

378
C.1

CONCURRENT BIST COST ESTIMATION DURING DATA PATH ALLOCATION

by

MAYA KODEIH

Submitted in partial fulfillment of the requirements
for the Degree of Master of Science

Thesis Advisor: Dr. Haidar M. Harmanani

Department of Computer Science
LEBANESE AMERICAN UNIVERSITY
November 2002

LEBANESE AMERICAN UNIVERSITY

GRADUATE STUDIES

We hereby approve the thesis of

MAYA KODEIH

candidate for the *Master of Science* degree*.

(signed)

(chair) Haider M. Harmouni

Dr. Nashaat Mansour

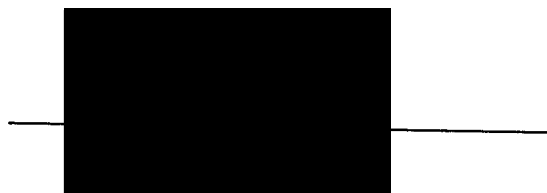
Dr. Walid Kairouz

Date:

Feb 27, 2003

**We also certify that written approval has been obtained for any proprietary material contained therein.*

I grant to the LEBANESE AMERICAN UNIVERSITY the right to use this work, irrespective of any copyright, for the University's own purpose without cost to the University or to its students, agents and employees. I further agree that the university may reproduce and provide single copies of the work, in any format other than in or from microforms, to the public for the cost of reproduction.



CONCURRENT BIST COST ESTIMATION DURING DATA PATH ALLOCATION

ABSTRACT

by

MAYA KODEIH

The increase in density that the advent of Very Large Scale Integration (VLSI) has allowed made the move to higher levels of design abstraction imperative. *High Level Synthesis* emerged as a result; however, most solutions 1) were not optimal; 2) did not incorporate testing at the system level. Recently, a new trend in high-level synthesis has emerged, with researches being aware of the importance of testability at the system level.

In this work we introduce a method for concurrent BIST cost estimation during testable data path allocation. A basic feature of this method is the integration of testability in the design process. The main objective is to develop a system-level synthesis tool set mapping a behavioral description onto an optimized and testable RTL design subject to user-defined constraints. The method considers test points cost concurrently with design cost. In order to measure the performance of our method, we use six data flow graphs which are widely adopted for benchmarking high-level BIST synthesis.

To my husband and children

ACKNOWLEDGMENTS

I would like to thank my Thesis advisor Haidar Harmanani for his guidance throughout my M.S. studies. Thanks is also due to Dr. Nashaat Mansour and to Dr. Walid Kairouz for being on my Thesis committee.

I would like to express my sincere gratitude to the Lebanese American University whose financial support during my graduate studies made it all possible.

Finally, I would like to thank my family for their patience and their long support.

Contents

1. Introduction.....	1
1.1. Synthesis: From Concept to Silicon.....	2
1.2. The Design and Test Process.....	5
1.2.1. Scheduling in High-Level Synthesis.....	6
1.2.2. Allocation in High-Level Synthesis.....	8
1.2.3. Modeling vs. Design.....	9
1.2.4. Testing for VLSI Built-In Self-Test.....	10
1.3. Illustrating High-Level Synthesis using An Example.....	12
1.4. Design For Testability.....	16
1.5. Built-In Self Test.....	18
1.6. Design and Test Tradeoffs.....	19
2. Review of Literature.....	21
2.1. Allocation Techniques.....	21
2.2. Allocation Technique Reducing Area Overhead.....	23
2.3. Allocation Techniques Based on ILP.....	24
3. Concurrent BIST Cost Estimation.....	28

3.1. Testable Data Path Synthesis.....	29
3.1.1. BIST Methodology.....	29
3.1.2. Data Flow Synthesis.....	29
3.1.3. Module Allocation Graph.....	30
3.1.4. Resources Allocation with Testability Consideration.....	30
3.2. Concurrent BIST Points Selection.....	31
3.2.1. Testability Tradeoff Model at the System Level.....	31
3.2.2. Data Path Representation.....	33
3.2.3. Pseudo-Merge of two TFBs.....	34
3.2.4. Select Input Registers.....	34
3.2.5. Select Output Registers.....	34
3.2.6. Self-Loop Breaking.....	36
4. Experimental Results.....	40

4.1. Background.....	40
4.2. 6 th Order FIR Filter.....	42
4.3. 3 rd Order IIR Filter Cascade Connection.....	43
4.4. 6-Tap Wavelet Filter.....	45
4.5. 4-point Discrete Cosine Transformation (DCT).....	45
4.6. Tseng.....	46
4.7. Paulin.....	46
4.8. Results.....	47
5. Conclusion.....	52
6. Bibliography.....	53

List of Figures

1.1 Design abstraction levels.....	4
1.2 A hypothetical Silicon Compiler Design Flow.....	4
1.3 Scheduled Data Flow Graph	7
1.4 Top-down design flow.....	10
1.5 Input behavioral specification or algorithmic model.....	13
1.6 a) The data flow representation of the algorithmic model, b) the module library indicating the speed and cost of specific resources.....	14
1.7 An RTL implementation, data-path structure.....	14
1.8 Testing a circuit using BIST.....	18
3.1. (a) Self Testable ALU with Self-Adjacent register, (b) Non-Testable ALU with Self-Adjacent registers.....	29
3.2. Basic model for testability tradeoffs.....	32
3.3. Merging procedure.....	32
3.4. (a)Example DFG, (b) Corresponding MAG.....	33
3.5 Paths that may cause functional self-loops.....	36
4.1. A 6 th order FIR filter.....	42
4.2. Data flow graph for the 6 th order FIR filter.....	43
4.3. Data flow graph of a 3 rd order IIR filter (cascade connection).....	44
4.4. Data flow graph of a 6-tap wavelet filter.....	45
4.5 Data flow graph of a 4-point discrete cosine transformation.....	46
4.6. Data flow graph of Tseng.....	46
4.7 Data flow graph of Paulin.....	47

List of Tables

1.1 State-Table, or "Controller".....	15
4.1. Characteristics of the circuits.....	41
4.2 Number of transistors of 8-bit test registers and multiplexers.....	42
4.3 Results from the FIR filter.....	48
4.4 Results from the IIR3 filter.....	49
4.5 Results from the wavelet 6 filter.....	49
4.6 Results from the DCT4 filter.....	49
4.7 Results from the diffeq Example.....	50
4.8 Results from the diffeq Example with a different binding.....	50
4.9 Results from the Tseng Example.....	50
4.10 Results Comparisons.....	51

Chapter 1

Introduction

Computer design aids for digital systems began as programs performing the routine tasks of bookkeeping. As designs grew, reliable analysis and optimization programs evolved to aid in the design process. This design complexity raised the level of abstraction at which integrated circuit is designed; thus, moving the designer to higher levels, releasing him of the details of the logic and circuit level. High-level synthesis (HLS) of digital systems is one step in the design process. It consists of transforming a behavioral (algorithmic) description of a design into a register transfer level description of the design. The HLS process is divided into the following three subtasks: 1) operation scheduling; 2) resource allocation; 3) resource binding. The *operation scheduling* assigns each operation in the design to a time step in which it will be executed. *Resource allocation* determines the types (e.g., adder, multiplier, or register) and the number of these types of resources that should be included in the design. Finally, *Resource binding* determines which resources should be used to implement each specific operation. Recently, a new trend in high-level synthesis has emerged, with researches being aware of the importance of testability at the system level. In this chapter we discuss the synthesis process in general and high-level synthesis in particular. In addition, we describe design for testability along with our approach to integrate both the design and test process at the system level. We later present the research motivation and the Thesis organization.

1.1. Synthesis: From Concept to Silicon

The design of electronic circuits can be tackled at various levels of abstractions, dealing with designs at different domains. The design process at each domain requires the development of specific tools to support and automate the various stages. This, every electronic system can be described at the *behavioral domain*, *structural domain* and *physical domain*. The behavioral domain describes the intended behavior of the system without any reference to the implementation. The structural domain deals with the system as a hierarchy of functional elements. Finally, the physical domain describes the structure as it is mapped to physical components.

The various design domains are best illustrated using the Y-chart shown in Figure 1.1, first introduced by [GaKu83]. Thus, every design can be described as a point along the three axes, with more abstract levels at the periphery, and all the levels converging to a common center point. Using the Y-chart, we define synthesis as the transition from the behavioral domain to the structural one. Depending on the behavior level of abstraction, the outcome of the synthesis process varies. The input to the synthesis process is described behaviorally, in terms of a hardware language while the output is defined in terms of structural components. Each component is defined by its own behavioral description which can be obtained through synthesis on a lower abstraction level. The ultimate *goal* of the synthesis process is to fully automate the design process – that is the transformation from behavior to structure. A software system that can provide this transformation is called a *silicon compiler*. The design flow of a hypothetical silicon compiler is illustrated in Figure 1.2. The system consists of a "pipeline" of synthesis tasks at various levels of abstractions. Every task is subsequently divided into subtasks which serve as a vehicle to introduce design

steps into synthesis and to provide a top-down design methodology. We distinguish among the following synthesis processes:

- *System level synthesis* which partitions the system into subsystems consisting of a set of communicating concurrent processes together with a behavioral description at the algorithmic level.
- *High-level synthesis* which generates a register-transfer level (RTL) description of a data path and a controller from an algorithmic description that defines the precise procedure for the computational solution of a problem.
- *Logic level synthesis* which generates a gate level hardware from a Boolean equations description. The logic synthesis task includes logic optimization through logic minimization, aiming at minimal area, in terms of number literals.
- *Technology mapping* which generates a physical implementation of an abstract network through library and technology mapping. In general, technology mapping is done at the logic level by covering the network with cells, resulting in different areas and delay values. However, some approaches perform technology mapping at the RTL level, after high-level synthesis; thus mapping RTL components to macro-blocks.

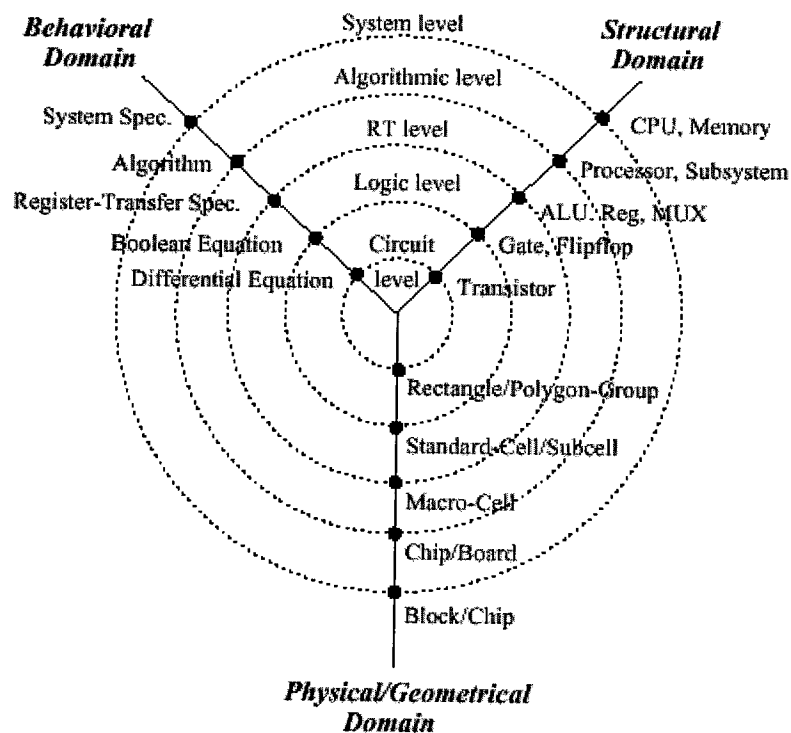


Figure 1.1: Design abstraction levels

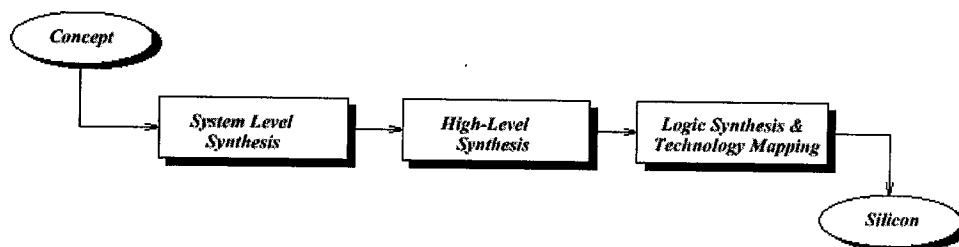


Figure 1.2: A hypothetical Silicon Compiler Design Flow

As the complexity of systems increase, so will the need for synthesis tools and design automation on higher, more abstract levels where functionality and tradeoffs are easier to understand. Note that the terms *specification* and *description* will not be used interchangeably in this Thesis; the first term will be used to describe the

behavior in terms of results while the latter will be used to describe the behavior in terms of procedure.

1.2. The Design and Test Process

Design and test are commonly viewed as being two sides of the same coin [Agra91]. The design process is quite mature at the logic and layout levels due to many existing professional tools for design synthesis and simulation. Due to the complexity in current VLSI technology, the field of high-level synthesis has recently emerged to address the need of design methods and techniques at the register-transfer level (RTL) [McPC90]. The aim of high-level synthesis is the automatic generation of an RTL design (data path/controller) from a behavioral level description, subject to a set of constraints. The actual circuit layout can be later generated using a silicon compiler. Currently, there are several such tools [DeMi90, Jain89, Thom88, Marw86, Raba88]. The behavioral synthesis of digital systems is a computationally intractable problem since most its sub-problems are known to be NP-hard or at least NP-complete. Thus, the synthesis process is further partitioned into subtasks. Two major subtasks in high-level synthesis are *operations scheduling* and *resources allocation*. This complexity means basically that 1) most designs are not optimum; 2) there is often room for improvement by restricting the design space and moving in the right direction. Just as with the design process, the test process, particularly test generation, has matured at the logic and circuit levels. For example, there are several tools for test generation, including some recent ones based on logic synthesis techniques [DAC90]. It has been estimated that the cost of testing and diagnostics goes at a higher rate than a factor of 10 per level [WiPa83]. This makes test considerations and solutions very attractive at the system level and makes design for testability especially important.

Two points have become clear: a) DFT is particularly important at the RTL or system level of the design hierarchy to achieve good test quality [WiPa83], b) DFT increases the chip cost due to the extra silicon area and may imply a performance degradation as well.

Thus, there is a tight relation between design and test, and tradeoffs between both disciplines can be best addressed if they are integrated in a system level design environment. This would result in various solutions and styles under various design constraints in a very short turnaround time. However, traditional design and test methodologies at the system level have always separated the two processes. Test is usually done as a post-design process, i.e., after the completion of the design process. To consider the testing issues only after the design has been completed leads to delays, designs which are hard to test and more area overhead than necessary. We describe next the synthesis process and propose later our view for testability at the system level.

1.2.1. Scheduling in High-Level Synthesis

In this section we talk in more detail about scheduling. Scheduling in high-level synthesis assigns the operators in the DFG to control steps that represent the clock cycles in the final design. The scheduling phase affects greatly the following factors in the final design:

1. *The design timing:* Scheduling fixes the overall timing of the design, illustrated by the maximum number of control steps. This determines the overall design performance.
2. *The number of resources:* While the cost of a design cannot be determined until allocation, the scheduling phase determines a lower bound on the number

of functional units and registers. The lower bound on functional units is the maximum number of a given resource scheduled concurrently at a given time step. The lower bound on the number of registers is the maximum number of data flow transfer which crosses the boundary of a given time step in the schedule.

Thus, one of the tasks of scheduling is to minimize the length of the schedule while minimizing the number of resources. One of the difficulties in scheduling is operators dependency which requires that an operator that produces a value be scheduled before an operator which consumes this value. When the two operators are scheduled in different control steps, this will imply that the result must be stored in a register until it is used. Furthermore, scheduling has to deal with control operations such as conditionals, loops and subroutines. Figure 1.3 shows the transformation of the DFG from Figure 1.6 to a Scheduled data flow graph (SDFG).

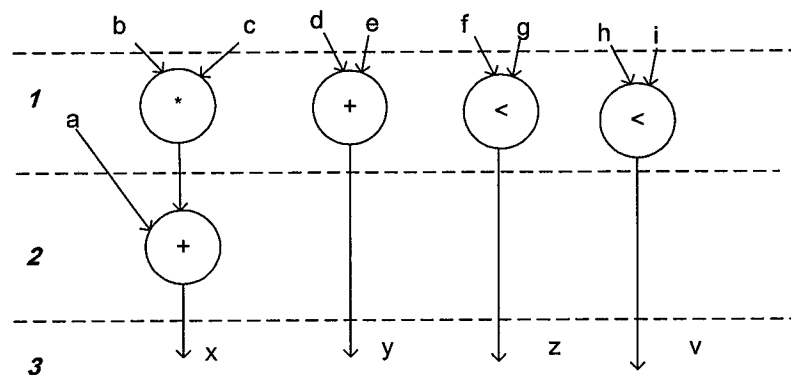


Figure 1.3: Scheduled Data Flow Graph

1.2.2. Allocation in High-Level Synthesis

Data path allocation is concerned with assigning operations and values to hardware so as to minimize the amount of hardware needed. During allocation, registers are allocated for variables, operations are assigned to functional units (FU), and connections which are multiplexers, buses, or a combination of both, are established between them. The allocation phase is constrained by the control step schedule which it implements. Thus, all operators in the scheduled DFG must be bound to ALUs; however, operators which are active simultaneously cannot share the same hardware. In the same token, values that are active across control steps boundaries are stored in registers. There may be additional constraints on the design which limit the total area, total design throughput, or delay.

Allocation techniques can be classified into two categories. The first category is *iterative/constructive* where an operation, value or interconnection to be assigned is selected, and the algorithm then iterates. The other category is based on global allocation that includes *graph heuristic techniques* such as in Facet [TsSi86]; *branch and bound techniques* such as in Mimola [Marw86] and in Splicer [Pang88] where additional heuristics are used to reduce the search space and a trade-off with design quality. Finally, the allocation problem can be formulated as a *mathematical problem* where a variable is created for each possible assignment of an operation, variable, or interconnection to hardware element. Constraints are then formulated, and an objective function that includes area or some other parameters is minimized.

Scheduling and allocation can be accomplished independently like in the Facet system and in MIMOLA or interdependently like in MAHA, HAL, ADPS, and Elf.

1.2.3. Modeling vs. Design

The first representation of a design is called a model, though it could also be called a specification. The reason we call the initial design a model is that it is typically the vehicle for design exploration. The designer may try several different variations of the model. For example, he may try out different implementations of an algorithm, or different combinations of resources. It is at this experimentation stage that the most fundamental decisions about the system are made. Things like how many processors, which algorithms will be used, which ones go in hardware and which in software, characteristics of the memory architecture, data connections, all are fundamental decisions which will shape the eventual implementation of the target design, and which may be made based on information gained from experimentation with the initial model. For this reason the initial model is often called an architectural model.

It is a short conceptual step from architectural model to system implementation. After all, the model is executable, and the target system is executable, so it is just a matter of transforming one into the other. The rub, of course, is in the details. The transformation must conform to the design constraints, which may not be captured in the architectural model. As the constraints are added, as well as the design features those constraints imply, the model becomes a more precise specification and, ultimately, a complete representation of the design. This is the very essence of top-down design (Figure 1.4).

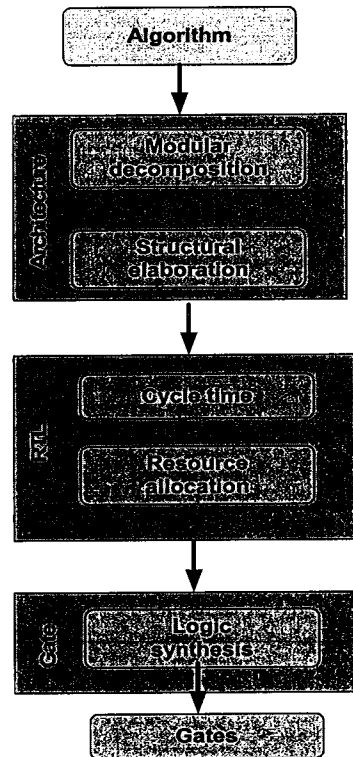


Figure 1.4: Top-down design flow

1.2.4. Testing for VLSI Built-In Self-Test

In very large scale integrated (VLSI) circuits, there exists a large device count and a relatively few input/output pins. This can produce complex structures for which test generation is difficult and results in long tests with high input/output traffic during testing. One approach to dealing with this difficult testing problem is to employ Built-In Self-Test (BIST). In the VLSI environment, desirable goals for BIST are to:

1. Eliminate as much test generation as possible,
2. Permit a fairly general class of failure modes,
3. Permit easy circuit initialization and observation,
4. Reduce input/output pin signal traffic, and
5. Reduce test length.

Although BIST techniques clearly realize a number of the goals listed, for very large circuits with extensive BIST resources, the testing time can still be quite long if the tests for the various parts of the circuits are executed one after the other. In such cases, in order to reduce testing time and fully exploit the power of the BIST resources, it is essential to control the testing process so that full use is made of the potential parallelism available.

In order to develop a perspective for parallelism in BIST, consider the testing of a block of logic within a VLSI chip. The inputs to the block under test (BUT) must be stimulated with an appropriate input sequence including initialization steps. The outputs of the BUT must be observed and the response analyzed to determine if the block is faulty or not. The observation of the response must typically be coordinated with the application of the input sequence.

In the typical BIST implementation for testing a block of logic, the original source of the stimuli is a set of one or more test pattern generators (TPG) and the final destination of the responses is a set of one or more compressors and/or response analyzers. It is possible that the test generators and response compressors and/or analyzers are directly attached to the block under test. Often, however, there is additional logic lying between the test generators and the BUT and between the BUT and the response compressors/analyzers. Thus, test control logic must which controls not only the test pattern generators and response compressors/analyzers but also this intervening logic. Typically, paths must be established from the test pattern generators to the inputs of the BUT and from the BUT to the response compressors/analyzers. In addition, the test control logic must interact with a higher level of control either on or off the chip. The blocks which are required to perform a test (test control logic, TPGs, compressors/analyzers, BUT, and any intervening logic)

are known as *test resources*. Test resources may be shared among BUTs. For example, testing schemes exist in which the response compressor for one BUT can be used as an input stimulus, i.e., as a TPG, for another BUT. Also for those blocks which lie on the periphery of the chip, a portion of the test resources may lie off-chip.

1.3. Illustrating High-Level Synthesis using An Example

High level synthesis generates register-transfer level designs from behavioral specifications, in an automatic manner. Traditional high-level synthesis tools accept as input:

1. An algorithmic description, or a behavioral specification, that is usually represented in text by an infinite loop containing a series of procedural statements to those that might be found in a "C" program. Two popular languages that are used for algorithmic level hardware descriptions are Verilog and VHDL.
2. Design constraints such as cost constraints, performance constraints, power consumption constraints, pin count or testability constraints, etc.
3. An optimization function.
4. A module library representing the available components at RTL.

As output, synthesis tools produce:

1. A register transfer level implementation that consists of a set of data-path resources (e.g., functional-units, registers, and multiplexers), interconnections between them, and a state-table to indicate the function that each resource is to perform at any given time.
2. A controller captured usually as a symbolic FSM.
3. Other attributes, such as geometrical information.

The goal of high level synthesis is to generate an RTL design that implements the specified behavior while satisfying the design constraints and optimizing the given cost function. The algorithmic description or behavioral specification is represented by a procedural and functional language, and is also represented by graphics notations as shown in Figure 1.5.

```
PROCEDURE Test;  
VAR  
a,b,c,d,e,f,g,h,I,x,y,z,v  
BEGIN  
Read(a,b,c,d,e,f,g,h,I,x,y,z,v)  
always  
x := a + (b * c)  
y := (d + e)  
z := ( f < g)  
v := (h < i)  
END;
```

Figure 1.5: Input behavioral specification or algorithmic model.

For easier machine representation and manipulation, the textual algorithmic specification is typically transformed into a data-flow graph form prior to synthesis. A simple example to illustrate the algorithmic, register transfer and intermediate data-flow graph representation is shown in Figure 1.6.

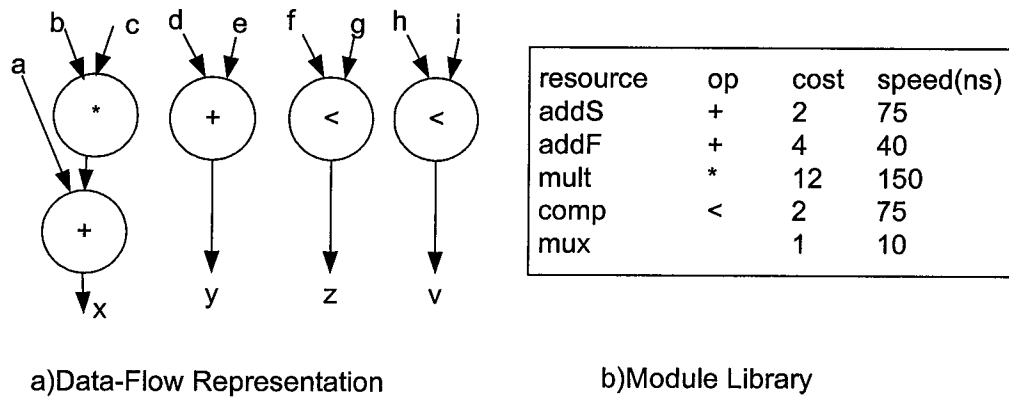


Figure 1.6: a) The data flow representation of the algorithmic model, b) the module library indicating the speed and cost of specific resources.

Figures 1.5 and 1.6(a) contain the same information but the data-flow representation is more graphical and is therefore easier to read. It also eliminates high-level language constructs. Figure 1.6(b) is a primitive module library that indicates the presence of two adders, an expensive fast adder and a slower less expensive adder.

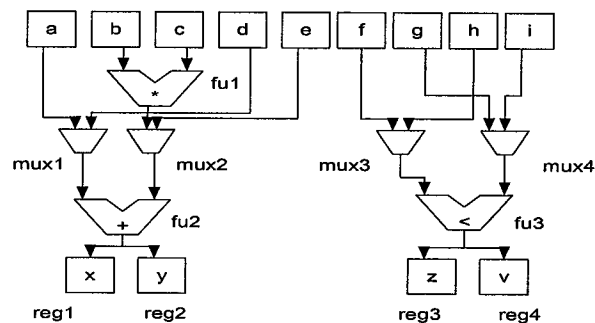


Figure 1.7: an RTL implementation, data-path structure.

The register transfer level implementation shown in Figure 1.7 is only one of many possible that properly implements the algorithmic specification using resources from the given primitive module library. For example, it would be possible to remove all multiplexers from the design by allocating an extra adder and an extra comparator.

The above register transfer level model implements the algorithmic specification in the following way. In step 1, fu1 begins to multiply operands b and c while fu2 adds operands d and e and fu3 compares operands f and g. To route the proper operands to functional units, fu2 and fu3, the multiplexers must be configured as shown in the first row of the state-table, see Table 1.1. Also in step 1, registers reg2 and reg3 must be loaded with the new values of y and z coming out of fu2 and fu3 respectively. In step 2, the multiply operation finishes (it requires more than one step) and the output is routed to fu2 via mux2 to be added to a that is being routed to fu2 via mux1. Also in step2, fu3 is now comparing h and I because the multiplexers have been re-configured as shown in row 2 of Table 1.1 to route these values to its inputs. Finally, in step 2, registers reg1 and reg4 are loaded with the new values for x and v that are coming out of fu2 and fu3 respectively.

Step	Next Step	Fu1	Fu2	Fu3	Mux1	Mux2	Mux3	Mux4	Reg1	Reg2	Reg3	Reg4
1	2	*	+	+	Right	Right	Left	Left	hold	load	load	hold
2	1	*	+	+	Left	Left	Right	Right	load	hold	hold	load

TABLE 1.1: State-Table, or "Controller".

Suppose our objective is to find a minimum cost design for a 2-step schedule length. According to the given primitive module library, the above implementation satisfies this objective. Thus there are two important goals for traditional high-level synthesis tools. First, the tool must construct a register transfer level implementation that complies with the specification. A *compliant* register transfer level design is one that implements the functionality specified in the algorithmic description without violating any design rules – an example design rule is that functional-units can perform at most one operation in a given control-step. Secondly, a good high-level synthesis tool attempts to optimize some objective function that includes factors such

as area, performance, power, etc. The high-level synthesis problem is typically divided into a number of sub-tasks which, though each can be performed independently, are highly interrelated in terms of their affect on overall design quality. Some of the common sub-tasks are:

- *Scheduling*: The scheduling problem is to place operations from the data-flow graph into specific control-steps to satisfy data-dependencies and, sometimes, global resource constraints. Scheduling was discussed in detail in section 1.2.1.
- *Module Selection*: The problem of module selection is to determine which type of resource from the primitive module library will be used to implement each type of operation in the graph.
- *Allocation*: The allocation problem is to determine how many instances of each type of resource will be required, sometimes in response to global performance constraints. The resources that are allocated include interconnection components (multiplexers, busses) and registers, as well as functional-units.
- *Binding*: The binding problem is to choose which of the allocated resources will be used to implement operations from the data-flow graph, and which registers and interconnections will be used to store and transport values.

1.4. Design For Testability

The success with test automation depends on testability. There are two aspects in testability, *visibility* and *control*. Visibility is our ability to observe the states and outputs of the software under test. Control is our ability to provide inputs and reach

states in the software under test. If we take a broader look at testability we find other aspects within the system to test that have to be ensured in order to have good testability:

- *Operability* should be maintained. The better the system works, the more efficiently it can be tested.
- *Controllability* of the system is another aspect that we should maintain in order to have good testability. It means that the input terminals or devices of the circuit under test can control the output.
- *Observability* has to be maintained as well. It means that the circuit under test can be observed at some output terminals or devices. Observability ensures that what we see is what we get.
- *Simplicity* of the design has to be ensured. The less there is to test the more quickly we can test it.
- *Understandability* of the design also has to be kept. The more information we have the smarter we test.
- *Suitability* of the design should be ensured as well. The more we know about the intended use of the software, the better we can organize our testing to find important bugs.
- *Stability* of the system is another addition that we can have. The fewer the changes, the fewer the disruptions to testing.

Testability Features are features that have been used to improve the visibility or controllability of the software. *Control features* include methods such as *exception seeding* (Instrument low level I/O code to simulate errors), *test points*, or

automatically filing memory. Test points allow data to be inspected, inserted or modified at points in the software.

Digital testing is concerned with revealing physical defects in circuits by applying test patterns to the circuit under test (CUT) and verifying the test responses (Figure 1.8). Three main issues are important during testing. The first issue is the *fault model* adopted to reveal the faults. The most common and simplistic *fault model* is the single *stuck-at* fault model. The second issue is concerned with *test pattern generations* which could be *deterministic*, *pseudorandom* or *exhaustive*. The last issue is the *test quality* which is usually referred to as *fault coverage*. Fault coverage is the percentage of modeled faults covered by the applied patterns and it is quantified by *fault simulation*.

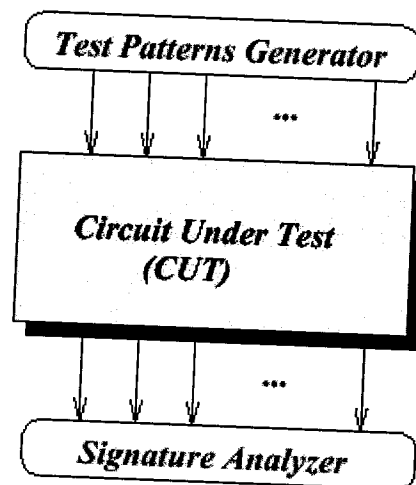


Figure 1.8: Testing a circuit using BIST

1.5. Built-In Self Test

The complexity in VLSI design process complicated the testing process of such systems as well. External equipments were used to generate test patterns which would be fed to special test input pins of the chip, and the responses would be

collected from output pins to be analyzed again by external equipments. However, the shrinking in design rules, and the significant increases in density and complexity in VLSI devices made the accessibility to the circuit very hard and made the traditional test generations and application methods very costly. Design For Testability (DFT) was an initiative in the computer hardware industry in the 1980's. It is part of a larger effort called error detection and fault isolation (EDFI). This required that running systems be able to detect errors and isolate them to originating components. Design For Testability (DFT) techniques emerged as a solution that aims at efficient and cost effective testing by enhancing the controllability and observability of the circuit of the circuit under test. Within the scope of DFT, Built-In Self-Test (BIST) techniques were proposed, as seen in section 1.2.2. The basic idea behind BIST is that the generation and verification of test occurs within the logic itself. Many BIST techniques were proposed such as Built-In Logic Block Observation (BILBO), store-and generate, syndrome testing, and autonomous testing. Along with the benefits, there are some penalties for using BIST techniques. These penalties include a hardware overhead, an additional design cost, an increase in the pin-count, and a possible degradation in the circuit performance [KiHT88].

1.6. Design and Test Tradeoffs

Incorporating DFT techniques in a given circuit may involve either the resynthesis of the design if DFT was not considered earlier or modifying the design by inserting extra hardware. The extra hardware may be in the form of extra logic added to configure registers as test registers during test mode or be even inserting dummy structures which will be active only during test mode. This extra hardware will affect the chip delay and final area. Thus, there is always a limit on how much extra

hardware can be inserted and a careful balance must exist between the amount of DFT used and the gain achieved. Furthermore, increasing the VLSI chip area for testing purposes results in an increase in power dissipation and a decrease yield [AbBF90]. Since testing is mainly concerned with faults identifications, and decreased yield leads to an increase in faulty chips produced, again careful balance must be reached between adding logic for DFT and yield. Normally, yield decreases linearly as chip area increases. Thus, if the additional hardware required to support DFT does not lead to an appreciable increase in fault coverage, then the defect level will increase.

Chapter 2

Review of Literature

There has been active research on high-level test synthesis that incorporates some testability features during high-level synthesis. High-level built-in self-test (BIST) synthesis aims to embed BIST capability for the synthesized circuit. Furthermore, testing has shifted from external, using automatic test pattern generation, to built-in self-test (BIST) techniques as millions of transistors are being put on a single chip with limited I/O making external testing very difficult. With the introduction of BIST techniques the chip is able to test itself. The problem of reducing area overhead without sacrificing the quality of the test has become an important area of research.

Incorporating BIST considerations into earlier stages of the design cycle can lead to a more efficient exploration of the design space, thus resulting in a circuit that achieves a desired fault coverage with minimal BIST area overhead and meets the area, throughput and other global requirements.

2.1. Allocation Techniques

One of the earliest high-level BIST synthesis methods was proposed by Papachristou *et al.* The approach is based on constraining allocation to generate a self-testable *template*, represented by a testable functional block, and hence results in exploring a small subspace of the testable design space. For their method, operations and variables are assigned to a testable functional block (TFB), which consists of input multiplexers, an arithmetic logic unit, and output registers. Also, the approach is to merge modules, registers and interconnect simultaneously thus not utilizing the flexibility provided by the separate optimization of these subproblems. The objective

of the assignment is to avoid self-adjacent registers (through which the input and the output of a module form a loop), which are undesirable in BIST due to high area overhead. Papachristou *et al.* have combined register and module allocation methods that generate self-testable designs that either have no self loops or have self-loops in a specific configuration.

Avra proposed a register allocation method to avoid or minimize self-adjacent registers in a design based on register conflict graphs. The assumption in this work is that very self adjacent register needs to be modified to be a BIST register, and thus the area overhead is high. Two variables of a data flow graph have conflict if they are an input and output of the same module. The merger of the two variables of a data flow graph has conflict if they are an input and output of the same module. The merger of the two variables in a register assignment results in a self-adjacent register, and hence it should be avoided.

Parulkar *et al.* investigated a method that maximizes the sharing of test registers to reduce the area overhead. During the register assignment phase, input and output variables of a data flow graph are merged to result in maximum sharing of the registers and to avoid self-adjacent registers. A reverse perfect vertex elimination scheme is employed to obtain maximal sharing of registers.

For all the methods mentioned so far, reducing area overhead in BIST synthesis is the main concern. Test time is not a concern in the design process and is determined from the synthesized circuit through a post process. To reduce test time in BIST, Harris and Orailoglu examined conditions that prevent concurrent testing of modules. They identified two types of conflicts, namely, hardware conflict and software conflict. The synthesis process is guided to avoid such conflicts in the

synthesized circuit. They reported that test time is reduced for example circuits (presumably at the cost of higher area overhead).

2.2. Allocation Technique Reducing Area Overhead

The technique presented by Parulkar *et al.* target testability and area overhead. A typical data path design contains registers and **functional modules**, such as adders and multipliers that are selected from a predesigned library. One way of testing such data paths with low BIST area overhead is *minimal intrusion BIST*. This method involves the modification of a subset of the functional registers to perform test functions such that all modules in a data path are tested using pseudorandom patterns. The methodology used by Parulkar *et al.* ensures that each functional module is tested or *covered* by BIST resources. The actual fault coverage of a functional module depends on the logic design of the module and the BIST resources, the seed and the polynomial chosen, and the number of test patterns. The rest of the data path comprising of multiplexer paths and registers is very well structured and hence easy to test using functional patterns. This form of testing that combines pseudorandom patterns for modules and functional patterns for the rest of the data path, assures a high fault coverage for the complete data path at very low cost. Depending on the required BIST functionality, such as generating test patterns and compressing test responses on-chip, four types of BIST registers can be designed. Each type has a different area overhead. The goal is to synthesize designs such as the minimal intrusion BIST methodology can be implemented with low area overhead. The method of Parulkar *et al.* starts with a scheduled data flow graph (DFG), where variables and data transfers are assigned to registers and interconnect in a way that the functional constraints are satisfied and the area overhead required for BIST is

minimal. The assignment is based on two key ideas: (1) sharing of BIST functions required to test different modules by a register, and (2) minimizing the number of BIST registers that would be *essential* in any BIST solution of the synthesized data path. The assignment techniques are designed to ensure that the *functional area* is not compromised in the quest for low BIST area overhead.

The methodology used for BIST is pseudorandom methodology, therefore, two test functionalities are necessary on chip: (1) capability of **generating** pseudorandom test patterns, and (2) capability of **compressing** test responses into a signature. In order to make a data path self-testable minimal intrusion BIST is used, where a subset of the **functional registers** in the data path are modified and given self test capabilities. In minimal intrusion BIST, in the test mode, some of the registers in the data path are reconfigured to support test pattern generation, some to support test response compression or signature analysis, and some to perform both of these test functionalities. The issue of concurrency of these functionalities arises when the functionalities are performed by the same register. The functionalities of generation and compression can be performed at different times (nonsimultaneous) or simultaneously.

2.3. Allocation Techniques Based on ILP

Integer linear programming (ILP) has been used to perform specific tasks in high-level synthesis. Hafer and Parker pioneered formulating a high-level synthesis problem into an ILP model in the 1980s. Since then, many researchers investigated ILP models to address synthesis problems. Various ILP formulations for scheduling and binding problems are available in the literature. A major advantage of an ILP-

based approach is that the obtained solution is optimal though computationally intensive due to the inherent nature of ILP, which involves an exhaustive search.

The ILP-based approach that is of interest to us is one that performs the three subtasks involved in register assignment of high-level BIST synthesis; these three subtasks are system register assignment, BIST register assignment, and interconnection assignment. Kim *et al.* proposed an ILP-based method that performs the three subtasks concurrently to achieve a global optimality. They present ILP formulations for high-level BIST synthesis with an objective of minimizing the area for each k -test session where k is 1, 2... N and N is number of modules. Hence, their ILP-based method tries to find N optimal (in area) BIST circuits of which a BIST circuit for a k -test session tests the entire modules in exactly k subtest sessions. This method offers a range of designs with different optimized area overhead and test time.

A high-level BIST synthesis for the parallel BIST architecture needs to assign a test-pattern generator to each input port of a module and a signature register to the output of the module. Kim *et al.* impose two constraints in their BIST synthesis. First, test-pattern generators and signature registers are reconfigured from existing system registers. All test registers function as system registers during normal operation. Second, extra paths are not added for testing. The constraints are met through reconfiguration of existing registers into four different types of test registers: A test pattern generator (TPG), a multiple input signature register, a built-in logic block observer (BILBO), or a concurrent BILBO (CBILBO). If a register should be a TPG and a signature register (SR) at the same subset session, it should be reconfigured as a CBILBO. If a register should behave as a TPG and an SR, but not at the same time, it should be reconfigured as a BILBO. Reconfiguration of a register into a CBILBO requires double the number of flip-flops of the register. Hence, it is

expensive in hardware costs. The number of subset sessions necessary for a test session is determined by the number of modules sharing the same SR, because an SR cannot be shared between modules tested in the same subset session, while a TPG can be shared between modules as long as each input of a module receives test patterns from different TPG's. Consider a DFG in which all operators are assigned to N modules. Through an appropriate register assignment, it is possible that the N modules can be tested at least once using exactly k subtest sessions where k is 1, 2 ... N . Test registers are reconfigured to TPG's and/or SR's in each subtest session, and a subtest of modules is tested in a subtest session. When a BIST design is intended to test all of the modules in k subtest sessions, the BIST is said to be a k -test session. As extreme cases, BIST design for one-test session tests all modules in one subtest session, while a BIST design for N -test session tests only one module in each subtest session.

The ILP formulations are for BIST register, which include system registers, and interconnections assignments. The formulations are solved to minimize an objective function for each k -test session. In Kim *et al.* the objective function represents hardware area in terms of the number of transistors. Their method performs the system register assignment, BIST register assignment, and interconnection assignment concurrently through exhaustive search as we have mentioned before, and finds an optimal (in area) BIST design for each k -test session. The method generates N optimal BIST designs, where each design tests the entire modules in k subtest sessions.

The objective of an ILP is to minimize an objective function, i.e., cost function, for the ILP formulations. The cost function to be minimized for the proposed ILP model represents hardware area (in terms of transistor count). The

hardware area is calculated using the number of system registers, SR's, TPG's, BILBO's, CBILBO's, and n -input multiplexers. The number of transistors for each type of hardware that have been mentioned is also included in the cost formula. To compute the value of the cost function it is necessary to compute the number of individual registers, BIST registers, and multiplexers. Then, the number of registers is multiplied by the number of transistors used per register.

The major limitation of this method lies in the long processing time for large designs. Hence, this method in its current form is impractical for large industry circuits. Kim *et al.* have investigated a new heuristic method to address the problem. The heuristic partitions a given data flow graph into smaller regions based on control steps and applies the ILP for each region successively. The heuristic reduces the processing time by several orders of magnitude, while the quality of the solution is slightly compromised. When the heuristic is applied in the discussed method, the method should be able to handle large industry circuits.

Chapter 3

Concurrent BIST Cost Estimation

The testability of a circuit depends largely on its interconnect structure as well as on the functions of its components. It has been generally recognized that in order to have a good design quality, there must be a tight coordination between the design and test processes. Given a behavioral circuit description, there exist different implementations with various structures. The testability cost may increase or decrease accordingly. Based on this observation, this work incorporates test constraints *during* the design process by tightly integrating the design and test processes.

The problem we address in this thesis is defined as follows: Given a behavioral level description of a circuit represented in the form of a scheduled DFG, a technology library and a set of constraints, generate *self-testable RTL data path structure* such that: 1) the datapath conforms to all the user constraints; 2) the overhead of test registers in the data path is minimized. The proposed method is based on the following approach:

- A model for the testable synthesis of RTL datapath structures. This is done through the synthesis of designs with structural properties proven to be good for testing.
- A test point selection scheme that concurrently explores, during the synthesis process, designs with low test and design cost.

3.1. Testable Data Path Synthesis

3.1.1. BIST Methodology

Assume we have the circuit in Figure 3.1(a). To be able to test such a circuit, we allocate the registers at the input ports as TPGRs. Test patterns are collected and observed at the output port, configured as an MISR. One of the difficulties in implementing BIST techniques is the register self-adjacency problem Figure 3.1(b) that is due to the fact that it is not possible to assign a register as both a pattern generator and a signature analyzer at the same time.

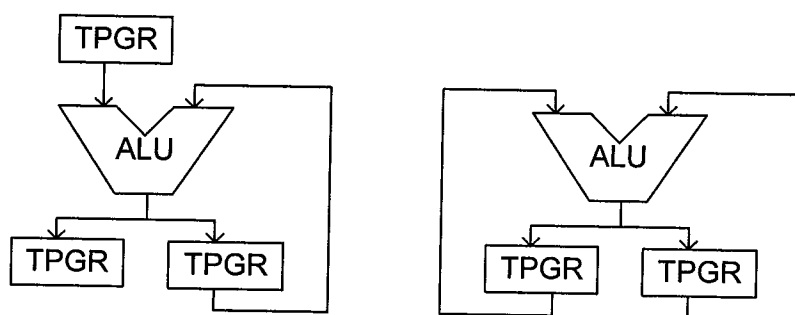


Figure 3.1: (a) Self Testable ALU with Self-Adjacent register, (b) Non-Testable ALU with Self-Adjacent registers

3.1.2. Data Flow Synthesis

Given a scheduled Data Flow Graph (DFG), a DFG node corresponds to 1) an operator that must be assigned to a functional unit during the control step in which it is scheduled; 2) a value that must be assigned to a register for the duration of its life time. Finally, data transfers are assigned to some path of connections, buses and multiplexers.

The allocation model is based on the notion of *structural testability*. The key element of the structural testability model is the *Testable Functional Block* (TFBs)

which are test kernels that do not have any self-loops. In this model, behavioral operations are mapped onto the ALU of the TFB, while the behavioral variables are mapped to the register at the TFB output.

Based on the above model, two TFBs are compatible if there is no resource conflict between the *operations* of the DFG nodes and if their merger does not result in self-adjacent registers that will hinder the design structural testability.

3.1.3. Module Allocation Graph

In order to illustrate the compatibility relations among the DFG nodes, we use a *Module Allocation Graph* (MAG). This is a directed levelized graph with its nodes corresponding to ones in the DFG operations and levels to the DFG schedule. An edge between two DFG nodes in the MAG indicates that both nodes are compatible; however, node A is scheduled before node B. A given path in the MAG corresponds to a list of compatible TFBs that can share resources. Clearly, nodes at the same level are not compatible, i.e., cannot be merged.

3.1.4. Resources Allocation with Testability Consideration

The testable allocation algorithm is an iterative refinement procedure. The method constructs an Initial Datapath Structure (IDP) that corresponds to an initial design point through the mapping of DFG nodes onto individual TFBs (Figure 3.4(a)). The initial design cost is incrementally improved through the merging of TFBs guided by the cost difference of the current and the intended data path configuration.

In order to achieve a good design quality in a relatively short time, we use local cost functions that we associate with every edge or path in the module allocation

graph. The cost function includes ALU, multiplexers, registers, and test overhead cost. The merging algorithm has an order complexity of $O(N^3)$.

3.2. Concurrent BIST Points Selection

In order to determine the exact effect of the above synthesis operation, we propose a methodology that takes the design for testability cost into consideration. In what follows, we describe the tradeoff model and describe the process of BIST cost estimation BIST during allocation (Figure 3.3).

3.2.1. Testability Tradeoff Model at the System Level

Assume *a module is random pattern testable* when random patterns are applied directly on its input ports and its output can be observed from its output ports.

Whether the module, inside the system, is testable or not depends on 1) whether the input patterns applied to this module are random enough and 2) whether the fault effects of this module can be sensitized through intermediate modules to an observable point. Only if these conditions cannot be satisfied, an observable or a controllable point needs to be inserted. Based on this observation, the tradeoff design approach is depicted in Figure 3.2.

- If the output patterns of module A, produced by feeding A with random patterns generated by R_1 is "random enough" then R_2 need not be a controllable point to exercise module B.
- If the faulty output response of A can "go through" block B and received by R_3 , then R_2 need not be an observable point.
- If we know, by investigation, that R_2 does not need to be either a controllable or an observable point, then BIST insertion overhead can be saved by leaving R_2 as a normal register.
- The randomness of an output pattern and the "transparency" of a module can be improved by increasing the testing time.

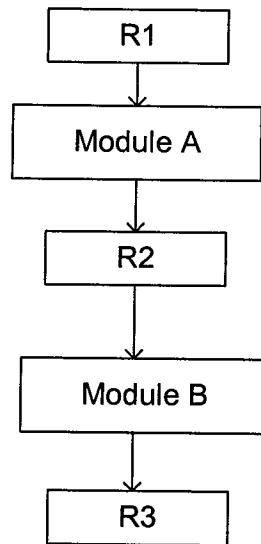


Figure 3.2: Basic model for testability tradeoffs

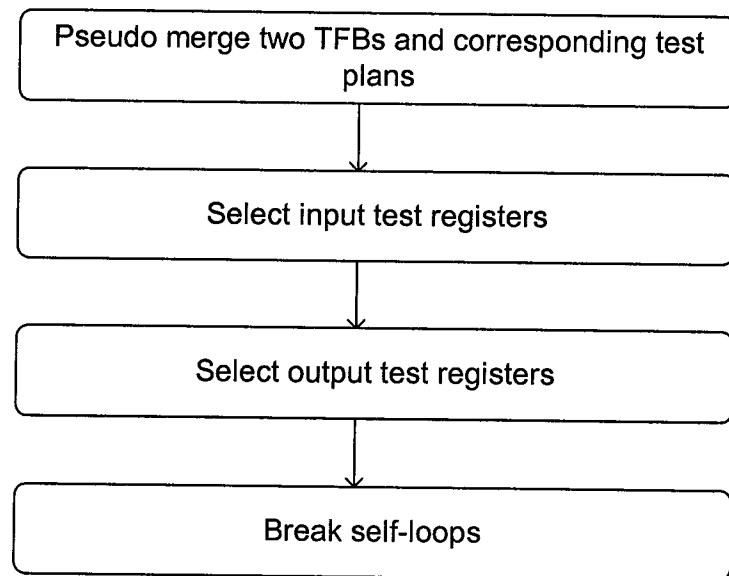


Figure 3.3: Merging procedure

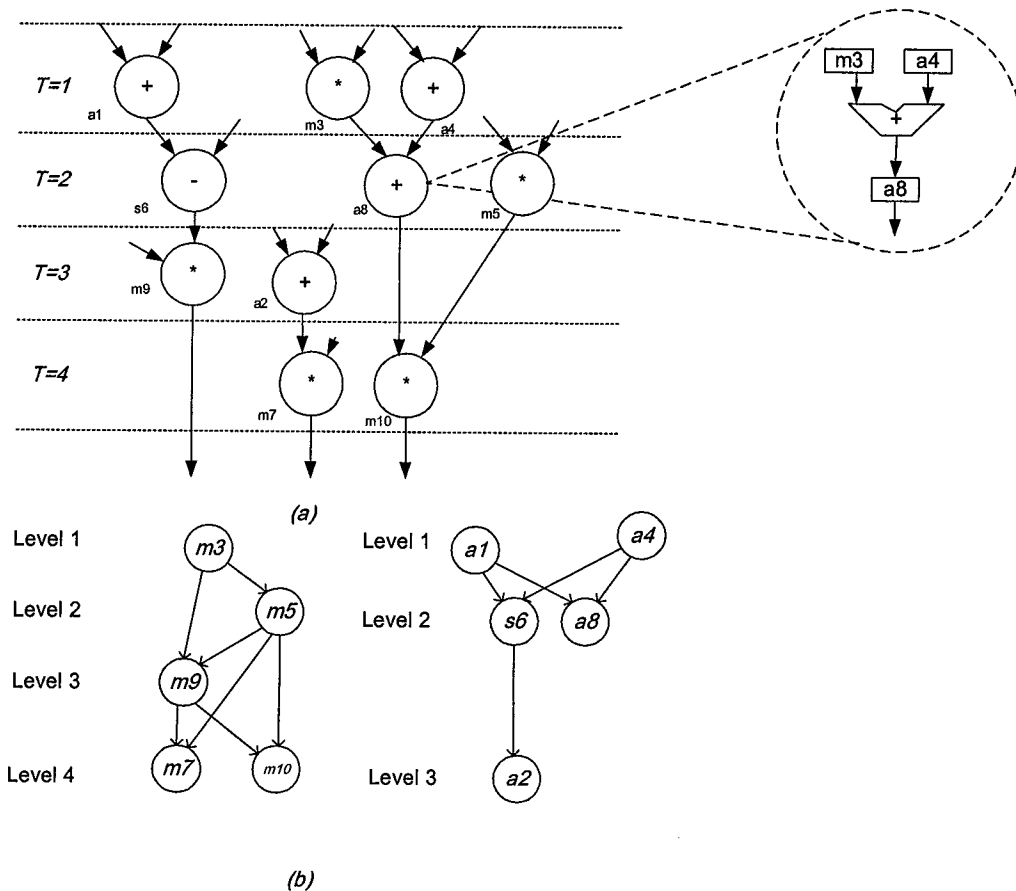


Figure 3.4: (a) Example DFG, (b) Corresponding MAG

3.2.2. Data Path Representation

The RTL data path consists of modules, registers as well as the connections between these entities. The usage of each test register, referred to as attribute table, can be *controllable*, *observable*, *pseudo-controllable*, or *pseudo-observable*. The final implementation attribute of a register is the union of the underlying register usage. This implementation can indicate the hardware overhead before and after merging of data flow nodes by computing the difference over two test plans where the given operators reside.

3.2.3. Pseudo-Merge of two TFBs

The pseudo merger of two TFBs combines the given two TFBs into one and link their associated test plans into a preliminary "merged test plan".

3.2.4. Select Input Registers

In order to test the datapath, we need to select a "Pattern Generator" for each input port of a datapath module resulting from merging the two given TFBs. Thus, for every CLB input port in the data path we select one register to provide random patterns during testing and remove the input port from all the "controllable" usage entry of the other input registers. The selection priority goes from "Normal," "Observable," "Controllable," to "Controllable & Observable." However, if the current TFB is transparent enough such that the testing time is acceptable after the adjustment, move the output port identifier list in "observable" to "pseudo observable" for all input registers. Note that we do not select the same input register to cover any two input ports of a module since this introduces dependencies between input test patterns to that module.

3.2.5. Select Output Registers

The selection of a "signature analyzer" is accomplished by avoiding the intended operator merger when the merged output register provides test patterns to different input ports of the same module. Next, the method selects accordingly one of the following cases:

1. *A BIST register combined with a BIST register:* If the merged test registers are a controllable and an observable point respectively, then move the identifier list from "controllable" to "pseudo-controllable," and we move the identifier

list from "observable" to "pseudo-observable". Assign implementation attribute to "normal" when the upper bound of test time is not exceeded. If, however, either register is attributed with both a controllable and an observable point, then we assign the implementation attribute "controllable & Observable" to the output register.

2. *A Non-BIST register combined with a Non-BIST register:* The resulting register in this case has an attribute "normal" except in the case when t_s exceeds user specification. If t_3 is above user specified upper bound, we have to give the attribute "controllable" when it originated from pattern randomness problem, or allocate the attribute "observable" when the problem comes from the module transparency deficiency.
3. *Non-BIST registers combined with BIST registers:* If the BIST register is a "controllable" point, we can remove this attribute when the test time is not exceeded. This is done by moving all items in the attribute entry "controllable" to "pseudo-controllable." Since now half of the time the patterns will be random, although the other half of the time the patterns are not, two times of the previous pattern generation time will be enough random to test the modules followed it. A similar argument applies for the case when the register attribute is "controllable." However, if due to the time limit a register needs to be assigned as a BIST register, then a BIST attribute is assigned to register after merging.

3.2.6. Self-Loop Breaking

Once the input and output test points have been selected, it is possible that some self-loops have been created due to the functional removal of test points. A self-loop problem arises only when there exists a path $F \rightarrow C$ and/or $C \rightarrow F$ where F and C are test registers (Figure 3.5). There are two cases that may result in a self-loop:

1. Test plan T_1 connects to test T_2 and the path $F \rightarrow C$ pass through the boundary.
2. The path $F \rightarrow C$ is in the same test plan.

Note that the above situation does not arise due to the structural property of the underlying data path design, but rather with the test methodology and tradeoff that we have chosen. In what follows, we describe the loop breaking algorithm with reference to Figure 3.5.

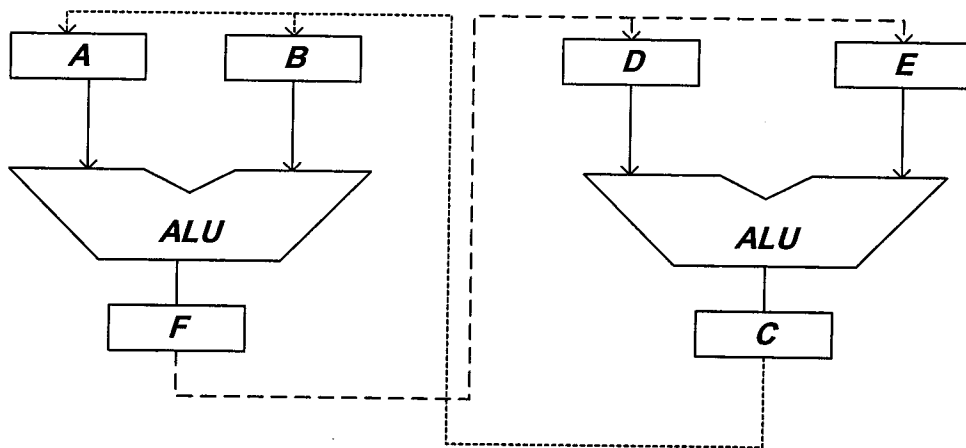


Figure 3.5: Paths that may cause functional self-loops

Both TFBs are in different test plans

Suppose that the case $F \rightarrow C$ is considered. Then, if F and C are in different test plans and at least one of register does not have a "normal" test attribute, there is no self-loop between F and C . Otherwise, we have the following:

1. Get all the children of F and store in a queue I.
2. For all nodes in I, pick a node A
 - a) If it is C, then there is a cycle; add to C an implementation attribute of either "controllable" or observable (the decision can be made randomly).(Modify the attribute table accordingly. Stop.
 - b) If it is a node that has already been visited then go to 2.
 - c) If it is a node with implementation attribute of either "controllable" or "observable" but is not one of the interface node, mark A visited and go to 3.
 - d) Add all its unvisited children to the queue I and mark A "visited".

Both TFBs are in the same test plan

Suppose that the case $F \rightarrow C$ is considered. If F and C are in the same test plan then:

1. If both implementation attributes are "normal," then we need to find a controllable and an observable point.
2. If the union of the implementation attributes is "controllable" (or "observable"), we need to find an observable point (or a "controllable" point)
3. If the union of their implementation attributes is "controllable and Observable", it is similar to the previous case.

Loop Breaking Algorithm

The loop breaking process proceeds as follows:

1. Start from F and store all its children in a queue I. Assign each node in I a label that is the same as the implementation attribute it bears.
2. Get a node A and mark it as "visited." Repeat until the queue is empty.

- a) If the label set union of the current node and F is 1) "controllable," "observable" or 2) "controllable & \observable", "observable", then there is no self-loop in this path. Goto 2.
 - b) If any children of the current node is C, then there is a self-loop. Record the current node label and the current node identifier that caused the self-loop.
 - c) Otherwise (i) assign the label set union of the current node and that of F to the "current label", (ii) assign all unvisited children of the current node with the "current label set" (iii) put these children into queue I.
3. Break-Loop
- a) Get the union of the attributes of registers F and C, call it START.
 - b) If START is "normal" and the union of all the labels recorded is (i) "normal," then assign all the recorded nodes "controllable" and the node F "observable" (ii) "controllable," then assign all the recorded nodes that have attribute "normal" with attribute "controllable" and the node F "observable" (iii) "Observable," then assign all the recorded nodes that have attribute normal with attribute "observable" and the node F "controllable" (iv) "controllable & \ observable", then assign all the recorded nodes that have attribute "normal" with attribute "observable," F "controllable and observable ". Once the implementation attribute has been added, modify the attribute table accordingly.
 - c) If START is "controllable" then add to the recorded registers the attribute "observable"; modify the implementation attribute and usage attribute accordingly.

- d) If START is "controllable & observable" then either F or C is "controllable & observable" while the other one is "normal." In this case, add to all the recorded registers "observable" attribute and modify the implementation attribute and usage attribute accordingly .

Chapter 4

Experimental Results

In this chapter, we present experimental results on the performance of our method.

4.1. Background

We measured the performance of our system using six data flow graphs, which are widely adopted for benchmarking high-level BIST synthesis. The data flow graphs include the ones studied by Tseng and Siewiorek, called *tseng*, and by Paulin and Knight, called *differential equation*. The other four data flow graphs are the 6th order FIR (finite impulse response) filter, a 3rd order IIR (infinite impulse response) filter, a 4-point DCT (discrete cosine transformation) circuit, and a 6-tap wavelet filter. Details of the circuits are shown in Table 4.1. Column headings of the table are described below:

ckt:	the name of the circuit.
var:	the number of variables in the DFG.
const:	the number of constants in the DFG.
op:	the number of operations in the DFG.
reg:	the minimum number of necessary registers (equivalent to the maximal horizontal crossing), and
modules:	the minimum number and types of necessary modules.

Ckt	Data Flow Graph			Data Path	
	Var	Const	Op	Reg	Modules
tseng	8	0	11	5	3 ALUs
diffeq	10	2	12	5	2 multipliers, 1 adder, 1 subtractor
fir6	13	7	27	7	2 multipliers, 1 adder
iir3	14	8	26	6	1 multiplier, 1 adder, 1 subtractor
dct4	15	4	23	6	2 multipliers, 1 adder, 1 subtractor
wavelet6	16	6	28	7	1 multiplier, 1 adder, 1 subtractor

Table 4.1: Characteristics of the circuits

In this thesis, the area of a circuit is represented by the transistor count of registers and multiplexers in the circuit. *Data path logic is not considered in the transistor count.* The number of transistors in test registers and multiplexers is based on the circuits of [KoZw79] and [WaMc86] and is given in Table 4.2. In the table, #Trs and #MuxIn denote the number of transistors and the number of multiplexer inputs, respectively. The heading "Avg" in Table 4.2(b) is the average number of transistors per multiplexer input.

The reference circuits, which were used to measure the area overhead of BIST designs, were obtained through an ILP for data path synthesis.

Type	Reg.	TPG	SR	BILBO	CBILBO
#Trs	208	256	304	388	596

(a) Test registers

#MuxIn	2	3	4	5	6	7	Avg
#Trs	80	176	208	300	320	350	-
#Trs/input	40	59	52	60	53	50	52

(b) Multiplexer

Table 4.2: Number of transistors of 8-bit test registers and multiplexers

4.2. 6th Order FIR Filter

The 6th order finite impulse response (FIR) filter is used in the results as a demonstrative circuit. This circuit implements the function:

$$y = h_0x_0 + h_1x_1 + h_2x_2 + h_3x_3 + h_4x_4 + h_5x_5 + h_6x_6$$

where h_n is a filter coefficient, and x_n is the delayed value of x_{n-1} . This equation requires seven multiplications and one summation. Figure 4.1 shows a block diagram of the filter.

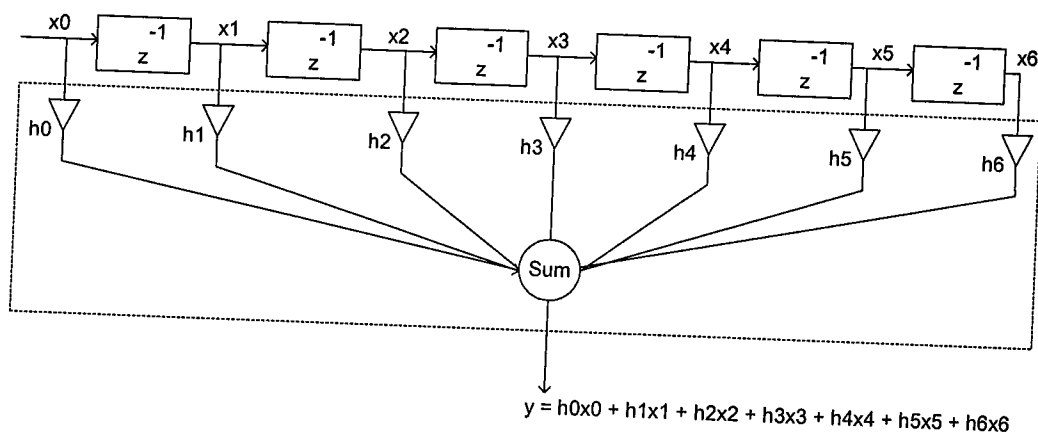


Figure 4.1: A 6th order FIR filter

Figure 4.2 shows a data flow graph for the 6th order FIR filter with the scheduling and module assignment completed.

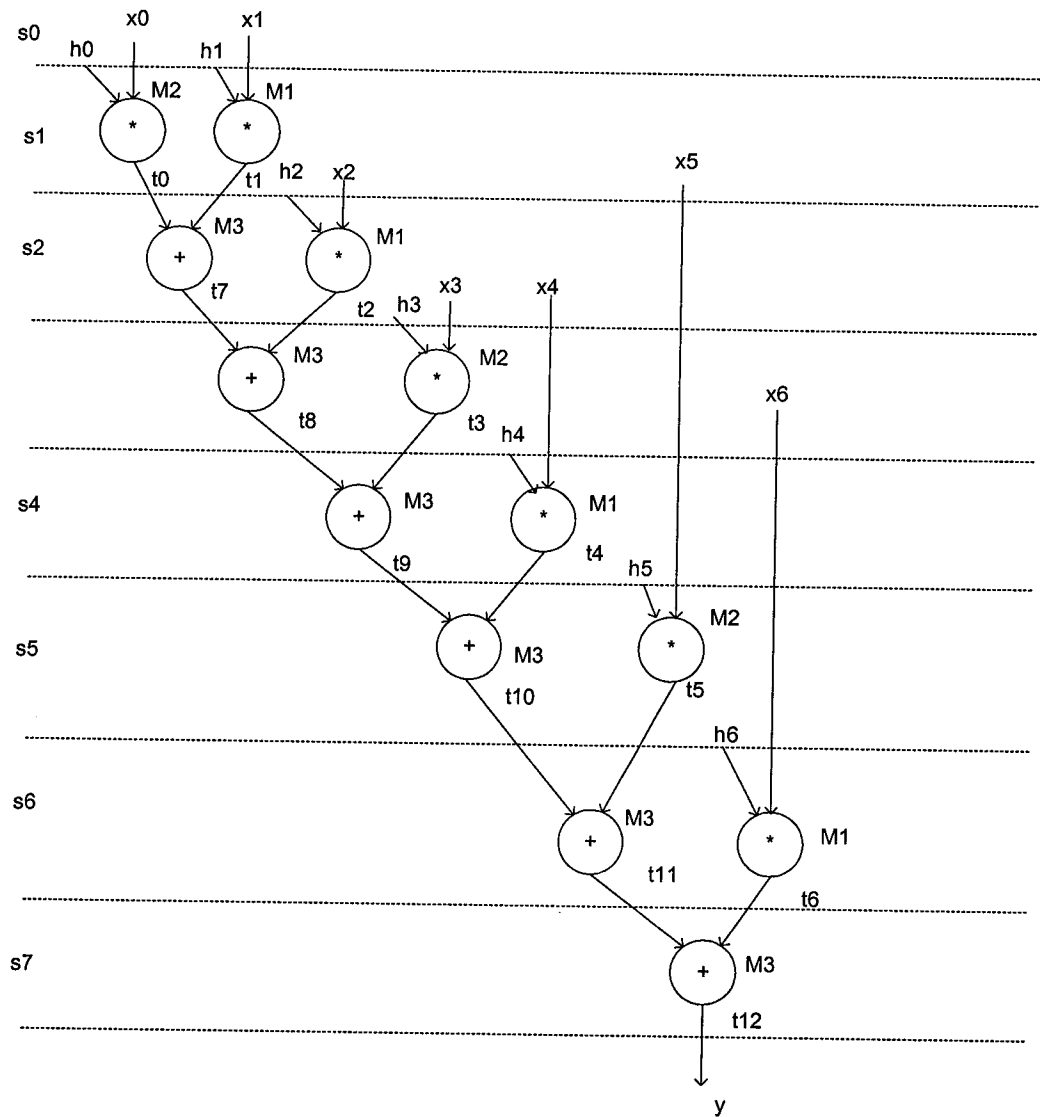


Figure 4.2: Data flow graph for the 6th order FIR filter

4.3. 3rd Order IIR Filter Cascade Connection

Figure 4.3 shows a data flow graph of a 3rd order IIR filter in which the scheduling and the module assignment are completed.

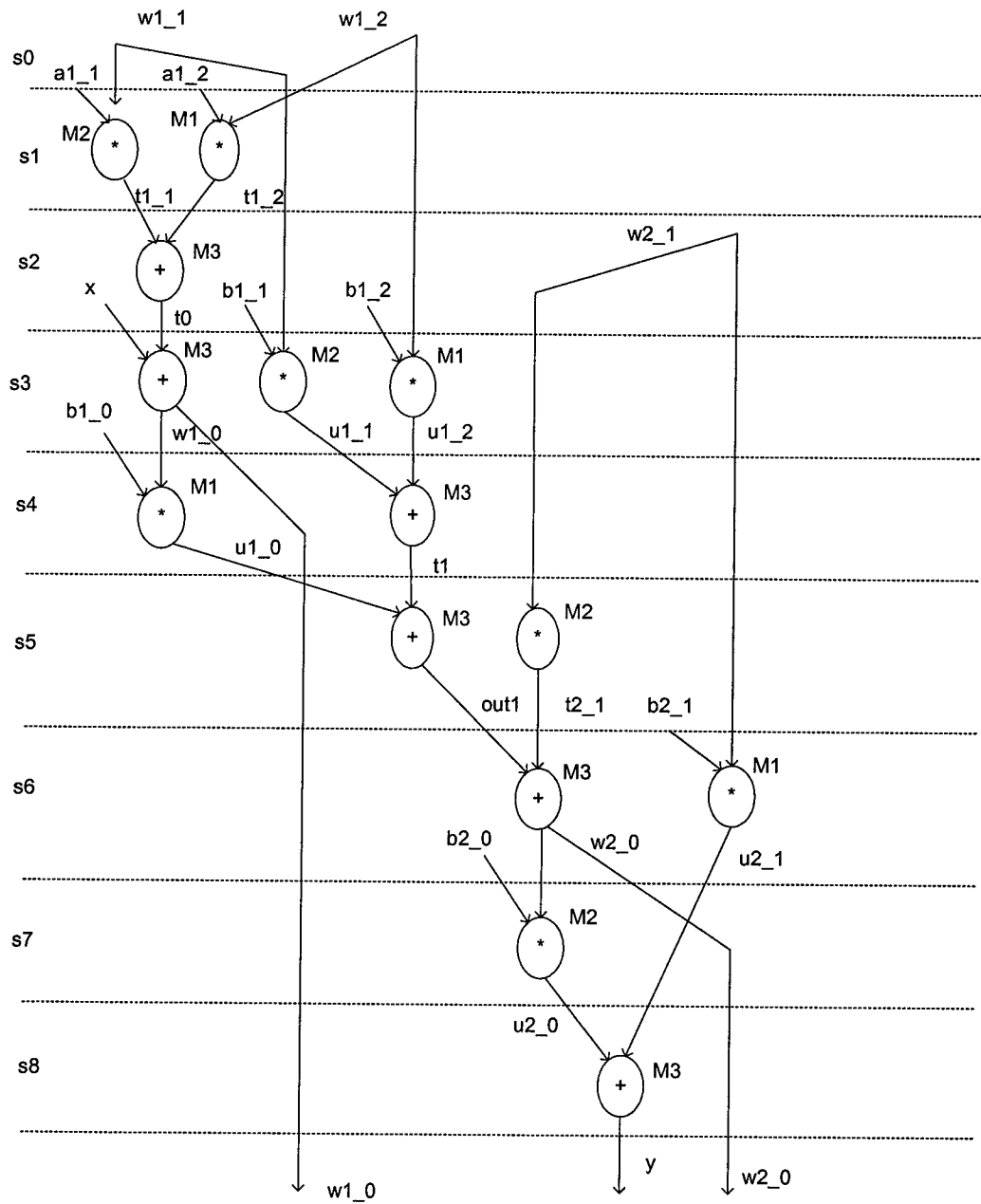


Figure 4.3: Data flow graph of a 3rd order IIR filter (cascade connection)

4.4. 6-Tap Wavelet Filter

Figure 4.4 shows a data flow graph in which the scheduling and the module assignment are completed.

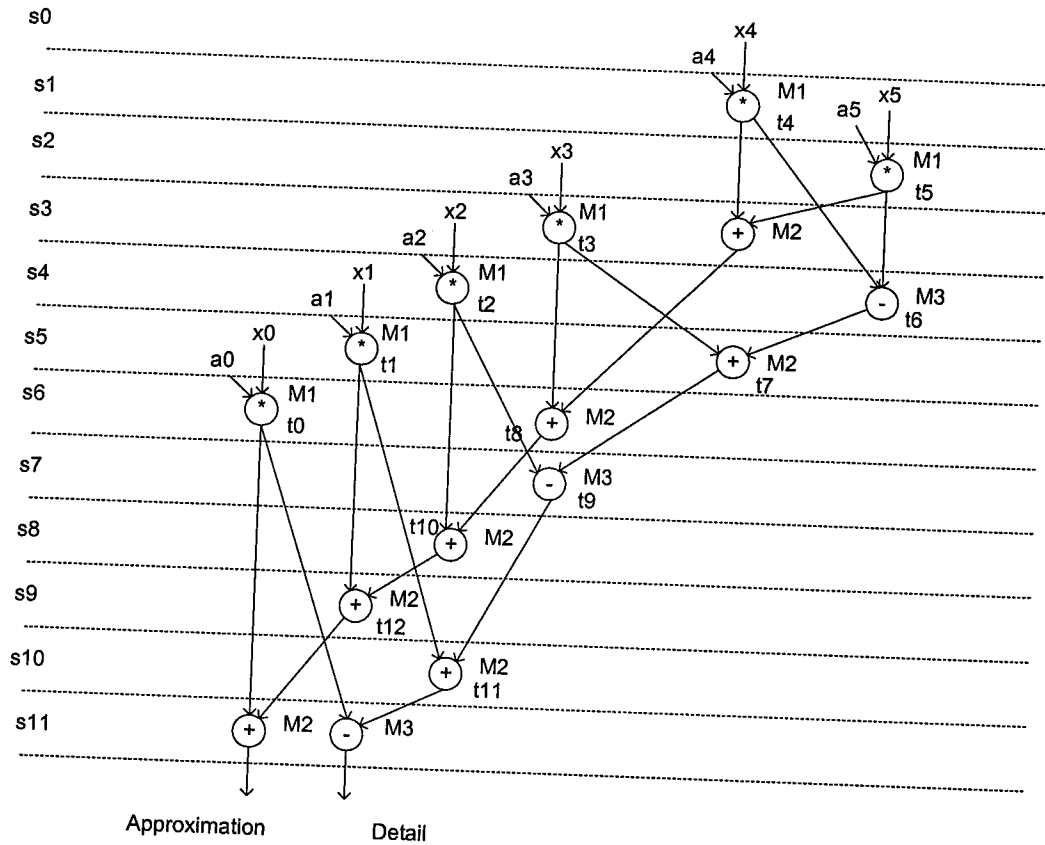


Figure 4.4: Data flow graph of a 6-tap wavelet filter

4.5. 4-point Discrete Cosine Transformation (DCT)

Figure 4.5 shows a data flow graph in which the scheduling and the module assignment are completed.

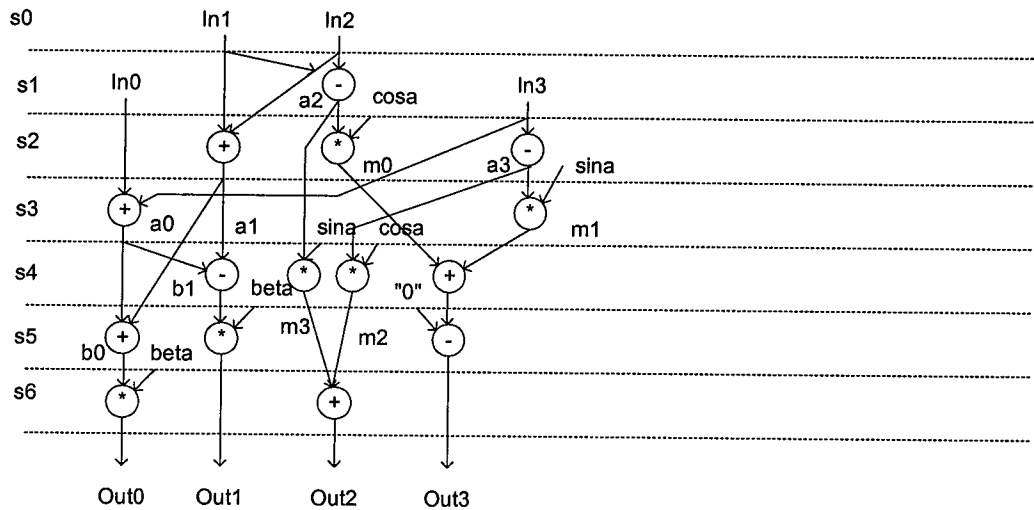


Figure 4.5: Data flow graph of a 4-point discrete cosine transformation

4.6. Tseng

Figure 4.6 shows a data flow graph which was studied by Tseng and Siewiorek.

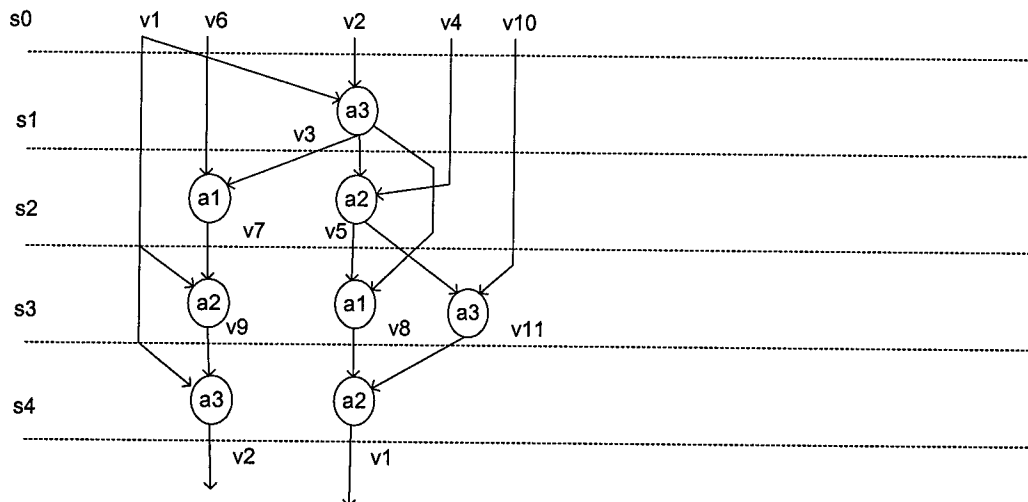


Figure 4.6: Data flow graph of Tseng

4.7. Paulin

Figure 4.7 shows a data flow graph which was studied by Paulin and Knight.

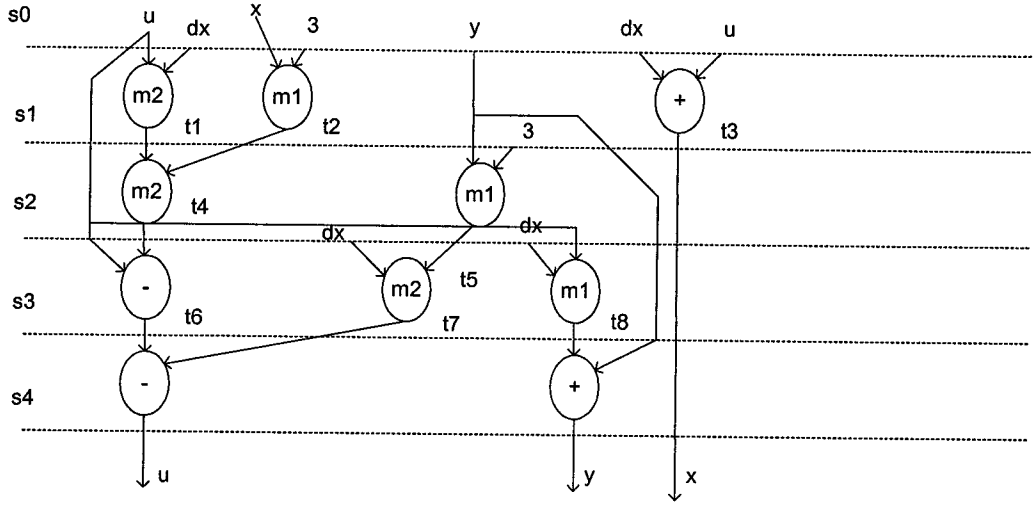


Figure 4.7: Data flow graph of Paulin

4.8. Results

We ran our methods on the above high-level synthesis benchmarks. Our experiment was to measure the performance of our proposed method. Detailed experimental results for the above benchmarks are shown in Tables 4.3-4.9. Area comparisons of the synthesized BIST circuits are shown in table 4.10. The first row for each circuit entry is the reference circuit that was generated without BIST consideration while the second row shows the results synthesized using our method based on the concurrent BIST selection scheme. The third row of each circuit shows the results for the benchmarks that were synthesized using the BILBO BIST method. Column headings for the table are explained below.

R : The total number of registers

TPGR : The total number of test pattern generators

MISR : The total number of signature registers

BILBO: The total number of BILBOs

M : The total number of inputs of multiplexers

Area : The number of transistors of the registers and the multiplexers

OH : The area overhead of the BIST design (%)

From the table, the area overhead for the synthesized designs derived using our method ranges from 8.89% to 27.46% while the overhead of the BILBO designs ranges from 20.87% to 34.56%. Note that the area overhead of two circuits is less than 10 percent(Differential Equation and IIR3). However, since the area overhead of a circuit is computed without considering the area for the data path logic modules, the actual area overhead will be much lower than the ones presented in the table.

The table shows that the results from our method use less overhead than BILBO. This is very obvious in all shown results. This leads us to conclude that our method has a better test overhead than BILBO.

Test Mode	ALUs	Number of Registers				# Mux Inputs
		<i>BILBO</i>	<i>MISR</i>	<i>TPGR</i>	<i>Normal</i>	
Concurrent Selection	2 (+), 2 (*)	0	2	8	5	16
BILBO	2(+), 2 (*)	4	0	11	0	16

Table 4.3: Results from the FIR filter

Test Mode	ALUs	Number of Registers				# of Mux Inputs
		<i>BILBO</i>	<i>MISR</i>	<i>TPGR</i>	<i>Normal</i>	
Concurrent Selection	2 (+), 2 (*)	0	2	5	8	21
BILBO	2(+), 2 (*)	4	0	11	0	19

Table 4.4: Results from the IIR3 filter

Test Mode	ALUs	Number of Registers				# of Mux Inputs
		<i>BILBO</i>	<i>MISR</i>	<i>TPGR</i>	<i>Normal</i>	
Concurrent Selection	2 (+), 2 (*), 1(-)	0	1	4	12	24
BILBO	2(+), 2 (*), 1(-)	5	0	11	0	25

Table 4.5: Results from the wavelet 6 filter

Test Mode	ALUs	Number of Registers				# of Mux Inputs
		<i>BILBO</i>	<i>MISR</i>	<i>TPGR</i>	<i>Normal</i>	
Concurrent Selection	2 (+), 2 (+-), 1(+)	0	1	6	8	27
BILBO	2(+), 2 (+), 1(-)	5	0	10	0	23

Table 4.6: Results from the DCT4 filter

Test Mode	ALUs	Number of Registers				# of Mux Inputs
		<i>BILBO</i>	<i>MISR</i>	<i>TPGR</i>	<i>Normal</i>	
Concurrent Selection	3 (*), 2 (-), (+), (>)	1	2	6	3	13
BILBO	3 (*), 2 (-), (+), (>)	6	0	6	0	13

Table 4.7: Results from the diffeq Example

Test Mode	ALUs	Number of Registers				# of Mux Inputs
		<i>BILBO</i>	<i>MISR</i>	<i>TPGR</i>	<i>Normal</i>	
Concurrent Selection	3 (*), 2 (-), (+), (>)	0	2	4	1	17
BILBO	3 (*), 2 (-), (+), (>)	4	0	5	0	18

Table 4.8: Results from the diffeq Example with a different binding

Test Mode	ALUs	Number of Registers				# of Mux Inputs
		<i>BILBO</i>	<i>MISR</i>	<i>TPGR</i>	<i>Normal</i>	
Concurrent Selection	(/), (+*), (-), (+&)	1	5	0	2	10
BILBO	(/), (+*), (-), (+&)	4	0	4	0	10

Table 4.9: Results from the Tseng Example

Ckt	Type	R	TPGR	MISR	BILBO	M	Area	OH(%)
Tseng	1	1248	0	0	0	728	1976	
	2	416	0	1520	388	400	2724	27.46
	3	0	1024	0	1552	400	2976	33.60
Diffeq(2)	1	1872	0	0	0	672	2544	
	2	208	1024	608	0	960	2800	9.14
	3	0	1280	0	1552	1056	3888	34.56
Diffeq(1)	1	2288	0	0	0	800	3088	
	2	624	1536	608	388	576	3732	17.26
	3	0	1536	0	2328	576	4440	30.45
DCT4	1	2704	0	0	0	1148	3852	
	2	1664	1536	304	0	1248	4752	18.94
	3	0	2560	0	1940	1088	5588	31.07
Wavelet6	1	2912	0	0	0	1312	4224	
	2	2496	1024	304	0	1112	4936	14.42
	3	0	2816	0	1940	1328	6084	30.57
IIR3	1	3120	0	0	0	1020	4140	
	2	1664	1280	608	0	992	4544	8.89
	3	0	2816	0	1552	864	5232	20.87
Fir6	1	2912	0	0	0	768	3680	
	2	1040	2048	608	0	752	4448	17.27
	3	0	2816	0	1552	752	5120	28.13

Table 4.10: Results Comparisons

Chapter 5

Conclusion

Testability is one of the most important requirements, along with other constraints such as performance and cost, to be taken into consideration when designing a circuit. Circuits with poor testability cause time as well cost losses during post-fabrication testing and testing for serviceability. Built-In Self-Test (BIST) is an improved technique for testability where the testing is done through built-in hardware features.

In this Thesis we implemented the described allocation and tradeoff scheme using six benchmark examples. The examples include the 6th order FIR (finite impulse response) filter, a 3rd order IIR (infinite impulse response) filter, a 4-point DCT (discrete cosine transformation) circuit, and a 6-tap wavelet filter. They also include the data flow graphs by tseng and paulin called diffeq. We show the detailed results using those examples in terms of components as well as number and types of test points.

Bibliography

- [AbBF90] M. Abramovici, M. Breuer, A. Friedman, Digital Systems Testing and Testable Designs, *Computer Science Press*, 1990.
- [AbBr85] M. Abadir, M. A. Breuer, "A Knowledge-Based System for Designing Testable VLSI Chips ", *IEEE Design & Test*, pp. 56-68, August .1985.
- [Agra91] V. Agrawal, Plenary Speech, International Conference on Computer Design, October 1991.
- [BaMc87] P. Bardell, W. McAnney, J. Savir, "Built-In Test for VLSI: Pseudorandom Techniques ", *John Wiley & Son*, 1987.
- [ChPa91] S. Chiu, C.A. Papachristou, "A Design for Testability Scheme with Applications to Data Path Synthesis" *Proc. 28th Design Automation Conference*, June 1991.
- [DAC90] "Testing Strategies for the 1990s," Panel Discussion, *Design Automation Conference*, 1990.
- [DeMi90] G. De Micheli, D. Ku, Frederic Mailhot, T. Tuong, "The Olypmus Synthesis System for Digital Design," *Technical Report*, Stanford University, 1990.
- [GaKu83] D. Gajski, R. Kuhn, "Guest Editors' Introduction: New VLSI Tools", *IEEE Computer*, 6 (12):11-14, 12 1983.
- [HaPa93] H. Harmanani, C. Papachristou, "An Improved Method for RTL Synthesis with Testability Trade-Offs", *In Proc. of the ICCAD*, Nov. 1993.
- [HuPe87] C.L. Hudson, G.D. Peterson, "Parallel Self-Test With Pseudo-Random Test Patterns", *Proc. International Test Conference*, pp. 954-971, Sept. 1987.
- [Jain89] R. Jain, K. Kukukcakar, M. Mlinar, A. Parker, "Experience with The Adam Synthesis System," *Proceedings of the 26th Design Automation Conference*, June 1989.

- [JaKu89] R. Jain, K. Kukukcakar, M. Mlinar, A. Parker, "Experience with The Adam Synthesis System", *Proceedings of the 26th Design Automation Conference*, June 1989.
- [KiHT88] K. Kim, D.S. Ha, and J.G. Tront, "On Using Signature Registers as Pseudorandom Pattern Generators in Built-in Self-Testing," *Proceedings of the IEEE Transactions on CAD*, pp. 919-928.
- [KiTh88] K. Kim, J.G. Tront and D.S. Ha, "Automatic insertion of BIST hardware using VHDL", *Proc. 25th Design Automation Conference*, pp. 9-15, June 1988.
- [KoZw79] Konemann, B.J. Mucha, and G. Zwiehoff, "Built-In Logic Block Observation Techniques," *Proc. Int'l Test conf.*, pp. 37-41, Oct. 1979.
- [KuWK85] S.Y. Kung, H.J. Whitehouse, T. Khailath, "VLSI and Modern Signal Processing", *Prentice Hall*, 1985, pp. 258-264.
- [Marw86] P. Marwedel, "A New Synthesis Algorithm for the MIMOLA Software System," *Proceedings of the 23rd Design Automation Conference*, June 1986, pp. 271-277.
- [McPC90] M. McFarland, A. Parker, and R. Compasano, "The High Level Synthesis of Digital Systems," *Proceedings of the IEEE*, Vol. 78, No. 2, February 1990, pp. 301 - 318.
- [PaCh91] C. Papachristou, S. Chiu, H. Harmanani, "A Data Path Synthesis Method for Self-Testable Designs", *Proc. 28th Design Automation Conference*, June 1991.
- [PaKo90] C. Papachristou, H. Konuk, "A Linear Program Driven Scheduling and Allocation Method Followed by an Interconnect Optimization Algorithm", *Proc. 27th Design Automation Conference*, June 1990, pp. 77-83.
- [PaKn87] P.G. Paulin J.P.Knight "Force -Directed Scheduling in automatic data path synthesis", *Proceedings of the 24th Design Automation Conference*, pp. 195-202, June 1987.
- [PaKn89] P.G. Paulin J.P.Knight "Force -Directed Scheduling for the Behavioral Synthesis of ASICs." *IEEE Trans. on Computer Aided Design*, Vol. 8, pp. 661-679, June 1989.

- [Pang88] B.M. Pangrle, "Splicer: A Heuristic Approach to Connectivity Binding," *Proceedings of the 25th Design Automation Conference*, June 1988, pp. 536-541.
- [Raba88] J. Rabaey, H. DeMan, J. Vanhoof, F. Cathoor, "Cathedral II: A Synthesis System for Multiprocessor DSP," *In Silicon Compilation*, pp. 311-360, 1988.
- [Thom88] Thomas D.E., E.M. Dirkes, R.A. Walker, J.V. Rajan and R.L. Blackburn, "The System Architects Workbench," *Proceedings of the 25th Design Automation Conference*, pp. 337-343, June 1988.
- [TsSi86] C. Tseng, D. P. Siewiorek, "Automated Synthesis of Data Paths in Digital Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, V. CAD-5, No. 3, pp. 379-395, July 1986.
- [WaMc86] L.-T. Wand and E.J. McCluskey, "Concurrent Built-In Logic Block Observer (CBILBO)," *Int. Symp. On Circuits and Systems*, pp. 1054-1057, May 1986.
- [WiPa83] T. Williams, K. Parker, "Design For Testability --- A Survey", *Proceedings of The IEEE*, Volume 71, Number 1, January 83, pp. 98-112.