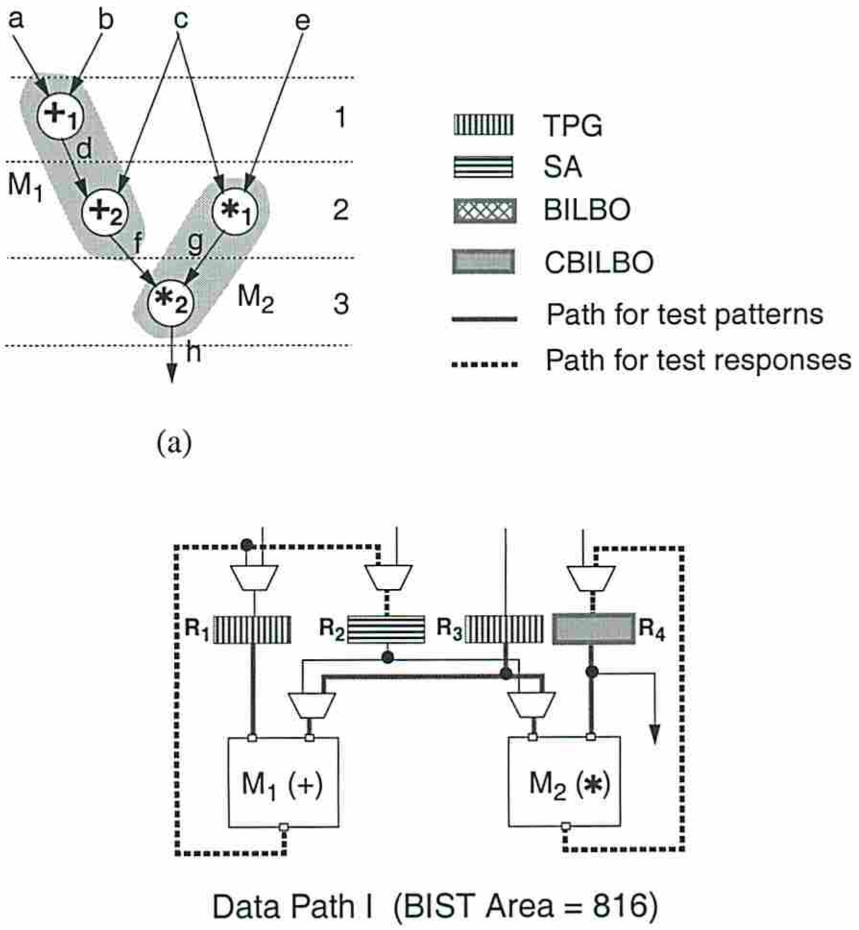
5.1 Introduction

The hardware assignment task in high-level synthesis comprises three subtasks: 1) assignment of operations to functional modules (module assignment), 2) assignment of variables to registers (register assignment), and 3) assignment of data transfers to interconnect and appropriate ports of functional modules (interconnect assignment). Various approaches to data path assignment with different ordering of these subtasks have been studied [22],[16], [25],[19]. Each has its merits and limitations. All three components of hardware assigned during the subtasks contribute to functional area. However, from the point of view of testability area overhead, the three components are different. In the self-testable version of the data path, registers are modified as BIST resources and interconnect is used to transfer test data to and from the modules. Modules are the hardware components that are *under test* in the proposed minimal intrusion BIST methodology. Hence the considerations for testability area overhead optimization in register and interconnect assignment are fundamentally different from those during module assignment. Complete information about the hardware to be tested (i.e. functional modules) makes estimation of required BIST registers more accurate and hence in this thesis we propose that register and interconnect assignment be performed independent from module assignment and after module assignment is known. Module assignment can be performed simultaneously with scheduling such that register and interconnect assignment that follows can benefit from it to optimize BIST area overhead. That is discussed in the next chapter. In this chapter, we focus on register and interconnect assignment assuming that scheduling and module assignment has already been performed.

5.2 BIST Considerations During Assignment

Consider a data path with a schedule and module assignment as shown in Fig. 5.1(a). Fig. 5.1(b) shows an implementation of the data path assuming a register and interconnect assignment that requires a minimum number of registers. The register and interconnect assignment has been done without any regard for BIST resources required to test the data path. The register assignment chosen is $R_1 = \{a, d\}$, $R_2 = \{b, f\}$, $R_3 = \{c\}$ and $R_4 = \{e, g, h\}$. For interconnect assignment, the left operands from the DFG are connected to the left input ports of the module and the right operands to the right input ports. The minimal intrusion BIST solution shown in Fig. 5.1(b) determined using the 0-1 ILP, has a minimum BIST area overhead of 816 cell units. Note that for the left input port of the adder and the right input port of the multiplier, there is only choice of test pattern generator, namely, R_1 and R_4, respectively. For the multiplier, R_4 is the only choice for test response compression also. Since R_4 does generation and compression for the multiplier, and it is the only register that can provide these test functionalities to the multiplier, the only choice is to modify it to a CBILBO.

An optimum BIST area overhead solution would select embeddings such that the BIST resources are shared between several functional modules. Traditional assignment techniques synthesize data paths without any consideration of how input



(a)

(b)

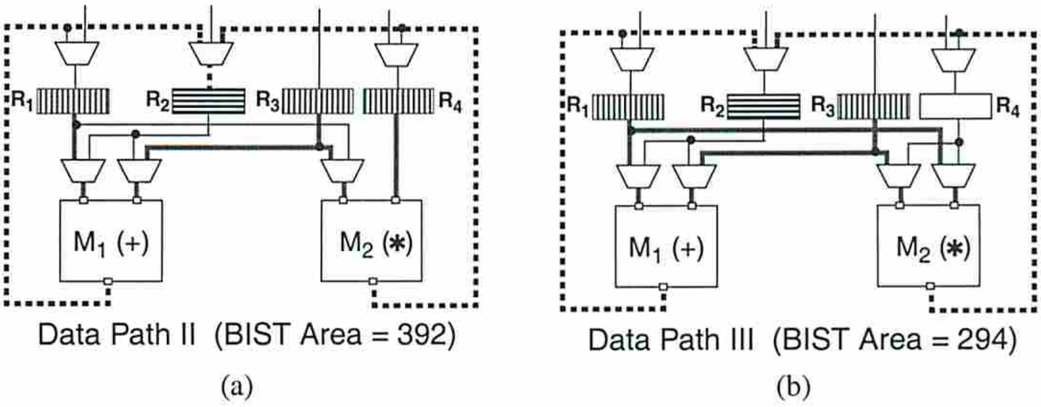
Figure 5.1: Assignment without BIST considerations

registers or output registers of modules can be shared between modules. For example, if a synthesized data path with m modules had disjoint sets of output registers, then each module would require a SA dedicated to it and hence m registers would have to be modified for test response compression capability. However, if the data path were synthesized such that the sets of output registers of the modules are not disjoint, then less than m registers could perform test response compression of the m modules. There are several other scenarios where if the same behavior was synthesized in an alternate way, lesser number of TPGs, SAs, BILBOs and CBILBOs would be required to test the data path.

The two factors contributing to a reduction in BIST resource cost are: 1) sharing of test functionality of BIST registers between different modules, and 2) minimizing the number of registers that would *necessarily* have to be modified as expensive BIST registers in the BIST version of a data path. In the second factor, we are primarily concerned with CBILBO registers because a CBILBO costs more than twice a normal register (as seen in Table 3.1).

Fig. 5.2(a) shows another hardware implementation of the DFG from Fig. 5.1(a). The schedule and module assignment is the same. However an alternate register assignment - $R_1 = \{a, f\}$, $R_2 = \{b, d, h\}$, $R_3 = \{c\}$ and $R_4 = \{e, g\}$ - is used for this data path. There are two things to note about the register assignment for Data Path II as compared to the one for Data Path I. Firstly, in Data Path II, variables d and h are assigned to the same register, namely, R_2. Variable d is an output variable of an operation assigned to module M_1 and variable h is an output variable of an operation assigned to M_2. Assigning these variables to R_2 makes the register a candidate for test response compression for both M_1 and M_2. The test functionality of response compression for the modules is shared by R_2 and it gets chosen as an SA in the minimal intrusion BIST solution as shown in Fig. 5.2(a). The second thing to be observed is that the register connected to the right input port of M_2, namely, R_4 does not have to be modified as a CBILBO. M_2 has a choice of R_2 and R_4 for test response compression and R_4 for test pattern generation. R_4 could be chosen as a CBILBO, but that would result in an expensive BIST solution. The flexibility accorded by the register configuration in Data Path II in the choice of test response compressors for M_2 is utilized to avoid an expensive BIST solution involving a CBILBO. The minimum area BIST solution for Data Path II has a cost of 392 compared to 816 in Fig. 5.1(b).

Data Path III in Fig. 5.2(b) is yet another implementation of the same DFG. The register assignment here is the same as in Fig. 5.2(a). However, the connectivity of the registers to the input ports has been changed. Module M_2 needs operands c and e in the second control step and operands f and g in the third control step. Based on the register assignment and the fact that c and f are *left* operands of M_2 and f and g are *right* operands, Data Path II is obtained. In Data Path III, advantage is taken of the commutativity property of multiplication and operands f



Data Path II (BIST Area = 392)

(a)

Data Path III (BIST Area = 294)

(b)

Figure 5.2: Assignment with BIST considerations (a) Register (b) Interconnect

and g are switched between the input ports. Now R_4, which holds g, is connected to the left input port and R_1, which holds f, is connected to the right input port. This creates a choice of R_1 and R_4 as test pattern generators for the right input port of M_2. Since R_1 is also a candidate for test pattern generation for M_1, the minimum area BIST solution selects R_3 as a TPG, resulting in saving R_4 as a BIST resource. The cost of the minimum BIST solution in this case is further reduced to 294.

The two alternate register assignments of the DFG in Fig. 5.1(a) use the same number of registers, 4, which is the minimum for this particular DFG. In fact, for this particular DFG, there are 288 distinct ways of assigning the variables to 4 registers. For a given scheduled DFG, a number of distinct assignments of variables to the minimum number of registers is possible. Only a subset of these are preferable in terms of interconnect complexity. Also only a subset of these result in data paths that can be self-testable with a low BIST area overhead. The discussion in this section shows that certain register assignments are beneficial to reducing BIST resources in a synthesized data path. Also, the interconnect assignment chosen can have an effect on the BIST resources. The properties of register and interconnect assignment that facilitate the reduction of BIST resources were discussed above. In the next sections, these properties are formalized and a structured technique for constructing assignments is presented.

5.3 Test Variables and BIST Registers

For directing assignments to low BIST area overhead data paths, a mechanism that relates variables to test functionality in a data path is required. For this purpose we define BIST functions and view each variable as a *test variable* in addition to a functional variable to be stored in a register.

A **BIST function**, f^B, is a subset of $\{\mathcal{G}, \mathcal{C}, \mathcal{NS}, \mathcal{S}\}$, where $\mathcal{G}$ represents test pattern generation functionality, $\mathcal{C}$ represents test response compression functionality, $\mathcal{NS}$ represents non-simultaneous $\mathcal{G}$ and $\mathcal{C}$ functionality and $\mathcal{S}$ represents simultaneous $\mathcal{G}$ and $\mathcal{C}$. The BIST function is defined for i) a register and iii) a single variable. The notation used in the two cases is the same and the context will be clear from the argument of f^B.

Definition 17 *The* **BIST function** *of*

1. *a register R, denoted by $f^B(R)$, is the combination of BIST functions that R could perform in a synthesized data path;*
2. *a variable v, denoted by $f^B(v)$, is the combination of BIST functions that a register could perform in a synthesized data path if the variable v was the only variable assigned to that register.*

Corresponding to the four combinations of test functionalities required for BIST, we define **candidacy** of registers as BIST registers. R being a **candidate** for a certain type of BIST register implies that R can be modified to that type in some valid BIST embedding. Note that it is not *necessary* to modify R for a valid BIST solution. R is just *one* of the choices for a valid BIST solution.

Definition 18 1. *If $\{\mathcal{G}\} \subseteq f^B(R)$, then R is a* **candidate for TPG**.

2. *If $\{\mathcal{C}\} \subseteq f^B(R)$, then R is a* **candidate for SA**.

3. *If $\{\mathcal{G}, \mathcal{C}, \mathcal{NS}\} \subseteq f^B(R)$, then R is a* **candidate for BILBO**.

4. *If $\{\mathcal{G}, \mathcal{C}, \mathcal{S}\} \subseteq f^B(R)$, then R is a* **candidate for CBILBO**.

In the above definition, note that a register can be a candidate for both a BILBO and a CBILBO. Moreover, a register that is a candidate for a CBILBO or a BILBO is always a candidate for a TPG and an SA. The BIST solution selects a subset of the candidate registers to perform test pattern generation and response compression for all the modules in the data path. The BIST solution will be denoted by using an asterisk $*$ as the superscript for the test functionality utilized for BIST. For example, a register R with $f^B(R) = \{\mathcal{G}, \mathcal{C}, \mathcal{NS}\}$ is a candidate for TPG, SA and BILBO. If the BIST solution selects R as TPG, then $f^B(R) = \{\mathcal{G}^*, \mathcal{C}, \mathcal{NS}\}$. Corresponding to the four test functions of BIST, we define four types of *test variables.*

Definition 19 *A variable v is called a*

1. **G**-variable *if it is only an input variable for modules. For a* **G**-variable*, $f^B(v) = \{\mathcal{G}\}$.*

2. **C**-variable *if it is only an output variable for modules. For a* **C**-variable*, $f^B(v) = \{\mathcal{C}\}$.*

3. **NSGC**-variable *if it is an input variable for a module and an output variable for a* different *module. For a* **NSGC**-variable*, $f^B(v) = \{\mathcal{G}, \mathcal{C}, \mathcal{NS}\}$.*

4. **SGC**-variable *if it is an input variable and an output variable for the same module. For a* **SGC**-variable*, $f^B(v) = \{\mathcal{G}, \mathcal{C}, \mathcal{S}\}$.*

In the register assignment process, as variables are assigned to a register, the BIST function of the register changes. The BIST function of the register at any point in the assignment process depends on the test functionalities of the variables assigned to it. If a variable v was the only variable assigned to a register R, the BIST function of v, $f^B(v)$, reflects the test functionalities that R could perform for BIST. For example, a register to which a **SGC**-*variable* is assigned would be a candidate for CBILBO in the BIST version of the data path. Hence, under the assumption of single variable assignment, $f^B(v) \equiv f^B(R)$. If more than one variable is assigned to R, then for each variable $v \in R$, $f^B(v) \subseteq f^B(R)$. When variables are merged (assigned to the same register), the register can perform the BIST functions of all the variables, and in addition the concept of simultaneity must now be considered. When variables of the same type are assigned to the same register, the register acquires the same test functionality as the variables. If variables of different type are assigned to the same register, the register acquires the test functionality of all the variables and in addition could acquire the $\mathcal{NS}$ and/or $\mathcal{S}$ functionalities. For a pair of variables v and w assigned to a register R, the following cases are possible.

Case (i): Homogeneous variables

(a) v and w are both input only variables (**G**-variables)

(b) v and w are both output only variables (**C**-variables)

Case (ii): Heterogeneous variables ($\mathcal{G} \in f^B(v), \mathcal{C} \in f^B(w)$)

(a) $\exists M_i$ and $M_j, i \neq j$ such that $v \in IVar(M_i), w \in OVar(M_j)$

(b) $\exists M_i$ such that $v \in IVar(M_i), w \in OVar(M_i)$

In Case(i)(a) and (b), since both the variables add only $\mathcal{G}$ or $\mathcal{C}$ functionality to R, simultaneity is not an issue. In Case(ii)(a) and (b), variable v adds $\mathcal{G}$ functionality and variable w adds $\mathcal{C}$ functionality. In Case(ii)(a), the $\mathcal{G}$ and $\mathcal{C}$ functionality is not required at the same time since M_i and M_j can be tested in separate test sessions. However in Case(ii)(b), the $\mathcal{G}$ and $\mathcal{C}$ functionality is required simultaneously to test M_i. Hence if Case(ii)(a) is satisfied, then $\mathcal{NS} \in f^B(R)$ and if Case(ii)(b) is satisfied, then $\mathcal{S} \in f^B(R)$. Note that variables v and w could satisfy both Case(ii)(a) and (b). The above discussion gives us the following lemma to determine the BIST function of a register.

Lemma 4 *The BIST function of a register R is*

$$f^B(R) = (\bigcup_{\forall v \in R} f^B(v)) \cup (\bigcup_{\forall v, w \in R} f^B_s(v,w))$$

$$\begin{aligned} \textit{where } f^B_s(v,w) &= \phi && \textit{for Case(i)} \\ &= \{\mathcal{NS}\} && \textit{for Case(ii)(a) only} \\ &= \{\mathcal{S}\} && \textit{for Case(ii)(b) only} \\ &= \{\mathcal{NS}, \mathcal{S}\} && \textit{for Case(ii)(a) and (b)} \end{aligned}$$

Table 5.1: Registers and their BIST functions for Data Path I (Fig. 5.1)

Register R	Variables in R	BIST function $f^B(R)$	Candidacy of R	Type in BIST soln.
R_1	$\{a,d\}$	$\{\mathcal{G}^*,\mathcal{C},\mathcal{NS},\mathcal{S}\}$	TPG, SA, BILBO, CBILBO	TPG
R_2	$\{b,f\}$	$\{\mathcal{G},\mathcal{C}^*,\mathcal{NS}\}$	TPG, SA, BILBO	SA
R_3	$\{c\}$	$\{\mathcal{G}^*\}$	TPG	TPG
R_4	$\{e,g,h\}$	$\{\mathcal{G}^*,\mathcal{C}^*,\mathcal{S}^*\}$	TPG, SA, CBILBO	CBILBO

Example 3 (Fig. 5.1)

*The classification of test variables of the DFG in Fig. 5.1(a) is as follows. Variables $a, b, c,$ and e do not belong to either $OVar(M_1)$ or $OVar(M_2)$ and they are only input variables (*G-variable*). Similarly h does not belong to either $IVar(M_1)$ or $IVar(M_2)$, hence it is a* C-variable. *Variable f belongs to $OVar(M_1)$ and $IVar(M_2)$, hence it is a* **NSGC**-variable. *Variables d and g are* **SGC**-variables *because $d \in OVar(M_1)$ and $d \in IVar(M_1)$; $g \in OVar(M_2)$ and $g \in IVar(M_2)$.*

The variables can be assigned to a minimum of four registers. Fig. 5.1(b) shows a data path synthesized using a register assignment corresponding to the minimum number of registers. The assignment was done oblivious of any BIST considerations. The modification for BIST is also indicated in the figure. Note that, many possible valid ways of making the data path self-testable exist, but the BIST solution depicted in the figure is one with a minimum area overhead which is 816 cell units. Table 5.1 shows the variables assigned to each register, the BIST function of each register, the candidacy of each register as a BIST register and the actual modification of each register in the self-testable version of the data path.

5.3.1 Sharing BIST Resources between Modules

As seen in Example 5.3 in the previous section, the BIST solution modifies some of the registers into BIST registers with the constraint that for each module, each input port has a BIST register with $\mathcal{G}^*$ functionality and each output port has a

BIST register with $\mathcal{C}^*$ functionality. The objective in making a data path self-testable is to perform the selection of BIST registers from candidates such that this constraint is met with minimum area overhead. Assignment of each variable to a register gives the register some test functionality, depending on the type of the variable. If the assignment is done considering which input and output variable sets the variables belong to, then the test functionalities of the registers can be shared between modules. Sharing of the test functionalities of registers between modules results in lower number (and area overhead) of BIST registers to provide all the test functionalities for each module in the design.

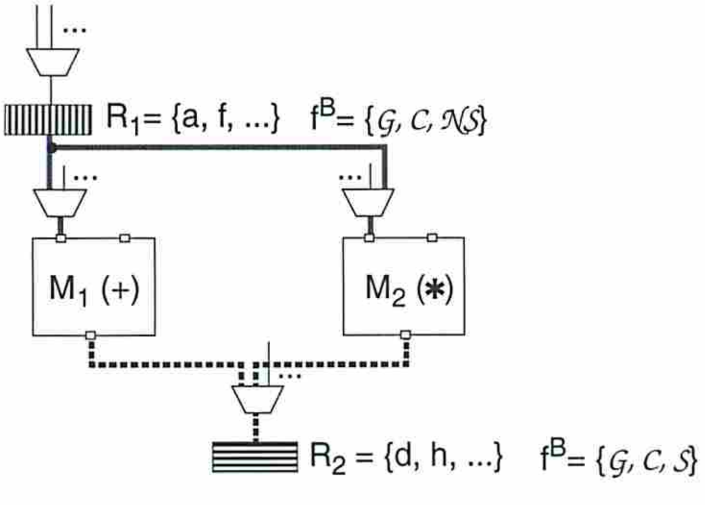


Figure 5.3: Assignment for sharing of test functionality

Fig. 5.3 shows a partially synthesized data path corresponding to the schedule and module assignment of Fig. 5.1. From Table 5.1, it can be seen that variable a is a **G**-variable and variable f is a **NSGC**-variable. Also, $a \in IVar(M_1)$ and $f \in IVar(M_2)$. Assigning a and f to register R_1 makes R_1 a candidate for providing $\mathcal{G}$ functionality to the input ports of M_1 and M_2. The multiplexers at the input ports of M_1 and M_2 would be required if the remaining input variables of the modules were assigned to registers other than R_1. Similarly, register R_2 has variables d and h assigned to it. Since $d \in OVar(M_1)$ and $h \in OVar(M_2)$, R_2 can share the $\mathcal{C}$ functionality for the output ports of M_1 and M_2. The effect of sharing of test functionalities between modules is quantified in the notion of **sharing degree** of variables and registers.

Definition 20 *1. The* **sharing degree of a variable** v, *$SD_{var}(v)$ is the sum of the number of modules for which v is an input variable and the number of modules for which v is an output variable.*
If $v \in IVar(M_j)$, let $X_j^v = 1$, else $X_j^v = 0$; if $v \in OVar(M_j)$, let $Y_j^v = 1$, else $Y_j^v = 0$. Then $SD_{var}(v) = \sum_{j=1}^{m}(X_j^v + Y_j^v)$, where m is the total number of modules assigned.

2. The **sharing degree of a register** R, *$SD_{reg}(R) = \sum_{j=1}^{m}(X_j^R + Y_j^R)$, where*

$$X_j^R = \bigvee_{\forall v \in R} X_j^v \quad \textit{and} \quad Y_j^R = \bigvee_{\forall v \in R} Y_j^v$$

Note that variables X_j^v and Y_j^v are treated as integers in 1) and as boolean variables in 2).

The sharing degree of a register R is the sum of distinct input variable sets and distinct output variable sets each of which contain at least one element of R. $SD_{reg}(R)$ is the number of modules for which R can provide either $\mathcal{G}$ functionality or $\mathcal{C}$ functionality.

Consider a register R that has been assigned some variables. Let the sharing degree of R after another variable v is assigned to it be denoted by $SD_{reg}(R \cup \{v\})$. Now $SD_{reg}(R \cup \{v\}) = SD_{reg}(R) + SD_{var}(v) - \sum_{j=1}^{m}(X_j^R * X_j^v + Y_j^R * Y_j^v)$. Using this measure, register assignment can be guided by choosing merges that result in larger increases in sharing degrees of registers over those resulting in smaller increases. This would result in registers with high sharing degrees, thereby requiring a fewer number of BIST registers globally in the design. The increase in the sharing degree of a register R as a result of assigning variable v to it is denoted by $\Delta SD^v(R)$, where $\Delta SD^v(R) = SD_{reg}(R \cup \{v\}) - SD_{reg}(R)$.

5.3.2 Essential BIST Registers

Two parameters can be associated with a functional register R when it is being considered to be a BIST register, namely, 1) the **BIST function** $f^B(R)$, and 2) the **sharing degree** $SD_{reg}(R)$. $f^B(R)$ indicates the type of BIST register for which R is a candidate. $SD_{reg}(R)$ indicates the number of modules for which R is a candidate, and hence the probability that R will be selected in an minimum BIST area overhead solution. A third factor affecting the selection of a candidate BIST register in the BIST solution is the availability of other candidates for the same BIST function for the same module. For example, if the output port of a module is connected to two registers, then both registers are candidates for test response compression and the one with a higher sharing degree will be chosen in the optimum BIST solution. However, if there was only one register connected to the output, then selecting this candidate for test response compression cannot be avoided.

Definition 21 *A register R in a data path is an* **essential BIST register** *of a certain type (TPG, SA, BILBO or CBILBO) if for any module in the data path, R is selected as that type of BIST register in all BIST embeddings of the module.*

A CBILBO candidate, in addition to providing $\mathcal{G}$ and $\mathcal{C}$ capabilities also provides the $\mathcal{S}$ capability (simultaneity of $\mathcal{G}$ and $\mathcal{C}$). From Table 1 it can be seen that a CBILBO register consumes more area than two normal registers. The area overhead of a CBILBO is a high price to pay for the added functionality of simultaneity since simultaneity is not *necessary* as long as the $\mathcal{G}$ and $\mathcal{C}$ requirements are satisfied for the input and output ports respectively. In previous work, assignment techniques incorporated the objective of minimizing CBILBOs in the design by minimizing self-adjacent registers in the data path [44], [48],[49]. However, self-adjacency is only a *necessary* condition for a register to be made CBILBO. Fig. 5.4 shows three scenarios with self-adjacent registers. In Fig. 5.4(a) register A is self-adjacent. But the presence of output register B precludes the need for a CBILBO. In Fig. 5.4(b) *both* the output registers A and B are self-adjacent, but because of the presence of other input registers, such as C, a CBILBO is not required. In Fig. 5.4(c), however, a CBILBO cannot be avoided since register A is essential to generate test patterns and compress responses for the module.

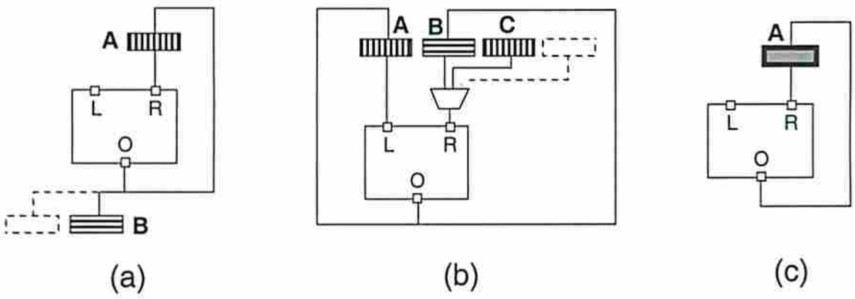


Figure 5.4: Self-adjacency and CBILBOs

The identification of such *essential* CBILBOs can help minimize BIST area overhead by performing register assignment that avoids (or minimizes) essential CBILBOs. Next we derive conditions for register assignment which, when followed by *minimum* interconnect assignment, leads to essential CBILBOs in the synthesized data path. A register R_i is a CBILBO candidate only if it can provide the $\mathcal{G}$ and $\mathcal{C}$ functionality for an input port of the same module. Hence it is necessary that $\{\mathcal{G}, \mathcal{C}, \mathcal{S}\} \subseteq f^B(R_i)$. R_i is an *essential* CBILBO to test M_k if it is essential to provide $\mathcal{G}$ functionality to an input port of M_k and essential to provide $\mathcal{C}$ functionality to the output port.

Definition 22 *An* **instance** i *of a module* M_k*,* M_k^i *is the ith operation executed by* M_k *from the operations assigned to it.*

Definition 23 *The* **temporal multiplicity** *of a module* M_k*,* $TM(M_k)$ *is the number of operations from* V *assigned to* M_k*,*

Lemma 5 *A register* R_x *is an essential CBILBO w.r.t. module* M_k *iff*

$OVar(M_k) \subseteq R_x$ **and** $R_x \cap IVar(M_k^j) \neq \phi$ *for* $j = 1, 2, \ldots, TM(M_k)$.

Proof: The assignment of input registers to input ports of a commutative operation maps to a double clique partitioning of a register compatibility graph, where each vertex is an input register and there is an edge between two registers only if they can be connected to the same input port [66]. From the two disjoint cliques, each one corresponds to registers connected to one input port. The rest of the vertices (input registers) are connected to both input ports. A *minimum* connectivity assignment is one that minimizes registers connected to both input ports [66].

$R_x \cap IVar(M_k^j) \neq \phi$ for $j = 1, 2, ..., TM(M_k)$, implies that every time module M_k executes its operation (i.e. executes instance j), it receives one of its operands from R_x. Hence, the vertex corresponding to R_x in the input register compatibility graph is disjoint for the vertices corresponding to the other input registers. Moreover since, the other registers hold operands of M_k that are required in different control steps (i.e. different instances j), they are fully connected to each other. A minimum connectivity assignment corresponds to R_x being the only input register connected to one input port and the rest of the input registers connected to the other input port.

If ($\Leftarrow$):

If $R_x \cap IVar(M_k^j) \neq \phi$ for $j = 1, 2, ..., TM(M_k)$, then from the above discussion, it can be seen that R_x is the only register connected to one input port of M_k. Hence it is essential to provide $\mathcal{G}$ functionality for M_k. Since $OVar(M_k) \subseteq R_x$, it is essential to provide $\mathcal{C}$ functionality for M_k. Therefore, R_x is an essential CBILBO.

Only if ($\Rightarrow$):

R_x is an essential CBILBO for M_k. Hence R_x is essential to provide the $\mathcal{G}$ functionality and essential to provide the $\mathcal{C}$ functionality to M_k. Since R_x is essential to provide $\mathcal{G}$ functionality, it must be the only register connected to an input port of M_k. This is possible only if the register assignment is such that, for every iteration of data through M_k, R_x supplies an operand. This implies $R_x \cap IVar(M_k^j) \neq \phi$ for $j = 1, 2, ..., TM(M_k)$. Since R_x is essential to provide $\mathcal{C}$ functionality, R_x holds the output variable of M_k after computation of each instance of M_k, implying $OVar(M_k) \subseteq R_x$. □

The above lemma enables us to check if a particular assignment would result in an *essential* CBILBO in the data path. The register assignment algorithm (described in the next section) includes this check at every assignment step to avoid assignments leading to essential CBILBOs.

Lemma 6 *Given a schedule S, a module assignment Π_M, a partial register assignment $\Pi_R = \{R_1, R_2, ..., R_p\}$ and a new variable to be assigned, x. Let Π_R^C corresponds to an assignment of x that creates an essential CBILBO and $\Pi_R^{no\_C}$ corresponds to an assignment that does not create an essential CBILBO. If $(c^C - c^B) > c^B$, then $Area_{BIST}(\Pi_R^{no\_C}) < Area_{BIST}(\Pi_R^C)$, where $Area_{BIST}$ denotes the area overhead of essential BIST registers in a data path.*

Proof: Let c^C, c^B, c^T and c^S be the cost of modification of a normal register to a CBILBO, BILBO, TPG and SA, respectively.

Consider register assignment Π_R^C and without loss of generality assume that x is assigned to R_1 making it an essential CBILBO. There are four possible choices for the test functionality of R_1 in Π_R. It could have been either an essential BILBO, an essential TPG, an essential SA or none of the above. If it were to become an essential CBILBO, the BIST area overhead would increase by $(c^C - c^B)$, $(c^C - c^T)$, $(c^C - c^S)$ or c^C, corresponding to the four cases, respectively. Of the four cases, the smallest increase is $(c^C - c^B)$.

Consider register assignment $\Pi_R^{no\_C}$ and without loss of generality assume that x is assigned to R_2 such that it does not create an essential CBILBO. In the worst case R_2 could become an essential BILBO and the BIST area overhead would increase by 0, $(c^B - c^T)$, $(c^B - c^S)$ or c^B, corresponding to the four cases, respectively. Of the four cases, the largest increase is c^B.

Since $(c^C - c^B) > c^B$, the smallest possible increase in area overhead due to essential BIST registers in Π_R^C is worse than the largest possible increase in the case of $\Pi_R^{no\_C}$. Hence $Area_{BIST}(\Pi_R^{no\_C}) < Area_{BIST}(\Pi_R^C)$. □

5.4 Register Assignment

The register assignment problem can be modeled as coloring of the *variable conflict graph.*

Definition 24 *A* **variable conflict graph** $C = (V_C, E_C)$*, is a graph with vertices corresponding to variables in a scheduled DFG with an edge between two variables only if they have overlapping lifetimes. A* **coloring** *of this graph corresponds to a valid register assignment with each color corresponding to a register.*

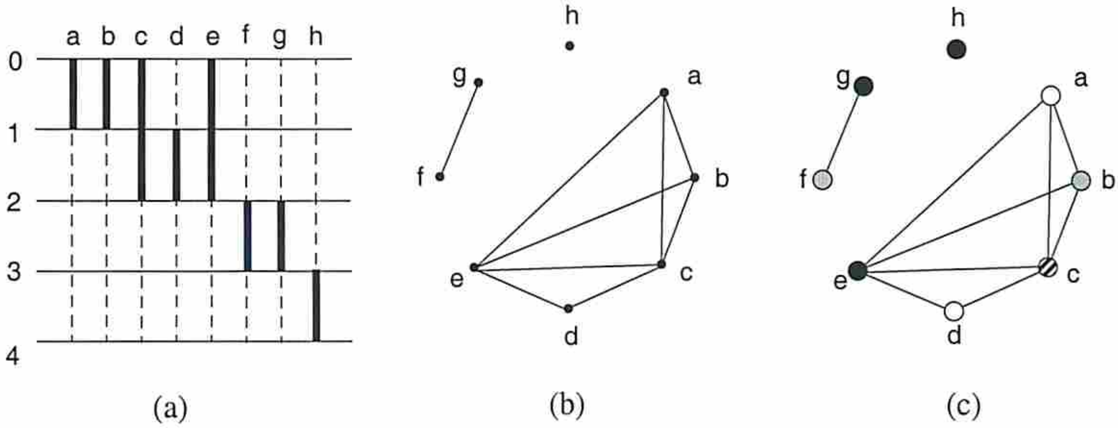


Figure 5.5: Conflict graph of variables

The lifetime table for the DFG in Fig. 5.1(a) is shown in Fig. 5.5(a) and the corresponding conflict variable graph in Fig. 5.5(b). A coloring of the graph corresponding to the register assignment in Fig. 5.1(b) is indicated in Fig. 5.5(c). In the rest of this paper we will use the terms color and register interchangeably. Similarly the terms coloring and assignment are used interchangeably. Minimum coloring of general graphs has been proven to be NP-complete [15]. However polynomial time algorithms exist for special graphs such as chordal graphs and interval graphs [67]. If a data flow graph does not contain mutual exclusion constructs, the resulting variable conflict graph is an interval graph [68].

The greedy optimum coloring algorithm uses the hereditary property of interval graphs which is defined through *simplicial* vertices. A vertex v of $C = (V_C, E_C)$ is simplicial if its *adjacency set* $Adj(v)$ induces a clique in C. The adjacency set is the set of all vertices that are connected to v. In Fig. 5.6(a), vertex b is a simplicial vertex because its adjacency set $\{a, e, c\}$ induces a clique shown highlighted. An interval graph has at least two simplicial vertices. If a simplicial vertex and all its incident edges are removed, the remaining graph is also an interval graph with at least two

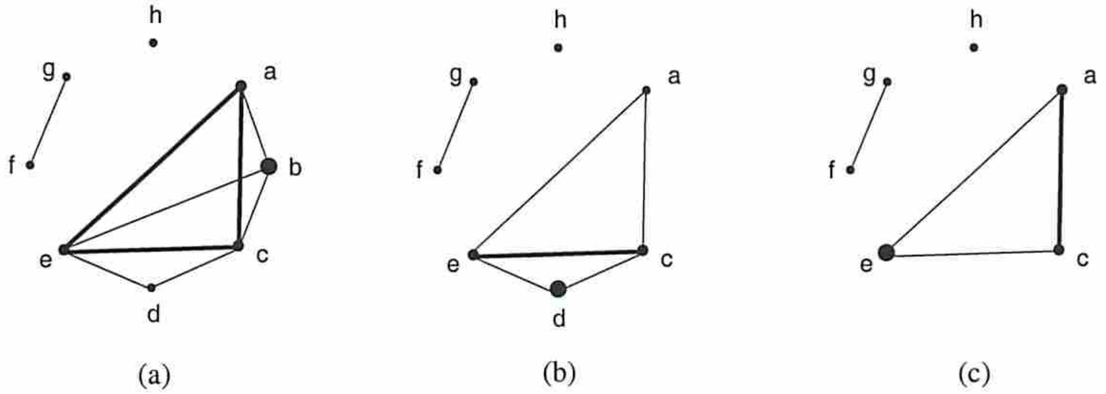


Figure 5.6: Simplicial vertices

simplicial vertices. Fig. 5.6(b) is the same graph in Fig. 5.6(a) with the vertex b removed. A simplicial vertex of the remaining graph, namely, d is shown. An ordering of the vertices such that each vertex is a simplicial vertex of the remaining graph is called *perfect vertex elimination scheme* (*PVES*). An interval graph has many such perfect vertex elimination schemes. One such $PVES$ for the example conflict graph is constructed in Fig. 5.6 and is b, d, e, after which the remaining vertices can be chosen in any order as all of them are simplicial vertices. The optimal coloring algorithm constructs one $PVES$ scheme *arbitrarily* and colors the vertices *greedily* in the reverse order (*reverse PVES*) [69]. Our heuristic is different from the optimal coloring algorithm in two respects: 1) it selects the $PVES$ in a more structured way taking into account information such as the sharing degree of variables and size of maximum cliques; and 2) the vertices are then colored using this scheme. However instead of assigning colors greedily, many more coloring possibilities are explored and the one most suited for maximizing the sharing of BIST resources for BIST is selected. We have already defined sharing degree of a variable v, $SD_{var}(v)$. We define another useful parameter, *maximum clique size*, that is useful in determining a $PVES$.

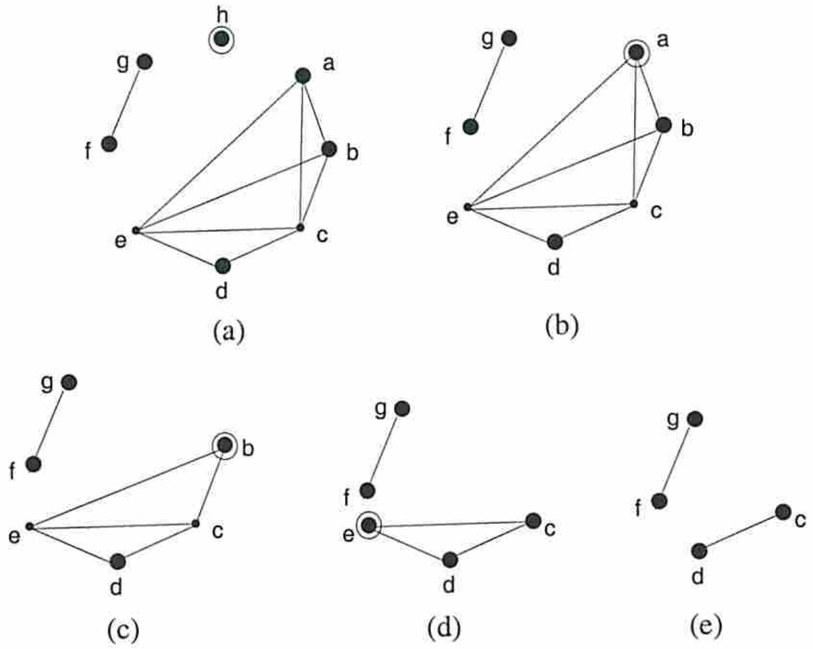
Definition 25 *The* **maximum clique size** *of a variable v, denoted by $MCS(v)$ is the size of the maximum clique in the variable conflict graph to which v belongs.*

The maximum clique size is an indication of the number of variables with which a given variable conflicts in terms of sharing a register. During incremental register assignment, a variable with a high MCS value is likely to conflict with a higher number of registers from the ones already assigned.

1. Selection of a $PVES$: With each vertex of the conflict graph we associate a sharing degree as per Definition 2. In addition we also find the size of the maximum clique to which each vertex belongs as per Definition 25. The vertices are ordered in increasing order of their sharing degrees. Among vertices with the same sharing degree they are ordered in increasing order of the maximum clique sizes. Thus the ordering of the vertices is such that if v is before w, then $SD_{var}(v) \leq SD_{var}(w)$ and if $SD_{var}(v) = SD_{var}(w)$ then $MCS(v) \leq MCS(w)$. At every step of constructing the $PVES$ there is a choice of simplicial vertices. The $PVES$ is now determined such that at each step a simplicial vertex that is earliest in this order is selected. Since vertices are colored in the *reverse* $PVES$ order, this results in vertices with higher sharing degrees to be considered earlier when there is maximum flexibility in the assignment of colors. Also since vertices with a higher MCS value are considered earlier more colors are fixed in the earlier stages thus creating more coloring options to explore. This enables the heuristic to search the design space more efficiently for finding a coloring with low testability area overhead keeping the number of colors close to optimum. Fig. 5.7 demonstrates the selection of a $PVES$ using the MCS and SD values. At each step there is a choice of many simplicial vertices indicated by the bold vertices. Among them the one circled is chosen by our algorithm. After the stage shown in Fig. 5.7(e), the vertices are chosen in the order g, f, d, and c. The $PVES$ chosen by our scheme is therefore h, a, b, e, g, f, d and c.

2. Coloring in reverse $PVES$ order: For the purposes of the following discussion the vertices will be referred to by their number in the reverse $PVES$. Let the coloring after the kth vertex is colored be denoted as $\Pi_R^k = (\ R_1^k, R_2^k, ..., R_{c_k}^k\ )$ where $R_i^k \cap R_j^k = \phi$ if $i \neq j$ and $\bigcup_{i=1}^{c_k} R_i^k = \{\ 1,\ 2,...,k\ \}$. The total number of colors after the kth vertex is colored is c_k. Coloring vertex $(k + 1)$ implies adding it to one of $R_1^k, R_2^k, ..., R_{c_k}^k$ or if it conflicts with all registers, creating a new set $R_{c_{k+1}}^{k+1} = \{\ k + 1\ \}$ to produce a new coloring Π_R^{k+1}, where $c_{k+1} = c_k + 1$.

Variables *v* :	a	b	c	d	e	f	g	h
Sharing degree *SD(v)* :	1	1	2	2	1	2	2	1
Max. clique size *MCS(v)*:	4	4	4	3	4	2	2	1



(a) (b) (c) (d) (e)

Figure 5.7: $PVES$ based on $SD(v)$ and $MCS(v)$

The vertex $(k + 1)$ is colored in the following way. If $(k + 1)$ conflicts with all registers $R_1^k, R_2^k, ..., R_{c_k}^k$ then a new register $R_{c_{k+1}}^{k+1} = \{ k + 1 \}$ is created. Otherwise, out of the registers that do not conflict with $(k + 1)$ pick a register R_i^k such that $\Delta SD^{k+1}(R_i^k) = SD_{reg}(R_i^k, k+1) - SD_{reg}(R_i^k)$ is maximum. Such an R_i^k corresponds to a register that can best utilize $(k + 1)$ to improve its sharing as a BIST resource. If there is more than one such register then the tie is broken by considering the sharing degree of the registers and the one which has the higher sharing degree is chosen. There are two cases in which a register other than R_i^k might be preferable. Case(i): Suppose variable $(k+1)$ is an output variable of some module M_j. If there is a register R_l^k with which $(k+1)$ does not conflict such that R_l^k already has an output variable of M_j assigned to it and if $SD_{reg}(R_l^k) > SD_{reg}(R_i^k, k + 1)$ then assigning $(k + 1)$ to R_i^k does not help the situation. On the other hand it might increase the interconnection cost. So it is preferable to assign $(k + 1)$ to R_l^k.

Case(ii): Suppose variable $(k+1)$ is an input variable of some module M_j. If among the non-conflicting registers there are two registers R_m^k and R_n^k such that each of them already has an input variable of M_j assigned then if their sharing degrees are higher than $SD_{reg}(R_i^k, k+1)$, it is preferable to assign $(k+1)$ to one of them.

In general both the cases can arise in which case a set of candidate registers is created of all such registers R_l^k, R_m^k and R_n^k and $(k+1)$ is assigned to the one which results in the highest increase in its sharing degree.

```
Step 1: $\Pi_R = \Phi$
Step 2: **While** $(L \neq \Phi)$
        Step 2.1: $cand\_reg = \Phi$
        Step 2.2: Remove first vertex $v$ from $L$
        Step 2.3: Find set of registers compatible with $v$, $compat\_\Pi_R$
        Step 2.4: **If** $(compat\_\Pi_R = \Phi)$
                Step 2.4.1: Allocate new register $R_{new}$ and assign $v$ to it
                Step 2.4.2: $\Pi_R = \Pi_R \cup \{ R_{new} \}$
        Step 2.5: **Else**
                Step 2.5.1: Select $R_i \in compat\_\Pi_R$ s.t. $\Delta SD^v(R_i)$ is maximum
                Step 2.5.2: **If** $\exists R_l \in output\_compat\_\Pi_R$ s.t. $SD_{reg}(R_l) > SD_{reg}(R_i, v)$
                                $cand\_reg = cand\_reg \cup \{ R_l \}$
                Step 2.5.3: **If** $\exists R_j, R_k \in input\_compat\_\Pi_R$ s.t.
                            $SD_{reg}(R_j) > SD_{reg}(R_i, v)$ and $SD_{reg}(R_k) > SD_{reg}(R_i, v)$
                                $cand\_reg = cand\_reg \cup \{ R_j, R_k \}$
                Step 2.5.4: **If** $(cand\_reg \neq \Phi)$
                            $R \leftarrow break\_tie(cand\_reg)$
                            $R = R \cup \{ v \}$
                Step 2.5.5: **Else**
                            $R_i = R_i \cup \{ v \}$
```

Algorithm for register assignment - *assign_reg(L)*

The optimality in the minimum coloring algorithm is guaranteed by assigning the vertex $(k+1)$ to the first R_i^k with which it does not conflict. Since our heuristic does not make such an assignment we cannot guarantee optimality in terms of the number of registers allocated. However the heuristic is near-optimal since it still relies on a $PVES$ of a conflict graph. In all the examples considered it resulted in the minimum number of registers.

The register assignment algorithm *assign_reg(L)* is shown above. It takes as input a list of vertices L in the *reverse* $PVES$ order and outputs a register assignment Π_R. The algorithm iterates over each vertex v selected from L. *compat*_Π_R is the set of registers that are compatible (do not conflict) with variable v. Let $v \in IVar(M_j)$ and/or $v \in OVar(M_k)$. *input_compat*_Π_R is a subset of *compat*_Π_R such that every register $R_i \in$ *input_compat*_Π_R is such that $R_i \cap IVar(M_j) \neq \phi$. These are registers that can provide $\mathcal{G}$ functionality to the module for which v is an input variable. Similarly, *output_compat*_Π_R is a subset of *compat*_Π_R such that every register $R_i \in$ *output_compat*_Π_R is such that $R_i \cap OVar(M_k) \neq \phi$. These are registers that can provide $\mathcal{C}$ functionality to the module for which v is an output variable. Registers from *output_compat*_Π_R and *input_compat*_Π_R that satisfy cases(i) and (ii) above form the set of candidate registers *cand_reg*. The procedure *break_tie* selects a register from *cand_reg* with the maximum increase in sharing degree.

Lemma 5 gives a condition that checks if a certain assignment leads to essential CBILBOs. The algorithm *assign_reg(L)* can be modified to *assign_reg_noCBILBO(L)* that includes this check. In *assign_reg_noCBILBO(L)*, a set of compatible registers *compat*_Π_R for a variable v is identified just as in *assign_reg(L)*. However a check is performed to identify if assignment of v to a register from *compat*_Π_R creates the conditions defined in Lemma 5. Such registers are dropped from consideration for assignment and a subset *noCBILBO_compat*_Π_R is created. The rest of the assignment procedure remains the same. It is possible that *noCBILBO_compat*_$\Pi_R = \phi$ while *compat*_$\Pi_R \neq \phi$ which implies that a new register will have to be allocated. But since it is desirable to keep the total number of registers close to minimum, in a case like this, *noCBILBO_compat*_Π_R is made the same as *compat*_Π_R implying that a CBILBO will be required. Our experiments indicated that the assignment space is large enough so that this situation does not occur frequently.

5.5 Interconnect Assignment

The register assignment algorithm does not take into account the effect on interconnect area except to resolve ties. The area due to interconnect hardware is usually abstracted to the area of multiplexers. For a given module assignment different register assignments have different effects on interconnect area. The typical situations that occur when two variables or intermediate registers are merged into one register are shown in Fig. 5.8. In the figure, the circles and ellipsoids indicate partial registers. In all cases, $f^B(v_1) = f^B(v_2) = \{\mathcal{G}, \mathcal{C}, \mathcal{NS}\}$ and $SD(v_1) = SD(v_2) = 2$.

In Fig. 5.8, cases (a) and (b) result in an extra multiplexer. In case (a), $f^B(\{v_1, v_2\})$ is the same as $f^B(v_1)$ (or $f^B(v_2)$) but $SD(\{v_1, v_2\}) = 4$. The $\mathcal{G}$ and $\mathcal{C}$ functionalities for the four modules can be covered by this one partial register instead of two. In case (b), merging v_1 and v_2 adds the $\mathcal{S}$ functionality and $f^B(\{v_1, v_2\}) = \{\mathcal{G}, \mathcal{C}, \mathcal{NS}, \mathcal{S}\}$ and $SD(\{v_1, v_2\}) = 3$. The $\mathcal{G}$ and $\mathcal{C}$ functionalities for the three modules can be covered by one partial register instead of two. Thus, in these two cases the extra multiplexer cost is compensated by the increase in the sharing of BIST functionality, which in turn reduces the BIST register requirement for the data path. In cases (c) and (d), $f^B(\{v_1, v_2\}) = \{\mathcal{G}, \mathcal{C}, \mathcal{NS}\}$ and $SD(\{v_1, v_2\}) = 3$. In both cases, after merging v_1 and v_2, the multiplexer requirement remains the same and one partial register is required for providing the BIST functionalities for the three modules.

The above discussion shows that a register assignment with consideration for BIST overhead and without any consideration for interconnect area will still result in a data path with lower *overall* area. However more reduction in the BIST area overhead can be achieved by assigning interconnect so as to make the most use of the register assignment. Given a module M_k and the set of input registers IR_k, each input register can be connected in one of the following three possible ways: 1) it is only connected to the "left" input port of M_k, 2) it is only connected to the "right" input port of M_k, and 3) it is connected to both the "left" and the "right" input port of M_k. Assignment of interconnect Π_I can be thus seen as a partition of IR_k into sets IR_k^L, IR_k^R and IR_k^{LR} corresponding to the cases 1), 2) and 3), respectively. For making the most use of the register assignment in reducing the BIST overhead, the

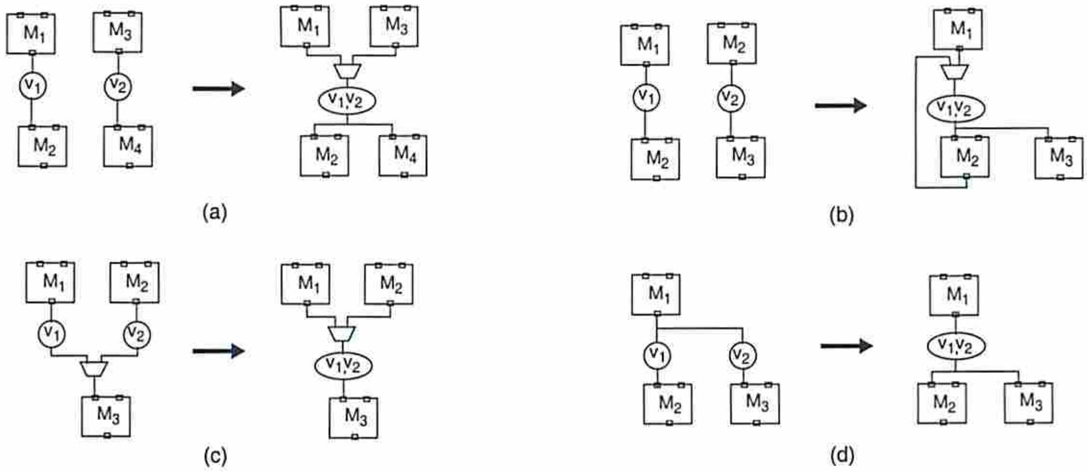


Figure 5.8: Trade-off between BIST functionality and interconnect

connectivity assignment can be directed to ensure that registers with high sharing degrees have a better chance of being selected as BIST registers. To test a module M_k, two registers each with $\mathcal{G}$ functionality and each connected to a distinct input port of M_k are required. Among all candidates for an input port, selection of a register with the highest sharing degree results in a globally minimum BIST area overhead. Hence it is advantageous to have a register with high sharing degrees in both, IR_k^L and IR_k^R. The output connectivity assignment has no effect on the selection of BIST resources.

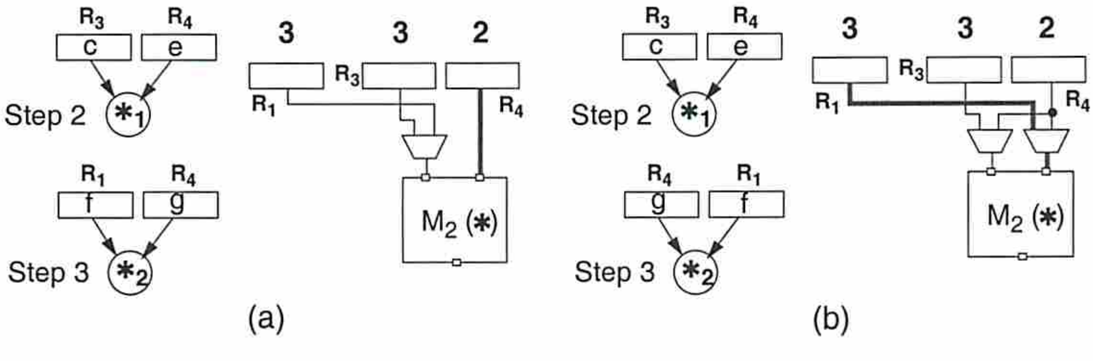


Figure 5.9: Effect of connectivity assignment on BIST registers

Consider the multiplier module M_2 from the example in the previous section. Fig. 5.9(a) shows the multiplier and its input registers and also the complete data path with BIST registers. The sharing degrees of the registers is shown on top of the registers: $SD_{reg}(R_1) = 3$, $SD_{reg}(R_3) = 3$, and $SD_{reg}(R_4) = 2$. Fig. 5.9(a) shows the input connectivity of R_1, R_3 and R_4 to the input ports of M_2. In this data path, R_4 is the only register connected to the right input port of M_2 and it has a lower sharing degree compared to the other two. Fig. 5.9(b) shows an alternate input connectivity assignment. Using the commutativity of the multiplier for the second instance of the multiplication in control step 3, R_1 and R_4 can be flipped between the left and right input ports. Now each input port is connected to a register with high sharing degree - the left input port has R_3 and the right input port has R_1. Data paths II and III in Fig. 5.2 correspond to Fig. 5.9(a) and Fig. 5.9(b), respectively. It can be seen that in Data Path III, R_1 can be shared as a TPG between the two modules resulting in further reductions in BIST area by saving R_4 as a TPG.

To determine an interconnect assignment that increases the possibility of registers with higher sharing degrees connected to both input ports, we use the following two parameters for each module M_k. The **maximum** sharing degree of the registers connected to the left input port of a module M_k, denoted by $MaxSD^L$, is the maximum of the sharing degrees of registers connected to the left input port of M_k. The **average** sharing degree of the left input port of a moduleM_k, denoted by $AvgSD^L$, is the average of the sharing degrees of registers connected to the left input port of M_k. Similarly for the right input port. For each module M_k an initial interconnect assignment is first determined for minimizing the number of multiplexers at its input ports [66]. Based on the connectivity, each instance j of a module M_k has a left input register and a right input register. For each instance j, new values of $MaxSD^L$ and $MaxSD^R$ are computed assuming that the input registers are flipped between the left and right input ports. The instance for which a flip results in a maximum value of $(MaxSD^L + MaxSD^R)$ is selected. In the case of more than one instance with the same maximum, the one that results in the lowest difference between $AvgSD^L$ and $AvgSD^R$ is selected. The interconnect assignment is constructed by flipping the pair of input registers corresponding to the selected instance and keeping the connectivity of the remaining instances unchanged.

Table 5.2: Flipping choices for connectivity assignment

Module	L port		R port		$MaxSD$		$MaxSD$	$AvgSD$		$AvgSD$
instance	Reg	SD	Reg	SD	L	R	$L+R$	L	R	$\|L-R\|$
1	A	1	B	4	4	5	9	3	3	0
2	C	2	D	4	4	5	9	2.75	3.25	0.5
3	E	3	F	5	5	4	9	2.75	3.25	0.5
4	G	3	H	2	3	5	8	2	4	2

Example 4 *Consider a module with registers A, C, E and G connected to the left input port and registers B, D, F and H connected to the right input port based on an initial interconnect assignment. Table 5.2 shows the sharing degrees of each register. The last six columns of the table refer to maximum and average sharing degree values assuming that the registers corresponding to that particular instance are flipped. Column 8 is the sum of $MaxSD^L$ and $MaxSD^R$, and column 11 is the difference between $AvgSD^L$ and $AvgSD^R$. It can be seen that flipping in the case of instances 1, 2 and 3 results in a higher sum of maximum sharing degrees, namely 9, than instance 4. Among them instance 1 has the least difference between the average sharing degrees of the left and right input ports and hence is chosen for flipping.*

5.6 Experimental Results

The assignment techniques proposed in this chapter were integrated into the Stanford CRC synthesis-for-test tool, TOPS [51]. To demonstrate the advantage of the proposed technique in synthesizing data paths with low BIST area overhead, experiments were conducted on some scheduled DFGs. For each DFG, a module assignment using a greedy approach was constructed. Then the scheduled DFGs were synthesized using two different register and interconnect assignment approaches. Approach I uses traditional assignment algorithms with the objective of minimizing registers and multiplexers. Approach II uses the assignment algorithms proposed in this chapter, which in addition to minimizing registers and interconnect, also minimizes BIST area overhead. The BIST registers required to test data paths

Table 5.3: Hardware characteristics of data paths

DFG	Schedule	Module Assignment	Approach I		Approach II	
			# Reg	# Mux Inputs	# Reg	# Mux Inputs
ex1	S_1	1+, 1*	4	10	4	13
	S_2	1+, 1*	4	10	4	13
Diffeqn	S_1	1+, 4*, 1-	7	26	7	30
	S_2	1+, 2*, 1-	5	30	5	30
Tseng	S_1	2+, 1*, 1-, 1&, 1|	4	17	4	14
	S_2	2+, 1*, 1-, 1&, 1|	4	17	4	14
AR_filter	S_1	4+, 8*	16	48	16	84
	S_2	2+, 4*	8	66	8	77
FIR_filter	S_1	4+, 4*	8	31	8	30
EW_filter	S_1	5+, 3*	10	50	10	72

synthesized using both these approaches were determined using an ILP formulation that minimizes BIST area overhead described in an earlier chapter. The BIST area overhead and the total area of the designs were then compared. All data paths were synthesized using a LSI Logic standard cell library and the area is given in cell units [52].

Table 5.3 summarizes the hardware characteristics of the designs synthesized using the two approaches. The module assignment was the same for both the approaches. It can be seen from Table 5.3 that the number of registers in Approach II is the same as that in Approach I. Approach I did not take into account any BIST considerations and tried to minimize the number of registers in the design. Our proposed approach, while taking into account BIST considerations, synthesized a design that is area competitive with that of Approach I in terms of number of functional registers. It can be seen that the number of multiplexers in the two approaches is not the same in all the cases.

Tables 5.4 show the actual number of BIST registers required to test data paths synthesized using Approach I. The reduction or increase in the number of BIST registers using Approach II over Approach I is indicated in brackets in each table

Table 5.4: BIST registers using Approach I (Approach II)

DFG	BIST registers			
	# CBILBO	# BILBO	# TPG	# SA
$ex1_{S_1}$	1 (-1)	0 (0)	2 (0)	1 (0)
$ex1_{S_2}$	1 (-1)	0 (0)	2 (0)	1 (0)
$Diffeqn_{S_1}$	1 (-1)	2 (+1)	2 (0)	2 (-1)
$Diffeqn_{S_2}$	0 (0)	0 (0)	3 (-1)	2 (-1)
$Tseng_{S_1}$	2 (-1)	1 (0)	0 (+1)	0 (0)
$Tseng_{S_2}$	2 (-1)	1 (0)	0 (+1)	0 (0)
$AR\_filter_{S_1}$	0 (+1)	7 (-4)	8 (+2)	1 (0)
$AR\_filter_{S_2}$	0 (+1)	8 (-5)	8 (+2)	0 (+1)
$FIR\_filter_{S_1}$	2 (-1)	4 (-2)	1 (+3)	1 (0)
$EW\_filter_{S_1}$	2 (-1)	0 (0)	5 (0)	2 (+1)

Table 5.5: BIST area overhead comparison

DFG	Approach I	Approach II	% Reduction
$ex1_{S_1}$	816	294	63.97
$ex1_{S_2}$	816	294	63.97
$Diffeqn_{S_1}$	1424	1056	25.84
$Diffeqn_{S_2}$	480	288	40.00
$Tseng_{S_1}$	1312	880	32.93
$Tseng_{S_2}$	1312	880	32.93
$AR\_filter_{S_1}$	1328	1176	11.45
$AR\_filter_{S_2}$	1408	1176	16.48
$FIR\_filter_{S_1}$	2272	1520	33.10
$EW\_filter_{S_1}$	1728	1296	25.00

Table 5.6: Comparison of register and multiplexer area

DFG	Area (Reg + Mux)		% Reduction in
	Approach I	Approach II	Area(Reg + Mux)
$ex1_{S_1}$	5232	5030	3.86
$ex1_{S_2}$	5232	5030	3.86
$Diffeqn_{S_1}$	5728	5552	3.07
$Diffeqn_{S_2}$	4256	3984	6.39
$Tseng_{S_1}$	3968	3376	14.92
$Tseng_{S_2}$	3968	3376	14.92
$AR\_filter_{S_1}$	6229	7081	-13.67
$AR\_filter_{S_2}$	6256	5987	4.23
$FIR\_filter_{S_1}$	7632	7232	5.24
$EW\_filter_{S_1}$	9456	9349	1.14

entry. For example in the case of $AR\_filter_{S_1}$, the number of BILBOs was reduced by 4 from 7 in Approach I to 3 in Approach II and the number of TPGs increased by 3 from 8 in Approach I to 10 in Approach II. The number and type of BIST registers were determined using the ILP formulation described in Chapter 3. It can be seen that fewer BIST registers are required in Approach II as compared to Approach I. Note that even if the number of cheaper resources (such as TPG and SA) increases, the number of more expensive resources (such as CBILBO) decreases, which results in significantly lower BIST area overhead. Table 5.5 shows the actual BIST area overhead and the percentage reduction achieved using the proposed approach. The percentage reduction achieved by Approach II is as high as 60%.

Since we assume the same schedule and module assignment for the two approaches, the functional module area is the same in both the cases. The trade-off in minimizing BIST area overhead is the functional area required for registers and interconnect (multiplexers). In Table 5.6, the register and multiplexer area is compared for the two approaches. Note that the register area is the area *after* BIST modification in both the cases. It can be observed that the area reduction is small for FIR_filter and EW_filter. These benchmarks do not provide much flexibility in the assignment phase for improvement in BIST area overhead. The savings that are achieved in BIST area overhead indicated in Table 5.5 are significant for these two benchmarks. However they get offset by the increase in multiplexer complexity. The other domains of high-level synthesis are more suited for BIST area overhead optimization of these benchmarks.

5.7 Summary

In this chapter we have presented a register and interconnect assignment approach to synthesize data paths with low BIST area overhead. The proposed assignment algorithms synthesize data paths in which the sharing of BIST registers between functional modules is maximized and the number of CBILBOs essential to test a data path is minimized. We have shown that there is sufficient flexibility during register assignment to explore solutions that are competitive in terms of functional area significantly better in terms of BIST area overhead. The proposed register assignment approach synthesizes data path with the same number of modules and registers but requiring lesser number of BIST registers. A small amount of multiplexer complexity is added to achieve that. However, the increase in multiplexer complexity is more than compensated for by reduction in BIST area overhead, resulting in lower area for a data path. Experimental results on examples demonstrate the ability of our algorithms to achieve a reduction of up to 60% in BIST area overhead of synthesized data paths.

Chapter 6

Scheduling for Reducing BIST Resources

In the previous chapter we described how register and interconnect assignment can be performed to minimize BIST resources. The assumption there was a schedule and module assignment was available. In this chapter we discuss how scheduling and module assignment can be performed to minimize BIST resources.

6.1 Introduction

Among the various types of data path components, registers are the BIST resources and interconnect, the test paths from and to the BIST resources. Modules are the components *under test*. Hence the properties of high-level synthesis tasks used for optimization of BIST area overhead are different for different data path components. Register and interconnect assignment affect BIST resources *directly* in that the objective of these tasks synthesizes the BIST resources. Module assignment, on the other hand, affects BIST resources *indirectly* by synthesizing a configuration of modules from the numerous possibilities that will require few BIST resources. From Chapter 4 we know that a module assignment that has a maximal independent operation set with very low input and output storage concurrency would require very few BIST resources. Module assignment is *tightly coupled* with scheduling. Scheduling assigns a temporal sequence on execution of operations in a DFG. The temporal sequence determines which operations can be assigned to the same module and which cannot be assigned to the same module. In the past, it has been shown that considering scheduling and module binding simultaneously results in a more efficient exploration of the design space [70], [71]. This, coupled with the fact that module assignment is inherently different than register and interconnect assignment for BIST resource minimization, makes a combined scheduling and module assignment approach very appropriate.

6.2 Scheduling and BIST Resources

In this section we will discuss how scheduling has an effect on BIST resources of a data path. Scheduling affects BIST resources in two ways. 1) It determines lifetimes of variables and hence affects the register assignment solution space. 2) It determines temporal sequence of operations and hence affects the module assignment solution space. 1) and 2) correspond to the concepts of *storage concurrency* and *maximal independent operation sets*, respectively, that were introduced in Chapter 4 and were shown to influence BIST resources. Formal definitions of a schedule, variable lifetimes and module assignment are given in Chapter 4 (Definitions 3, 5 and 8).

Definition 26 *The* **as-soon-as-possible** *value $ASAP_i$ of an operation o_i is the earliest control step in which o_i can be scheduled. The* **as-late-as-possible** *value $ALAP_i$ of an operation o_i is the latest control step in which o_i can be scheduled. $ASAP_i = \min_{\forall S} S(o_i)$ and $ALAP_i = \max_{\forall S} S(o_i)$ where the minimum and the maximum is over all schedules $S : V \rightarrow \{1, 2, ..., L_{min}\}$.*

Definition 27 *The* **mobility** *$\mathcal{M}(o_i)$ of an operation o_i is the interval $[ASAP_i, ALAP_i]$, and the* **slack** *of an operation o_i is $slack(o_i) = \mid \mathcal{M}(o_i) \mid -1$ or $(ALAP_i - ASAP_i)$.*

6.2.1 Effect of Scheduling on Variable Lifetimes

Two variables can be assigned to the same register only if their lifetimes do not overlap (Definition 6). Fig. 6.1 shows a portion of a DFG scheduled in 4 control steps. Assume that all the shaded nodes are of a different type (i.e. not addition) and the slack of all the shaded nodes is 0 because of constraints not indicated in the figure. Since they are of a different type they do not interfere with the module assignment of the addition operations. Since the slack of these operations is 0, they have to be scheduled in the control steps as shown in the two schedules. The mobilities of the addition operations are $\mathcal{M}(+_1) = [1,2]$, $\mathcal{M}(+_2) = [2,3]$ and $\mathcal{M}(+_3) = [2,3]$. Also assume that 2 adder modules M_1 and M_2 are available to which the addition operations can be assigned. Fig 6.1(a) shows one valid schedule. Given this schedule, two module assignments are possible, $\Pi_M^1 : M_1 = \{+_1\}; M_2 = \{+_2, +_3\}$ and $\Pi_M^2 : M_1 = \{+_3\}; M_2 = \{+_1, +_2\}$.

For a module, any register to which an input variable of one of its operations is assigned can provide $\mathcal{G}$ functionality to that module. Similarly, any register to which the output variable of one of its operations is assigned can provide $\mathcal{C}$ functionality. To reduce the number of registers required to supply test patterns to M_1 and M_2, we need to assign the input variables of operations assigned to the two modules to as few registers as possible. Consider all possible register assignments of this scheduled DFG.

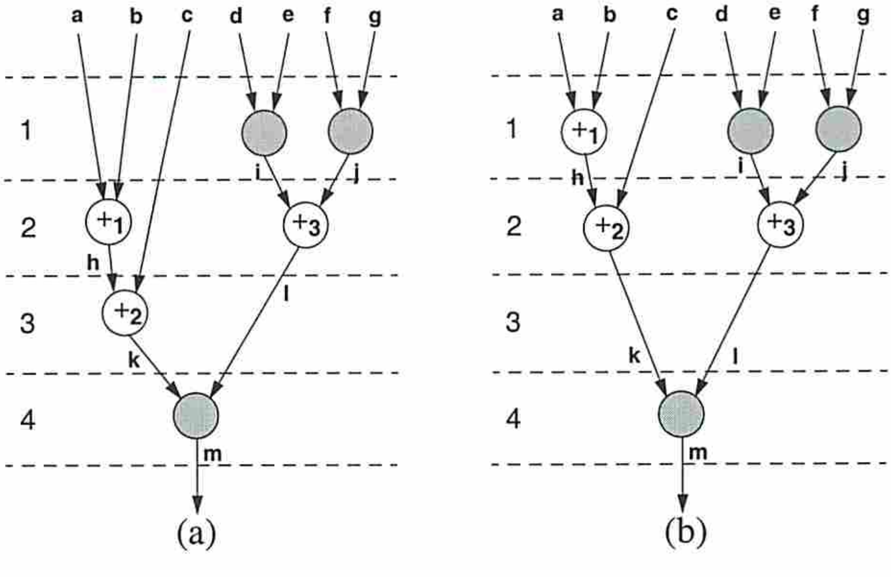


Figure 6.1: Scheduling and variable lifetimes: (a) Schedule I (b) Schedule II

For Π_M^1, a register assignment that results in minimum BIST area overhead includes registers $R_1 = \{c, ...\}, R_2 = \{b, ...\}$ and $R_3 = \{a, h, k, ...\}$. R_1 and R_2

are TPGs and R_3 is a CBILBO. Note that input variables of $+_1$, namely a and b, can share a register with only one input variable of an operation belonging to M_2, namely h. The variable h is the only output variable for module M_1 and hence the register to which it is assigned has to be a $\mathcal{C}$-resource as well. For Π_M^2, a register assignment that results in minimum BIST area overhead includes registers $R_1 = \{c, ...\}, R_2 = \{i, ...\}, R_3 = \{l, ...\}$ and $R_4 = \{h, j, ...\}$. R_1 and R_2 are TPGs, R_3 is a SA and R_4 is a CBILBO. The output variable of M_1, namely l, cannot share a register with either variable h or k which are the output variables for M_2. Hence two registers are required for $\mathcal{C}$ functionality. Also at least three registers are required for $\mathcal{G}$ functionality.

Now consider an alternative schedule as shown in Fig. 6.1(b). Given this schedule, two module assignments are possible, $\Pi_M^3 : M_1 = \{+_2\}; M_2 = \{+_1, +_3\}$ and $\Pi_M^4 : M_1 = \{+_3\}; M_2 = \{+_1, +_2\}$. For Π_M^3, a register assignment that results in minimum BIST area overhead includes registers $R_1 = \{b, ...\}, R_2 = \{c, ...\}$ and $R_3 = \{a, h, k, ...\}$ where R_1 and R_2 are TPGs and R_3 is a CBILBO. The optimum BIST solution for this case has the same cost as module assignment Π_M^1 in the case of schedule I. Module assignment Π_M^4 in the case of schedule II is the same as Π_M^2 of schedule I. However, the optimum BIST solution in this new case is different. Since input variables of M_1, namely i and j, can share registers with input variables a and b of M_2, only two registers with $\mathcal{G}$ functionality are required. In addition, output variable l of M_1 can share a register with h which is an output variable of M_2. Hence a register assignment which includes registers $R_1 = \{a, i, ...\}, R_2 = \{b, j, ...\}$ and $R_3 = \{h, l, ...\}$, where R_1 and R_2 are TPGs and R_3 is a SA, gives the optimum BIST solution.

Schedules I and II gave different optimum BIST solutions for the same module assignment $\Pi_M^2 (\equiv \Pi_M^4)$. Schedule II changed the lifetimes of the input and output variables of M_1 and M_2 such that the variables were able to share registers. These registers can then be shared as BIST resources between the modules. A quantitative measure of overlap of lifetimes of input and output variables of operations can be used to compare schedules. Let $E_{o_i} \subset E$ denote the set of edges in a DFG incident on o_i (operands and result of o_i).

Definition 28 *The* **disassociation** *between two operations o_i and o_j in a scheduled DFG is*

$$d(o_i, o_j) = \frac{\mid \{(x,y) \mid (x,y) \in E_{o_i} \times E_{o_j} \;\; s.t.\; x,y \; have\; overlapping\; lifetimes\} \mid}{\mid E_{o_i} \times E_{o_j} \mid}$$

Definition 29 *The* **S_degree of disassociation** *of a schedule S is*

$$D(S) = \frac{\sum_{\forall (o_i,o_j), i \neq j} d(o_i, o_j)}{\mid \{(o_i, o_j) \mid i \neq j\} \mid}$$

Definition 30 *The* **S_Π_degree of disassociation** *of a schedule S and a module assignment Π_M is*

$$D(S, \Pi_M) = \frac{\sum_{\forall (o_i,o_j), \Pi_M(o_i) \neq \Pi_M(o_j)} d(o_i, o_j)}{\mid \{(o_i, o_j) \mid \Pi_M(o_i) \neq \Pi_M(o_j)\} \mid}$$

The lower the degree of disassociation, the higher the possibility of sharing of BIST resources between different modules and the lower the BIST area overhead. Table 6.1 shows the module assignments, BIST area overheads assuming 16-bit registers and the **S_degree of disassociation** for Schedules I and II and the **S_Π_degree of disassociation** for module assignments Π_M^1, Π_M^2, Π_M^3 and Π_M^4.

6.2.2 Effect of Scheduling on Module Assignment

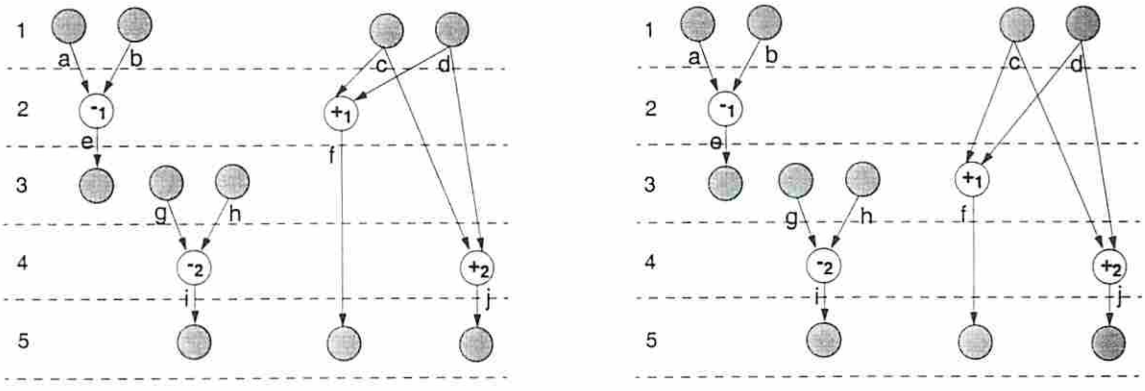
Scheduling determines which operations can be assigned to which modules from a set of available modules. Operations that are concurrent (scheduled in the same control step) have to be assigned to different modules. Also an operation can be assigned to a module only if the module can perform that *type* of operation.

Consider a portion of a scheduled DFG shown in Fig. 6.2(a). Assume that all the shaded operations have zero slack. The mobilities of the addition and the subtraction

Table 6.1: Effect of scheduling on variable lifetimes

Schedule		Module		Optimum	BIST
S	$D(S)$	Assignment	$D(S, \Pi_M)$	BIST	Overhead
		Π_M		Solution	(cell units)
I	0.48	Π_M^1	0.44	2T, 1C	740
		Π_M^2	0.56	2T, 1S, 1C	846
II	0.29	Π_M^3	0.44	2T, 1C	740
		$\Pi_M^4 (\equiv \Pi_M^2)$	0.17	2T, 1S	318

operations are $\mathcal{M}(-_1) = [2,2], \mathcal{M}(-_2) = [4,4], \mathcal{M}(+_1) = [2,4]$ and $\mathcal{M}(+_2) = [2,4]$. Furthermore, assume that two modules are available, M_1 and M_2, such that M_1 can perform only subtraction and M_2 can perform both addition and subtraction. For schedule I (shown in Fig. 6.2(a)) only one module assignment Π_M^1 is possible. Both $+_1$ and $+_2$ have to be assigned to M_2 since it is the only module that can perform addition.



(a)

(b)

Figure 6.2: Scheduling and module assignment: (a) Schedule I (b) Schedule II

Since $-_1$ is concurrent with $+_1$ and $-_2$ is concurrent with $+_2$, neither of the subtraction operations can be assigned to M_2 and hence they are both assigned to M_1. Of all possible register assignments, the one that results in an optimum BIST solution includes $R_1 = \{a, ...\}, R_2 = \{b, ...\}, R_3 = \{c, ...\}, R_4 = \{d, ...\}$ and $R_5 = \{e, j, ...\}$. R_1, R_2, R_3 and R_4 are TPGs and R_5 is a SA. The input variables of $-_1$ and $-_2$ cannot share the same registers with the input variables of $+_1$ and

Table 6.2: Effect of scheduling on module assignment

Schedule S	$D(S)$	Module Assignment Π_M	$D(S,\Pi_M)$	Optimum BIST Solution	BIST Overhead (cell units)
I	0.50	Π_M^1	0.64	4T, 1S	530
II	0.49	Π_M^2	0.44	2T, 1S	318

$+_2$ and hence 4 TPGs are required. Note that the input variables of $-_1$ and $-_2$ can share registers since they have disjoint lifetimes. However the schedule and the resulting module assignment prevents the two subtractions to be assigned to different modules, thus precluding the possibility of the shared register acting as test resources for different modules.

Consider an alternative schedule of the same DFG shown in Fig. 6.2(b). By scheduling $+_1$ in control step 3, a different module assignment Π_M^2 is possible. $+_1$ and $+_2$ still have to be assigned to M_2 since this is the only module that performs addition. However, since $-_1$ is not concurrent with any of the additions and since M_2 performs subtraction also, $-_1$ can be assigned to M_2. Now a register assignment that includes registers $R_1 = \{a, g, ...\}, R_2 = \{b, h, ...\}$ and $R_3 = \{e, i, ...\}$ is possible which corresponds to an optimum BIST solution of 2 TPGs (R_1 and R_2) and 1 SA (R_3).

Schedules I and II have about the same **S_degree of disassociation**. However Schedule II allows a module assignment which has a lower **S_Π_degree of disassociation**. Table 6.2 summarizes how a schedule effects BIST area by influencing only module assignment and not influencing the lifetimes of variables in DFG.

6.3 Scheduling Procedure

In the previous section we have shown how scheduling can affect the BIST solution of a data path. Given a schedule, a module assignment that satisfies the schedule and a constraint on resources, different register assignments are possible. Using register assignment techniques such as the ones presented in Chapter 5 and in [49] and [44], a data path can be synthesized that requires a low number of BIST resources. However, the optimum BIST area overhead that can be achieved is constrained by the chosen schedule and module assignment. In general, different schedules and module assignments that have a desirable latency and functional area can differ significantly in their BIST resource requirement. For a given schedule and module assignment, establishing what can be achieved in terms of register and interconnect assignment to minimize BIST resources is a key question in incorporating BIST area overhead optimization techniques in the scheduling phase of high-level synthesis. Theory on estimating lower bounds on BIST resources from scheduled DFGs has been developed in Chapter 4.

We propose a scheduling approach in which the module assignment is done simultaneously with scheduling. It has been shown that considering scheduling and module binding simultaneously results in a more efficient exploration of the design space [70], [71]. Our approach has two phases: 1) adding temporal testability constraints between selected operations, and 2) performing scheduling and module assignment of each operation on the DFG with the modified constraints. In Phase 1, pairs of operations o_i and o_j are selected such that it is beneficial (in terms of BIST area overhead) for the operations to execute in different control steps and be assigned to different modules. The scheduling of such operations is constrained to occur in different control steps by adding a temporal relationship (an edge in the DFG) between the operations. Phase 1 can be viewed as *coarse* scheduling, where only concurrency and sequentiality of operations is influenced but the exact control step is not assigned. Phase 1 requires an analysis of the DFG which is done in a preprocessing step. In Phase 2, *detailed* scheduling is performed where each operation is assigned to a control step and to a module taking into account the constraints added in Phase 1.

6.3.1 Phase 1: Adding Temporal Constraints

A schedule induces a temporal sequence on operations. Operations that are executed sequentially can share functional modules. Sequentiality of operations is thus beneficial to minimizing functional resources and desirable when the objective is to minimize functional area. Also a pair of sequential operations has low **disassociation**, i.e. the operations have a high possibility of sharing input and output variables. Hence if sequential operations are *distributed* across modules (as opposed to sharing modules) then BIST resources and testability area overhead can be minimized. We define the concepts of *strictly sequential*, *strictly concurrent* and *weakly sequential* (or *weakly concurrent*) pairs of operations for an unscheduled DFG.

Definition 31 *Operations o_i and o_j are*

1. **strictly sequential** *if* $\forall S, S(o_i) \neq S(o_j)$,
2. **strictly concurrent** *if* $\forall S, S(o_i) = S(o_j)$,
3. **weakly sequential** *(or* weakly concurrent*) if* $\exists S_1$ *such that* $S_1(o_i) = S_1(o_j)$, *and* $\exists S_2$ *such that* $S_2(o_i) \neq S_2(o_j)$,

where S, S_1 *and* S_2 *are schedules with latency* L_{min}.

Strict sequentiality implies that the two operations have to be scheduled in different control steps for a valid schedule of latency L_{min} and strict concurrency implies that the two operations have to be scheduled in the same control step for a valid minimum latency schedule. Weak sequentiality, on the other hand, implies that it is possible but not necessary to schedule the two operations in the same control step for a minimum latency schedule. In the DFG shown in Fig. 6.3(a), operations $*_1$ and $*_2$ are strictly sequential, operations $*_1$ and $*_2$ are strictly concurrent, and operations $*_5$ and $+_1$ are weakly sequential.

The temporal (order of execution) and spatial (assignment to hardware units) relationships of a pair of operations determines the nature of the data path as shown in the Table 6.3. Two operations executing in the same control step cannot be assigned to the same module. The other three cases are functionally possible. If two operations are strictly concurrent, they fall in the first column (same control step) and test optimization is not possible. Strict sequentiality falls in the second column (different control steps). Scheduling has no control over strict concurrency and strict

Table 6.3: Temporal-spatial relation of operations

	Control steps	
Modules	Same	Different
Same	-	Does not contribute to BIST resource reduction
Different	High storage concurrency of input/output variables (cannot share BIST resources)	Low storage concurrency of input/output variables **(can share BIST resources)**

sequentiality if minimum latency schedules are desired. Strict sequentiality can be leveraged to reduce BIST resources by deciding an appropriate control step and assigning the operations to different modules. This *fine* (i.e. detailed) scheduling is addressed in Phase 2. The case of interest in Phase 1 is weak sequentiality which spans the whole table. Weak sequentiality can be exploited to add temporal constraints such that the operations fall in the second column. Phase 2 can then take advantage of the constraints to push the operations in the direction of the lower right hand box in Table 6.3. A temporal constraint between a pair of operations that constrains the operations to occur in different control steps is formally defined below.

Definition 32 *A* **temporal constraint** *on a pair of operations o_i and o_j is defined as a modification of $G = (V, E)$ to $G' = (V, E')$ where $E' = E \cup \{o_i, o_j\}$ or $E' = E \cup \{o_j, o_i\}$. However, there is no variable associated with the added edge. The constraint is denoted as $(o_i \nleftrightarrow o_j)$.*

Theorem 10 *If operations o_i and o_j are weakly sequential operations in an unscheduled DFG, $G = (V, E)$, then adding a temporal constraint $(o_i \nleftrightarrow o_j)$ still guarantees a schedule of G with latency L_{min}.*

Proof: The proof follows directly from the definition of weakly sequential operations (Definition 31). Since there exists a minimum latency schedule S_2 such that $S_2(o_i) \neq S_2(o_j)$, an edge can be added from o_i to o_j (if $S_2(o_i) < S_2(o_j)$) or from o_j to o_i (if $S_2(o_j) < S_2(o_i)$) and schedule S_2 would be still valid. □

Theorem 10 allows us to constrain the scheduling of a DFG in a manner that is beneficial for BIST overhead without compromising the latency of the schedule. However, adding a temporal constraint can have an adverse effect on the number of modules required. In Phase 1 of our approach, pairs of weakly sequential operations are identified and testability temporal constraints are added to selected pairs. The following theorem is used to identify pairs of weakly sequential operations.

Theorem 11 *Operations o_i and o_j are* **weakly sequential iff** *all of the following conditions are true.*
1) there is no path from o_i to o_j or from o_j to o_i
2) $[ASAP_i, ALAP_i] \cap [ASAP_j, ALAP_j] \neq \phi$
3) at least one of $slack(o_i)$ and $slack(o_j)$ is not equal to 0.

Proof: If ($\Leftarrow$):
Since the operations are weakly sequential, from Definition 31, $\exists$ S_1 such that $S_1(o_i) = S_1(o_j)$. Therefore, $S_1(o_i)(= S_1(o_j)) \in [ASAP_i, ALAP_i]$ and $S_1(o_j)(= S_1(o_i)) \in [ASAP_j, ALAP_j]$ which implies Condition 2. Also since the synthesis model does not allow chaining of operations, scheduling operations o_i and o_j in the same control step implies Condition 1.

$\exists S_1$ such that $S_1(o_i) = S_1(o_j)$, and $\exists S_2$ such that $S_2(o_i) \neq S_2(o_j)$. A minimum of two *distinct* values of control steps are required to satisfy these S_1 and S_2. Since $S_1(o_i)$, $S_2(o_i) \in [ASAP_i,\ ALAP_i]$ and $S_1(o_j)$, $S_2(o_j) \in [ASAP_j,\ ALAP_j]$, at least one of $[ASAP_i,\ ALAP_i]$ and $[ASAP_j,\ ALAP_j]$ should have two control steps which implies that at least one of the operations has non-zero slack (Condition 3).

Only if ($\Rightarrow$):
Condition 2 implies that $\exists$ a control step c such that $c \in [ASAP_i, ALAP_i]$ and $c \in [ASAP_j, ALAP_j]$. According to Condition 1, there is no path from one operation to another and hence they can be scheduled as per schedule S_1 where $S_1(o_i) = S_1(o_j) = c$. (1)

As per Condition 3, let us assume that $slack(o_i) > 0$. So even if o_j is scheduled in control step $c \in [ASAP_i, ALAP_i]$ and $c \in [ASAP_j, ALAP_j]$, o_i can be scheduled in control step $c' \neq c$. It is possible to schedule o_i in c' after scheduling o_j in control step c, because of Condition 1. Hence a schedule S_2 exists such that $c' = S_2(o_i) \neq S_2(o_j) = c$. (2)

From (1) and (2), o_i and o_j are weakly sequential. □

The minimum latency of the DFG is not affected by adding temporal constraints between weakly sequential operations according to Theorem 10. The number of registers required for the DFG is also not affected because it has been shown to be insensitive to different schedules [72]. But two other effects need to be considered: 1) effect on the mobility of other operations, and 2) effect on the number of functional modules required. For a weakly sequential pair of operations, the effect of adding a temporal constraint on the mobility of other operations is quantified as the total change in *slack* of operations, Δ_{slack}. All weakly sequential operation pairs are considered in an increasing order of the value of Δ_{slack}. If a temporal constraint between a pair of operations violates the constraint on the number of functional modules, the pair is dropped from consideration. Of the remaining candidates, a pair is selected such that it is most beneficial for BIST resources. Note from Table 6.3 that operations in the lower right hand box of the table are most beneficial in this regard. Hence, candidate pairs that have a high probability of being assigned to different modules are selected. If operations in a candidate pair are of different *types*, this probability is 1. A temporal constraint is added to a pair of operations selected in this manner. The ALAP and ASAP values of all operations are updated after addition of a temporal constraint.

Example 5 *The addition of temporal constraints to the* Diffeqn *benchmark [61] is demonstrated in Fig. 6.3. The original $[ASAP_i, ALAP_i]$ values of each operation o_i are indicated in Fig. 6.3(a). Consider the pair or weakly sequential operations $(*_5, +_1)$. The addition of an temporal constraint between these two operations changes the mobility of operation $+_1$. (The mobilities that change are highlighted in the figure.) In Fig. 6.3(b), the total change in* slack *of operations, Δ_{slack}, is 2. However, if a temporal constraint is added to the operation pair $(*_5, +_2)$, as shown in Fig. 6.3(c), $\Delta_{slack} = 1$ and hence this constraint is preferred. Note that a temporal constraint has a preferred* precedence *relationship. The same temporal constraint as in Fig. 6.3(c) with the* opposite *precedence relationship is shown in Fig. 6.3(d). In* *this case,* $\Delta_{slack} = \Delta_{slack(*_5)} + \Delta_{slack(*_6)} + \Delta_{slack(+_2)} = 1 + 2 + 2 = 5$. *This constrains the schedule severely and hence the temporal constraint with the precedence order in Fig. 6.3(c) is preferred.*

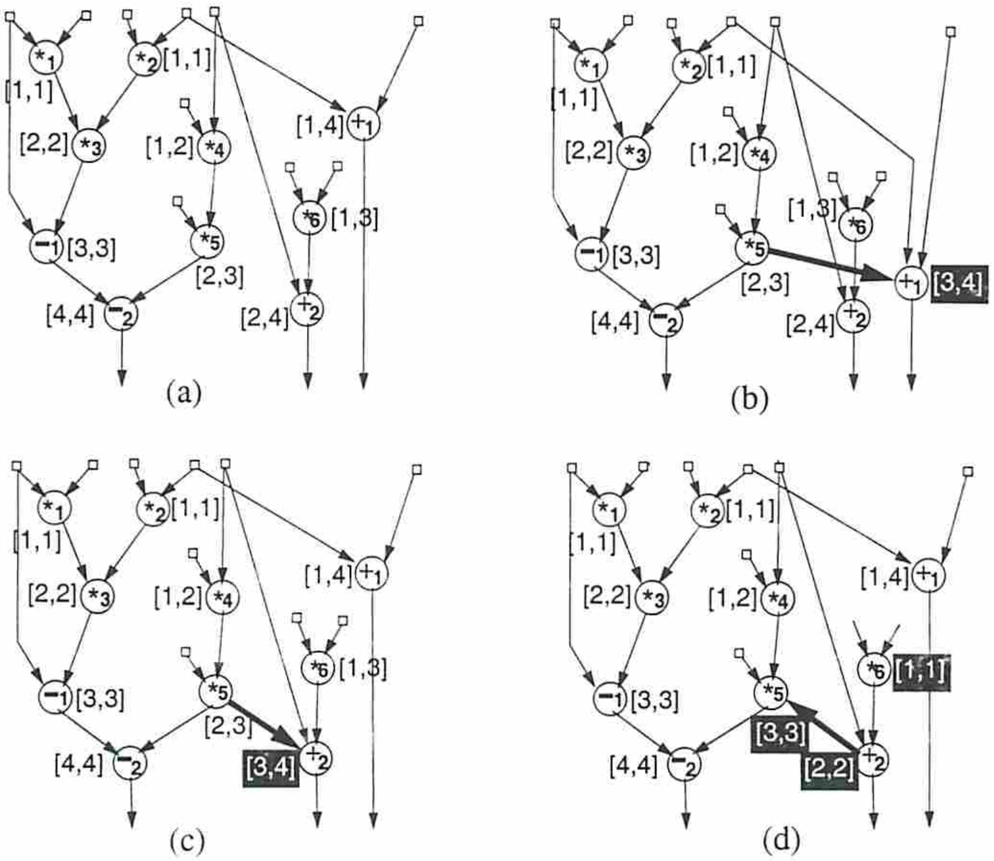


Figure 6.3: Phase 1 - Adding temporal constraints

6.3.2 Phase 2: Detailed Scheduling and Module Assignment

The proposed scheduling procedure is based on *list scheduling* techniques [16]. List scheduling techniques are widely used in high-level synthesis because of low computation complexity and near-optimum solutions. We use *slack* of an operation as the priority function in picking an operation to be scheduled since we desire minimum latency schedules. In addition to assigning operations one by one to a control step, the procedure simultaneously assigns them to the available modules. The types of modules and the number of instances of each type available, are known to the procedure.

```
INPUT: $DFG$, $L_{min}$ and $M = \{M_1, M_2, ..., M_m\}$

Step 1: Pick an unscheduled $o_i$ with least *slack*
Step 2: Find all possible tuples $T_j =< CStep_j, Mod\_index_j >$ s.t. $o_i$
        can be assigned control step $CStep_j$ and module $M_{Mod\_index_j}$
Step 3: **For each** tuple $T_j$ s.t. $M_{Mod\_index_j}$ is not empty
        Step 3.1: Calculate $Cost(T_j) = \Delta_{BIST} - \Delta_{MUX}$
Step 4: **If** there exists $T_j$ s.t. $Cost(T_j)$ is positive
        Step 4.1: **then** select $T_j$ with highest cost
        Step 4.2: **else** select $T_j$ s.t. $M_{Mod\_index_j}$ is empty
Step 5: Assign $o_i$ to selected $T_j$
Step 6: Go to Step 1
```

Procedure for Phase 2 - *Schedule_and_Assign()*

The procedure is iterative in nature and at every step from the operations that have not yet been scheduled and assigned, an operation o_i with the smallest *slack* is selected. The set of all control step and module assignment tuples ($T_j = < CStep_j, Mod\_index_j >$) is then determined such that o_i can be scheduled in control step $CStep_j$ and assigned to module $M_{Mod\_index_j}$ from the set of available modules $M = \{M_1, M_2, ..., M_m\}$. Note that while considering such tuples for o_i some other operations are already scheduled and assigned to modules. A cost function, $Cost(T_j)$, is computed for each tuple T_j to determine the control step and the module to which an operation should be assigned. $Cost(T_j)$ has two components. The primary component of the cost is Δ_{BIST}, the decrease in BIST area overhead corresponding to the assignment as defined by the tuple. A decrease in the number of BIST resources can adversely affect the multiplexing complexity. Hence the second component of $Cost(T_j)$ is Δ_{MUX}, the increase in multiplexer area corresponding to the assignment defined by T_j. A tuple T_j is chosen chosen such that the decrease in BIST area overhead after compensating for an increase in multiplexer area ($Cost(T_j) = \Delta_{BIST} - \Delta_{MUX}$), is maximum. The detailed description of the components of the cost function is given next.

6.3.2.1 Estimation of BIST Area Overhead

The estimation of BIST area overhead is first discussed for a complete schedule and module assignment. The extension to partial schedules and module assignments and use as Δ_{BIST} is straightforward.

Let $\mathcal{L} = \{V^1_{I_{max}}, V^2_{I_{max}}, \ldots, V^l_{I_{max}}\}$ denote the list of all maximal independent operation sets. Since $V^i_{I_{max}}$ contains one operation from each module, the registers to which variables of $IVar(V^i_{I_{max}})$ are assigned correspond to a BIST solution, where these registers provide $\mathcal{G}$ functionality for all modules in the data path. Similarly, the registers to which the variables of $OVar(V^j_{I_{max}})$ have been assigned correspond to one BIST solution where these registers provide $\mathcal{C}$ functionality for all modules. The actual *type* of the test register (i.e. TPG, SA, BILBO or CBILBO) is determined by how the variables in $IVar(V^i_{I_{max}}) \cup OVar(V^j_{I_{max}})$ $(1 \leq i, j \leq l)$ are distributed across the BIST registers. Let $IVar(V^i_{I_{max}}) \cup OVar(V^j_{I_{max}})$ be denoted by $TestVar^{i,j}$, the set of *test variables*.

Remark 1 *Given a scheduled DFG and a module assignment, for any register assignment, the registers to which variables in $TestVar^{i,j}$ have been assigned define a minimal intrusion BIST solution for the synthesized data path. In this BIST solution a register R is a*

1) TPG, if $\forall v \in TestVar^{i,j}$ that are assigned to it, $v \in IVar(V^i_{I_{max}})$, $v \notin OVar(V^j_{I_{max}})$

2) SA, if $\forall v \in TestVar^{i,j}$ that are assigned to it, $v \in OVar(V^j_{I_{max}})$, $v \notin IVar(V^i_{I_{max}})$

3) BILBO, if it is assigned

$x, y \in TestVar^{i,j}$ such that $x \in OVar(V^j_{I_{max}})$, $y \in IVar(V^i_{I_{max}})$, and the operation of which x is the output and the operation of which y is the input do not belong to the same module.

(Note: x and y could be the same variable)

4) CBILBO, if it is assigned

$x, y \in TestVar^{i,j}$ such that $x \in OVar(V^j_{I_{max}})$, $y \in IVar(V^i_{I_{max}})$, and the operation of which x is the output and the operation of which y is the input belongs to the same module.

(Note: x and y could be the same variable)

Example 6 *Consider the DFG in Fig. 6.4(a) with a schedule as shown and a module assignment such that $M_1 = \{*_1, *_3\}$, $M_2 = \{*_2, *_4, *_5\}$, $M_3 = \{-_1, -_2\}$ and $M_4 = \{+_1\}$. Out of all the maximal independent operation sets consider $V^1_{I_{max}} = \{*_1, *_5, -_2, +_1\}$ and $V^2_{I_{max}} = \{*_3, *_2, -_2, +_1\}$. The corresponding sets of input and output variables are $IVar(V^1_{I_{max}}) = \{b, c, j, m, n, o, e, h\}$, $OVar(V^1_{I_{max}}) = \{i, o, p, e\}$ and $IVar(V^2_{I_{max}}) = \{i, j, d, e, n, o, h\}$, $OVar(V^2_{I_{max}}) = \{l, j, p, e\}$*

Consider $TestVar^{2,1} = IVar(V^2_{I_{max}}) \cup OVar(V^1_{I_{max}})$. Fig. 6.4(b) shows a data path that has been synthesized using one register assignment of the several possible. It can be seen that registers to which the variables of $TestVar^{2,1}$ have been assigned (shown shaded in the figure) define one BIST solution for the data path. According to Remark 1, each of the shaded registers has the following functionality.

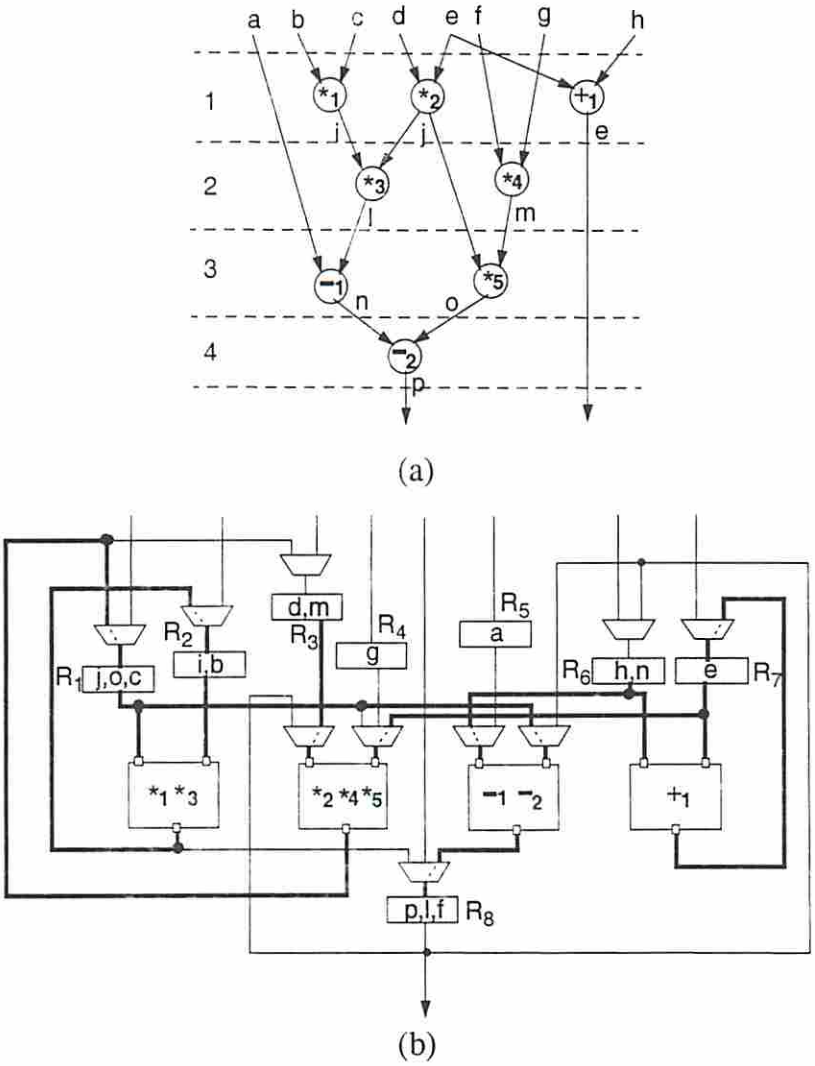
1. *$R_6 = \{h, n\}$ is TPG since $h, n \in TestVar^{2,1}$; $h, n \in IVar(V^2_{I_{max}})$; $h, n \notin OVar(V^1_{I_{max}})$. Similarly $R_3 = \{d, m\}$ is TPG.*

2. *$R_8 = \{p, l, f\}$ is SA since $p \in TestVar^{2,1}$; $p \in OVar(V^1_{I_{max}})$; $p \notin IVar(V^2_{I_{max}})$.*

3. *$R_1 = \{j, o, c\}$ is BILBO since $j, o \in TestVar^{2,1}$ such that $o \in OVar(V^1_{I_{max}})$, $j \in IVar(V^2_{I_{max}})$, and the operation of which o is the output ($*_5$) and the operation of which j is the input ($*_3$) belong to different modules (M_2 and M_1, respectively).*

4. *$R_7 = \{e\}$ is CBILBO since $e \in TestVar^{2,1}$ such that $e \in OVar(V^1_{I_{max}})$, $e \in IVar(V^2_{I_{max}})$, and the operation of which e is the output ($+_1$) and the operation of which e is the input ($+_1$) belong to the same module (M_4). Similarly $R_2 = \{i, b\}$ is CBILBO because of variable i.*

The test functionality of the registers with respect to each of the 4 modules is summarized in Table 6.4. In Fig. 6.4(b) the paths used to transport test data are shown highlighted. The functionality of the test registers derived above and the correctness of the BIST solution can be verified from Table 6.4.



(a)

(b)

Figure 6.4: BIST Solution from $TestVar^{i,j}$

For a given schedule and a module assignment, each $TestVar^{i,j} (1 \leq i, j \leq l)$ gives a minimal intrusion BIST solution as defined in Remark 1. Depending upon the register assignment chosen, the number of registers defined in 1), 2), 3) and 4)

Table 6.4: Test functions of BIST registers in Fig. 6.4(b)

Module	Test pattern generation		Test response compression
	Left input	Right input	
M_1	R_1	R_2	R_2
M_2	R_3	R_7	R_1
M_3	R_6	R_1	R_8
M_4	R_6	R_7	R_7

of Statement 1 changes, thus changing the cost of the BIST solution. Assuming complete flexibility in register assignment, the minimum number of registers of each type could be found which would give a lower bound on the cost (area overhead) of the BIST solution, $lb_{BIST}(i,j)$, corresponding to $TestVar^{i,j}$. The minimum of the lower bounds over all $TestVar^{i,j}$ ($i = 1, 2, ..., l$ and $j = 1, 2, ..., l$) would give an estimate on the optimum BIST solution, $LB_{BIST}(\mathcal{L})$, of the final data path.

Definition 33 *Let $TestVar^{i,j} = T^{i,j} \cup S^{i,j} \cup B^{i,j} \cup C^{i,j}$ where $T^{i,j}$, $S^{i,j}$, $B^{i,j}$ and $C^{i,j}$ are all disjoint and defined as*
1) $T^{i,j} = \{v \mid v \in IVar(V^i_{I_{max}}),\ v \notin OVar(V^j_{I_{max}})\}$,
2) $S^{i,j} = \{v \mid v \notin IVar(V^i_{I_{max}}),\ v \in OVar(V^j_{I_{max}})\}$,
3) $B^{i,j} = \{v \mid v \in IVar(V^i_{I_{max}}),\ v \in OVar(V^j_{I_{max}})$, v is not an operand and result of operations belonging to the same module $\}$,
4) $C^{i,j} = \{v \mid v \in IVar(V^i_{I_{max}}),\ v \in OVar(V^j_{I_{max}})$, v is an operand and result of operations belonging to the same module $\}$.

Theorem 12 *Given a scheduled DFG and a module assignment, a lower bound on the BIST area overhead of the BIST solution corresponding to $TestVar^{i,j}$ is given by*

$$\begin{aligned} lb_{BIST}(i,j) = {} & c^C \cdot SC(C^{i,j}) \ + \ c^B \cdot (SC(B^{i,j} \cup C^{i,j}) - SC(C^{i,j})) \\ & + c^T \cdot (SC(B^{i,j} \cup C^{i,j} \cup T^{i,j}) - SC(B^{i,j} \cup C^{i,j})) \\ & + c^S \cdot (SC(B^{i,j} \cup C^{i,j} \cup S^{i,j}) - SC(B^{i,j} \cup C^{i,j})), \end{aligned}$$

where c^C, c^B, c^T and c^S are costs of modification of a normal register to CBILBO, BILBO, TPG and SA, respectively, and $c^C > c^B > c^T + c^S$.

Proof: The storage concurrency of a set of variables gives the *minimum* number of registers to which those variables can be assigned. According to Definition 33, $C^{i,j}$ is the set of variables that when assigned to a register, require the register to be CBILBO for a correct BIST solution. Similarly $B^{i,j}$ is the set of variables that when assigned to a register, require the register to have *at least* the BILBO functionality. If two variables, $b \in B^{i,j}$ and $c \in C^{i,j}$, were assigned to the same register, the register would have to be CBILBO. (Note that, a CBILBO performs the functionality of a BILBO as well).

The minimum number of CBILBOs required is therefore the minimum number of registers that can store the variables in $C^{i,j}$ which is given by $SC(C^{i,j})$. Similarly the minimum number of registers that can store the BILBO variables $B^{i,j}$ is $SC(B^{i,j})$. However, if a BILBO variable shares a register with a CBILBO variable, the register would become a CBILBO. $SC(B^{i,j} \cup C^{i,j})$ gives the minimum number of registers that have to have at least the BILBO functionality. Since $c^C > c^B$, the minimum overhead of these registers would result when a minimum number of the $SC(B^{i,j} \cup C^{i,j})$ registers are CBILBOs and the rest are BILBOs. Hence the contribution of the CBILBO component is $c^C \cdot SC(C^{i,j})$ and that of the BILBO component is $c^B \cdot (SC(B^{i,j} \cup C^{i,j}) - SC(C^{i,j}))$. (1)

Similarly, a register to which an input only variable ($\in T^{i,j}$) is assigned is required to have *at least* the test pattern generation capability, i.e. the register could be TPG, BILBO or CBILBO. Some of the input only variables could share registers with the BILBO and CBILBO variables. Assuming assignment of BILBO and CBILBO variables corresponding to (1), the minimum number of TPGs would be $(SC(B^{i,j} \cup C^{i,j} \cup T^{i,j}) - SC(B^{i,j} \cup C^{i,j}))$. (2)

With an argument similar to the TPG case, the minimum number of SAs is $(SC(B^{i,j} \cup C^{i,j} \cup S^{i,j}) - SC(B^{i,j} \cup C^{i,j}))$. (3)

If variables $t \in T^{i,j}$ and $s \in S^{i,j}$ were to share a register, the register would be BILBO as per Remark 1. However, since $c^B > c^T + c^S$, this case does not correspond to the minimum cost solution. So from (1), (2) and (3), we have the theorem for the minimum cost of a BIST solution corresponding to $TestVar^{i,j}$. □

Note that the relationship between the costs for the modification of registers to test registers used in Theorem 12 is true for the cell library used in this work [52]. An alternative library could be used such that a different relationship exists between the costs. In that case, the principle behind the theorem can be easily applied to derive a modified expression for the lower bound. The scheduling and module assignment procedure, and all other concepts discussed in this chapter can be used without any change.

Theorem 13 *Given a scheduled DFG, a module assignment and* $\mathcal{L} = \{V^1_{I_{max}}, V^2_{I_{max}}, ..., V^l_{I_{max}}\}$*, a lower bound on the area overhead of an optimum minimal intrusion BIST solution is given by*

$$LB_{BIST}(\mathcal{L}) = \min_{i,j=1}^{l}(lb_{BIST}(i,j)).$$

The lower bound on the optimum BIST area overhead can also be calculated for a *partial* data path, corresponding to a partial schedule and module assignment and the associated list of maximal independent operation sets. This is used as Δ_{BIST} in every step of the scheduling procedure. Since all the operations are not scheduled, the lifetimes of all the input and output variables of scheduled operations is not known. The storage concurrencies required for computing the lower bound are determined by the worst case lifetimes of the variables. These can be easily found from the ASAP and ALAP values of the appropriate unscheduled operations.

6.3.2.2 Estimating BIST Cost, Δ_{BIST}

Consider the i^{th} iteration of procedure *Schedule_and_Assign()*, when o_i is being scheduled. Assume that up to this point in the scheduling and module assignment, out of the $M_1, M_2, ..., M_m$ modules available, k modules, M_1 through M_k have been assigned one or more operation and the other $(m - k + 1)$ modules have not been assigned any operation yet. A list of maximal independent operations $\mathcal{L}_{i-1}$ corresponding to the k modules exists after the $(i-1)^{th}$ operation was scheduled and assigned. Let $T_j = < CStep_j, Mod\_index_j >$ be a possible control step and module assignment for o_i such that $1 \leq Mod\_index_j \leq k$. If o_i is assigned to $M_{Mod\_index_j}$, a new list of maximal independent operations, $\mathcal{L}_i^j$ is created corresponding to the assignment of o_i to tuple T_j. The new list $\mathcal{L}_i^j$ contains all the maximal independent operation sets from $\mathcal{L}_{i-1}$. But in addition, it also contains maximal independent operation sets each containing o_i and one operation each from rest of the $(k-1)$ modules. Using Theorem 13, the lower bound on BIST area overhead for the partial data path resulting from assignment of o_i to tuple T_j can be computed as $LB_{BIST}(\mathcal{L}_i^j)$. The decrease in the lower bound, $\Delta_{BIST} = LB_{BIST}(\mathcal{L}_{i-1}) - LB_{BIST}(\mathcal{L}_i^j)$, is the BIST cost of the assignment to tuple T_j.

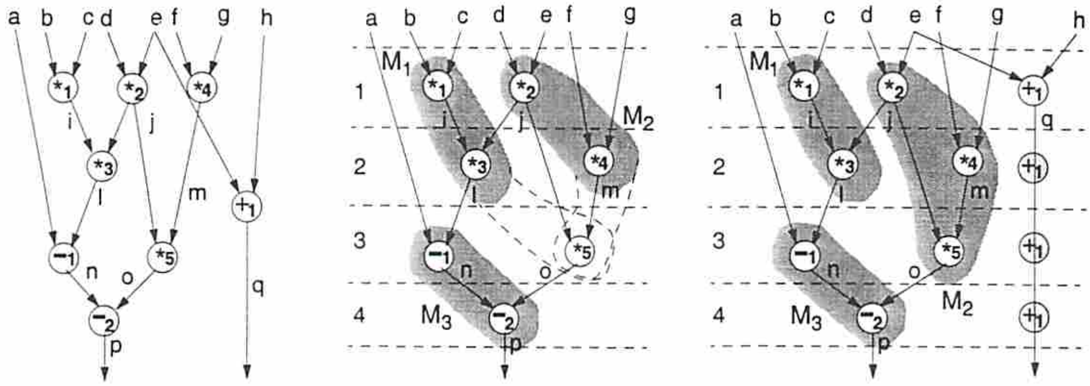
6.3.2.3 Estimating Multiplexer Cost, Δ_{MUX}

Minimizing BIST resources for minimal intrusion BIST in a data path corresponds to an increase in multiplexing area. Registers can be shared as test response compressors only if they multiplex output data from different modules. If a module has more than one input register connected to an input port, a multiplexer is required for that port. When two modules share registers as test pattern generators, the number of registers connected to an input port of each of the modules increases. Hence reduction in the number of BIST resources (due to sharing), results in an increase in multiplexers for a fixed number of modules. Estimating *total* multiplexer area at the scheduling stage of high-level synthesis is difficult because the total number of registers is not known. However, the number of test registers is known at this stage according to the theory presented in this section. The change in the number of BIST resources resulting from an assignment is also known. Hence the corresponding *increase* in multiplexer area can be estimated. The estimated increase in multiplexer cost has two components: 1) Δ_{mux}^{reg}, the increase in multiplexer cost at the input of registers (i.e. multiplexing of output variables), and 2) Δ_{mux}^{mod}, the increase in multiplexer cost at the input of modules (i.e. multiplexing of input variables).

At the i^{th} iteration of the scheduling procedure, Δ^{reg}_{mux} can be found from the change in the number of test registers that compress the responses of modules resulting from an assignment of o_i to tuple T_j. A decrease in the number of such test registers implies that the same number of output variables (one from each module) have to be multiplexed into a smaller number of registers. The corresponding increase in multiplexer area in the worst case can be calculated from the available library. For computing Δ^{mod}_{mux} corresponding to assignment of o_i to tuple T_j, only module $M_{Mod\_index_j}$ to which o_i is assigned needs to be considered. The input variables of $M_{Mod\_index_j}$ that are assigned to the shared test pattern generation registers and the registers that hold the rest of its input variables are multiplexed onto the input ports of $M_{Mod\_index_j}$. If the numbers of these registers after the $(i-1)^{th}$ step and after assignment of o_i to T_j in the i^{th} step are known, then Δ^{mod}_{mux} can be computed. The total increase in multiplexer area, Δ_{MUX}, is computed as the sum of Δ^{mod}_{mux} and Δ^{reg}_{mux}.

For every tuple T_j, the cost of the tuple assignment is calculated as $Cost(T_j) = \Delta_{BIST} - \Delta_{MUX}$. A high value of this cost refers to a decrease in total area and is hence preferred. The tuple corresponding to the highest cost is selected and operation o_i is appropriately assigned. If there is no tuple for which $Cost(T_j) > 0$, then o_i is assigned to a module that is non-empty. If such a module is not available, then a tuple T_j is selected such $Cost(T_j)$ is the highest (least negative). This assignment corresponds to a scheduling move in the direction of a worse testability solution but preserves the functional module requirement and hence keeps the functional area within acceptable bounds. The scheduling and module assignment is followed by register and interconnect assignment to synthesize the complete data path.

Example 7 *In this example estimation of BIST cost in Phase 2 is described using the unscheduled DFG in Fig. 6.5(a). The DFG requires 4 modules to implement it - 2 multipliers M_1 and M_2, one subtractor M_3 and one adder M_4. Operations $*_1, *_2, *_3, -_1$ and $-_2$ have a slack of 0 for a minimum latency schedule. Hence they get scheduled in control steps 1, 1, 2, 3, and 4, respectively. Operations $*_1$ and $*_3$ are assigned to multiplier M_1 and $*_2$ to multiplier M_2. The unscheduled operation $*_4$ has a slack of 1 and is considered next. Assigning it to control step 1 would require an additional multiplier, hence it is assigned to control step 2 and module*



(a) (b) (c)

Figure 6.5: Phase 2 - Detailed scheduling and module assignment

M_2 (since $*_3$ is already assigned to M_1). The next unscheduled operation is $*_5$ and that case is shown in Fig. 6.5(b). The possible tuples for $*_5$ are $T_1 =< 3, M_1 >$ and $T_2 =< 3, M_2 >$. The minimum storage concurrencies for the two tuples are

$$T_1: \quad \min_i SC(IVar(V^i_{I_{max}})) = 3;\; min_i SC(OVar(V^i_{I_{max}})) = 1,$$

$$T_2: \quad \min_i SC(IVar(V^i_{I_{max}})) = 2;\; min_i SC(OVar(V^i_{I_{max}})) = 1.$$

Tuple T_2 is preferable since it corresponds to a data path with a smaller requirement of BIST resources. Hence $*_5$ is assigned to control step 3 and module M_2. Fig. 6.5(c) shows the next scheduling step, that of scheduling operation $+_1$. The operation can be assigned only to the adder module M_4. However, there is a choice in terms of the control step in which it can be scheduled. The possible tuples are $T_1 =< 1, M_4 >$, $T_2 =< 2, M_4 >$, $T_3 =< 3, M_4 >$ and $T_4 =< 4, M_4 >$. Depending upon the tuple chosen the lifetimes of the input and output variables of $+_1$ change which in turn affects the BIST resources. The minimum storage concurrencies for all the tuple are

$$T_1: \quad min_i SC(IVar(V^i_{I_{max}})) = 2;\; min_i SC(OVar(V^i_{I_{max}})) = 2,$$

$$T_2: \quad min_i SC(IVar(V^i_{I_{max}})) = 4;\; min_i SC(OVar(V^i_{I_{max}})) = 4,$$

$$T_3: \quad min_i SC(IVar(V^i_{I_{max}})) = 4;\; min_i SC(OVar(V^i_{I_{max}})) = 4,$$

$$T_4: \quad min_i SC(IVar(V^i_{I_{max}})) = 4;\; min_i SC(OVar(V^i_{I_{max}})) = 4.$$

Table 6.5: Synthesis algorithms used in experiments

SYNTHESIS TASK	TYPE	DESCRIPTION
Scheduling	Without Testability **SWT**	Traditional scheduling algorithms such as ASAP
	For Testability **SFT**	Scheduling and module assignment algorithm proposed in this chapter
Assignment	Without Testability **AWT**	Traditional register and/or module assignment without BIST Consideration
	For Testability **AFT**	Register assignment with BIST consideration (Chapter 5)

Hence tuple T_1 is preferred and $+_1$ is scheduled in control step 1 and assigned to M_4.

6.4 Experimental Results

The proposed scheduling and module assignment procedure has been integrated into the Stanford CRC synthesis-for-test tool, TOPS [51]. To demonstrate the use of the proposed scheduling technique in synthesizing data paths with low BIST area overhead, experiments were conducted on the following benchmarks: 1) the 2nd order differential equation - *Diffeqn* [61], 2) the auto regression filter element - *AR_filter* [63], 3) an 8-point FIR filter - *FIR_filter* [73], and 4) the elliptic wave filter - *EW_filter* [64]. Different synthesis flows were experimented on using two scheduling techniques and two register assignment techniques shown in Table 6.5. Using a combination of these techniques, the following three synthesis flows were used to synthesize data paths.

- Flow I (SWT-AWT): This is traditional high-level synthesis without any BIST consideration. A traditional scheduling technique is followed by traditional assignment techniques. Neither of them consider optimization for testability.

- Flow II (SFT-AWT): This flow uses the scheduling and module assignment technique proposed in this chapter, followed by a traditional register assignment that is oblivious to BIST considerations. The data paths synthesized using this flow demonstrate the effect of scheduling on savings in BIST overhead, independent of optimization during register assignment phase.

- Flow III (SFT-AFT): This flow uses the scheduling and module assignment technique presented in this chapter followed by a register assignment algorithm that can make the best use of the schedule to further minimize BIST overhead (from Chapter 5).

Tables 6.6, 6.8, 6.10 and 6.12 show the characteristics of the data paths synthesized from *Diffeqn*, *AR_filter*, *FIR_filter* and *EW_filter*, respectively. The functional module requirements used in Flow II and Flow III that use the proposed SFT approach were derived from the requirements of Flow I. It can be seen from the results that the latency and the total number of functional modules in all the three synthesis flows is preserved for all the benchmarks. However, there is an increase in the number of multiplexers as BIST considerations are incorporated into the various stages of synthesis. In the case of *EW_filter* (Table 6.12), the SWT-AWT technique results in more registers than the synthesis flows that incorporate testability.

The data paths synthesized by each synthesis flow were made self-testable using the minimal intrusion BIST methodology described in Chapter 3. The 0-1 ILP model was used to find a minimum area BIST solution for the data paths. Tables 6.7, 6.9, 6.11 and 6.13 compare the areas of synthesized data paths, both, before and after making them self-testable. Components designed using a macro-cell library supplied by LSI Logic Corp. were used for synthesis and the area is given in cell units [52]. It can be observed that for all four benchmarks, the BIST area overhead, B in the case of Flow II is less than that of Flow I and the BIST area overhead B for Flow III is less than that for Flow II. This decrease in BIST area overhead is accompanied by an increase in the area A of the original data paths (non self-testable versions). In the results, the only exception to this is the *EW_filter* benchmark (Table 14), in which case the area of the non self-testable version *improves* for Flow II and Flow III. The increase in the area of the non self-testable version in the other benchmarks

Table 6.6: Characteristics of data paths synthesized from *Diffeqn*

Synthesis	Latency	Registers	Modules			Muxes	
Technique	L		Mult	Add	Sub	2:1	3:1
SWT-AWT	4	5	3	1	1	6	4
SFT-AWT	4	5	3	1	1	8	3
SFT-AFT	4	5	3	1	1	9	3

Table 6.7: Area overhead comparison of *Diffeqn*

Synthesis	Area before	BIST	% BIST	Total	% decrease
Technique	BIST	overhead	overhead	area	in total area
	A	B	$(B/A \cdot 100)$	$C = A + B$	w.r.t. $C^{\dagger}_{WT}$
SWT-AWT	5306	1552	29.25	6858	-
SFT-AWT	5306	1136	21.40	6442	5.50
SFT-AFT	5402	1008	18.66	6410	6.53

$^{\dagger}C_{WT}$ is the total area using SWT-AWT technique

Table 6.8: Characteristics of data paths synthesized from *AR_filter*

Synthesis	Latency	Registers	Modules		Muxes				
Technique	L		Add	Mult	2:1	3:1	4:1	5:1	6:1
SWT-AWT	8	16	4	8	11	2	2	0	3
SFT-AWT	8	16	4	8	11	4	2	3	0
SFT-AFT	8	16	4	8	14	5	3	3	0

Table 6.9: Area overhead comparison of *AR_filter*

Synthesis	Area before	BIST	% BIST	Total	% decrease
Technique	BIST	overhead	overhead	area	in total area
	A	B	$(B/A \cdot 100)$	$C = A + B$	w.r.t. $C^{\dagger}_{WT}$
SWT-AWT	14496	2560	17.66	17056	-
SFT-AWT	14748	1792	12.15	16540	3.2
SFT-AFT	15137	1072	7.08	16209	5.1

$^{\dagger}C_{WT}$ is the total area using SWT-AWT technique

Table 6.10: Characteristics of data paths synthesized from *FIR_filter*

Synthesis	Latency	Registers	Modules		Muxes			
Technique	L		Add	Mult	2:1	3:1	4:1	5:1
SWT-AWT	5	8	4	4	6	1	0	1
SFT-AWT	5	8	4	4	6	2	2	1
SFT-AFT	5	8	4	4	7	2	0	2

Table 6.11: Area overhead comparison of *FIR_filter*

Synthesis Technique	Area before BIST A	BIST overhead B	% BIST overhead $(B/A \cdot 100)$	Total area $C = A + B$	% decrease in total area w.r.t. $C_{WT}^{\dagger}$
SWT-AWT	7363	3392	46.06	10755	-
SFT-AWT	8099	2592	32.00	10691	0.6
SFT-AFT	8110	2224	27.42	10334	4.0

[†] C_{WT} is the total area using SWT-AWT technique

Table 6.12: Characteristics of data paths synthesized from *EW_filter*

Synthesis	Latency	Registers	Modules		Muxes					
Technique	L		Add	Mult	2:1	3:1	4:1	5:1	6:1	7:1
SWT-AWT	14	10	5	3	8	3	2	2	1	1
SFT-AWT	14	8	5	3	6	7	6	2	0	0
SFT-AFT	14	8	5	3	5	6	7	1	1	0

Table 6.13: Area overhead comparison of *EW_filter*

Synthesis Technique	Area before BIST A	BIST overhead B	% BIST overhead $(B/A \cdot 100)$	Total area $C = A + B$	% decrease in total area w.r.t. $C_{WT}^{\dagger}$
SWT-AWT	10905	1776	16.28	12681	-
SFT-AWT	10608	736	6.94	11344	10.54
SFT-AFT	10661	736	6.90	11397	10.13

[†] C_{WT} is the total area using SWT-AWT technique

is due to the increase in the number of multiplexers. Even in the case of *EW_filter* where A decreases, the multiplexer area increases as the BIST overhead decreases. The total area of the self-testable data path, C decreases from Flow I to Flow II and from Flow II to Flow III. The last column in Tables 6.7, 6.9, 6.11 and 6.13 indicates the reduction in total area of the self-testable versions of the data paths with respect to the self-testable version of the data path synthesized without any BIST considerations (Flow I). It can be seen that in the case of *Diffeqn* and *EW_filter*, most of the reduction in total area comes from SFT and in the case of *FIR_filter* most of it comes from AFT. In the case of the *FIR_filter*, SFT produces a schedule that requires a register assignment algorithm capable of utilizing the testability optimization potential of the schedule to synthesize a data path with low BIST area overhead. The results indicate that the proposed scheduling and module assignment technique give a reduction of 30-50% in BIST area overhead and up to 10% in total area over traditional high-level synthesis techniques.

6.5 Summary

In this chapter we have shown how scheduling can affect the BIST area overhead of a data path. For minimizing BIST area overhead, it is desirable to share BIST resources across functional modules. We have shown how scheduling affects lifetimes of variables and module assignment and thus influences BIST overhead. The properties of schedules and module assignments that influence BIST overhead have been incorporated into a 2-phase scheduling technique. In Phase 1, coarse scheduling is performed such that the latency and module requirement of the final data path is not compromised. In Phase 2, detailed scheduling is done by assigning operations to control steps and modules. The data paths synthesized by the proposed scheduling technique, while having significantly lower BIST overhead, are competitive in terms of performance and area of registers and functional modules with those synthesized by traditional techniques. However, the savings in BIST area are at the expense of increased multiplexer cost. The results indicate a reduction of up to 10% in overall area of the BIST version of the synthesized data path.

Chapter 7

Computational Redundancy and BIST Resources

In the previous chapters we presented scheduling and assignment approaches that reduce BIST resource cost of a data path. However, the degree of freedom that can be exploited during scheduling and assignment to minimize BIST resources is often limited by the data and control dependencies of a behavior. In this chapter we propose transformation of a behavior by introducing redundant computations such that the resulting data path is testable using few BIST resources. The transformation makes use of the *spare capacity* of modules to add redundancy that enables test paths to be shared among the modules. A technique for identifying potential BIST resource sharing problems in a behavior and resolving them by redundant computation is presented. Introduction of redundant computations is performed without compromising the latency and functional resource requirement of the behavior.

7.1 Introduction

The degree of freedom available during scheduling and assignment for minimizing BIST resources is often limited by the data and control dependencies of a behavior. In such cases, alternate behavioral descriptions need to be explored. At the logic level, it has been shown that introduction of redundancy can be beneficial for certain testability objectives [74]. In this chapter we propose transformations that introduce redundant computations in a behavior for achieving designs with a reduced number of BIST resources. The data flow graph (DFG) representation used in behavioral synthesis has proved to be a suitable representation to perform all kinds of optimizations. Transformations have been applied for optimization of a great variety of goals, including area, performance, fault tolerance and partial scan overhead [10],[11],[12],[13],[14]. We propose a semantic-preserving DFG transformation aimed at optimizing the cost of BIST resources required to make a synthesized data path self-testable.

7.2 Redundancy in RTL Data Paths

7.2.1 Spare Capacity of Modules

Most synthesized data paths do not have 100% utilization of all modules during a computation. Typically, most functional modules perform useful computation in some clock cycles and are idle during other cycles. For example, consider the DFG G shown in Fig. 7.1. Suppose that a data path is synthesized from this DFG that executes in 3 clock cycles and the addition is performed by an adder module A, the multiplication by a multiplier module M and the two subtraction operations by a subtractor module S. Modules A and M perform computations only in clock cycle 1 and are idle during clock cycles 2 and 3. Module S is idle only in clock cycle 1 and computes in clock cycles 2 and 3. A and M are utilized 33% of the time and S is utilized 66% of the time. By performing *redundant* computations in the modules during the clock cycles they are not performing *useful* computations, sharing of paths that carry test data can be increased.

Fig. 7.2(a) shows the state (busy or idle) of the read and write ports of the various operations in DFG G of Fig. 7.1, assuming a minimum latency schedule. The shaded boxes represent when the input ports (read ports) or the output ports (write ports) are busy. Clearly the ports are idle some of the time. Also, note that the read and write ports of the addition and multiplication operations are busy at the *same* time. By introducing redundant reads and writes during idle times, paths to operations can be created that can be shared for transporting test data. For example, in Fig. 7.2(b), redundant read and write actions have been introduced

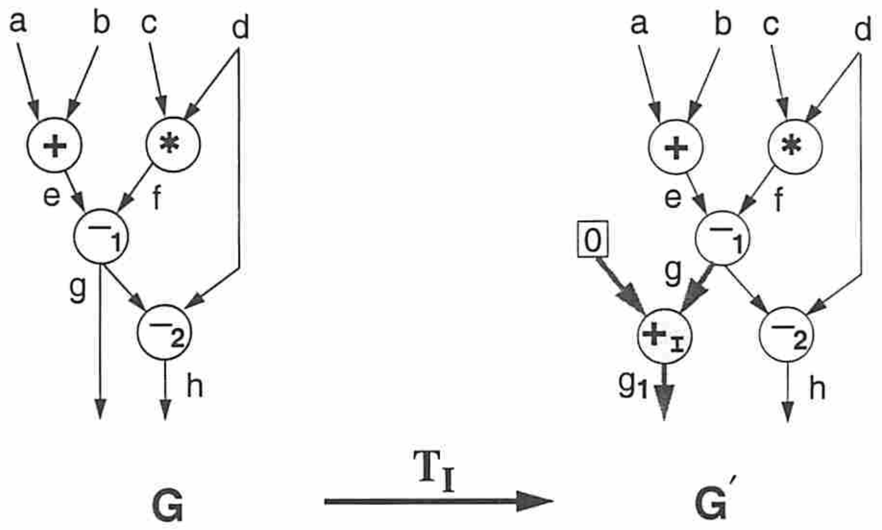


Figure 7.1: I-transformation

for the addition operation during idle time (shown hashed in the figure). Since the redundant read of the addition occurs at a different time than the read of the multiplication, the two reads can share hardware for the data transfer. The shared hardware could be used to transport test data for both the operations. A redundant write can similarly enhance sharing of test paths between the operations.

Definition 34 *The* **percentage spare capacity** *of an operation type is defined as the percentage of clock cycles during which the modules allocated for that operation type do not perform useful computation.*

It can be seen from Table 7.1 that a significant amount of spare capacity is available in the well known benchmark algorithms [54]. The available spare capacity gives a great deal of flexibility in introducing redundant computations to be performed by modules with the goal of optimizing BIST resources.

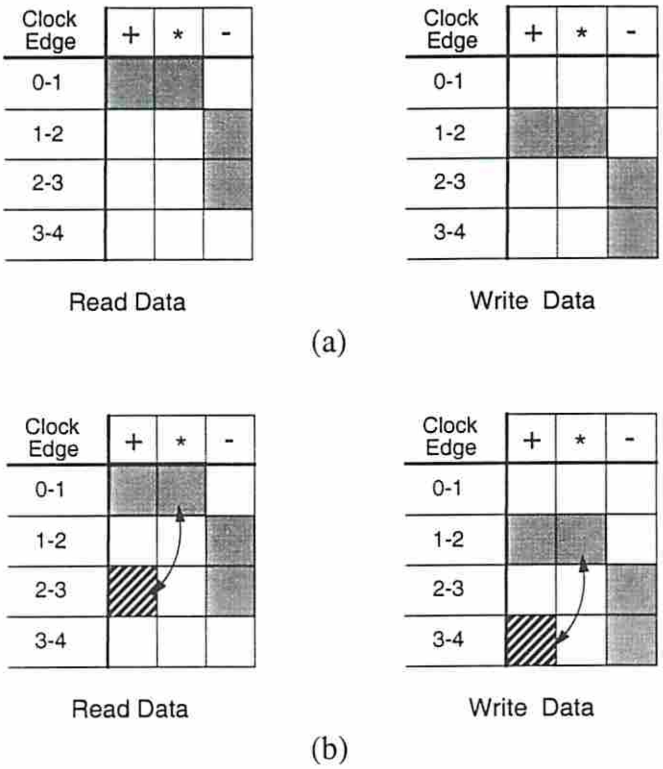


Figure 7.2: Read and Write Cycles of Operations

7.2.2 Adding Redundant Computations in Data Flow

Many operations used in a behavioral specification have an identity value associated with them. For example, a multiplication operation has an identity value of 1 and addition operation has an identity value 0. If one of the operands of these operations is equal to the identity value then the output of the operation is the same as the other operand.

Definition 35 *An identity mode DFG node* (**I-node**) *is a node whose operation type has an identity value and one of its operands is set to a constant corresponding to the identity value.*

Table 7.1: Available spare capacity of benchmark algorithms

Algorithm	L	# Operations			# Modules			% Spare capacity		
		+	*	-	+	*	-	+	*	-
Diffeqn	4	2	6	2	1	2	1	50.0	25.0	50.0
EW_filter	14	26	8	-	5	3	-	56.7	84.6	-
AR_filter	8	12	16	-	4	4	-	62.5	50.0	-
FIR_filter	5	4	4	-	1	4	-	0	80.0	-

An I-node can be added between any two operations of a DFG without changing the functionality of the DFG. Consider a DFG, $G = (V, E)$. It can be transformed into a DFG, $G' = (V', E')$ by the following transformation.

Definition 36 **(I-transformation)** *Introduce an I-node o_k^I between any two operation nodes o_i and o_j, $(o_i, o_j) \in E$. The node is introduced such that the output (outgoing edge) of o_i becomes one of the inputs (incoming edge) of o_k^I and the output (outgoing edge) of o_k^I becomes the input (incoming edge) of o_j. The other input (incoming edge) of o_k^I corresponds to a constant, the identity value of the I-node operation.*

We denote the transformation as $G \overset{T_I}{\to} G'$. Fig. 7.1 shows a DFG G which is transformed into the DFG G' by the addition of an I-node (shown shaded in the figure). A DFG has two pseudo-nodes (not shown in the figure) that correspond to the primary inputs (*source* node) and primary outputs (*sink* node) of the data flow. In the case of Fig. 7.1, the I-node is added between operation node $-_1$ and the sink node. The variable input g of the I-node is the same as the output variable of $-_1$ and a new variable g_1 is created that is the output of the new addition I-node. The value of the variable g_1 in G' is the same as variable g in G. The concept of exploiting identity modes of functional modules in *synthesized* data paths for transporting test data was suggested in [75]. An I-node is an analogous concept at the behavioral level with the distinction that the identity mode is used to move *functional* data without changing it.

Theorem 14 *The data flow graph G' obtained from G by an I-transformation is functionally equivalent to G.*

7.3 Effect of I-nodes on BIST Resources

We propose the use of I-transformations to modify DFGs which, when followed by scheduling and hardware assignment, will lead to cost-effective BIST data paths. The basic idea is to introduce redundant paths that can be used to 1) transport test data, or 2) enable sharing of non-redundant paths in transporting test data. Corresponding to 1) and 2), we have two types of I-transformations, Type 1 and Type 2, respectively.

7.3.1 Type 1 I-transformation

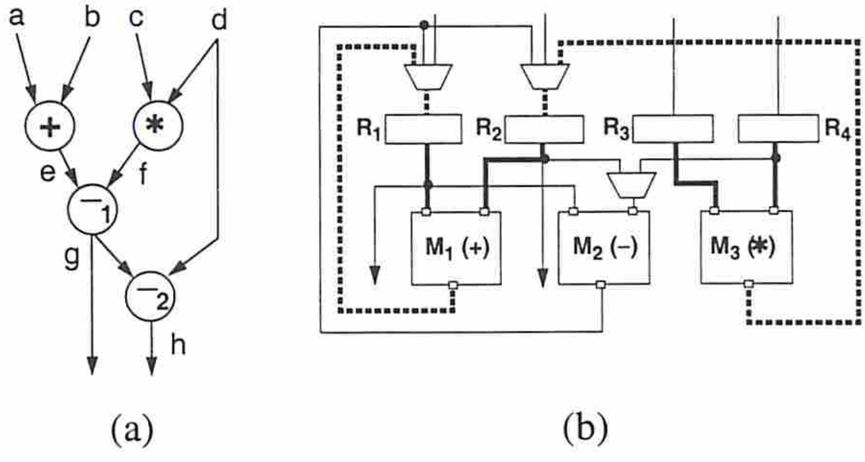
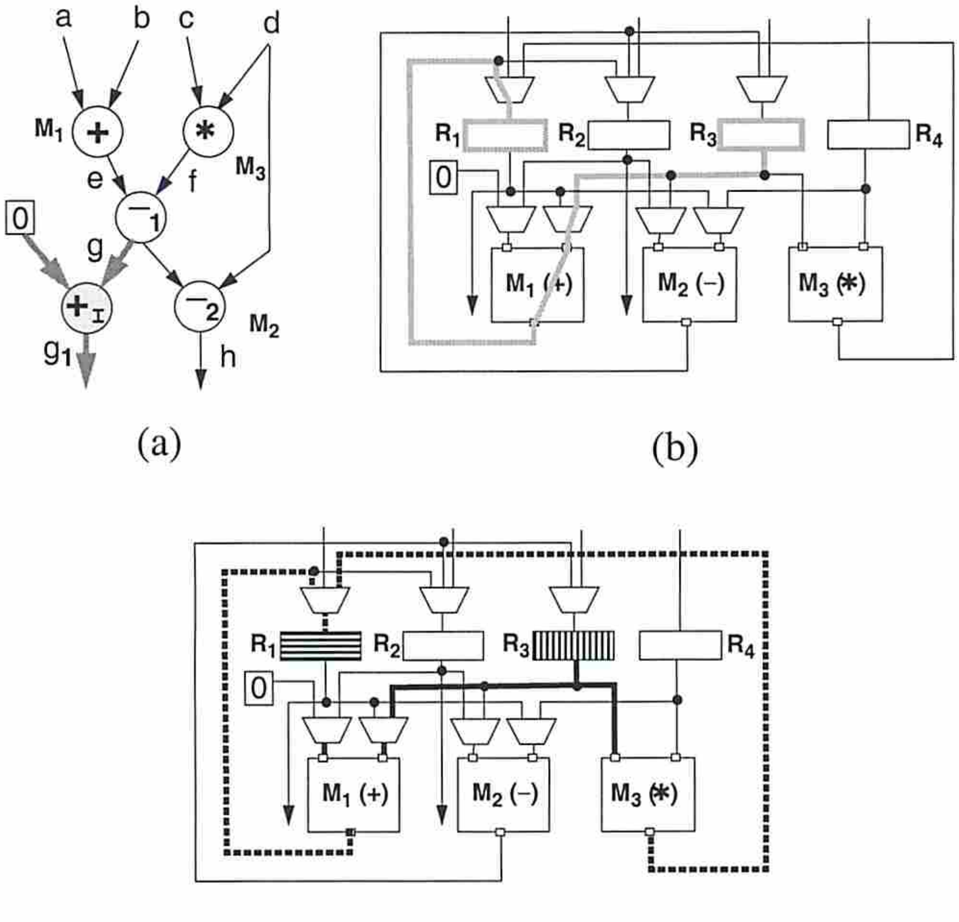


Figure 7.3: DFG1 and synthesized data path

The simple DFG shown in Fig. 7.3(a) illustrates the Type 1 I-transformation. The minimum latency achievable for this DFG is 3. An adder, a multiplier and a subtractor are required to implement the scheduled behavior. A data path synthesized with the objective of minimizing BIST area overhead using assignment techniques from Chapter 5 is shown in Fig. 7.3(b). The adder uses R_1 and R_2 as input, and the multiplier uses R_3 and R_4. Thus the adder and multiplier cannot share any test pattern generators. Also two distinct registers are required to compress the responses of the multiplier and the adder, namely R_1 and R_2 respectively. The scheduling, module assignment or register and interconnect assignment cannot improve the sharing of test resources between the modules. The minimal intrusion BIST solution for the data path involves modification of R_1 to CBILBO, R_2 to BILBO and R_3 and R_4 to TPGs for a BIST area overhead of 980 cell units (for a 16-bit data path using library [52]).



(a)

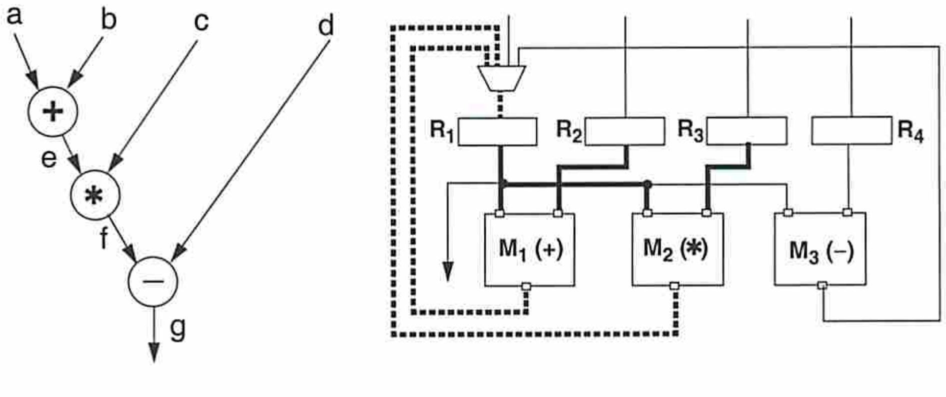
(b)

(c)

Figure 7.4: Type 1 I-transformation of DFG1

Fig. 7.4(a) shows the same DFG with an I-node inserted. The I-node performs a redundant addition in control step 3 when the allocated adder is idle. A data path synthesized using this modified DFG is shown in Fig. 7.4(b) where the functionally redundant path is shown highlighted. The redundant paths from 1) variable g to the I-node, and 2) from the I-node to variable g_1, make it possible for the adder and multiplier to share BIST resources. Variables f (output variable of the multiplication operation) and g_1 (output variable of the addition operation created by redundancy) can be assigned to the same register, R_1, that can provide $\mathcal{C}$ functionality for both the modules. Similarly, input variables of multiplication and addition, c and g, respectively, can be assigned to the same register, R_2, that can provide $\mathcal{G}$ functionality for both the modules. As shown in Fig. 7.4(c), the redundant paths created through the adder by the addition of I-node can be shared with other functional paths through the multiplier, resulting in common BIST resources for the two modules. The minimal intrusion BIST solution for the data path in Fig. 7.4(c) is R_3 is modified to CBILBO and R_1 and R_4 to TPGs for a BIST area overhead of 724 cell units.

7.3.2 Type 2 I-transformation



(a) (b)

Figure 7.5: DFG2 and synthesized data path

In contrast to Type 1 transformation, Type 2 transformation creates redundant paths that *enable* sharing of other non-redundant functional paths as test paths. In this case the redundant path does not transport any test data. This is illustrated on the DFG shown in Fig. 7.5(a). The minimum latency is 3 and the module requirement is one adder, one multiplier and one subtractor. Fig. 7.5(b) shows a data path synthesized from this DFG. It can be seen that it is possible to assign registers such that the test response compression of the adder and multiplier can be done by the same register, namely, R_1. However, a distinct register is required for $\mathcal{G}$ functionality for the right input port of the adder and multiplier, R_2 and R_3, respectively. R_1 is modified to CBILBO and R_2, R_3 and R_4 to TPGs in the minimal intrusion BIST solution and the overhead is 816 cell units. Fig. 7.6(a) shows the same DFG modified using a multiplication I-node. The corresponding synthesized data path is shown in Fig. 7.6(b) with the redundant path highlighted. The use of the multiplier in the redundant path corresponds to the redundant multiplication in control step 1. The addition of redundancy creates a data transfer to the right input port of the multiplier. This enables the input variable of the multiplication, c_2, and an input variable of the addition, b, to be assigned to the same register, R_2. R_2 can thus be shared as a $\mathcal{G}$-resource between the adder and multiplier as shown in Fig. 7.6(c). The minimal intrusion BIST solution modifies R_1 as CBILBO and R_3 and R_4 as TPGs for an area overhead of 720 cell units. Note that in this type of transformation, the redundant path does not transport test data. It *enables* sharing of test paths, thus reducing BIST area overhead.

7.4 Properties of I-transformation

The introduction of an I-node does not change the behavior. However, it does change the structure and some properties of a DFG. It is essential to characterize the effect of the I-transformation on the properties of the DFG to be able to apply the transformation in an efficient and beneficial manner.

Definition 37 *A* critical path *in a DFG $G = (V, E)$ is the longest path in the DFG.*

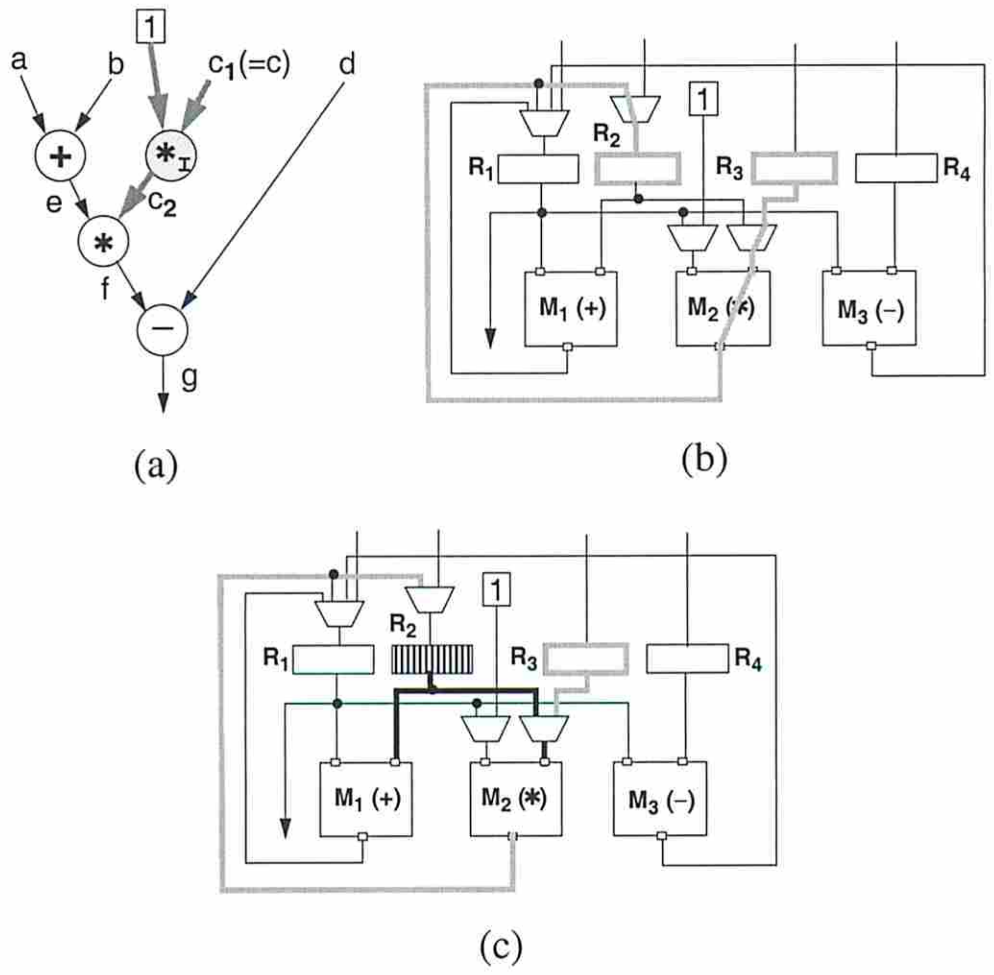


Figure 7.6: Type 2 I-transformation of DFG2

Lemma 7 *If the I-transformation is applied such that o_k^I is introduced in a non-critical path of the DFG, then the minimum latency for which the DFG can be scheduled remains unchanged.*

The length of the critical path is a lower bound on the achievable latency of a DFG. Lemma 7 enables the introduction of I-nodes that do not effect the minimum latency of the DFG. For optimizing BIST area overhead we have to maximize the sharing of registers as BIST resources between different modules. The assignment of variables to registers depends on the lifetimes of the variables. The lifetimes of variables and hence the minimum number of registers required to implement a data path is determined by the schedule of a DFG. For an unscheduled DFG G, the minimum number of registers required for *any* schedule can be determined from cuts of graph $G_{s,t}$ derived from G (refer Fig. 3.1). In the following discussion $CS_j \,|\, CS_{j+1}$ denotes the control step boundary between the jth and $(j+1)$th control steps of a scheduled DFG.

Definition 38 *A* **feed-forward cut** *(S,T) of DFG $G_{s,t} = (V,E,s,t)$, is a partition of nodes in $G_{s,t}$ into S and T such that $s \in S$, $t \in T$ and any edge $e \in E$ between a node $v_i \in S$ and a node $v_j \in T$, if it exists, is directed from v_i to v_j. The number of such edges e is called the* **size** *of the feed-forward cut and is denoted by $size(S,T)$.*

Lemma 8 *Given a DFG $G_{s,t}$, for any schedule S_i, the variables alive at any control step boundary $CS_j \,|\, CS_{j+1}$ correspond to a feed-forward cut (S,T) of $G_{s,t}$ and $size(S,T)$ is the number of such variables.*

Proof: Consider the $CS_j \,|\, CS_{j+1}$ control step boundary of schedule S_i of $G_{s,t}$. Let S be the set of operations in $G_{s,t}$ scheduled in control steps 1 to CS_j and T be the set of operations scheduled in control steps CS_{j+1} to L. According to Definition 38, (S,T) is a feed-forward cut and $size(S,T)$ is the number of variables alive at $CS_j \,|\, CS_{j+1}$. □

Lemma 9 *An I-transformation increases the number of variables in the DFG by 1, but the minimum number of registers required to store all the variables remains the same.*

Proof: Consider a DFG $G_{s,t}$ which transforms to $G'_{s,t}$ after an I-transformation. Let o^I_k be the I-node replacing edge (o_i, o_j) in $G_{s,t}$.

Consider a feed-forward cut (S,T) of $G_{s,t}$. Depending on the partitions to which o_i and o_j belong, the cut (S,T) of $G_{s,t}$ is transformed into a feed-forward cut (S',T') of $G'_{s,t}$. There are three possible types of feed-forward cuts. (1) $o_i, o_j \in S$, (2) $o_i, o_j \in T$, and (3) $o_i \in S, o_j \in T$.

1. $\underline{o_i, o_j \in S\text{: (Fig. 7.7(a))}}$ Cut (S,T) of $G_{s,t}$ is transformed to cut (S',T') of $G'_{s,t}$ where $S' = S \cup \{o^I_k\}$ and $T' = T$. Since $o_i, o_j, o^I_k \in S'$, $size(S',T') = size(S,T)$.

2. $o_i, o_j \in T$: (Fig. 7.7(b)) $size(S', T') = size(S, T)$ (similar to Type 1).

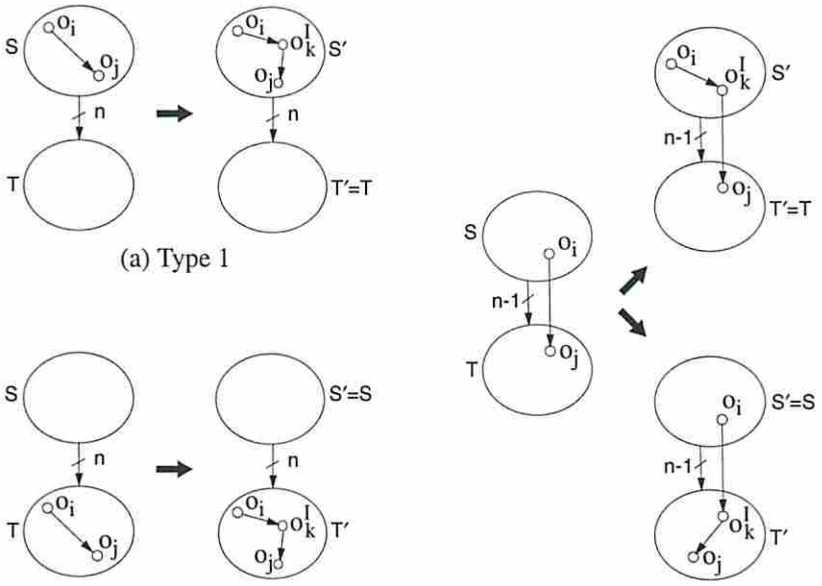
3. $o_i \in S, o_j \in T$: (Fig. 7.7(c)) Cut (S, T) of $G_{s,t}$ can be transformed to two cuts depending on the partition to which o_k^I belongs. If $o_k^I \in S$, then the edge (o_i, o_j) in cut (S, T) is *replaced* by edge (o_k^I, o_j) in the transformed cut (S', T'). Similarly, if $o_k^I \in T$, then the edge (o_i, o_j) in cut (S, T) is *replaced* by edge (o_i, o_k^I) in the transformed cut (S', T'). Hence $size(S', T') = size(S, T)$ for both the transformed cuts.

If $G_{s,t}$ has p, q and r feed-forward cuts of types 1, 2 and 3, respectively, they get transformed into p, q and $2 \cdot r$ feed-forward cuts in $G'_{s,t}$. From Lemma 8, any schedule of latency L of a DFG $G_{s,t}$ corresponds to a set of L feed-forward cuts of $G_{s,t}$. From the above discussion, it can be seen that after an I-transformation a schedule is still possible such that a cut of type 1 in $G_{s,t}$ is replaced by a cut of type 1 in $G'_{s,t}$, a cut of type 2 in $G_{s,t}$ is replaced by a cut of type 2 in $G'_{s,t}$ and a cut of type 3 in $G_{s,t}$ is replaced by *any one* cut of type 3 in $G'_{s,t}$. Since the maximum size of a feed-forward cut for a schedule is the minimum number of registers for that schedule, it remains unchanged after the transformation. Since the argument is valid for any schedule of $G_{s,t}$, the lemma is proven. □

Lemma 9 demonstrates that introduction of I-nodes is not harmful in terms of storage resource requirement of a data path. The minimum storage requirement (i.e. minimum number of registers required to implement the data path) before and after any I-node transformation is the same. Thus we see that I-nodes can be introduced such that the number of modules, number of registers and the latency remains the same. However the transformation can have an adverse effect on the interconnect complexity. A trade-off exists between reduction in BIST area overhead and interconnect complexity.

Lemma 10 *The number of multiplexer inputs at the input ports of a module to which n_I I-nodes have been assigned increases by at most $n_I + 1$.*

Proof: The n_I I-nodes have $2 \cdot n_I$ operands out of which n_I are *constants* equal to the identity value of the module and n_I are *variables*. The n_I constant values can



(a) Type 1

(b) Type 2

(c) Type 3

Figure 7.7: Transformation of feed-forward cuts

be supplied to the same input port, resulting in one additional multiplexer input for that port. The n_I variable operands need to be supplied at the other input port of the module. In the worst case, these variables cannot share registers with each other or with the original operands of the module. In this worst case, n_I new input registers would be created for the input port, resulting in n_I additional multiplexer inputs. Hence a total of $(n_I + 1)$ additional multiplexer inputs are required in the worst case. □

The increase in the number of multiplexers is due to the fact that one additional operation (corresponding to the I-node) is assigned to a module and additional operands need to be supplied to the module. However, one of the operands of the I-node is a constant corresponding to the identity of the I-node operation. Taking advantage of this fact, multiplexer area can be optimized. This can be achieved in two ways.

1. Storing identity value in primary input register: The identity value of an I-node corresponds to an extra multiplexer input at the input port of a module to which the I-node is assigned. The identity value is required only during the

clock cycle in which the redundant computation is performed, hence a *primary input register* that does not contain a valid value during that clock cycle can be used to supply the identity value during that clock cycle. For example, in Fig. 7.8(a), the primary input register R_2, does not have valid data during clock cycle 3 as indicated by the utilization chart of the registers in Fig. 7.8(b) (shaded boxes imply register has valid data). The identity value required for the redundant operation in clock cycle 3 can be loaded from R_2. Since R_2 is connected to the same input port as the identity value, the multiplexer at that input port can be eliminated (Fig. 7.8(c)).

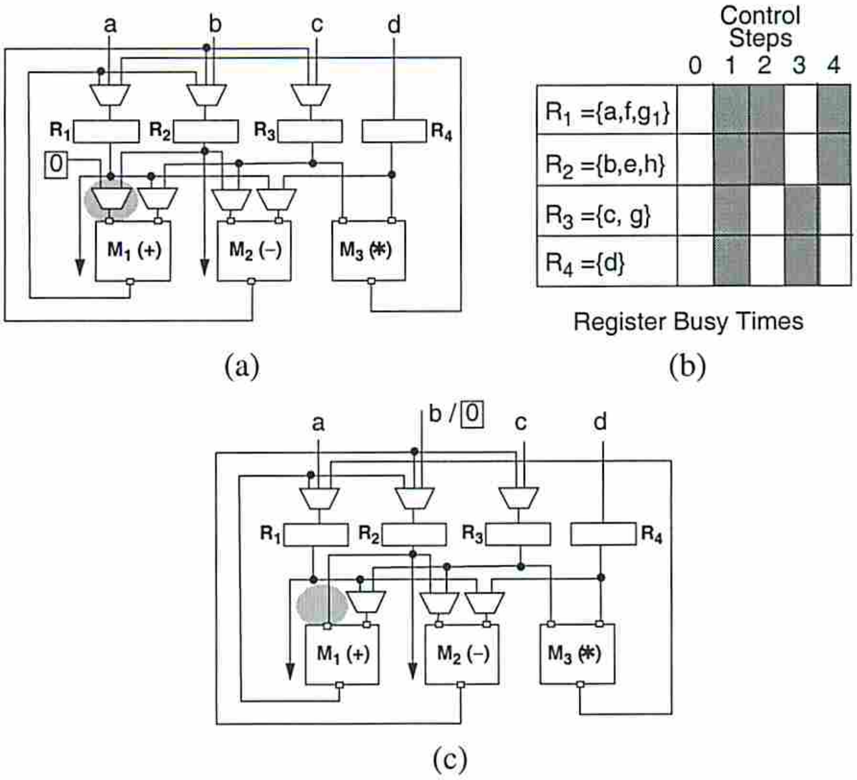
2. Logic optimization of multiplexers: Sometimes it is not possible to find a primary input register for storing the identity value. In such cases multiplexer overhead can be reduced by logic optimization. For example, in Fig. 7.9(a), the multiplexer input necessary to supply a 0 identity value can be replaced by a simple AND gate as shown in Fig. 7.9(b).

7.5 Strategy for Introduction of I-nodes

The strategy for introduction of I-nodes involves two tasks, namely 1) identify pairs of operations that will have a potential problem with sharing of BIST resources in subsequent stages of synthesis; and 2) resolve the problem identified in Task 1 by introducing an I-node.

7.5.1 Identifying BIST Resource Sharing Problem (Task 1)

A BIST resource sharing problem between a pair of modules arises out of the inability to share registers by the input and output variables of pairs of operations assigned to those modules. Hence, the task of identifying BIST resource sharing problems in a DFG involves identification of pairs of operations that have 1) a high probability of being assigned to different modules, and 2) a high probability of having disjoint sets of BIST resources that cannot be shared. We identify pairs of operations that get assigned to different modules with *certainty*.



	Control Steps 0	1	2	3	4
R_1 ={a,f,g_1}		▨	▨		▨
R_2 ={b,e,h}		▨	▨		▨
R_3 ={c, g}		▨		▨	
R_4 ={d}		▨		▨	

Figure 7.8: Elimination of extra MUX input

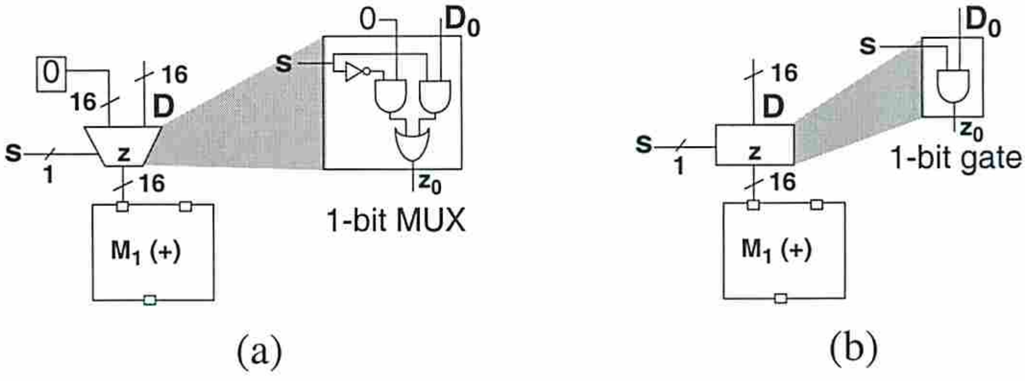


Figure 7.9: Logic optimization of extra MUX input

Lemma 11 *A pair of operations o_i and o_j get assigned to different modules for all minimum latency schedules and corresponding module assignments only if*

1. $\mid \mathcal{M}(o_i) \mid = \mid \mathcal{M}(o_j) \mid = 1$ *and* $\mathcal{M}(o_i) = \mathcal{M}(o_j)$, *or*

2. $type(o_i) \neq type(o_j)$.

If the operations are of different type then they will definitely get assigned to different modules. If they are of the same type, then they will get assigned to different modules with certainty only if they are concurrent in all possible schedules. We target only such pairs of operations for the following two reasons. Firstly, since many parameters of the data path are not fixed at this point, it is efficient to use information that has a *high degree of certainty*. Secondly, from our experiments we find that the process of introduction of I-nodes follows the *law of diminishing returns*. As more I-nodes are added, the resulting savings in BIST area overhead become marginal. Hence we target those pairs of operations that have a potential problem with sharing of BIST resources to the *highest degree* and with the *highest certainty*. The targeted pairs of operations are ranked based on the degree of difficulty in sharing test resources. Since the schedule is not determined at this point, the lifetimes of the variables are not known. Based on the probabilities of assigning operations to a control step, a probabilistic estimate of the number of registers for an unscheduled DFG can be derived. Let $P_{o_i}(CS)$ be the probability that o_i is scheduled in control step CS. A simple way of computing this probability is given in [76].

Lemma 12 *The probability that edge (o_i, o_j) crosses the $CS_k | CS_{k+1}$ clock boundary is*

$$P_{edge}(o_i, o_j, CS_k) = \left(\sum_{CS=ASAP_i}^{CS_k} P_{o_i}(CS) \right) \cdot \left(\sum_{CS=CS_{k+1}}^{ALAPj} P_{o_j}(CS) \right).$$

Proof: Given two operations o_i and o_j, where o_j is a direct successor of o_i, and given a control step CS_k, the edge (o_i, o_j) crosses the boundary between control steps CS_k

and CS_{k+1} only if o_i is scheduled in CS_k or any clock step before that and o_j is scheduled in CS_{k+1} or any clock step after that. The probability that o_i is scheduled in CS_k or any clock step before that is given by the sum of the probabilities $P_{o_i}(CS)$ for values of CS from $ASAP_i$ to CS_k. Similarly, the probability that o_j is scheduled in CS_{k+1} or any clock step after that is given by the sum of the probabilities $P_{o_j}(CS)$ for values of CS from CS_{k+1} to $ALAP_j$. □

If two or more edges coming from the same node (other than the source node) cross a given control step boundary, only one register is needed. Since for a given node o_i and control step CS_k, there can exist more than one edge from o_i crossing the boundary CS_k and CS_{k+1}, we consider the maximum probability,

$$P_{edge}^{max}(o_i, CS_k) = \max_{\forall o_j, (o_i, o_j) \in E} P_{edge}(o_i, o_j, CS_k).$$

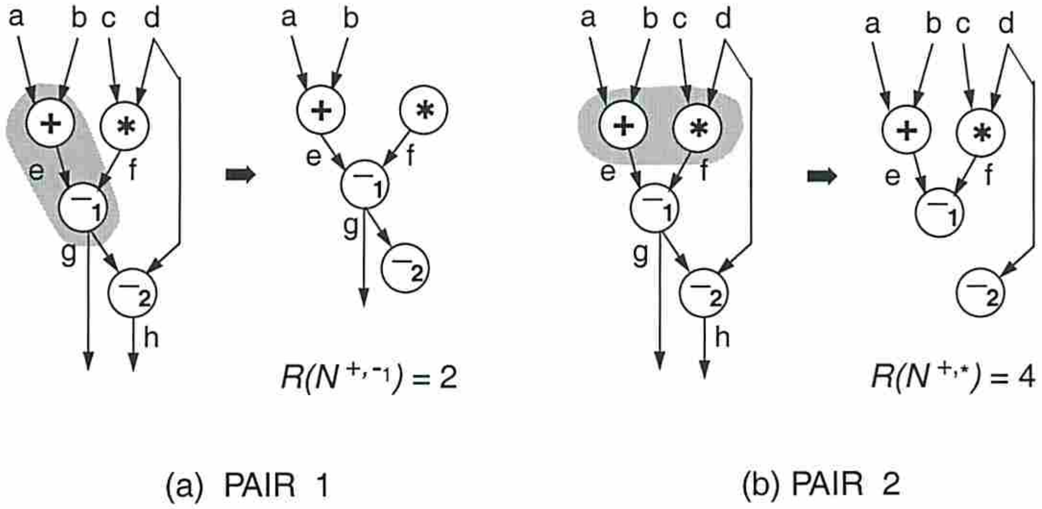
For a given control step CS_k, the sum of probabilities of edges coming from different nodes that cross the boundary represents the estimated number of registers needed for this boundary. And the *maximum* of this value over all the control step boundaries is the estimated number of registers for DFG G,

$$\mathcal{R}(G) = \max_{\forall CS} \left\{ \sum_{\forall o_i} P_{edge}^{max}(o_i, CS_k) \right\}.$$

The degree of difficulty of sharing of test resources for a pair of operations is proportional to the number of registers required for the input and output variables of the pair of operations. The register estimation technique mentioned above is used on a subgraph of the DFG subtended by the input and output variables of the pair of operations under consideration.

Definition 39 *A* **neighborhood graph** $N^{i,j} = (V_N, E_N)$ *corresponding to a pair of operations o_i and o_j in DFG G is a subgraph of G such that $E_N = \{e \mid e$ corresponds to an input or an output variable of o_i or $o_j\}$ and $V_N = \{o_k \mid o_k$ is an operation that consumes or defines a variable in $E_N\}$.*

The subgraph $N^{i,j}$ contains only the edges corresponding to the input and output variables of o_i and o_j and the operations that determine the lifetimes of these variables. For an operation o_k in $N^{i,j}$, $ASAP_k$ and $ALAP_k$ are the same as in the original DFG. Hence, the register estimate on subgraph $N^{i,j}$, $\mathcal{R}(N^{i,j})$, is an estimate of the number of registers required to supply test patterns and compress test responses to the pair of modules to which the pair of operations is assigned. A pair of operations with the highest degree of difficulty, (largest value of $\mathcal{R}(N^{i,j})$) is targeted as the BIST resource sharing bottleneck to be solved in Task 2.



(a) PAIR 1

(b) PAIR 2

Figure 7.10: Finding sharing bottlenecks

Example 8 *In Fig. 7.10, two pairs of operations are considered for determining the degree of difficulty of sharing BIST resources. In Fig. 7.10(a), the register estimate of the neighborhood graph of operations* $+$ *and* $-_1$ *is the expected number of registers required to store the input and output variables of operations* $+$ *and* $-_1$*, namely,* a, b, e, f *and* g*. Fig. 7.10(b) shows the register estimate of the neighborhood graph of another pair of operations, namely,* $+$ *and* $*$*. Since the register estimate in (b) is greater than the register estimate in (a), the adder and multiplier has a greater degree of difficulty in sharing BIST resources. Hence pair 2 is chosen as the problem pair in Task 1.*

7.5.2 Choosing an I-node (Task 2)

The second task of choosing an I-node to resolve the problem identified in Task 1 has two parameters: 1) the operation type of the I-node, e.g. addition or multiplication, and 2) the position of the I-node in the DFG. As shown in section 4, the introduction of an I-node anywhere in the DFG preserves its functionality but changes some of the properties of the original DFG and affects the synthesized data path in different ways. To ensure that I-nodes are introduced in a manner that is beneficial for BIST but not harmful in terms of functional constraints, the introduction of an I-node should satisfy the following criteria: 1) not violate timing constraints; 2) not violate resource constraints; 3) not limit scheduling mobility; 4) maximize re-usage of interconnect to keep interconnect complexity low; and 5) optimize BIST area overhead. Criteria 1 and 2 are strict in the sense that I-nodes that violate them are not allowed. Criteria 3 and 4 are desirable and I-nodes with least impact on scheduling mobility and interconnect complexity are preferred. From the I-nodes that qualify for the first four criteria, an I-node with the highest impact on the BIST resource sharing problem is selected. The impact of an I-node on BIST resources is quantified using the probabilistic register estimate used in Task 1.

Criterion 1 can be satisfied by ensuring that I-nodes are introduced on non-critical paths (Lemma 7). Efficient techniques for computing resource requirements for a DFG are available [16]. Using these techniques the change in functional resource requirement due to the presence of an I-node can be determined. This information can be used to avoid I-nodes that violate criterion 2. The scheduling mobility in criterion 3 is the degree of freedom available in scheduling operations in a DFG in different control steps.

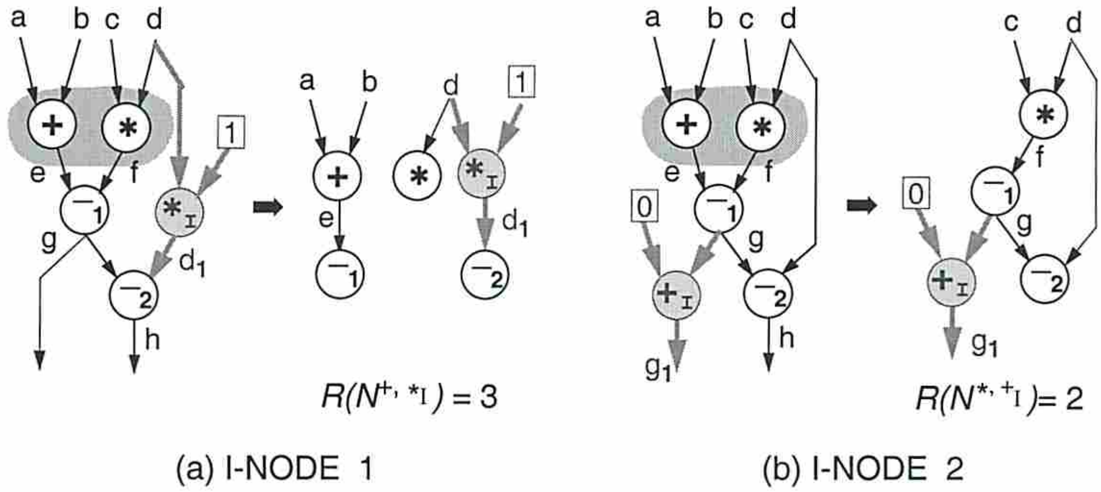
Definition 40 *The* **scheduling mobility**, *$SM(G)$ of a DFG $G = (V, E)$, is the sum of the slack of all operations in the DFG. $SM(G) = \sum_{o_i \in V} slack(o_i)$.*

Introduction of an I-node can change the $ASAP$ and $ALAP$ values of some of the operations. The effect of introducing an I-node on the scheduling mobility of a DFG G is denoted by $\Delta SM = SM(G) - SM(G')$. Note that only the slack of operations in the original DFG G is included in $SM(G')$, i.e. none of the I-nodes are considered. Criterion 3 is satisfied by selecting an I-node with the smallest ΔSM. The scheduling mobility is also used to determine whether more I-nodes should be added to a DFG. I-nodes are introduced only while the scheduling mobility remains over a certain predetermined threshold. If the scheduling mobility drops below the threshold, no more transformations are performed.

The process of choosing an I-node o_k^I for solving the BIST resource sharing problem of pair $\{o_i, o_j\}$ proceeds as follows. Based on criteria 1 and 2, the set of all possible I-nodes is pruned to a few choices. Criteria 3 is used to determine a preference order among the choices of I-nodes with I-nodes with the smallest ΔSM having the highest preference. Associated with every candidate o_k^I is a tuple $T_k =< type(o_k^I), edge(o_k^I) >$. $type(o_k^I)$ is the same type as o_i or o_j. $edge(o_k^I)$ is the edge in the DFG that is replaced by the redundant operation. To evaluate the tuples with respect to their effect on improving the BIST resource sharing potential of the pair of operations, a cost function $Cost(T_k)$ is associated with each tuple. To compute $Cost(T_k)$, the register estimates are computed for the neighborhood graphs of the I-node candidate paired with o_i and with o_j. When $type(o_i) \neq type(o_j)$, the register estimate of the neighborhood graph of I-node (o_k^I) and the operation which is not the same type as $type(o_k^I)$ is considered. When $type(o_i) = type(o_j) = type(o_k^I)$, neighborhood graphs of both pairs $\{o_i, o_k^I\}$ and $\{o_j, o_k^I\}$ are considered and the *maximum* of the register estimates is considered. The cost of T_k is now defined as,

$$Cost(T_k) = \begin{cases} \mathcal{R}(N^{i,k}) & \text{if } type(o_k^I) = type(o_j)(\neq type(o_i)) \\ \mathcal{R}(N^{j,k}) & \text{if } type(o_k^I) = type(o_i)(\neq type(o_j)) \\ \max\{\mathcal{R}(N^{i,k}), \mathcal{R}(N^{j,k})\} & \text{otherwise.} \end{cases}$$

The register estimate indicates the extent to which the introduction of o_k^I will solve the BIST resource sharing problem of pair $\{o_i, o_j\}$. The lower the value of the register estimate on a neighborhood graph with the I-node, the greater the impact of the I-node in sharing of BIST resources. Hence, from the I-node candidates with the least impact on scheduling mobility, an I-node o_k^I with the lowest $Cost(T_k)$ is selected for insertion.



(a) I-NODE 1

(b) I-NODE 2

Figure 7.11: Choosing an I-node

Example 9 *Fig. 7.11 shows two possible I-node choices for solving the pair targeted in Task 1 (Fig. 7.10(b)). Since the types of operations in the pair are different we consider only one neighborhood graph for each I-node. For the multiplication I-node in Fig. 7.11(a), the register estimate is 3 and for the addition I-node in Fig. 7.11(b), the register estimate is 2. This implies that the addition I-node increases BIST resource sharing of the adder and multiplier to a greater extent since then number of registers estimated for BIST resources is smaller than the multiplication I-node. Hence the I-node in Fig. 7.11(b) is chosen for insertion.*

7.6 Experimental Results

The I-node transformation was performed on some example DFGs. *ex* is the example from Fig. 7.1 and *Diffeqn*, *Tseng* and *AR_filter* are benchmarks from [54]. Some of the characteristics of these DFGs are shown in Table 7.2. Number of operation types is the distinct types of operations used in the DFG such as addition and multiplication. Scheduling mobility per operation is the slack of all operations averaged over the number of operations in the DFG. This number is an indicator of the flexibility available in scheduling, a higher number indicating more flexibility. All DFGs were analyzed for BIST resource sharing problems and the DFGs were transformed by introducing I-nodes. TOPS [51], a synthesis-for-test tool was used to synthesize data paths from the original and transformed DFGs which were then made self-testable using minimal intrusion BIST. Components designed using a macro-cell library supplied by LSI Logic Corp. were used for synthesis and the area numbers in the tables are given in cell units [52].

The DFG *ex* was used to study the effect of introducing I-nodes incrementally. *ex* has a minimum module requirement of one adder, one multiplier and one subtractor. In Table 7.3, *ex-0* is the original DFG and the rest are transformed versions using redundant operations. The number and type of redundant operations are shown in the column labeled *I-nodes.* Table 7.4 shows the actual areas of the data paths before and after they were made self-testable. The bottom half of the table represents, the same data paths but with multiplexer optimization as described in an earlier section. It can be seen that introduction of one addition I-node in *ex*-1 results in a significant reduction in BIST area overhead and also total area. However, adding one more redundant addition in *ex*-2 does not give any significant improvement over *ex*-1. In *ex*-3, more redundant operations are added which result in a very large reduction in BIST area overhead. However, the multiplexer complexity goes up and the savings in total area are not as significant. The experiments on *ex* indicate that introduction of I-nodes follows the *law of diminishing returns.* Also, as more I-nodes are introduced, multiplexer complexity becomes an overriding factor.

Tables 7.5 and 7.6 show similar data for the *Diffeqn* benchmark. Version *Diffeqn*-1 has one redundant addition and one multiplication as compared to *Diffeqn*-0. The savings of about 6% in area are achieved at the expense of about 22% reduction in the scheduling mobility of the DFG. The data for the *Tseng* DFG is shown in Tables 7.7 and 7.8. For the *AR_filter* (Tables 7.9 and 7.10), after adding 6 redundant computations, the BIST area overhead is reduced significantly (from 16% to 8%) sacrificing 60% of the scheduling mobility. However only about 2% saving in total area is achieved after multiplexer optimization. From Table 7.2, it can be seen that the *AR_filter* has a relatively higher scheduling mobility per operation and

Table 7.2: Comparisons of benchmark DFGs

DFG	Number of Operations	Scheduling Mobility per Operation	Number of Operation Types
ex	4	0	3
Diffeqn	10	0.20	3
Tseng	7	0	5
AR_filter	28	0.36	2

Table 7.3: Characteristics of data paths synthesized from *ex*

DFG Version	L	#Reg	#Mod			#Mux			I-nodes	Scheduling Mobility
			*	+	-	2:1	3:1	4:1		
ex-0	3	4	1	1	1	6	0	0	-	100
ex-1	3	4	1	1	1	5	2	0	1+	100
ex-2	3	4	1	1	1	5	3	0	2+	100
ex-3	4	4	1	1	1	4	4	1	2+, 2*	100

Table 7.4: Area comparisons of data paths synthesized from DFG *ex*

DFG Version	Area before BIST A	BIST overhead B	% BIST overhead $(B/A \cdot 100)$	Total area $C = A + B$	% decrease in total area w.r.t. $C_0^{\dagger}$
ex-0	6812	2272	25.01	9084	-
ex-1	7260	1088	13.03	8348	8.10
ex-2	7580	768	9.20	8348	8.10
ex-3	8188	576	6.57	8764	3.53
same data with multiplexer optimization					
ex-0	6812	2272	25.01	9084	-
ex-1	**7228**	1088	**13.08**	**8316**	**8.46**
ex-2	**7324**	768	**9.49**	**8092**	**10.92**
ex-3	**7900**	576	**6.80**	**8476**	**6.70**

$^{\dagger}C_0$ is the total area of Version ex-0

Table 7.5: Characteristics of data paths synthesized from *Diffeqn*

DFG	L	#Reg	#Mod			#Mux				I-nodes	Scheduling
Version			$*$	+	-	2:1	3:1	4:1	5:1		Mobility
Diffeqn-0	4	6	3	1	1	5	2	2	1	-	100
Diffeqn-1	4	6	3	1	1	10	2	1	0	1+,1$*$	77.78

Table 7.6: Area comparisons of data paths synthesized from DFG *Diffeqn*

DFG Version	Area before BIST A	BIST overhead B	% BIST overhead $(B/A \cdot 100)$	Total area $C = A + B$	% decrease in total area w.r.t. $C_0^{\dagger}$
Diffeqn-0	6033	1072	15.09	7105	-
Diffeqn-1	5916	736	11.06	6652	6.38
same data with multiplexer optimization					
Diffeqn-0	6033	1072	15.09	7105	-
Diffeqn-1	**5900**	736	**11.09**	**6636**	**6.61**

$^{\dagger}C_0$ is the total area of Version *Diffeqn*-0

Table 7.7: Characteristics of data paths synthesized from *Tseng*

DFG	L	#Reg	#Mod					#Mux			I-nodes	Scheduling
Version			$*$	+	-	$\wedge$	$\vee$	2:1	3:1	4:1		Mobility
Tseng-0	4	4	1	2	1	1	1	1	2	1	-	100
Tseng-1	4	4	1	2	1	1	1	2	2	1	1+	100

Table 7.8: Area comparisons of data paths synthesized from DFG *Tseng*

DFG Version	Area before BIST A	BIST overhead B	% BIST overhead $(B/A \cdot 100)$	Total area $C = A + B$	% decrease in total area w.r.t. $C_0^{\dagger}$
Tseng-0	4078	1296	24.12	5374	-
Tseng-1	4094	1024	20.01	5118	4.76
same data with multiplexer optimization					
Tseng-0	4078	1296	24.12	5374	-
Tseng-1	**3998**	1024	**25.62**	**5022**	**6.56**

$^{\dagger}C_0$ is the total area of Version *Tseng*-0

Table 7.9: Characteristics of data paths synthesized from *AR_filter*

DFG Version	L	#Reg	#Mod		#Mux				I-nodes	Scheduling Mobility
			*	+	2:1	3:1	4:1	6:1		
AR_filter-0	8	16	8	4	11	0	4	2	-	100
AR_filter-1	8	16	8	4	25	4	2	2	2+,4*	40.00

Table 7.10: Area comparisons of data paths synthesized from DFG *AR_filter*

DFG Version	Area before BIST A	BIST overhead B	% BIST overhead $(B/A \cdot 100)$	Total area $C = A + B$	% decrease in total area w.r.t. $C_0^{\dagger}$
AR_filter-0	14000	2656	15.95	16656	-
AR_filter-1	15504	1440	8.50	16944	-1.73
same data with multiplexer optimization					
AR_filter-0	14000	2656	15.95	16656	-
AR_filter-1	**14928**	1440	**9.65**	**16368**	**1.73**

$^{\dagger}C_0$ is the total area of Version *AR_filter*-0

Table 7.11: BIST resources for synthesized data paths

Data Path	# Registers	BIST Resources			
		# CBILBO	# BILBO	# TPG	# SA
ex-0	4	1	1	2	0
ex-1	4	0	1	2	1
ex-2	4	0	0	2	2
ex-3	4	0	0	2	1
Diffeqn-0	6	1	1	2	1
Diffeqn-1	6	0	1	3	2
Tseng-0	4	1	3	0	0
Tseng-1	4	0	4	0	0
AR_filter-0	16	0	7	8	1
AR_filter-1	16	0	3	5	2

hence there is not much scope for BIST resource optimization by transforming the behavior.

Tables 7.4, 7.6, 7.8 and 7.10 show the actual BIST area overhead of various data paths. Table 7.11 shows the number of different types of BIST resources in the minimal intrusion BIST solutions of all the synthesized data paths. The cost of the four different types of resources varies significantly (refer Table 3.1). From Table 7.11 it can be seen that significant reduction in BIST area overhead is achieved even if the number of cheaper resources (such as TPG and SA) increases because the number of more expensive resources (such as CBILBO) is decreased.

The experimental results indicate that a few well-chosen I-nodes can reduce BIST resources significantly. Introduction of I-nodes follows law of diminishing returns - as more I-nodes are introduced, the savings gained in BIST area overhead by each introduction decrease significantly as indicated by Table 7.4. The I-transformation is especially useful where scheduling or assignment does not have sufficient degree of freedom. From Table 7.2 it can be seen that *ex*, *Diffeqn* and *Tseng* have a small degree of freedom in scheduling and assignment and hence the I-transformation is very efficient in these cases in terms of BIST area overhead. In terms of number of BIST resources, the *AR_filter* also shows a significant reduction.

7.7 Summary

We have demonstrated how the spare resource capacity available in DFGs can be exploited for reducing BIST resources. A transformation that utilizes the spare resource capacity in a behavior and introduces redundant computations has been described. A technique based on identifying potential testability problems and resolving them by adding I-nodes has been presented. Experimental results demonstrate that very few well-placed I-nodes can significantly reduce the BIST resource requirements of a data path. The transformation is especially useful in cases where there is not enough freedom in the subsequent scheduling and assignment stages to optimize for BIST resources.

Chapter 8

Conclusions

High-level synthesis encompasses a variety of synthesis tasks, such as behavioral transformations, scheduling, allocation and assignment, each of which has the potential to influence a number of design parameters such as area, power and speed. Testability of a design has traditionally been addressed after the design has been synthesized to the logic level, and in recent years, to the RT-level. A need to consider testability during high-level synthesis has clearly been established. It is also widely recognized that testability of a circuits is dependent on the testing methodology adapted. The main contribution of this dissertation is addressing testability during high-level synthesis, while the RTL structure is being designed, for a built-in self-test (BIST) methodology of testing digital circuits. In this chapter we present a summary of the contributions made in this thesis, and discuss some open problems that might stimulate further research. As per the organization of this dissertation, we will discuss the RTL BIST methodology, BIST resource estimation and the three sub-tasks of high-level synthesis individually.

8.1 BIST at RT-level

BIST is a favored test methodology in that it provides support for a broad set of fault models, is conducive to field test, requires short test development and application time and results in a reduced need for expensive automatic test equipment. Two main issues in a BIST methodology are the *quality of test* measured in terms of fault coverage, and the *area overhead* incurred in modifying a design to perform BIST functions. The accepted fault models are defined at the gate-level and hence identification of testability-determining circuit properties that are *independent* of gate level specifics is the main challenge in considering testability during high-level synthesis. Good fault coverage of a module under test using BIST can be ensured by a gate-level implementation of the module that is easily testable using random patterns and appropriate design of BIST registers. What can be ensured at the RT-level for good fault coverage is the existence of BIST registers in a design such that each module can be supplied with uncorrelated pseudo-random patterns at its input ports, and the test responses of each module can be gathered from the output ports and compressed into signatures. Also it is desirable to ensure a very low cost of implementation, which at the RT-level translates to minimizing the number of BIST registers required to ensure the high quality of test.

Taking advantage of the nature of RTL data paths, we proposed a test methodology that is a combination of BIST for the hard-to-test modules in a data path and use of functional patterns for the remaining portions. Based on the combinations of test functionalities required, the BIST methodology uses four types of BIST resources with varying costs. A minimal intrusion BIST methodology was described such that each module gets tested with pseudo-random patterns of high quality with minimal area impact on the whole data path. A 0-1 ILP was formulated for determining which registers to modify to BIST registers and to which *type*, such that the BIST area overhead of a RTL data path is minimum. Testing of the functional modules using minimal intrusion BIST and the rest of the data path using functional test patterns assures a high quality of test for the complete data path at very low cost.

8.2 BIST during High-level Synthesis

By considering optimization of BIST resources during high-level synthesis two objectives are achieved: 1) data paths with lower BIST area overhead and total area are synthesized as indicated by experimental results in the dissertation, and 2) the design cycle time is reduced because iterations due to violation of area constraint after BIST insertion are no longer necessary. One of the most significant aspects of the thesis is categorizing components of a behavior depending on their implication on testability and identifying properties of each synthesis sub-task that affect these components. Variables in a behavior are *potential BIST resources* and contribute to BIST area overhead directly. Operations in a behavior are correlated to modules which form the *logic under test*. Data transfers between variables and operations correspond to interconnection paths that are potential *test paths*. Previous research that addressed BIST area overhead did not make this distinction in searching the design space. In this thesis, we have shown the importance of this categorization in determining the contribution of each synthesis sub-task to minimizing BIST resources. Depending on the actual behavior, the degree of BIST resource optimization in a specific domain varies. Some behaviors lend themselves easily to BIST resource optimization in the assignment domain, some in the scheduling domain, while others require transformation of the behavior. Together, these techniques provide an efficient way of cutting down design cycle time and minimizing BIST resource overhead and total area of synthesized data paths.

8.2.1 Estimation of BIST Resources

Estimation plays a central role in a synthesis environment. The ability to estimate design parameters during synthesis enables 1) efficient exploration of the design space, and 2) different synthesis approaches and algorithms to be compared on a common basis. Estimation of *functional resources* among other functional design parameters has been well studied in the past. This thesis is the first one to address estimation of *test resources*. We have developed a theory for estimation of lower bounds on the number of BIST resources required to test a data path. The concepts of storage concurrency and maximal independent module sets have been developed. Storage concurrency relates to variables that contribute to potential BIST resources and maximal independent operation set relates to the modules under test. These two basic concepts are the underlying theme in the thesis. The lower bounds can be computed for a schedule and a module assignment or even partial module assignments and partial register assignments. Efficient algorithms for computing the lower bounds have been designed.

8.2.2 Register and Interconnect Assignment

The hardware assignment task in high-level synthesis comprises three subtasks: 1) assignment of operations to functional modules (module assignment), 2) assignment of variables to registers (register assignment), and 3) assignment of data transfers to interconnect and appropriate ports of functional modules (interconnect assignment). Register and interconnect assignment affect BIST resources *directly* in that the objective of these tasks synthesizes the BIST resources and the test paths. Module assignment, on the other hand, affects BIST resources indirectly by synthesizing a configuration of modules, from the numerous possibilities, that will require few BIST resources. Hence we separate the assignment tasks and consider register and interconnect assignment together, while module assignment is performed simultaneously with scheduling.

All variables in a behavior are treated in a homogeneous manner by traditional assignment techniques since all of them are assumed to contribute to storage of only *functional data*. We have developed a classification of variables into different types of *test variables* with the objective of keeping track of how registers acquire BIST functionalities as different variables are assigned to them. The classification is based on the relation of variables to inputs and outputs of modules. We have identified two properties of data paths that contribute to reduction of BIST area and are influenced by register and interconnect assignment: 1) large number of registers that share BIST functions of different modules, and 2) low number of expensive BIST registers (CBILBO) that are *essential* to test any module. Previous work in assignment made simplistic assumptions for requirement of CBILBO in data paths, such as the presence of self-adjacent registers. We have identified exact assignment conditions that identify structures that require CBILBOs. A register assignment algorithm based on a greedy optimal algorithm has been developed. The effect of interconnect assignment on these two properties has been integrated into an interconnect assignment algorithm. Experimental results show that our assignment technique, assigns a minimum number of registers, assigns them in such a way that all modules can be tested with a low BIST area overhead.

8.2.3 Scheduling and Module Assignment

In Chapter 4 we showed that a module assignment with a maximal independent operation set that has very low input and output storage concurrency would require very few BIST resources. In register and interconnect assignment, the key concept used was *storage concurrency* of some specific sets of variables. The key concept considered during module assignment is *mutually independent* operations. Module assignment is *tightly coupled* with scheduling. This, along with the fact that module assignment is inherently different than register and interconnect assignment for BIST resource minimization, makes our proposed combined scheduling and module assignment approach very effective. We have demonstrated how scheduling affects BIST resources in two ways: 1) it determines lifetimes of variables and hence affects the register assignment solution space; and 2) it determines temporal sequence of operations and hence affects module assignment solution space. A 2-phase scheduling technique was developed that synthesizes schedules that are BIST resource-efficient or are such that they are amenable to register assignment to minimize BIST resources. In previous research the scheduling stage of synthesis was not investigated for reducing testability area overhead. We have shown that BIST resource optimization of some benchmarks is done most effectively during the scheduling stage of synthesis, while others lend themselves to BIST resource optimization during assignment.

8.2.4 Redundancy and BIST Resources

The degree of freedom that can be exploited during scheduling and assignment to minimize BIST resources can be limited in some behaviors. In Chapter 7 we have identified a property, namely, *redundancy* of operations in a behavior that can be exploited in some cases to reduce BIST resources. We have proposed a transformation technique that introduces redundant computations in a behavior in such a way that the synthesized data path is testable using few BIST resources. We introduced the concept of an *Identity-node* and a transformation called *I-transformation* that makes use of the *spare capacity* of modules to add redundancy that enables test paths to be shared among the modules. As we move to higher levels of abstraction, the number of design options increase exponentially and it is critical to identify those characteristics of a behavior that impact testability the most. Using probabilistic estimates we can find "bottlenecks" in a behavior that hinder sharing of BIST resources. A technique for identifying such bottlenecks in a behavior and resolving them by introducing redundant computation was developed. Introduction of redundant computations is performed without compromising the latency and functional resource requirement of the behavior. Transformation of a behavior has been studied for objectives such as latency, functional area, fault tolerance, power, test generation complexity and partial scan overhead. In this thesis we have addressed the objective of BIST area overhead via transformations. Experimental results show that the I-transformation is effective in reducing BIST resources, especially in behaviors where the degree of freedom in scheduling and assignment is very limited.

8.3 Future Work and Extensions

This thesis has addressed BIST resource considerations during all stages of high-level level synthesis. The scope for future work extends along the two dimensions of this work, namely, *high-level synthesis* and *testability.*

8.3.1 High-level Synthesis

The synthesis model used in this work assumes single-cycle operations, individual register storage and point-to-point multiplexer connectivity. The model can be extended to include multi-cycle and chained operations. In the *multi-cycling* model, a single operation can be performed over multiple control steps and in the *chaining* model, multiple operations can be chained in one control step. The basic concepts of this thesis can be used for these models of operation execution. In the case of multi-cycling, the same register and interconnect assignment techniques can be used, but the scheduling and module assignment would necessitate modifications. For chained operations, the register and interconnect assignment also would be affected since some variables are not required to be stored in registers. In the case of multi-cycling, the test control circuitry has to be modified to ensure that the generation of test patterns and collection of test responses for the multi-cycle operations is performed with the correct period for the clocks feeding the BIST registers in the test mode.

The storage model can be extended to the register file model, in which registers are grouped into files that can be accessed through single (or multiple) read and write ports. For a single port register file, only one register from a group of registers assigned to the register file can be accessed. This adds the constraint that registers that are selected to generate test patterns for each input port and a register that is selected to compress test responses of a module have to reside in distinct register files. In the case of multiple port register files, this constraint does not exist if the number of ports of a register file is equal to or greater than the maximum number of ports of a module in the data path.

An alternative connectivity model used in high-level synthesis is a bus-based model. It has its advantages and disadvantages. A bus-based model increases global interconnect area, but also increases sharing of interconnection paths between modules and registers. Complex arbitration circuitry and tri-state drivers are required for the proper functioning of buses. Usually, based on the parallelism desired, multiple buses are used in a data path. In such a case, the problem of sharing of BIST resources between modules translates to a problem of sharing of buses between different modules because the registers connected to the shared buses can be shared as BIST resources between modules.

8.3.2 Testability

With regards to the BIST methodology that we have chosen, there are two issues that need to be addressed in future work. The choice of BIST resources affects the *test application time* and the *test control complexity*. If two modules share $\mathcal{G}$-resources they can receive test stimulus concurrently. However, if they share a $\mathcal{C}$-resource they cannot compress test responses concurrently because the test paths are exclusive paths through a multiplexer. Hence sharing of $\mathcal{C}$-resources decreases test concurrency and increases test application time. The trade-off between BIST resource sharing and test application time needs to be studied. In this work we have focussed on data path synthesis. Considerations of BIST during data path synthesis affect functional control as well as test control complexity. Functional control synthesis has been well characterized in past research. However, effect of synthesis on test control complexity needs to be studied.

The BIST model for RTL data paths can be extended to include the concept of *I-paths* through functional modules [75]. In this thesis test paths from registers to modules and from modules to registers through only multiplexers are considered. Most functional modules in RTL data paths have an identity mode in which patterns at one input port can be transported to the output port without any change if the other input port is held at the identity value of the module. Such test paths can enrich the BIST solution space and reduce BIST resources further though at the expense of test control complexity.

This thesis has demonstrated the importance of considering *BIST* during high-level synthesis. With regards to high-level synthesis for testability in general, several other testability criteria and test methodologies remain a rich area to be explored.

Reference List

[1] *The National Technology Roadmap for Semiconductors.* Technical Report, Semiconductor Industry Association (SIA), San Jose, California, December 1997.

[2] V.D. Agrawal and S.C Seth. *Test Generation for VLSI Chips.* IEEE Computer Society Press, 1988.

[3] T.W.Williams and K.P. Parker. Design for Testability - A Survey. In *Proc. of the IEEE*, pages 98–112, January 1983.

[4] R. Gupta, R. Gupta, and M.A. Breuer. The BALLAST Methodology for Structured Partial Scan Design. *IEEE Trans. on Computers*, 39(4):538–544, April 1990.

[5] K-T. Cheng and V.D. Agrawal. A Partial Scan Method for Sequential Circuits with Feedback. *IEEE Trans. on Computers*, 39(4):544–548, April 1990.

[6] M. Abramovici, M.A. Breuer, and A.D. Friedman. *Digital Systems Testing and Testable Design.* IEEE Press, 1990.

[7] E.J. McCluskey. Built-in Self-test Techniques. *IEEE Design and Test of Computers*, pages 21–28, 1985.

[8] K.M. Kavi and B.P. Buckles. A Formal Definition of Data Flow Graph Models. *IEEE Trans. on Computers*, 35(11):940–948, November 1986.

[9] A.V. Aho, R. Sethi, and J.D. Ullman. *Compiler Principles, Techniques and Tools.* Addison-Wesley Publishing Company, 1986.

[10] A.P. Chandrakasan, M. Potkonjak, J. Rabaey R. Mehra, and R. Brodersen. Optimizing Power Using Transformations. *IEEE Trans. on Computer-Aided Design*, 14(1):12–31, January 1995.

[11] L. Guerra, M. Potkonjak, and J. Rabaey. High-level Synthesis for Reconfigurable Datapath Structures. In *Proc. Intn'l Conf. on Computer-Aided Design*, pages 26–29, November 1993.

[12] R. Karri and A. Oraiglu. Transformation-based High-level Synthesis of Fault-tolerant ASICs. In *Proc. 29th Design Automation Conf.*, pages 662–665, June 1992.

[13] S. Dey and M. Potkonjak. Transforming Behavioral Specifications to Facilitate Synthesis of Testable Designs. In *Proc. Intn'l Test Conf.*, Oct. 1994.

[14] M. Potkonjak and J. Rabaey. Optimizing Resource Utilization Using Transformations. *IEEE Trans. on on Computer-Aided Design*, 13(3):277–292, March 1994.

[15] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, 1979.

[16] G. D. Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, Inc., 1994.

[17] C.H. Gebotys and M.I.Elmasry. Global Optimization Approach for Architectural Synthesis. *IEEE Trans. on Computer-Aided Design*, 12(9):1266–1278, September 1993.

[18] C-Y. Wang and K.K Parhi. High-level DSP Synthesis Using Concurrent Transformations, Scheduling and Allocation. *IEEE Trans. on Computer-Aided Design*, 14(3):274–295, March 1995.

[19] D.E. Thomas, E.D. Lagnese, R.A. Walker, J.A. Nestor, J.V. Rajan, and R.L. Backburn. *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench.* Kluwer Academic Publishers, 1990.

[20] A.C. Parker, J.T. Pizarro, and M. Mlinar. MAHA: A Program for Datapath Synthesis. In *Proc. 23rd Design Automation Conf.*, pages 496–499, June 1986.

[21] P. Michel, U. Lauther, and P. Duzy. *The Synthesis Approach to Digital System Design.* Kluwer Academic Publishers, 1992.

[22] D. Gajski, N. Dutt, A. Wu, and S. Lin. *High-level Synthesis.* Kluwer Academic Publishers, 1992.

[23] R.A. Bergamaschi, R.A. O'Connor, L. Stok, M.Z. Moricz, S. Prakash, A. Kuehlmann, and D.S. Rao. High-level Synthesis in an Industrial Environment. *IBM J. Res. Develop., Vol.39*, pages 131–148, January/March 1995.

[24] J. Biesenack, M. Koster, T. Langmaier, S. Ledeux, S.Marz, M.Payer, M.Pilsl, S.Rumler, H.Soukup, A. Stoll, N. Wehn, and P. Duzy. The Siemens High-level Synthesis System CALLAS. *IEEE Trans. on VLSI Systems*, 1(3):244–253, September 1993.

[25] J. Vanhoof, K.V. Rompaey, I. Bolsens, G. Goossens, and H.De Man. *High-level Synthesis for Real-time Digital Signal Processing.* Kluwer Academic Publishers, 1993.

[26] C.-J. Tseng, R.-S. Wei, S.G. Rothweiler, M.M Tong, and A.K. Bose. Bridge: A Versatile Behavioral Synthesis System. In *Proc. 25th Design Automation Conf.*, pages 415–420, July 1988.

[27] R.W. Hunter, T. Fuhrman, and D.E. Thomas. Working Chips from High-level Synthesis: A Case Study from Industry. In *Proc. of the IEEE Custom Integrated Circuits Conf.*, pages 144–147, May 1994.

[28] T. Fuhrman. Industrial Extensions to University High-level Synthesis Tools: Making it Work in the Real World. In *Proc. 28th Design Automation Conference*, pages 520–525, June 1991.

[29] B. Tuck. Finally, Behavioral Synthesis is Production-ready. *Computer Design*, pages 57–63, July 1997.

[30] P.G. Paulin. DSP Design Tool Requirements for the Nineties: An Industrial Perspective. In *Proc. 6th Intn'l Workshop on High-level Synthesis*, November 1992.

[31] E. Roza, J. Biesterbos, B. De Loore, and J. Van Meerbergen. On the Application of Architectural Synthesis in the Design of High-volume Production ICs for Consumer Applications. In *Proc. 6th Intn'l Workshop on High-level Synthesis*, pages 2–15, November 1992.

[32] P.H. Bardell, W.H. McAnney, and J. Savir. *Built-in Test for VLSI: Pseudorandom Techniques.* John Wiley & Sons, 1987.

[33] T-C. Lee, W.H. Wolf, N.K. Jha, and J.M. Acken. Behavioral Synthesis for Easy Testability in Data Path Allocation. In *Proc. Intn'l Conf. Comp. Design*, pages 29–32, 1992.

[34] T-C. Lee, W.H. Wolf, and N.K. Jha. Behavioral Synthesis for Easy Testability in Data Path Scheduling. In *Proc. Intn'l Conf. Computer-aided Design*, pages 616–619, 1992.

[35] T. Lee, N. Jha, and W. Wolf. Behavioral Synthesis of Highly Testable Datapaths under the Non-scan and Partial Scan Environments. In *Proc. 30th Design Automation Conf.*, pages 292–297, June 1993.

[36] S. Bhatia and N.K. Jha. Genesis: A Behavioral Synthesis System for Hierarchical Testability. In *Proc. Euoropean Design Automation Conf.*, pages 272–276, February 1994.

[37] S. Bhatia and N.K. Jha. Behavioral Synthesis for Hierarchical Testability of Controller/Data Path Circuits with Conditional Branches. In *Proc. Intn'l Conf. Computer-Aided Design*, Oct. 1994.

[38] A. Mujumdar, K. Saluja, and R. Jain. Incorporating Testability Considerations in High-level Synthesis. In *Proc. FTCS*, pages 272–279, 1992.

[39] S. Dey, M. Potkonjak, and R.K. Roy. Synthesizing Designs with Low-Cardinality Minimum Feedback Vertex Sets for Partial Scan Application. In *Proc. VLSI Test Symp.*, pages 2–7, April 1994.

[40] S-P Lin, C.A. Njinda, and M.A. Breuer. A Systematic Approach for Designing Testable VLSI Circuits. In *Proc. Intn'l Conf. on Computer-Aided Design*, pages 496–499, November 1991.

[41] I.G Harris and A. Orailoğlu. Microarchitectural Synthesis of VLSI Designs with High Test Concurrency. In *Proc. 31st Design Automation Conf.*, pages 206–211, June 1994.

[42] I.G. Harris and A. Orailoğlu. SYNCBIST:SYNthesis for Concurrent Built-In Self-Testability. In *Proc. Intn'l Conf. Comp. Design*, pages 101–104, October 1994.

[43] A. Orailoğlu and I.G. Harris. Microarchitectural Synthesis for Rapid BIST Testing. *IEEE Trans. on Computer-Aided Design*, 16(6):573–586, June 1997.

[44] L. Avra. Allocation and Assignment in High-level Synthesis for Self-testable Data Paths. In *Intn'l. Symp. on Circuits and Systems*, pages 463–472, August 1991.

[45] L.T. Wang and E.J. McCluskey. Concurrent Built-In Logic Block Observer (CBILBO). In *Intn'l. Symp. on Circuits and Systems*, pages 1054–1057, 1986.

[46] L. Avra. Orthogonal Built-in Self-test. In *Proc. COMPCON'92*, pages 452–457, 1992.

[47] L. Avra and E.J. McCluskey. Synthesizing for Scan Dependence in Built-in Self-testable Designs. In *Proc. Intn'l Test Conf.*, pages 734–743, 1993.

[48] C. Papachristou, S. Chiu, and H. Harmanani. A Data Path Synthesis Method for Self-Testable Designs. In *Proc. 28th Design Automation Conf.*, pages 378–384, June 1991.

[49] H. Harmanani and C. Papachristou. An Improved Method for RTL Synthesis with Testability Tradeoffs. In *Proc. Intn'l Conf. on Computer-Aided Design*, pages 30–35, November 1993.

[50] C. Papachristou, S. Chiu, and H. Harmanani. A Data Path Synthesis Method for Self-Testable Designs. In *Proc. 28th Design Automation Conf.*, pages 378–384, June 1991.

[51] L.J. Avra, L. Gerbaux, J-C. Giomi, F. Martinolle, and E.J. McCluskey. *A Synthesis-for-Test Design System.* Tech. Report CSL TR 94-622, Computer Systems Laboratory, Stanford University, May 1994.

[52] Data Book. *G10-p Cell-Based ASIC Products.* LSI Logic Corp., May 1996.

[53] L. Schrage. *LINDO: An Optimization Modeling System.* Scientific Press, 1991.

[54] N. Dutt and C. Ramachandran. *Benchmarks for the 1992 High-level Synthesis Workshop.* Tech. Report 92-107, Univ. of California, Irvine, 1992.

[55] R. Jain, A.C. Parker, and N. Park. Predicting System-Level Area and Delay for Pipelined and Non-pipelined Designs. *IEEE Trans. on Computer-Aided Design*, 11(8):955–965, August 1992.

[56] M. Rim and R. Jain. Estimating Lower-Bound Performance of Schedules Using a Relaxation Technique. In *Proc. Intn'l Conf. Comp. Design*, pages 290–294, October 1992.

[57] Y. Hu, A. Ghouse, and B.S. Carlson. Lower Bounds on the Iteration Time and the Number of Resources for Functional Pipelined Data Flow Graphs. In *Proc. Intn'l Conf. Comp. Design*, pages 21–24, October 1993.

[58] A. Sharma and R. Jain. Estimating Architectural Resources and Performance for High-Level Synthesis Applications. *IEEE Trans. on VLSI Systems*, 1(2):175–190, June 1993.

[59] S. Chaudhari and R.A. Walker. Computing Lower Bounds on Functional Units before Scheduling. In *Proc. 7th Intn'l Symp. on High-level Synthesis*, pages 36–41, May 1994.

[60] S.Y. Ohm, F.J. Kurdahi, and N. Dutt. Comprehensive Lower Bound Estimation from Behavioral Descriptions. In *Proc. Intn'l Conf. Computer-Aided Design*, pages 182–187, October 1994.

[61] P.G. Paulin and J.P. Knight. Force-Directed Scheduling for the Behavioral Synthesis of ASICs. *IEEE Trans. on Computer-Aided Design*, 8(6):661–679, June 1989.

[62] C. Tseng and D.P. Siewiorek. Automated Synthesis of Data Paths in Digital Systems. *IEEE Trans. on Computer-Aided Design*, 5(7):379–395, July 1986.

[63] R. Jain, M. Mlinar, and A. Parker. Area-time Model for Synthesis of Non-pipelined Designs. In *Proc. Intn'l Conf. on Computer-Aided Design*, pages 48–51, November 1990.

[64] S.Y. Kung, H.J. Whitehouse, and T. Kailath. *VLSI and Modern Signal Processing*. Prentice-Hall, 1985.

[65] K. Küçükçakar and A. Parker. Data Path Trade-offs using MABAL. In *Proc. 27th Design Automation Conf.*, pages 511–516, June 1990.

[66] B.M. Pangrle. On the Complexity of Connectivity Binding. *IEEE Trans. on Computer-Aided Design*, 10:1460–1465, 1991.

[67] M.C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, 1980.

[68] D.L. Springer and D.E. Thomas. Exploiting the Special Structure of Conflict and Compatibility Graphs in High-Level Synthesis. In *Proc. Intn'l Conf. on Computer-Aided Design*, pages 254–257, November 1990.

[69] F. Gavril. Algorithms for Minimum Coloring, Maximum Clique, Minimum Covering by Cliques, and Maximum Independant Set of a Chordal Graph. *SIAM J. Computing*, pages 180–187, June 1972.

[70] S. Devadas and A.R. Newton. Algorithms for Hardware Allocation in Datapath Synthesis. *IEEE Trans. on Computer-Aided Design*, 8(7):768–781, July 1989.

[71] R.J. Clotier and D.E. Thomas. The Combination of Scheduling, Allocation and Mapping in a Single Algorithm. In *Proc. 27th Design Automation Conf.*, June 1990.

[72] L. Stok. *Architectural Synthesis and Optimization of Digital Systems*. PhD thesis, Eindhoven University, The Netherlands, 1991.

[73] N. Park and A.C. Parker. SEHWA: A Program for Synthesis of Pipelines. In *Proc. 23rd Design Automation Conf.*, pages 454–460, July 1986.

[74] A. Karasniewski. Can Redundancy Enhance Testability? In *Proc. Intn'l Test Conf.*, pages 483–491, Oct. 1991.

[75] M.S. Abadir and M.A. Breuer. A Knowledge-Based System for Designing Testable VLSI Chips. *IEEE Design & Test of Computers*, pages 56–68, August 1985.

[76] R. Moreno, R. Hermida, and M. Fernandez. Short Note: Register Estimation in Unscheduled Dataflow Graphs. *ACM Trans. on Design Automation of Electronic Systems*, 1(3):396–403, July 1996.