Order Number 9501997

Resource allocation and reallocation techniques in high-level synthesis with testability constraints

Harmanani, Haidar M., Ph.D.

Case Western Reserve University, 1994

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

RESOURCE ALLOCATION AND REALLOCATION TECHNIQUES IN HIGH LEVEL SYNTHESIS WITH TESTABILITY CONSTRAINTS

by

HAIDAR M. HARMANANI

Submitted in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

Thesis Advisor: Dr. Christos A. Papachristou

Department of Computer Engineering and Science
CASE WESTERN RESERVE UNIVERSITY
May 1994

CASE WESTERN RESERVE UNIVERSITY

GRADUATE STUDIES

We hereby approve the thesis of

Haidar M. Harmanani

candidate for the *Doctor of Philosophy*
degree*.

(signed) _____

(chair)

_____

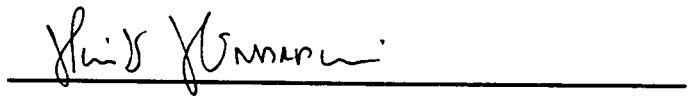
_____

_____

_____

date Feb 2, 1994

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



RESOURCE ALLOCATION AND REALLOCATION TECHNIQUES IN HIGH LEVEL SYNTHESIS WITH TESTABILITY CONSTRAINTS

ABSTRACT

by

HAIDAR M. HARMANANI

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.

In this Work, we propose a prototype high-level synthesis system with self-testability, SYNTEST, that alleviates the above problems. SYNTEST is based on a model that treats testing as a structural design property during *data path allocation.* Thus, the design is testable by construction and there is no need for the traditional post-design test insertion methods. The most significant aspect of this work is that it covers the void between the fields of high-level synthesis and design for testability. This allows to exploit the tight relation that exists between both disciplines in an integrated system level design environment. The allocation method incorporates a test points (registers) selection method which trades test overhead for fault coverage. We follow the allocation method with a *reallocation* method which aims at exploring any possible design improvements which may be due to the non-optimal nature of the design process. By considering placement and routing, in addition to component cost, the reallocation modifications become more effective and more realistic. Another motivation for the reallocation process

is that it may be desireable to reuse the old data path in order to generate an alternate structure under a different technology. Thus, the designer may reuse the old structure to generate a new one, optimized under a different cost function. The reallocation phase is based on a *rip-and-rebind* approach. Finally, we automatically generate VHDL output from SYNTEST in order to complete the silicon compilation iteration. The output is a mixed behavioral and structural VHDL description of a testable data path and a controller. We link SYNTEST to the COMPASS Design Automation tools, through VHDL, which serves as our means to validate our design model and architecture. We validate our approach using various design and benchmark examples and several chips layouts were generated.

To my parents

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Christos Papachristou for his guidance throughout my M.S. as well as my Ph.D. studies. The various interactions we had have helped shaping a lot of the ideas in this Thesis. Thanks is also due to Dr. Frank Merat, Dr. George Ernst and Dr. Steven Garveric for being on my Thesis committee.

I would like to express my sincere gratitude to the Hariri Foundation, whose financial support during my undergraduate as well as my graduate studies made it all possible.

I would like also to thank Mehrdad Nourani and Joan Carletta for all the discussion and working environment in the Design Automation Group. Thanks also is due to Ricardo Shaffner who kept up with me during the various graphical development stages in SYNTEST and to Misha Baklashov for developing the VHDL parser. They all helped shaping SYNTEST the way it is right now. I would like also to thank all my friends at Case especially Kyu Kwak, and Hassan Noureddine.

Finally, I would like to thank my family for their long support. They have kept up with me being away for all these years.

Contents

1. Introduction **1**

1.1 Synthesis: From Concept to Silicon 1

1.2 The Design and Test Process 5

1.3 High Level Synthesis Task 6

1.3.1 Scheduling in High-Level Synthesis 8

1.3.2 Allocation in High-Level Synthesis 11

1.4 Why High-Level Synthesis? 12

1.5 Design For Testability 12

1.6 Design and Test Tradeoffs 14

1.7 High-Level Synthesis and Design for Testability 14

1.8 Problem Definition and Thesis Outline 15

2. Background **17**

2.1 Overview of Synthesis Systems 17

2.1.1 The Hardware ALlocator 17

2.1.2 The VHDL Synthesis System 19

2.1.3 The Facet System 20

2.1.4 MAHA/REAL 21

2.2 Overview of Synthesis Systems with Testability 23

2.2.1 CATREE 23

2.2.2 The Register Allocator System (RALLOC) 24

2.3 The Testable Design Expert System 25

2.3.1 Comments on TDES 27

3. The SYNTEST Design System **28**
3.1 Motivation 28
3.2 Design Representation 30
3.2.1 Input 31
3.2.2 Output 31
3.3 Tools Overview 33
3.3.1 DFG Generation 33
3.3.2 DFG Scheduling 33
3.3.3 Testable Data Path Allocation 34
3.3.4 Design Cost Estimation Tool 35
3.3.5 Test Cost Estimation Tool 36
3.3.6 Layout and Area Estimation Tools 37
3.4 User's End View 37
3.4.1 Graphical Interface 37
3.4.2 User Interaction with SYNTEST 38

4. Testable Data Path Allocation **42**
4.1 Design and Test Methodology 43
4.1.1 Background 43
4.1.2 Improved Model for Synthesis with Testability 45
4.1.3 Design and Test Space 46
4.2 Testable Data Path Allocation 49
4.2.1 Requirement Analysis 50
4.2.2 System Library 51
4.2.3 Module Allocation Graph 52
4.2.4 Resources Allocation with Testability Consideration 54
4.3 Synthesis Aspects 60
4.3.1 Pipelining 60
4.3.2 Conditionals 61

4.4 Test Points Selection 63
4.5 Test Tradeoffs Illustration Using an Example 65
4.6 Results 67

5. Datapath Reallocation 75
5.1 Introduction 75
5.1.1 Problem Definition and Motivation 75
5.1.2 Related Research 76
5.2 Reallocation Approach 77
5.2.1 Background 77
5.2.2 Transformations 79
5.3 Design Representation 82
5.3.1 Structural model 82
5.3.2 Grid representation 82
5.4 Datapath Reallocation 83
5.5 Phase I: Components Reallocation 85
5.5.1 ALUs reallocation 85
5.5.2 Registers Reallocation 93
5.6 Phase II: Reallocation with Layout Consideration 95
5.7 Results 96

6. VHDL Generator: A Bridge to Lower Level Synthesis 99
6.1 VHDL: Quick Overview 99
6.2 VHDL: A High-Level Synthesis Perspective 100
6.3 Describing The Output 105
6.3.1 Data Path Description 105
6.3.2 Controller Synthesis 108
6.4 From Concept to Silicon: An Example 112
6.4.1 The Differential Equation Example 112

6.4.2 Design capture and scheduling 112
6.4.3 Testable allocation and test points selection 114
6.4.4 Synthesizing The Output 116

7. Conclusion and Future Work **132**
7.1 Resource Allocation and Reallocation Techniques in High Level Synthesis with Testability Constraints 132
7.2 Future Research 133

A. Publications **135**

Bibliography **136**

List of Figures

1.1 Design abstraction levels 2
1.2 A hypothetical Silicon Compiler Design Flow 3
1.3 Two sides for VLSI Design: design and manufacturing 4
1.4 The high-level synthesis task 7
1.5 (a) Behavioral description in VHDL, (b) Scheduling alternatives . 9
1.6 Allocation possibilities corresponding to various schedules 10
1.7 Testing a circuit using BIST 13

3.1 SYNTEST General Organization 29
3.2 Basic Testable Functional Block (TFB) 30
3.3 Allocation in SYNTEST: design and test cost estimation determines the next allocation move. 36

4.1 (a) Testable Functional Block, (b) Self Testable ALU with Self-Adjacency, (c) Non-Observable ALU due to Self-Adjacency 44
4.2 Illustrating the structural testability notion with different designs 48
4.3 Design and Test Spaces 49
4.4 (a) Self Testable ALU with Self-Adjacent register, (b) Non-Testable ALU with Self-Adjacent registers 51
4.5 (a) Biquad filter DFG, (b) Biquad Module Allocation graph. . . . 52
4.6 Initial data path (IDP) for the biquad example 53
4.7 Intermediate solution after merging TFB *a0* and TFB *a1* 53
4.8 Datapath Allocation Algorithm 56
4.9 Merging Two TFBs 57
4.10 (a) Initial TFBs, (b) Resulting merged one 59

4.11 (a) VHDL case statement; (b) CDFG reprsentation (shaded lines are control lines) 61
4.12 Biquad final solution 67
4.13 Fault coverage for the wave filter example 71
4.14 Area decrease in the wave filter after test points selection and trade-offs 72
4.15 Fault coverage for the AR filter example 72
4.16 Area decrease in the AR filter after test points selection and trade-offs 73
4.17 Scheduled DFG for the trigo example 73
4.18 Fault coverage for the trigo example after injecting one, two, and three additional test points 74
4.19 Relative relationship between test overhead and test time for the trigo example 74

5.1 Design and Test Spaces 78
5.2 The Hal example before reallocation 81
5.3 A partial grid like representation for the differential equation datapath 82
5.4 Reallocation Approach 83
5.5 Simple example 85
5.6 Move tables for the runing example 91
5.7 Data Path charts for our runing example 91
5.8 (a) Move table for − using Figure 5(a), (b) Data path after reduce ALU1, (c) After reduce ALU2. 91
5.9 The Hal example after reallocation 92
5.10 Registers Reallocation: initial solution from wave 94
5.11 Registers Reallocation Algorithm 94
5.12 Registers Reallocation: final solution from wave after reallocation 95
5.13 Results from wave filter after applying Reduce strategies (Components area only) 97

5.14 Results from wave filter after Phase I and Phase II 98

6.1 From behavior to silicon: Process 101
6.2 SYNTEST Design Model 105
6.3 Data Path Description in VHDL 106
6.4 Linear Feedback Shift Register 106
6.5 BILBO register: (a) Schematic; (b) Cell description; (c) Modes of operations 108
6.6 Ring State Transition Diagram 109
6.7 Controller State Diagram with Conditional Branches 111
6.8 HAL Differential Example in VHDL 113
6.9 Differential equation schedule 114
6.10 Differential equation data path 115
6.11 Top level chip description 117
6.12 Top level chip description: Controller 118
6.13 Top level chip description: data path 119
6.14 Signals connecting the data path and the controller components . 120
6.15 Partial controller behavioral description (Time step 3 is not shown) 121
6.16 Partial data path description (some components along with components and signals declarations are not shown) 122
6.17 Chip structural view in COMPASS 123
6.18 Register behavioral description in VHDL 124
6.19 BILBO behavioral description in VHDL 125
6.20 Four-to-one behavioral multiplexer description in VHDL 126
6.21 A complex ALU behavior in VHDL 127
6.22 Data path structural view using COMPASS 128
6.23 Partial Data path structural view using COMPASS 129
6.24 Partial controller view using COMPASS 130
6.25 Chip final layout 131

List of Tables

4.1 Designs Comparisons for the HAL Differential Equation example . 69

4.2 Designs Comparison for the FACET Example 69

4.3 Test overhead before and after tradeoffs 70

4.4 Test results summary from the wave filter 70

4.5 Test results summary from the AR Filter filter 71

6.1 Design comparisons for the Differential Equation example after using COMPASS 116

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 from many of the details of the logic and circuit levels. Within the design hierarchy, high-level synthesis is placed at the register-transfer level. In the following 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. Thus, every electronic system can be described at the *behavioral domain*, *structural domain* and the *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.

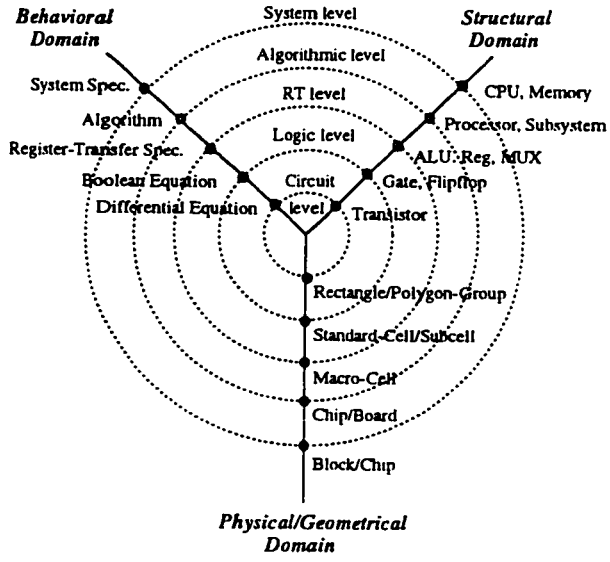


Figure 1.1: Design abstraction levels

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

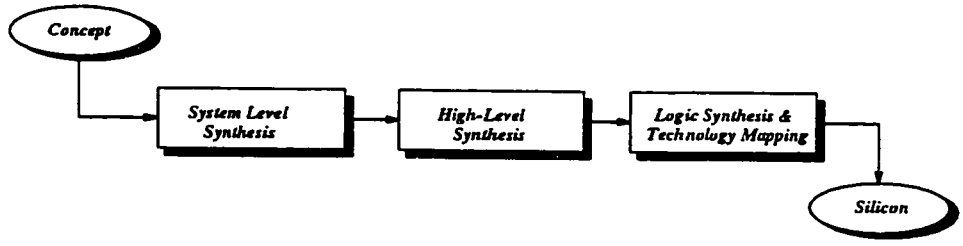


Figure 1.2: A hypothetical Silicon Compiler Design Flow

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 datapath 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 of 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.

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.

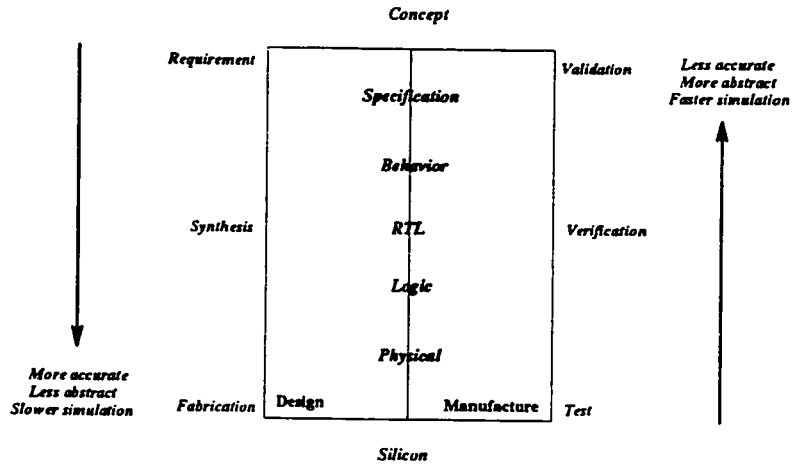


Figure 1.3: Two sides for VLSI Design: design and manufacturing

One of the main premises of synthesis is that the resulting designs are correct by construction. However, an important question arises, which deals basically with the verification and validation (V&V) of synthesis tools and algorithms. Verification refers to the set of activities that ensure that the design correctly implements the intended function and it can be best summarized with the question: are we building the product right? Validation on the other hand refers to a different set of activities that ensure that the hardware is "traceable" to the initial behavior and it is best summarized with the question: are we building the right product? This problem is not an easy one to solve due to the problem complexity. Furthermore, the verification process may require a detailed design model that requires sophisticated algorithms. There are two directions to solve this problem [Gajs92]. The first approach, the *capture-and-simulate* approach, believes that good design knowledge cannot be automated, and design automation tools should capture various aspects of design and verify them predominantly through simulation. This approach relies mostly on building components in a bottom-up fashion. The second approach, *describe-and-synthesize* approach, believes that design automation tools can outperform human designers and that CAD algorithms can search the

design space more thoroughly in a top-down fashion and find nearly optimal designs. In this approach, designers describe the intent of the design and design automation tools add the detailed electrical and physical structures.

In this Thesis, we subscribe to the second approach, and we intend to investigate design models, synthesis algorithms and a novel synthesis for test environment. We believe that this approach is more suitable for the design of complex systems and that VLSI technology has reached the point where such methodologies, especially high-level synthesis, can be very effective.

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], but the high-level synthesis area is not mature yet [McPC90]. 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.

1.3 High Level Synthesis Task

High-level synthesis is the design and implementation of a digital circuit from a behavioral description that describes the function to be performed by the circuit in an algorithmic form given a set of goals and constraints. From the input specification, the synthesis system produces a description of a data path, that is, a network of registers, functional units, multiplexers, and buses. The synthesis

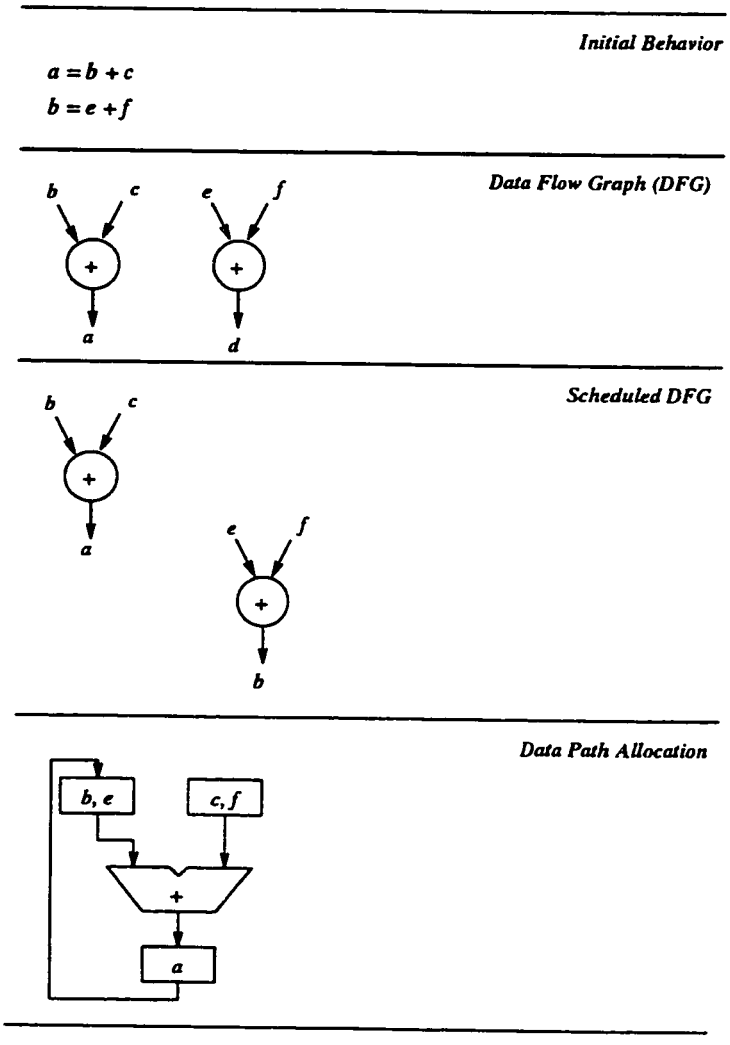


Figure 1.4: The high-level synthesis task

must also produce the specification of the control path. There are many different structures that can be used to realize a given behavior. One of the main tasks of high-level synthesis is to find the structure that best meets the constraints while minimizing other costs. For example, the goal might be to minimize area while achieving a certain required processing rate [McPC90]. The system to be designed is usually represented at the algorithmic level by a programming language such as Pascal or by a hardware description language such as Verilog or VHDL. The algorithmic description is parsed and represented by a Control Data Flow Graph (CDFG) [OrGa86] that preserves the data flow and the control information. The nodes in such graphs represent the operations that are performed on the data (edges) coming into them. Arcs or edges leaving the nodes correspond to the results produced by the operations represented by the nodes. The next two steps in the synthesis process, *scheduling* and *allocation*, are considered the core of transforming the behavior into a structure.

1.3.1 Scheduling in High-Level Synthesis

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:

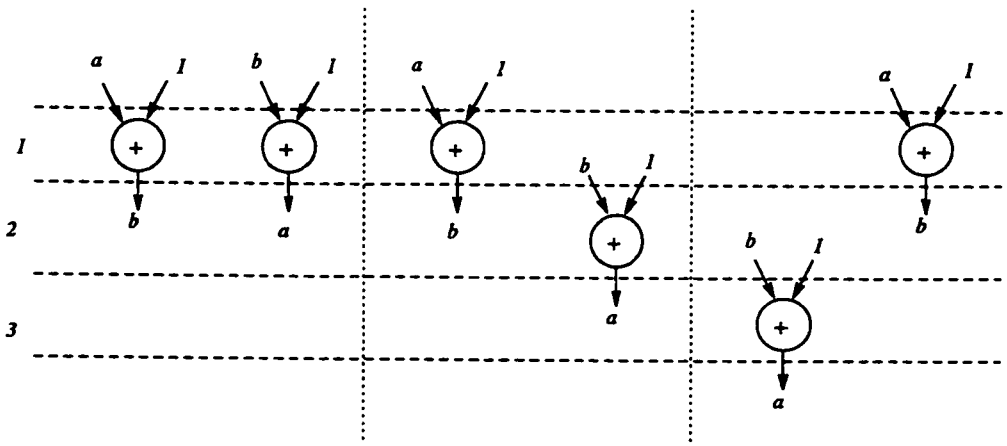
1. *The design timing:* Scheduling fixes the overall timing of the design, illustrated by the maximum number of control steps. This determines the overall design performance.

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.

```
Entity Test is
  Port(a, b: InOut bit);
End Test;

Architecture test_behavior of Test is
begin
      b <= a + 1;
      a <= b + 1;
end test_behavior;
```

(a)



(b)

Figure 1.5: (a) Behavioral description in VHDL, (b) Scheduling alternatives

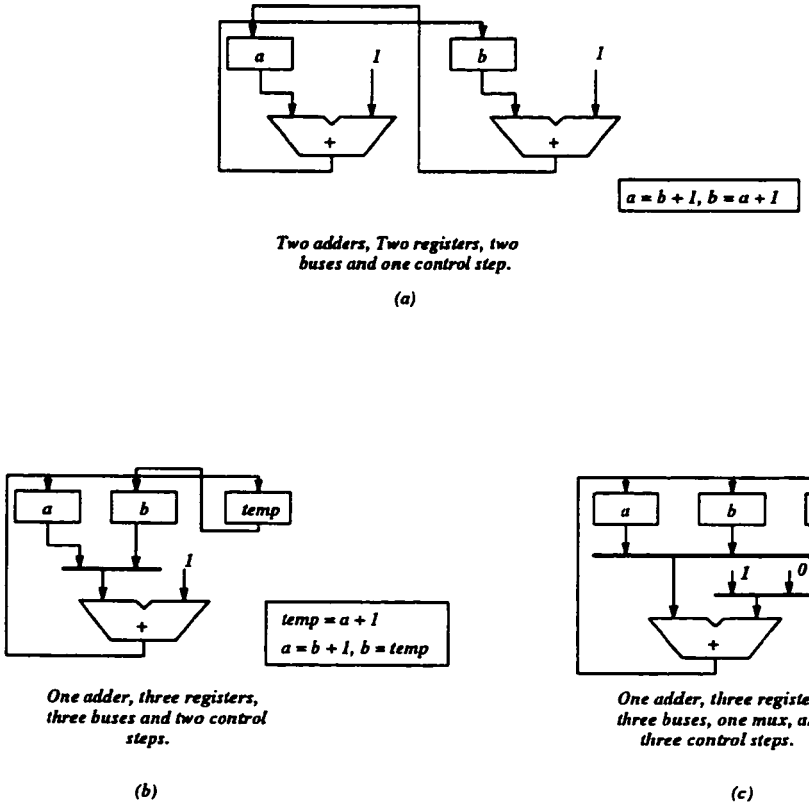


Figure 1.6: Allocation possibilities corresponding to various schedules

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.

1.3.2 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, busses, 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 as 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.

1.4 Why High-Level Synthesis?

Computer-aided design tools in general and high-level synthesis in particular are gaining acceptance in industry. There are many reasons for this trend, in particular:

- *Increase designer productivity* by moving the design process to a higher level of abstraction. Thus, the designer can now think at a conceptual level rather than at the gate level.
- *Improve design quality* since high-level synthesis tools allow for quick design exploration due to the automation of the optimization algorithms. Thus, designer can easily explore design alternatives.
- *Consistency among abstraction levels* since the *pre-* and *post-* synthesis models are guaranteed to be equivalent because of the automated bridge between the behavioral level, the RTL level and the gate level.
- *Reduction of silicon literacy requirement* since the designer does not have to be familiar any more with the details and peculiarities of each ASIC vendor's library technology specific aspects of design.
- *Technology-independent designs* since high-level descriptions are technology-independent in nature and thus retargetable, enabling the deferring or changing of vendor library selection.

The trend toward design automation is expected to continue to increase in the next decade.

1.5 Design For Testability

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 1.7). 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*.

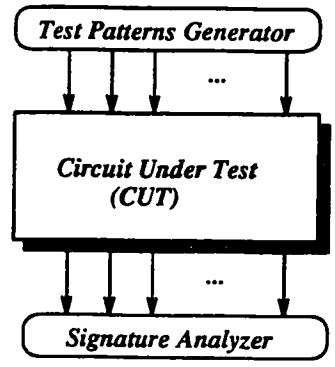


Figure 1.7: Testing a circuit using BIST

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.*

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) techniques emerged as a solution that aims at efficient and cost effective testing by enhancing the controllability and observability of the circuit under test. *Controllability* means that the input terminals or devices of the circuit under test can control the output, while observability means that the circuit under test can be observed at some output terminals or devices. Within the scope of DFT, Built-In Self-Test (BIST) techniques were proposed. 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 by 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 decreased 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.

1.7 High-Level Synthesis and Design for Testability

Recent advances in high level synthesis made DFT consideration at higher levels imperative in order to keep pace with the capability of such systems. However, only few researches addressed the testability problem in coordination with high

level synthesis [Been90, Catt89, AbBr85, GeEl88, Avra91]. In essence, there are currently three approaches to this problem in high level synthesis:

1. *A macro template* approach which ensures that DFT is used as a building block for structural level system design. The testing objective is achieved by handling each macro cell separately. However, this approach does not allow any flexibility in design space exploration which is one of the important aspects of high-level synthesis.

2. *A testability insertion back-end* approach [AbBr85, GeEl88] where a post-synthesis module for the synthesis system is added for converting a register-transfer level design into a testable one. Some of these systems provide a redesign loop that can be be initiated if the design does not fulfill the constraints supplied by the user. However, the main problem with this approach is that the post-synthesis test embedding process can reveal serious problems too late — problems with performance and area, for example, that the system could have thought up front if testability was considered at an earlier level.

3. *Test registers allocation* such as in [Avra91] where a register allocation method which generates self-testable designs based on BIST schemes is used. The approach aims at minimizing the number of self-adjacent registers in the design and resolving their conflicts using some costly test structures (CBILBOs, see chapter 4). This method uses pseudorandom techniques and assumes that the ALU binding has been generated. Thus, there is not much room for coordination and tradeoffs between the register allocator and the binder.

In conclusion, none of the above systems consider testability as a fully coordinated process with the design phase. In order to reduce the penalties described previously, insertion of BIST hardware should be tightly coupled with the design process, and the insertion should be fully automated in order to permit the exploration of various alternatives and to allow various design and testability tradeoffs at the system level.

1.8 Problem Definition and Thesis Outline

This research aims to develop new methods and tools for the automatic synthesis of data path at the register-transfer level (RTL). A major effort in this research

involves the development of a prototype high-level synthesis system that satisfies the following criterias:

1. *Rapid exploration of various design and test tradeoffs* in a short turnaround time. This includes tradeoffs that are common between both disciplines.

2. *Introducing testing during the design process as a structural property.* This is accomplished by introducing an *allocation method with self-testability* followed by a test point selection method. The data paths generated are self-testable by construction under the BIST test methodology.

3. *Design resynthesis* which is accomplished by modifying designs already synthesized in order to meet new design requirements and satisfy new constraints.

4. *The capability to bridge the gap between behavioral synthesis and logic synthesis.* This is accomplished using a logic and layout synthesis tool. We use VHDL as our interface vehicle, and validate our approach using an educational tool, ALLIANCE,[1] in addition to a commercial tool from COMPASS Design Automation.

In this Thesis, we discuss and investigate solutions for the above issues as they are incorporated into our prototype synthesis for testability system, SYNTEST. Further, we investigate the tradeoffs between various design and test decisions. In chapter 2, we review works which are related to our research. Later, we describe SYNTEST in chapter 3, and discuss in depth the datapath allocator in chapter 4. The reallocation process is discussed in chapter 5. We discuss the VHDL generator along with the underlying data path and controller hardware models in chapter 6. We put it all together by illustrating a typical journey from behavior to silicon in the same chapter.

[1] ALLIANCE is available in public domain from Université Pierre et Marie Curie, France

Chapter 2

Background

Many synthesis systems evolved in the past decade. This chapter discusses some of the most successful ones, including two for synthesis with testability. The chapter reviews also an expert system for testability insertion, TDES, which was proposed as a post-synthesis tool for testability back-end insertion.

2.1 Overview of Synthesis Systems

We present some of the early high-level synthesis first. While many other systems evolved in the past decade such as Mimola [Marw86], Olympus [DeMi90], ADPS [PaKo90], we elect to discuss the HAL system, the VSS system, the Facet system, and MAHA among the high level synthesis systems; RALLOC and CATREE among synthesis with test systems.

2.1.1 The Hardware ALlocator

HAL is one of the early systems, and was developed at Carleton University in Canada by Pierre Pauline [PaKn89]. One of HAL's innovations is the introduction of a new effective scheduling scheme, *forced directed scheduling* (FDS). HAL is mainly comprised of the following three modules:

InHAL Module

The InHAL module takes a DFG, number of control steps, clock cycle time, propagation delays of operation types, and a set of functional unit types as input. It

generates a scheduled DFG in two runs in coordination with the MidHAL module.

InHAL first schedules the DFG in as soon as possible (ASAP) and as late as possible (ALAP). The results from both schedules are combined in order to determine the time frame for every operation. A time frame of an operation is a rectangular area with height equals to the number of control steps in which it can be scheduled in, and width equals to one over the height. The time frames concept comes from the fact that an operation can be assigned to any control step between its ASAP and ALAP control steps. The width of the box containing a particular operation represents the probability that the operation will eventually be placed in a given time slot. Next, Distribution Graphs (DGs) are constructed by adding up the probabilities of each type of operation at each control step of the DFG. The resulting distribution graphs (DGs) indicate the concurrency of similar operations. For each DG, the distribution in control step i is given by:

$$DG(i) = \sum_{Operation\ Type} Prob(Operation, i) \tag{2.1}$$

where $Prob(Operation, i)$ is the probability of an operation at control step i.

The next step is to compute the force associated with each node. This is done by computing the *self forces* first. The self forces are quantities that reflect the effect of an attempted control step assignment on the overall operation concurrency. The self force associated with the assignment of an operation to a control step j ($t \leq j \leq b$) is given by:

$$SelfForce(j) = \sum_{i=1}^{b} [Force(i)] \tag{2.2}$$

For a given operation whose initial time frame spans control steps t to b ($t \leq b$), the force is given by:

$$Force(i) = DG(i) * X(i) \tag{2.3}$$

where DG(i) is computed by equation 2.1, and X(i) is the change in the operation's probability.

Finally, to compute the total force, the *predecessor and successor* forces must be accounted for. This comes from the fact that scheduling an operation is equivalent to reducing its time frame to one control step. This modification will propagate to the time frames of the predecessor and/or successor operations.

Once all forces are computed, the operation-control step pair with the largest negative force (or the least positive force) is scheduled first. The distribution graphs and forces are updated and the process is repeated until all nodes are scheduled.

MidHAL Module

The MidHAL is implemented as a rule-based expert-system. It accepts as input the scheduled DFG, and returns the type of ALU's allocated, the number of ALU's of each type as well as their propagation delay and their area cost which includes interconnect cost.

ExHAL Module

The ExHAL module performs binding of DFG operations to allocated ALU's, registers and interconnect allocation to map the DFG onto the final data path.

2.1.2 The VHDL Synthesis System

The VHDL Synthesis System (VSS) was developed at the University of California, Irvine by Gajski's group [LiGa88]. The system has the advantage of being the first to use VHDL as an input and output interface media.

VSS defines a restricted subset of VHDL which is supposed to be suitable for synthesis. The *graph compiler* parses the input description and generates a control data flow graph (CDFG). Each section statement in the VHDL input is represented by a flow graph, interconnected based on read and write accesses of common signals. The system provides a *graph critic* which transforms the various

VHDL representations into a unique construct, which represents the hardware being described. The optimized CDFG is then processed by the *design compiler* which has two modes. The first mode is a register-transfer level mode and in which the number of units and connections are optimized by sharing. The second mode is a behavioral mode and in this mode the number of registers, units and connections are optimized through sharing. Nodes in the flow graph are then replaced by a single or a combination if available architectural components from the generic component library. Finally, a *net list generator* creates a structural VHDL description of the output.

One comment about VSS is its lack of testability analysis and incorporation. This means that the designer will either have to make the design testable manually or by the use of some testability insertion tool. We note that this comment is valid in all the subsequent systems.

2.1.3 The Facet System

The Facet system [TsSi86] is one of the earliest synthesis systems. It was developed at Carnegie-Mellon as a part of the Design Automation Assistant (CMU-DAA). Facet takes a value-trace (VT) like specification for input which preserves all the data flow and control flow information. The system compacts the original operation sequences in an As Early As Possible (AEAP or ASAP) manner. One of the innovations of the Facet system is the use of the *Clique Partitioning Problem* to minimize the number of storage elements, data operators, and interconnection units.

To allocate the registers, Facet constructs a graph where each vertex corresponds to a variable. An edge between two vertices exists if the two variables life spans are disjoints. The graph is then partitioned into cliques. The number of these cliques corresponds to the number of registers needed, and a register is allocated for those variables in the clique. For the operation assignment problem, a

compatibility graph is constructed of the nodes corresponding to the operations. An edge exists between two vertices if the operations are not scheduled in the same control step. The graph is partitioned into cliques, and functional units are allocated for these cliques. Finally, the connection allocation is performed in a similar manner. A graph is constructed where each vertex represents a point-to-point connection, and where an edge will exist between two vertices if the two corresponding connections will not be used simultaneously. After partitioning into cliques, Facet allocates a bus for every clique.

The clique partitioning problem has been proved to be NP complete; however, Facet uses a heuristic algorithm based on a neighborhood property to solve this problem in a reasonable time.

Comments on Facet

1. Facet uses ASAP scheduling which is local both in the selection of the next operation to be scheduled and in where it is placed [McPC90]. Furthermore, ASAP scheduling assumes unlimited number of resources.
2. During the register allocation process, the effect of such allocation on the interconnection cost is not taken into consideration.

2.1.4 MAHA/REAL

The Modified Automatic Hardware Allocator

The Modified Automatic Hardware Allocator (MAHA) is a part of a larger system developed at the University of Southern California, The Advanced Design AutoMation System (ADAM) .

MAHA [PaPM86] is divided into four sections: library generator, data flow graph partitioner, critical path analyzer, and allocator. The allocation is bounded by maximum time and/or area cost for the data flow graph. MAHA selects the hardware library, and analyzes each type of operation performed at a node and generates a list of all components in the library which can perform it. Each list is

reduced later to one entry which has as its properties the average speed and cost of all components in the original list which perform the same operation.

Next, MAHA determines the critical path or paths in the DFG by using another program, Clocking Scheme Synthesis Package (CSSP). The maximum delay of any node is usually chosen as the cycle time. Then the DFG is optimally partitioned into n stages where n is less than or equal to the number of nodes in the critical path. After partitioning the DFG into n stages, if the critical path delay exceeds the time bound set by the user, CSSP makes an attempt to partition the DFG into n-1 steps. The operations on the critical path will be next assigned to average components on a first-come first-served basis.

For the off-critical path nodes, MAHA introduces a new notion, the *freedom* of a node which is defined as the time range in which operation can be performed without lengthening the critical path. The off-critical path node with the smallest freedom is chosen first for allocation. If the freedom is too small which means that it must occur at a specific stage, that stage is assigned to the node. Otherwise, it is assigned to the first control step that does not necessitate an extra resource to be selected. If all allowable control steps necessitate an extra resource, the operation is scheduled in the earliest control step.

Once the stage has been assigned to the chosen off-critical path node, allocation occurs in an identical manner to that of the critical path nodes. After allocating the chosen off-critical path node, MAHA starts again by calculating new freedoms for all remaining nodes. This process will continue until all nodes have been allocated.

The REgister ALlocation

REAL [KuPa87] is also a part of the ADAM System, and it is responsible for the register allocation for the system. REAL makes use of an algorithm for track assignment in channel routing, the Left Edge Algorithm (LEA). The life span of

a variable is mapped onto the left and right edges. REAL produces an optimum allocation of registers for non-pipelined designs given a lifetime table for a specific DFG.

A quick comment on REAL is that despite the fact that LEA allocates minimum number of registers, it does not take into consideration the effect of such connections on the connection cost.

2.2 Overview of Synthesis Systems with Testability

2.2.1 CATREE

CATREE (Computer Aided TREE) is a high-level synthesis system that incorporates test considerations at the system level. The system parses a given input specification, described in an internal format and constructs a binary search tree data structure. The mincut partitioning algorithm is then used in order to balance the tree operators based upon a score that includes assigned weights for the same operator type, shared variables and non-conflicting fire times. The functional units are then allocated using a bottom up traversal algorithm. The assignment is done based on non-conflicting operators whose functionality exists in the library. Operators from different trees are then clustered.

The next stage in CATREE is to repartition the functional unit tree, add test operator leaves and perform test scheduling. The target in this phase is to balance the functional units useful for test scheduling, multiple chain definition, or non-uniform test incorporation. A test operator leaf is added to each functional unit subtree thus transforming the test tree. The test operators have test input and output variable which together describe the test mode design behavior in a test trace. It is assumed that the test methodologies are implemented in order to

allow test patterns to be applied to, generated at, or observed at the functional units inputs or outputs. Finally, CATREE allocate registers and interconnect in addition to floorplanning. The register allocation is performed in a bottom up tree traversal algorithm while the interconnect allocation involves a top down traversal algorithm. The floorplanning phases assigns x and y dimensions at each level in the tree; thus, transforming the binary search tree into a two dimensional binary tree.

Comments on CATREE

The several phases in CATREE treat all various tasks in the synthesis process as being independent. Thus, the interaction among functional units allocation, test allocation and register allocation is non-existing. This may lead to violation of initial synthesis constraints after test incorporation which is a post synthesis process. CATREE proposed a feedback loop after test insertion; however, such loop was never implemented or at least reported.

2.2.2 The Register Allocator System (RALLOC)

The register allocation method (RALLOC) proposed by Avra [Avra91] is similar to the allocation method which we proposed earlier [PaCH91] with the difference that the method does not prevent self-adjacent registers but rather minimizes them. In what follows, we describe the method briefly.

RALLOC starts with a data flow description and assumes that functional units binding is already accomplished using a synthesis tool. RALLOC constructs next a register conflict graph where a node represents a register that must store a variable and edges between nodes indicates that the variables associated with the nodes cannot be assigned to the same register. The conflict among registers has two sources: a scheduling source and a testing source. The scheduling conflict exists when different values are alive in the same clock cycle. The testing conflict

arises whenever one node represents a variable that is an input to an operation and the other node represents a variable that is the output of the same operation. The reason is to guarantee that no self-adjacent registers are synthesized. There is one exception to this rule and that is when two operations in consecutive clock cycles in the data flow graph are assigned to a single function block and the output operation is the input to the other. In this case, RALLOC assigns the node associated with both the input and the output of the function block to be a CBILBO registers. Next, RALLOC colors the nodes and assigns the nodes with the same color to the same register. Finally, multiplexer assignment is performed using a straightforward method.

Comments on RALLOC

1. RALLOC assumes that binding was generated using a synthesis tool; thus, no interaction is possible between the test allocator and the functional units binder.

2. The CBILBO structures are very expensive in cost (about 1.75 the cost of a BILBO). The presence of such structures could be prevented if there was an interaction between the binder and the test allocator.

2.3 The Testable Design Expert System

The Testable Design Expert System (TDES) [AbBr85] is a knowledge-based expert system for designing testable VLSI chips. The system, proposed at USC and used later as a back-end tool for the ADAM system, evaluates and makes choices among numerous existing DFT approaches.

TDES takes for input a register-transfer level design possibly generated using a high-level synthesis tool. The system database has several DFT techniques and Testable Design Methodologies (TDMs). Frames are used for representing knowledge about the TDMs. The information slots in the TDM frame can be classified into three parts. The first part is a *template* that describes the TDM's

structural architecture and conveys information about the type, style, and size of the testable subcircuit or *kernel* to which the TDM is applicable. The template also describes the BIST structures needed by the methodology and the connection paths that must connect them to the kernel. The second part in the TDM frame is a *test plan* that describes the sequence of actions which must be performed to execute a test. Finally, the TDM frame contains information about *test measures.* Some of the test measures are constants, while some others are functions of the actual circuit such as the PLA personality matrix.

The first step in processing a given circuit is to partition it into testable subcircuits or *kernels* where every combinational kernel has a well defined design style. The kernels correspond to the maximal basic structures, that is a basic structure not contained within a larger basic structure. Structures in the neighborhood of a kernel can be used to aid in the testing of that kernel. TDES will then assign various weights to every kernel. These weights include a size weight, a regularity weight, a type weight, a criticality weight, an observability weight, and a controllability weight.

The system processes next the kernels with higher weights first since these kernels may have the least number of applicable TDMs. An applicable TDM is sought for each kernel using a matching process. This is done in two steps:

1. Match every structure in the TDM template with a structure in the circuit.
2. Match every connection between structures in the TDM template with an I-path[1] between the corresponding circuit structures. This embedding may require *modifying or adding new structures to the circuit.*

If more than one TDM is applicable to a kernel, TDM measures are used to choose between them. Once TDES is done, every part of the original circuit will have its own test methodology, and will be tested on its own.

[1] An I-path [AbBr85] or an identity-transfer path from output port X of a structure S_1 to an input port Y of another structure S_2 denoted by P(S_1:X $\rightarrow$ S_2:Y) exists if the data at port X can be transferred unchanged to port Y.

2.3.1 Comments on TDES

1. An immediate comment is that there is no interaction with the high level synthesizer. There is no feedback to the synthesizer to explore trade-offs between design and test decisions and test parameters.

2. The test implementation is based on a local structural enhancement without global considerations.

Chapter 3

The SYNTEST Design System

3.1 Motivation

SYNTEST deals with a new approach to high-level synthesis with self-testability. What distinguishes SYNTEST from other high-level synthesis systems is the consideration of testing cost (test time and overhead) in addition to the conventional constraints (area and performance) during the design process.

The SYNTEST system, whose block-diagram is shown in Figure 3.1, operates on a behavioral input description and generates structurally testable RTL designs including optimal selection of test points (test registers). Currently, we have implemented a design path for BIST methodology [McCl85]. At present and in the following we assume a test register is a LFSR which may be configured, in circuit test mode, as either a test pattern generator register (TPGR), a multiple input signature register (MISR), or possibly a Built-In Logic Block Observer (BILBO) register [KoMZ79]. There are two important aspects underlying the SYNTEST system:

- A structural aspect based on a *structural testability model* at the register-transfer level.
- A functional aspect based on *the module design and function as well as on its test characteristics.*

The key element of the structural testability model is the *Testable Functional Blocks* (TFBs) which consist of a combinational block and a register at the output, Figure 3.2. The TFB structure is directly related to the DFG nodes. Thus, the

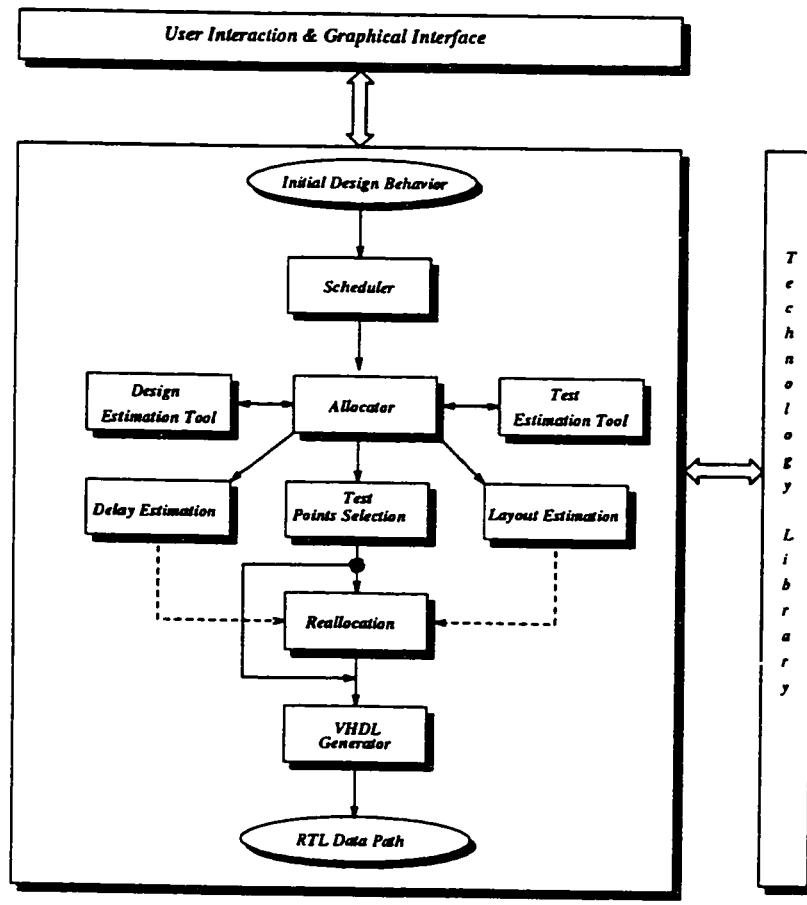


Figure 3.1: SYNTEST General Organization

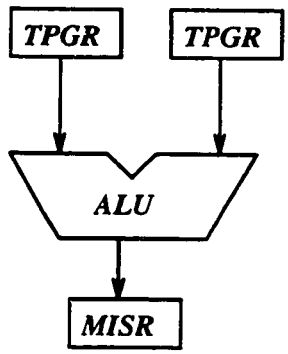


Figure 3.2: Basic Testable Functional Block (TFB)

operation will be mapped to the ALU of the TFB and the DFG variable to the register at the TFB output. The basic requirement of the structural testability model is that it does not allow I/O self-loops that involve TFB input ports and the same TFB's output register. This is because we cannot use the same register both as a TPGR and an MISR simultaneously in a loop.

The second important aspect of SYNTEST concerns the exploitation of the logic module (ALUs) characteristics during allocation with the following objectives: (a) reduce BIST hardware overhead (test registers) while satisfying fault coverage requirements; (b) trade-off BIST cost with module (ALU) operator cost during high level synthesis; (c) provide testability estimation, such as fault coverage and number of test patterns, during the system level design process.

These two important aspects have had a major influence in the development of the SYNTEST components. This is reflected in the block diagram in Figure 3.1 which shows the tight interaction between the resource allocator and the test estimator. A brief description of SYNTEST follow.

3.2 Design Representation

The initial circuit behavior in SYNTEST is described using VHDL which is parsed into an intermediate data flow representation. The design constraints are specified

interactively at various stages of the design process using dialog boxes. The output of the system is a structural VHDL description for the controller and the datapath. This provides a means to later verify the functionality and correctness of the synthesized via simulation.

3.2.1 Input

The input to SYNTEST is a VHDL description of a circuit. VHDL has become a standard for describing, designing and simulation of IC circuits. Thus, it has many semantics which are suitable for simulation rather than synthesis. VHDL is a "multilevel" simulation language, which can be described at the *behavioral*, *dataflow* or *structural* level. The input to high level synthesis is a behavioral level description leading to a structural one, in the form of a state-machine implementation. The dataflow description is usually used for logic synthesis and will not be addressed in this Thesis. We use SYNTEST in order to transform a VHDL behavioral description to a structural one. We discuss the synthesis subset along with the VHDL generation process in chapter 6.

3.2.2 Output

The output of SYNTEST is a hierarchy of structural components, mainly registers, ALUs, and multiplexers. The next step in the *synthesis pipeline* is to synthesize these individual structural models and attach pads to them. In the current version of SYNTEST, we took two approaches to accomplish that. The first approach is based on an educational tool, ALLIANCE while the second approach is based on a more powerful, industrial tool, COMPASS. In what follows, we describe briefly the interface process which was done merely through VHDL. We note that SYNTEST has currently two options to interface to either tool through graphical buttons.

Linking SYNTEST to ALLIANCE

To link SYNTEST to ALLIANCE, the following procedure was followed. First, the behavior of the cells (adders, multipliers, muxes, registers, ...) was described behaviorally at the logic level using boolean equations. The cells behavior was then synthesized in order to create one bit structural components using ALLIANCE logic synthesis tool (logic). Multiple bits cells were then created manually by replicating the individual cells. Those individual cells were declared by SYNTEST as COMPONENTS and used consequently. The last stage in this process was deriving layout for the data path, and which was accomplished using ALLIANCE standard cell and router (scr). We derived layouts (GDSII format) for most of the benchmark examples. It was not possible to generate layout for the controller since ALLIANCE cannot handle the process statement.

Linking SYNTEST to COMPASS

Linking SYNTEST to COMPASS Design Automation tools was much more efficient and fully automated due to the fact that COMPASS already has a VHDL ASIC Synthesizer. Our task in this case was automated as follows.

Once the high-level synthesis process is completed, SYNTEST creates a VHDL structural description for the datapath along with a finite state machine description for the controller. SYNTEST also creates a behavioral VHDL description for every cell in the datapath. The root VHDL files are then passed to the COMPASS ASIC Synthesizer which synthesizes the individual components. The logic details for the cells could be viewed using COMPASS Logic Assistant while the chip layout can be later derived using COMPASS Chip Assistant. We will describe this process in more details using an example in chapter 6.

3.3 Tools Overview

SYNTEST takes a behavioral circuit description as described above and performs next various synthesis tasks such as *scheduling* and *data path allocation.* We discuss next the main components in the SYNTEST system including the DFG generator, the scheduler, the allocator, and the estimation tools.

3.3.1 DFG Generation

SYNTEST parses the initial behavior and generates a data flow description, expressed in a transfer language suitable for synthesis. The language has input and output ports and supports various constructs such as conditionals and loops. The DFG generator DFG generator tool supports a graphical editor that allows the designer to extract a VHDL or data flow description out of a graphical description. This makes the tool very desireable to use especially if the designer is not familiar with VHDL.

3.3.2 DFG Scheduling

Scheduling assigns operations to control steps so as to minimize a given objective function while meeting the constraints [McPC90]. The scheduling phase determines the hardware cost-speed trade-offs of the design, and has been proven to be NP-complete. Thus, heuristics are introduced to obtain a fast schedule with respect to time required to execute the original code sequence or a balanced schedule with respect to operations concurrency. To implement an appropriate scheduler in our system the following requirements were sought:

1. *It should be fast* so that to allow the designer to explore various design possibilities in relatively short turnaround time.
2. *It should be able to incorporate our test constraints* easily so that to yield a schedule suitable to our testable allocation scheme.

3. *It should be able to optimize the schedule* under time or hardware constraints and handle different synthesis applications.

Based on the above requirements, we have developed the *Move Frame Scheduling (MFS)* (MFS), which is the main scheduling tool in SYNTEST. In addition, SYNTEST currently supports As-Soon-As Possible (ASAP) , As-Late-As-Possible (ALAP), and in fact it can support any scheduling method.

The *MFS scheduling method* [NoPa92] uses a transformation technique between the scheduling space and the dynamic system space and designates moves in the scheduling space which correspond to moves towards the equilibrium point in the dynamic system space in order to generate a well balanced schedule. The main advantage of this scheduling method over known techniques is its speed in searching the design space and providing alternative solutions. The MFS tool schedules any given data flow graph under fixed time or hardware constraints and supports various synthesis applications such as mutually exclusive operations, loop folding, multi-cycle operations, chained operations, structural and functional pipelining.

The MFS method also includes an option to schedule and allocate hardware resources simultaneously [NoPa92]. Our motivation is to explore the tight interaction that of scheduler and the allocator. However, this may increase the design space complexity.

3.3.3 Testable Data Path Allocation

Once the behavior has been scheduled, SYNTEST allocates hardware by binding operations to ALU's, variables to registers, and establishing connections between them. Our allocation method merges the data flow variables simultaneously with the motivation to generate easily testable, regularly structured data paths [PaCH91, HaPa93]. The allocation method interacts with SYNTEST design and test cost estimation tools described next. In addition to the method described

shortly, SYNTEST supports an integer linear programming optimization option for the allocation and which is used for near-optimal solutions [Harm91] but at the expense of much longer computational time.

The testable allocator in SYNTEST is based on a stepwise refinement process guided by the difference in data path cost between two consecutive iterations. The allocator first maps the generated schedule to an initial data path structure. Thus, every operation is mapped initially onto an ALU and its variables are mapped to a register; the connections will be established consequently. The allocator chooses all initial register configurations to be both TPGR and MISR. Clearly, the allocator's task is to optimize this initial data path structure by merging all compatible TFBs into more complex ones while, at the same time, selecting a TPGR and an MISR for every TFB, and removing all unnecessary TPGR and MISR . We note here that two TFBs are compatible if it is possible to merge them without causing a resource conflicts or creating self-loops [HuPe87]. All this intermediate information is stored in a multitree graph that we call *module allocation graph (MAG)*. At every iteration, based on the estimators discussed next, the allocator selects two TFBs from MAG and merges them in the intermediate data path. The *test and design estimation tools* will select the modules to be merged by estimating all the cost gains and losses involved based on a synthesis algorithm described in [PaCH91]. The allocation method will be discussed in details in chapter 4.

3.3.4 Design Cost Estimation Tool

The design cost estimation tool predicts the cost of selected components to be shared among different modules. The process produces estimation in terms of *gain* and *loss* in the data path which we have introduced in [PaCH91, HaPa93]. The gain gives a measure of the reduction in the data path cost at every iteration while the *loss measure* is a lookahead indicator during the same process. We discuss the design estimation tool in details in chapter 4.

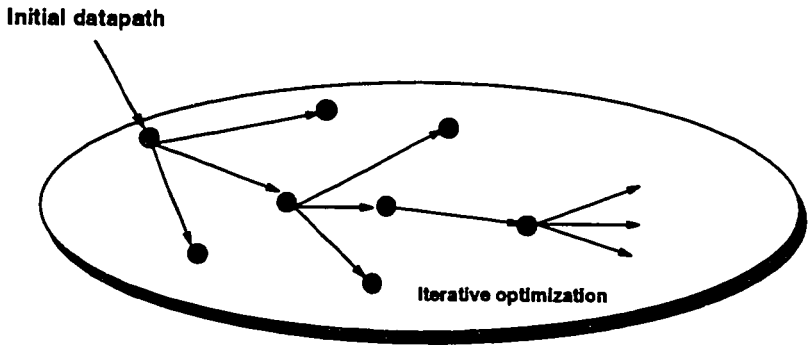


Figure 3.3: Allocation in SYNTEST: design and test cost estimation determines the next allocation move.

3.3.5 Test Cost Estimation Tool

SYNTEST currently supports the BIST methodology based on linear feedback shift registers for both random pattern generation and signature analysis. Based on a system level testability model described next, this cost estimation tool: 1) predicts the controllability and the observability within the intermediate data path generated; 2) selects system registers to be converted into either test pattern generators or signature analyzers.

The Test Cost Estimation Tool chooses a controllable and an observable point for each port in the design. The tool makes use of the functionality of the individual modules and explores possible trade-offs between testing and hardware overhead. The basic trade-off model in SYNTEST is to assign a controllable or an observable point if: 1) the inputs to the module are not random enough; 2) the fault effects of this module can not be sensitized through intermediate modules to an observable point. The above conditions are checked in SYNTEST by means of the following measures, presented in detail in [ChPa91a]:

- *Transparency measure*: This measure is an estimate of the fault propagation ability of the underlying module. A probabilistic approach was used here to decide whether a given register should be observable or not.
- *Output Randomness measure*: An information theoretic approach was used in order to determine whether a given register in the circuit need be controllable

or not. Two measures were developed here: 1) a randomness measure to check whether the module output patterns are random enough; 2) a uniformity measure to test whether the module can generate all possible patterns.

The result of this estimation is to obtain the test cost of the desired design points.

3.3.6 Layout and Area Estimation Tools

The *layout estimator* estimates the area of RTL datapaths by considering the physical length of components within an actual layout model, using analytical formulas in a constructive algorithm. The layout estimator is non-probabilistic and estimates the overall layout with emphasis on area minimization [NoPa92].

The delay estimator analyzes an RTL structure by: a) constructing the *propagation delay graph* from the given control/data flow graph; b) detecting the false paths created by the three sources; c) applying a modified version of the depth first search algorithm to compute the critical path delay and then measuring all register-to-register transfer operations so as to determine the maximum frequency.

3.4 User's End View

SYNTEST supports a graphical interface that provides an efficient means for interaction between the designer and the system. This provides a better design environment for feedback and interaction with the system. Thus, moving the designer from an *iterative synthesis environment to an interactive one.* The general aspects of the system interface are discussed next.

3.4.1 Graphical Interface

The SYNTEST system is supported by an X-Window graphical interface that solicits feedback and design decisions at various stages while keeping the designer

informed on various decisions. The graphics interface supports one level pull-right windows to select among the various major components of the system, and to set up various options related to the specific tools. The system also supports dialog boxes for all the operations that require text inputs and feedback since these have been proven to be a more effective means in graphics interface designs [Drag89]. *Zoom in* and *zoom out* functions are also provided at various levels in the user interface. At any phase during the design stage, any window can be spooled to a PostScript file or to PostScript printer in order to generate a hardcopy. The system graphics interface supports on-line help for the various design steps, and which is critical for those exploring the system or users using the system for the first time. The system supports an effective error recovery and report mechanism as well.

3.4.2 User Interaction with SYNTEST

The user can interact with SYNTEST through the graphics interface. Once the system is started, the system main window frame appears. The frame contains four options (buttons), notably the *behavior*, *scheduler*, *allocator*, and the *register-transfer level optimizer* options. The user will have initially to choose the behavior description file that he or she intends to implement. The behavior option will generate an intermediate file to be fed to the scheduler. We will discuss the DFG generator, the scheduler, the allocator, and the delay and area estimators next.

User Interaction with the DFG Generator

When synthesizing a given behavior in SYNTEST, the user can take one of two approaches. The first approach consists of describing the behavior in VHDL using a text editor. However, a second and easier approach consists of describing the behavior graphically using SYNTEST graphics editor. In what follows, we describe the second approach.

When the user selects the *DFG Generator* button, a pull-right window appears. The user can then invoke the graphics editor which has a set of icons corresponding to different arithmetic and logical operations. The user can select the desired operations (DFG nodes) and place them graphically on the editor. The user can also set the bit width of the nodes graphically which will be reflected later in the synthesized code. The nodes can be later interconnected using line segments in order to establish data and control dependencies in the behavior.

Once the behavior has been described, it can be saved it to a file using one of two formats: SYNTEST DFG format or using VHDL. However, if the user decides to use VHDL, then the VHDL parser must be invoked in order to generate a DFG file. It should be noted here that the VHDL file description will consist of a single process with a restricted data type, sensitive to all input nodes.

User Interaction with the Scheduler

Once the *scheduler* is selected, a pull-right option appears and the file specified in the DFG window will be selected. The user can specify some initial constraints such as the number of overall clock cycles the design need, the type and number of cycles available for the design, and the scheduling type, normal or pipelined. Finally, a scheduling method can be selected from the methods available in the system. If none of the previous options are specified, the system selects the MFS scheduling method by default. The user can then *run* the scheduler and *display the schedule* graphically. The user can then toggle between the nodes *operations*, the nodes *names* and can *show the time steps* at which the nodes are scheduled. At this stage, the *manual rescheduling* option can be used; it allows the user to reposition the scheduled objects (nodes) anywhere between their successors and their predecessors using the mouse. The structure of the original schedule is maintained, and all relations (data dependencies) to the selected object follow the mouse movement as well. The modifications can be saved in a file to be used by

the allocator in the next step.

User Interaction with the Allocator

Once the initial behavior has been scheduled, the *allocator* can be invoked in order to allocate hardware resources for the behavior. The user can select a technology library on which the allocator will base its decisions and optimization techniques; otherwise, the default library will be used. Furthermore, the user can choose between various options which include a set of *design* and *test* options. These options (ALU cost, MUX cost, test registers cost, test style, or a combination of the previous parameters) allow the testable allocator to direct the optimization process in order to suit the user preference.

In the next step, the user can *run* the data path allocator in *one shot*. That means basically that the user will obtain the final solution directly in contrast to the *step by step* option which allows the user to trace the optimization process *graphically* and can thus follow the system design decisions *step by step* by looking at every intermediate data path structure. The *step-by-step* option provides a better feel for the design process as well as a better understanding of the system decisions.

The user can specify at this stage the number of graphic layers or *levels* in which he/she would like to display the data path. The module allocation graph can be shown graphically as well. At this stage, SYNTEST will also generate a symbolic control table for the circuit under design, as well as a test plan for every TFB. In addition, a detailed data path and a controller description in VHDL is generated as well. Note that we use the term test plan here in the context of which registers will be used to test a given TFB, and the hardware configuration of all such registers. The user can obtain such information by clicking on the design entity.

User Interaction with the Delay and Area Estimators

Once the data path has been generated, the layout and area estimators can be invoked in order to estimate the area and delay of the generated design. The layout estimator estimates the interconnect area along with the final layout area. The user can specify the desired aspect ratio along with the underlying technology library.

The next step is to find the critical paths in the schedule and in the data path. This is accomplished by runing the delay estimator. The user can include the components wires in the estimation process as well, resulting in a more accurate delay model. The critical path in the schedule can be shown graphically as well. However, in the case of the data path, we distinguish between two types of critical path. The first type is what we call "the static critical path" which is the worst delay path from primary input to primary output. The second type is "the dynamic critical path" and is due to the fact that the synthesized chip is based on an underlying algorithmic description. Thus, by tracing the algorithm in the final chip, one can estimate the "actual" critical path. In case where there are more than one critical path in the schedule or in the data path, the user can toggle back and forth among them.

Chapter 4

Testable Data Path Allocation

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 a *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.

We present in this chapter a method to solve the above problem. Our method deals with a new approach to high-level synthesis with self-testability. What distinguishes our approach from other high-level synthesis systems is the consideration of testing cost (test time and overhead) in addition to the conventional constraints (area and performance) during the design process. The approach is based on a *synthesis for testability* model that addresses the shortcomings of previous works. The ultimate goal is to *explore the tradeoffs that exist between the design and test processes.* Our method has the following contributions:

- *An improved model for the testable synthesis of RTL datapath* structures from behavioral descriptions based on the BIST methodology. An important feature of this method is that it is driven by a technology library based on a graph allocation heuristic.
- *The flexibility during the allocation phase to handle various testability design styles* depending on the user constraints; that is, strictly and loosely testable designs.
- *A test point selection scheme* which provides the capability to tradeoff design area and delay with test quality (fault coverage). The tradeoff scheme can control *the depth of the sequential path* from controllable points to observable points by removing or injecting additional controllable/observable registers. This injection capability is performed interactively by the designer with the aid of the testable synthesis system.

In section 4.1 we describe the test methodology along with our allocation model and the basis of our test trade-offs scheme. Section 4.2 discusses the testable allocation method along with the cost functions, while section 4.4 describes the test points selection method that we use. Section 4.5 illustrates the test trade-offs scheme using an example. Finally, results are presented and discussed in section 4.6.

4.1 Design and Test Methodology

4.1.1 Background

Design for Testability (DFT) has two main aspects, controllability and observability; the control and observation of a circuit are central to implementing its test procedures. Within DFT, self-testing, specifically Built-In Self-Test, has been getting considerable more attention recently in high-level synthesis systems with the aim to allocate self-testable data path from behavioral description. We base our method on pseudorandom BIST (PR-BIST). The pseudorandom patterns are generated using *Test Pattern Generation Registers* (TPGRs) which are based on autonomous Linear Feedback Shift Register (LFSR) design. The test responses are evaluated using *Multiple Input Signature Registers* (MISR) [AbBF90]. We determine the fault coverage using *logic level fault simulation* and select the test length so as to achieve an acceptable level of fault coverage.

Assume we have the circuit in Figure 4.1(a). To be able to test such a circuit, we allocate the registers at the input ports as TPGRs or simply *controllable points.* Test patterns are collected and observed at the output port which is allocated as an MISR or simply an *observable point.* One of the difficulties in implementing BIST techniques is the register self-adjacency problem,[1] arising due to structures

[1] A register is self-adjacent if an output of that register feeds through combinational logic and back into itself.

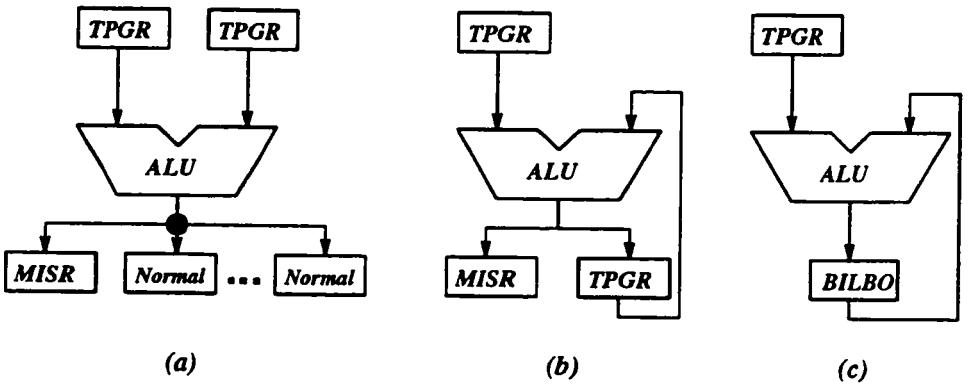


Figure 4.1: (a) Testable Functional Block, (b) Self Testable ALU with Self-Adjacency, (c) Non-Observable ALU due to Self-Adjacency

similar to the one shown in Figure 4.1(c). In this circuit, a *Built-In Logic Block Observation* register (BILBO) is a register that, in addition to its normal operation mode, operates during test mode as an MISR or as a TPGR [KoMZ79]. However, it is not possible to assign the output register as both a pattern generator and a signature analyzer at the same time. Some researchers used the MISR outputs, which are essentially random vectors, as random patterns [KiHT88]. The problem in this approach is that errors in the MISR propagate erroneous test patterns that are applied to the ALU, which then tend to produce errors in the *BILBO.* This problem can be rectified using a *concurrent built-in logic-block observation* (CBILBO) register [AbBF90]. The CBILBO register can operate simultaneously as an MISR and a TPGR. The disadvantage of the CBILBO is that it is very costly in area (about 1.75 times the size of a BILBO [Avra91]) and induces more delay during normal operation mode.

Hudson [HuPe87] showed that when self-adjacent registers are configured as test pattern generators, the additional feedback that made the register self-adjacent can greatly reduce its ability to retain error information as a signature analyzer. Kim [KiHT88] showed that random patterns generated by signature registers are rarely repeated when the number of test patterns is relatively small compared

to the number of possible patterns concluding that signatures registers can be used as test pattern generators. However, we note that this is not the case with self-adjacent registers since, when configured as signature registers, they have an increased probability of aliasing.

4.1.2 Improved Model for Synthesis with Testability

We base our allocation model on the notion of *structural testability* at the register-transfer level. The key element of the structural testability model is the *Testable Functional Block* (TFBs) which was introduced in [PaCH91, HaPa93], and is shown in Figure 4.1(a). A TFB consists of an ALU and a set of input and output registers. There are two registers at the input ports of a TFB which are configured as TPGRs during test mode. The output port of a TFB is connected to a set of registers, one of which is configured as an MISR in test mode. The number of registers at the TFB output port is determined during allocation and is optimized during the synthesis process. Although a basic TFB contains only three BIST registers, two TPGRs and one MISR, it should be noted that these BIST registers may be shared by other TFBs in the datapath. This means that some of these registers may have to be configured as BILBO registers[2]. Furthermore, some other non-BIST registers at the TFB output port may have to be configured as TPGRs *only* if they control the input port of other TFBs.

The main advantage of the model is that we can map operations whose variables list have overlapped life spans to the same TFB. The rationale is to avoid the dislocation of variable instances from their corresponding operations, keeping them close together during the entire allocation process. This proximity is preserved topologically in the generated data path and it contributes to its structural testability. Thus, we map the conflicting variables to different registers at the output layer of the TFB. The resulting data path is guaranteed to be self-testable by

[2]TPGR and MISR, but in different test sessions

construction since it is composed of connected TFBs satisfying the self-adjacency constraint (section 4.2.1).

Another advantage of our model is its ability to control the *sequential depth* in the circuit, which we define informally as the maximum depth from a controllable point to an observable point for a given module in the circuit.[3] Since we always observe the faults at the output port of the ALU, we guarantee a sequential depth of *one* from the TPGRs to the MISRs. However, this depth can be relaxed by the system or interactively by the designer using the notion of functional bypassing to perform test tradeoffs, as we will illustrate in sections4.4 and 4.5. Our method has a minimal or near minimal test hardware overhead and provides favorable results even in comparison to non-testable designs (Section 4.6).

4.1.3 Design and Test Space

As stated earlier, the basic requirement of our allocation model is that it does not allow self-adjacent registers that violates testability consideration, as we have shown earlier. We define designs from this class as *strictly structurally testable* designs; the generic design *a* shown in Figure 4.2(a) is such a design. In this design style, registers at the TFB input ports are configured as TPGRs while registers at the TFB output port are configured as BILBOs. To improve area and performance, we may relax the self-loop connection rule as shown in Figure 4.2(b) *and/or* remove some test attributes[4] which are not needed to test the datapath ALUs as shown in Figure 4.2(c). Designs following this rule are called *loosely structurally testable* designs. The two design examples described above share the characteristic that some components in the datapath cannot be exercised or tested. For example, the MUX at the left input port of design *b* is not testable while the small portion of the bus at the TFB output port (shown in thick line) cannot

[3]Note that this definition is different from the one in [ChAg90, Lee88] who are not using the BIST methodology.

[4]Test attributes are removed using the test points selection method described in section 4.4

be tested in the generic design c. Note that designs in which the self-adjacency rule is totally violated such as in Figure 4.2(d) are called *non-structurally testable designs* and are not acceptable nor generated by our method.

The rationale of considering loosely structurally testable designs is that they provide *a good tradeoff between design cost and testability cost.* Design *a* is better than design *b*, *c* and *d* in terms of testability, but *d* may well be superior to *a*, *b* and *d* in terms of area cost. Designs *b* and *c* are a compromise between *a* and *d*, and it may well be a good compromise in the following sense. Transforming designs from type *a* to type *b* or *c* is a relatively easy process within our testable allocation scheme. However, we claim that moving designs from type *d* into subspace type *b* or *c* is considerably more difficult because of the testability requirements that could involve extensive and costly modifications.

Another advantage of our method is its ability to control the *sequential depth* in the circuit, defined earlier in the previous section. A long sequential depth results in long test sequences to be generated by the test pattern generators which again increases the test time to be spent in testing.

In order to explore design and test tradeoffs possibilities, the logic modules (ALUs) characteristics are exploited during allocation, based on test metrics described in [ChPa91a]. The Specific objectives are: (a) *reduce BIST registers hardware overhead* while satisfying fault coverage requirements; (b) *trade-off BIST cost with components cost* (ALUs, MUX) during high level synthesis; (c) *provide testability estimation*, such as fault coverage and number of test patterns, during the system level design process while controlling the circuit sequential depth (Sections 4.4 and 4.5 for details).

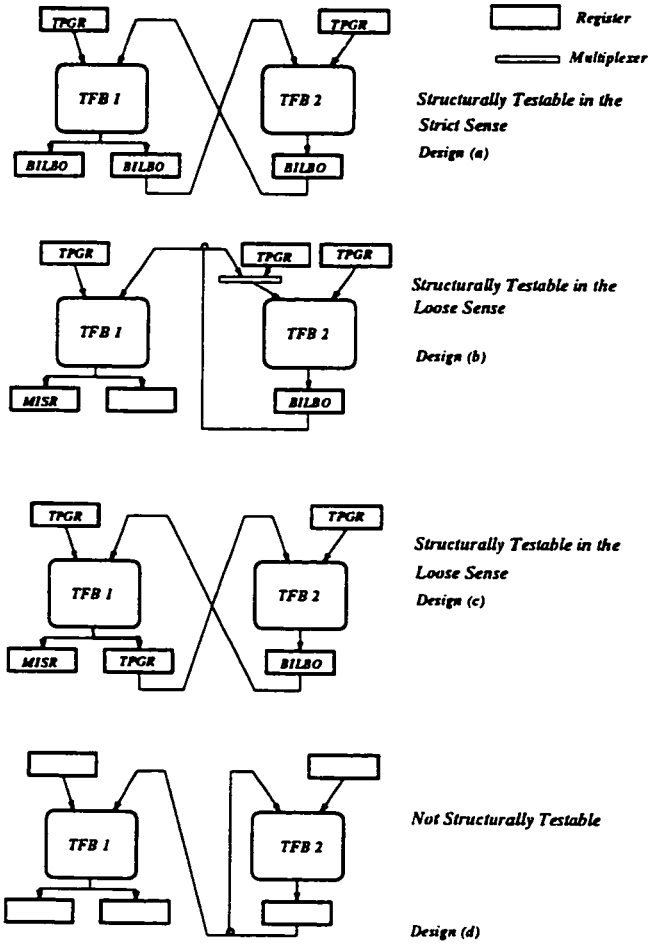


Figure 4.2: Illustrating the structural testability notion with different designs

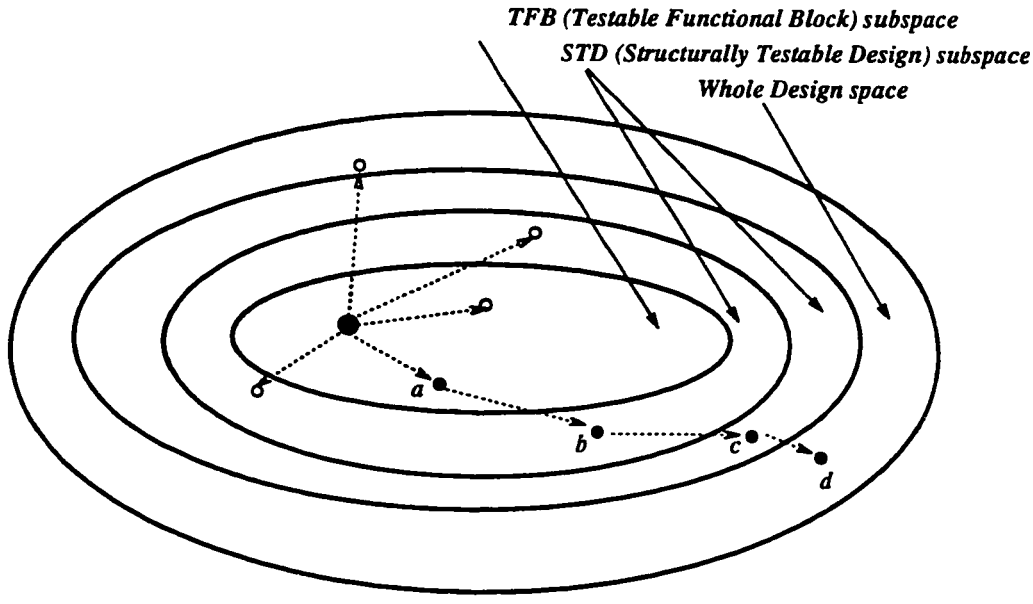


Figure 4.3: Design and Test Spaces

4.2 Testable Data Path Allocation

Our testable data path allocation starts with a scheduled data flow description of a circuit and uses an *iterative improvement optimization* method. The scheduling can be accomplished using any method reported in the literature [PaKn89, PaPM86]; however, we use our own scheduling method reported in [NoPa92]. The allocation method tends to eliminate self-adjacent registers and allocates a sufficient number of registers in order to enhance the datapath controllability and observability. Several key elements characterize our allocation technique: 1) A levelized compatibility graph; 2) a technology library of modules and components; 3) a cost function that is associated with every path in the graph. In what follows, we describe our method in reference to a simple scheduled data flow graph, the biquad filter, shown in Figure 4.5(a).

4.2.1 Requirement Analysis

Given a scheduled data flow graph (DFG), a DFG node corresponds to 1) an operator which must be assigned to a functional unit during the control step in which it is scheduled; 2) a value which must be assigned to a register for the duration of its life time[5]. Thus, overlapping life times cannot not be assigned to the same register. Finally, data transfers are assigned to some path of connections, buses and multiplexers.

Consider a DFG node associated with the variable instance V, its corresponding operation O(V), and life span L(V). Then the TFB A(V) of V is the 3-tuple:

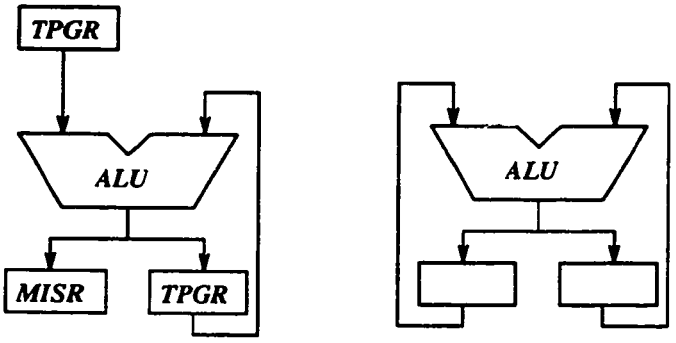
A(V) = [V; O(V); L(V)]

For the example Figure 4.5(a), the TFB of a_0 is: [a_0; *Addition*; 2 4]. It follows from this definition that each DFG node represents a TFB whose input edges are connected to the outputs of other TFBs. A more complex TFB is generated by merging two or more TFBs. The intuitive rationale is to avoid the dislocation of variable instances from their corresponding operations, keeping them close together during the entire allocation process. This proximity is preserved topologically in the generated data path and it contributes to its structural testability. The specific objective of our scheme is to allow the mapping of DFG nodes into TFBs which are the building blocks of the testable data path generated by the proposed allocation.

Two TFBs are compatible if there is no resource conflict between the *operations* of the DFG nodes; that is, the nodes are assigned to different time steps in the schedule. However, in addition to the above restriction, there are additional rules that should be satisfied for the successful merging of two TFBs:

- *Rule 1:* Do not merge the TFBs if such merging will result in a module which does not exist in the technology library. For example, if the resulting ALU

[5]The register life span or life time is the union of the intervals when its variables are first introduced and last used by other variables



(a) *(b)*

Figure 4.4: (a) Self Testable ALU with Self-Adjacent register, (b) Non-Testable ALU with Self-Adjacent registers

has an operation set such as {+,*,/} which is not in the library, such merging can not take place from the technology library standpoint.

- *Rule 2:* Allow self-adjacent registers *only* if the resulting structure is testable. Thus, we would not generate designs such as the ones depicted in Figure 4.1(c), while designs as in Figure 4.4(a) are testable and thus acceptable.

Rule 2 in our model is responsible for the structural testability of our RTL datapaths designs. We *emphasize* that designs similar to the ones depicted in Figure 4.4(b) can not be generated by our synthesis method unless there are *enough* additional registers controlling the input ports through multiplexers.

4.2.2 System Library

Minimizing the number of components (functional units, registers, ...) is not sufficient to guarantee a good design since some components may be more expensive than others under a given technology. We use a *technology library* as an input to our allocation method. This provides a good estimate of the data path cost during optimization and the computation of the cost formula.

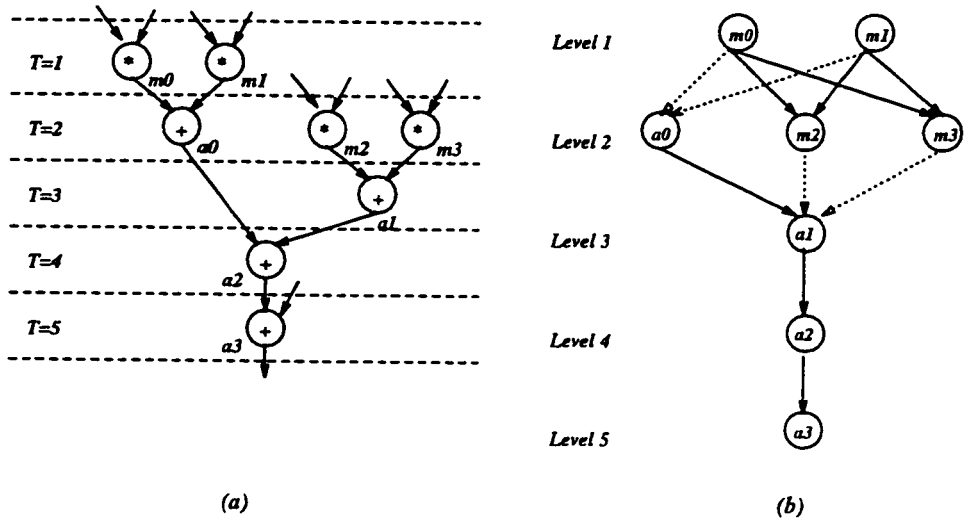


Figure 4.5: (a) Biquad filter DFG, (b) Biquad Module Allocation graph.

The technology library provides information about the layout area of every individual component (width, height and break points) along with the delay propagation. The library components are parametrized by their bit length and some may come in different implementation styles, for example, a slow but area efficient multiplier versus a fast (combinatorial) but area expensive multiplier. Furthermore, each component is associated with a set of test metrics which we obtained by fault simulation. These include the fault coverage of individual ALUs along with a set of *new* test metrics (*randomness* and *transparency* metrics) introduced in [ChPa91a].

4.2.3 Module Allocation Graph

In order to illustrate the compatibility relations among the DFG nodes, we use a special graph which we call *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 from node A to node B in the MAG indicates that both nodes are compatible; however, node A is scheduled before node B. A

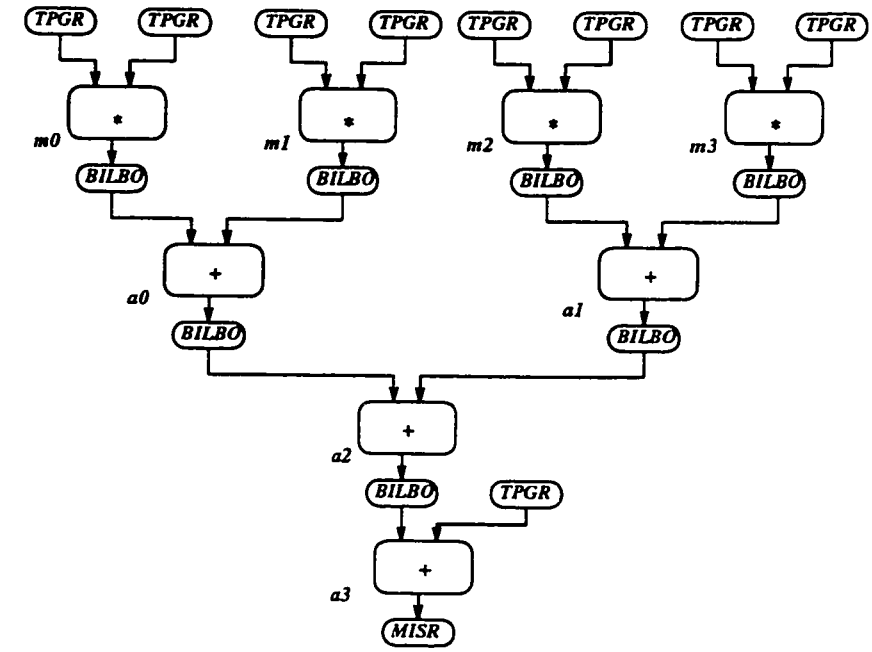


Figure 4.6: Initial data path (IDP) for the biquad example

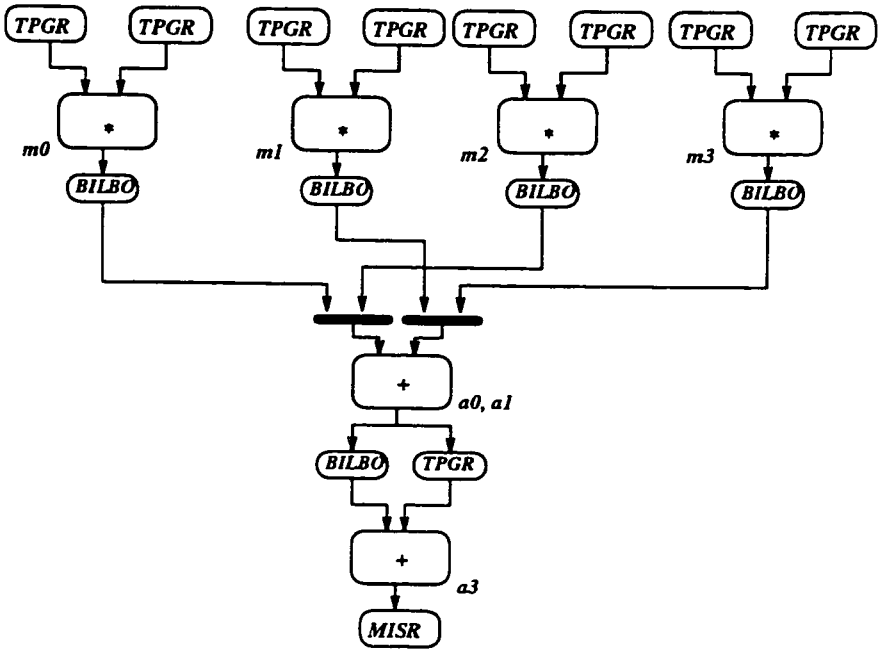


Figure 4.7: Intermediate solution after merging TFB *a0* and TFB *a1*

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.

The motivation for the MAG is twofold. First, by having directed edges, a top-bottom optimization process is possible by considering two levels at a time, pruning the number of paths in the MAG; thus, reducing the number of merging possibilities. Second, we associate a *local cost (gain) function* with every path in the MAG. The cost function will guide the algorithm in order to select the best merging possibility (path) at every iteration. The MAG is technology dependent and driven by the *system library*. Thus, for different libraries, we have different MAGs, and consequently different bindings due to *Rule 1*, discussed above.

The construction of the MAG is quite simple: we assign each node in the MAG to the level at which it was scheduled. We add directed edges between the compatible nodes using a top-bottom fashion by considering two levels at a time, k and $k+1$. The reason, as stated above, is to prune the number of paths in the MAG and thus reduce the number of edges in the graph. For example, in Figure 4.5(b), path *m0a1* (corresponding to merging operations $m0$ and $a1$) is redundant since it is covered by path *m0a0a1*. Edges are added to the MAG as long as they do not cause resource conflicts. This basically means that the MAG contains the schedule's edges (less the redundant ones) in addition to any edges that indicate no resource conflicts. The *module allocation graph* for the DFG in Figure 4.5(a) is shown in Figure 4.5(b) where dotted edges correspond to incompatibilities due to library considerations.

4.2.4 Resources Allocation with Testability Consideration

The ultimate goal of the allocation phase is to optimize the cost of the datapath, which is a weighted sum of the number of functional units (ALUs), registers and interconnection elements. We accomplish that by merging operations, variables

and interconnects, *simultaneously*. The reason for this is that the merging order may adversely affect the cost. For example, doing register merging first may increase the overhead of operation merging, and vice versa.

Our allocation maps DFG nodes into individual TFBs. Thus, we construct an initial datapath structure (IDP) which corresponds to an initial design point. The initial data path for the biquad example of Figure 4.5 (a) is shown in Figure 4.6. Based on an iterative improvement method, and guided by local cost functions, we improve the initial design cost through incremental merging of TFBs resulting in fewer but possibly more complex TFBs. In order to achieve a good design quality in a relatively short time, we introduce local cost functions which we associate with every edge or path in the module allocation graph. We define the local *cost gain* of the datapath resulting from merging TFBs A and B into a third TFB C as:

$$g_{local}(C) = g_{ALU} + g_{Mux} + g_{Reg} + g_{Controller}$$

where $g_{ALU}, g_{Mux}, g_{Reg}$, and $g_{Controller}$ are described in detail shortly.
We define next the local cost function f associated with a directed edge from A to B in the module allocation graph as:

$$f = \sum g_{local}(x) + \sum g_{local}(y) - g_{local}$$

where x and y are all the nodes connected to A and B, respectively, and g_{local} is the local cost function associated with merging A and B.

The merging or optimization algorithm is illustrated in Figure 4.8. Briefly, we start with the nodes in the first two levels of the levelized compatibility graph (MAG) and find the level with the least outgoing edges. Obviously, on this level, the nodes with the least number of outgoing edges are more restricted than the others. Among these restricted nodes, we choose the nodes with a minimum cost f. The local gain functions g_{local} can be used to break any ties. We thus create at every merging step a more complex TFB. After exhausting all the nodes at the level into consideration, we proceed to the following one and update the local

Input: Module Allocation Graph (MAG) and an Initial Data Path (IDP).
Output: Optimized Self-Testable RTL Structure.

```
i = 1
while (i < number of levels in the MAG) do {
Repeat {
 1. Compute the cost functions for all edges between levels i
    and i+1.
 2. Do not take into consideration the edges between level i and
    i+1 that we cannot merge based on the library restriction
    (Rule 1).
 3. Choose the level which has the least outgoing edges.  If
    both levels have the same number of outgoing edges, choose
    either one arbitrarily.
 4. Of the chosen level in the above step, select the nodes with
    the minimum number of incoming or outgoing edges.
 5. Of the selected nodes, choose the nodes with minimum cost
    and merge the corresponding TFBs in the IDP.
 6. Update the IDP.
 7. Update the graph and recompute the cost functions.
  } Until all nodes at levels i and i+1 have been processed.

Move remaining nodes to level i+1
Increment i
}
```

Figure 4.8: Datapath Allocation Algorithm

cost functions so as to reflect the new structure in the data path. We repeat this process until all nodes in the MAG have been processed. We illustrate in Figure 4.10 the merging procedure for two TFBs and describe the process in Figure 4.9. The complexity of the allocation algorithm, described in Figure 4.8, is of the order of $O(N^3)$.

Input: TFBs A and B
Output: New TFB C

- Construct the new ALU operation set such that: $Oper(C) = Oper(A) \cup Oper(B)$.
- Construct two MUXes on top of TFB C. Assign the input registers of TFBs A and B to the new MUXes. Try to share common signals by swapping signals which belong to Commutative operation.
- Construct a list of the register set of TFBs A and B. Apply the Left Edge Algorithm and merge the registers which 1) don't have overlapped life spans and 2) do not create ''bad'' self-adjacent registers (Figure 4.1(c)). Next, maintain the connections of these registers to the other TFBs.

Figure 4.9: Merging Two TFBs

Module Allocation

Every time we merge two ALUs, *ALU1* and *ALU2*, associated with two different TFBs, we are reducing the data path cost by the cost of one ALU. Assuming that the combined operation set, $ALU1 \cup ALU2$, already exists in the library, we define the local cost gain function as:

$$g_{ALU} = Cost(ALU_1) + Cost(ALU_2) - Cost(ALU_1 \cup ALU_2).$$

where $Cost(ALU_1)$, $Cost(ALU_2)$ and $Cost(ALU_1 \cup ALU_2)$ are provided by the library.

Mux Allocation

When mapping a commutative operation to an TFB, there exist two possible configurations to assign the input ports to the TFB, left or right mux. For non-commutative operations such as subtraction, the configuration is unique. In order to obtain a good MUX cost estimation, we use *incremental operands alignment*. We explain the method next.

When considering two TFBs for merging, we assign the non-commutative operations to the multiplexers at the input ports of the resulting TFB first. The reason is that these signals can not be swapped in order to explore sharing possibilities with other signals. The remaining assignment is done so as to reduce the number of multiplexer inputs, right or left, through register alignment. If $Cost(MUX_1)$ and $Cost(MUX_2)$ are the multiplexer cost of the original TFBs and $Cost(MUX)$ is the multiplexer cost of the resulting one provided by the library, then the local cost gain function is defined as:

$$g_{MUX} = Cost(MUX_1) + Cost(MUX_2) - Cost(MUX).$$

Registers Allocation

The register cost gain function estimates the reduction in registers resulting from merging two TFBs. The idea is to simply estimate the cost of registers at the output port of the TFB. Obviously, the fewer the registers, the better the solution. We apply locally the Left Edge Algorithm (LEA) [KuPa87] incrementally at the output ports of the *resulting* TFB. The restriction is that we do not merge registers that may create self-adjacent registers, similar to the ones depicted in Figure 4.1(c) (Rule 2). Thus, we guarantee the datapath self-testability. The LEA runing time is extremely fast since we are looking at a small list of registers corresponding to only two TFBs at a time. The register cost function chooses an observable register (MISR) at the output of the resulting TFB.

Let n be the number of registers at the output port of the TFB (Figure 4.1(a)), and let l be the number of TPGRs. Then, we estimate the cost of registers of a given TFB as:

$$Cost_{reg} = Observable\_Cost + l * TPGR\_Cost + (n - l - 1) * Cost\_Normal$$

where $Observable\_Cost$ is the cost of an MISR or a BILBO depending on whether it needs to control another port in the datapath. Note that all cost factors are provided by the library.

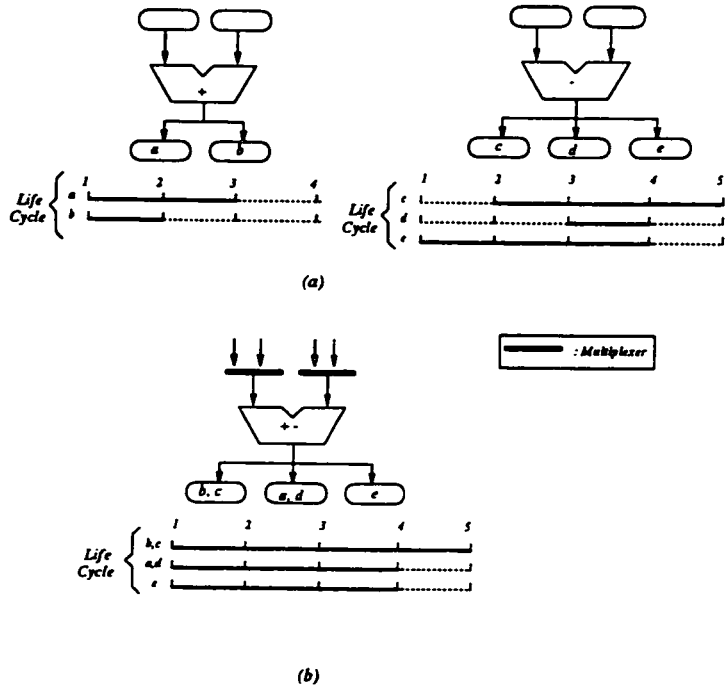


Figure 4.10: (a) Initial TFBs, (b) Resulting merged one

Merging two TFB sets implies merging their output registers as well. We apply the *left edge algorithm* to the register sets of both TFBs.

If $Cost_{reg1}$ and $Cost_{reg2}$ are the register cost of the two initial TFBs and $Cost_{reg}$ is the register cost of the resulting one, then the local register cost function is defined as:

$$g_{reg} = Cost_{reg1} + Cost_{reg2} - Cost_{reg}$$

Controller Cost Consideration

Every intermediate data path has less registers but more multiplexers and possibly more complex ALUs. This provides a good tradeoff measure among various merging possibilities, with respect to the controller cost.

If we assume a PLA model implementation, the controller area is computed as: $A_{PLA} = W_{PLA} * H_{PLA}$. Based on the area estimation model discussed in [Gajs92],

the width could be estimated as the sum of the width of the input AND array, the width of product-term buffers, and the width of the OR array. The height of the PLA is computed as the sum of a latch height, a buffer height, and the height of the of the AND-OR plane. Thus, the PLA area reduces to:

$$A_{PLA} = ((n+m) * MAX(l_w, b_w) + W_p) * (l_h + b_h + r * p)$$

where n and m are the number of PLA inputs and outputs respectively; b_h, b_w are the height and width of a buffer; l_h, l_w are the the height and width of a latch; r is the transistor pitch while p is the number of distinct product terms.

If PLA is the resulting PLA after merging TFB1 and TFB2, and $PLA1$ and $PLA2$, are the PLAs due to TFB1 and TFB2 before the merging process, then the gain can be described as:

$$g_{Controller} = Cost_{PLA1} + Cost_{PLA2} - Cost_{PLA}.$$

4.3 Synthesis Aspects

4.3.1 Pipelining

Pipelining is a powerful methodology for designing fast digital circuits. The job is split into sub-tasks allocated to separte hardware units. Two simple types of pipelining have been attempted in high-level synthesis, *functional and structural pipelining*. Currently, we support the later type of pipelining in our method. Thus, we divide an operation into s stages or s sub-operations. The stages are pipelined using functional sub-units as soon as these units are available instead of waiting for the whole operation to be completed as in the non-pipelined case. Thus, if we are using pipelined functional units, operations are merged if the number of stages corresponding to them can be pipelined.

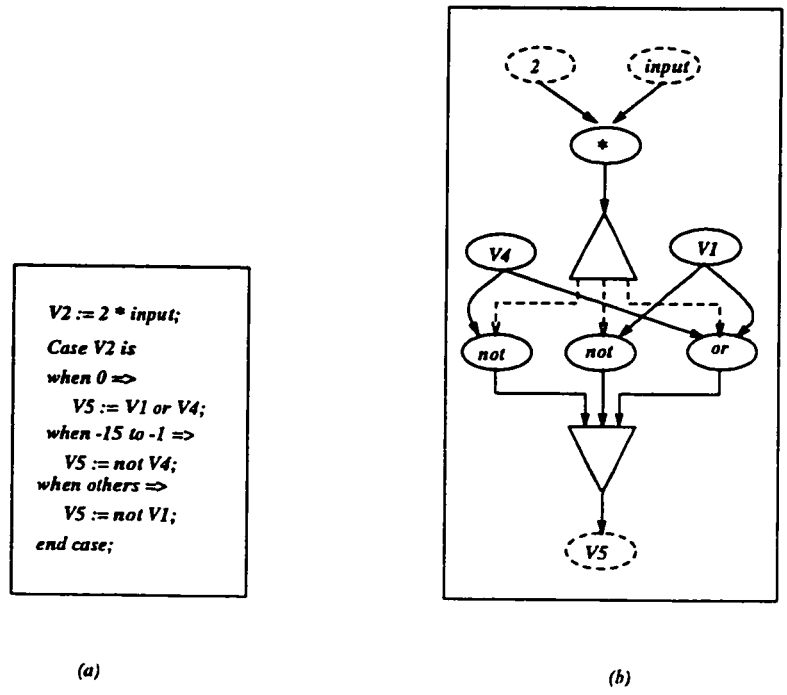


Figure 4.11: (a) VHDL case statement; (b) CDFG reprsentation (shaded lines are control lines)

4.3.2 Conditionals

In order to be able to handle conditionals, we use additional information in the DFG which retains to the control part. There are two kind of conditional models reported in the literature, *the data select model* and *the control select model* [HaPa83]. Both models are represented using a hybrid control and data flow representation that embeds all control and sequencing information explicitly within the data flow graph [Gajs92]. We implemented both conditional models in our method. We describe next how both approaches are handled during the allocation scheme.

Data Select Model

The data select model is based on a concurrent representation of the data flow graph where all mutual exclusive paths are executed concurrently and the final result is selected based on the value of the conditional. The merging algorithm and consequently the module allocation graph are still valid at this level since this control model corresponds to a pure data flow model. The allocator will have to route all necessary signals, including some redundant ones, to their corresponding ALUs. The disadvantage of this model is that it results in more components since operations and variables which are on a mutual exclusive paths cannot share information.

Control Select Model

The control select model is another scheme in which only active conditional paths are executed, rather than all branches of a conditional test. The representation uses diverge and merge nodes [OrGa86] to determine which path of the data flow graph to execute. This basically implies that operations or data stores in mutual exclusive branches can share resources eventhough they are scheduled at the same time step. This violates our merging scheme and we explain next how this is resolved in our method.

The initial levels in the module allocation graph correspond to the initial time steps in the schedule. This means that nodes scheduled in the same time step will never be on the same path in the module allocation graph. However, in the case of conditionals we resolve this problem by adding the additional compatibility rule:

> *Rule 3:* Nodes on mutual exclusive paths are compatible if such merging will result in modules in the library.

Thus, during the module allocation graph construction, mutual exclusive nodes on the same scheduling level are checked for compatibility using the above rule.

If two nodes are compatible, then either one is postponed to the next level in the MAG. We note here that all mutual exclusive paths in the schedule are *colored* during the scheduling phase.

4.4 Test Points Selection

Our allocation method guarantees self-testable RTL datapaths based on the TFB model: the controllable and the observable registers are "close" to their individual modules. Thus, *the sequential depth* for any module in the datapath is always *one* due to the unique construction style. The resulting datapath is characterized by a high fault coverage rate, with a relatively *short test pattern generation time*, but at the expense of test hardware (in terms of test registers). To further improve the design cost by reducing the test overhead, we perform test point (registers) selection while keeping the fault coverage at an acceptable level. The selection is done in two phases. In the initial phase, controllable points are removed while keeping the sequential depth to be equal to one. In a later phase, the system provides the ability to interactively remove/inject test registers to explore the design cost, test overhead and fault coverage (section 4.5). As a result of both reduction phases, the datapath area decreases significantly. It should be noted here that even without test point reduction, our overhead is still much better than the reported BIST techniques with high-level synthesis [Avra91]. The test overhead here refers to the additional area required to convert *normal* registers to *test* registers.

The initial test point selection eliminates some controllable registers (TPGRs) which are not necessary to test the datapath ALUs. This may reduce the ability to test some registers and multiplexers in the datapath, resulting in what we defined in section 4.1.3 as *loosely testable designs*. However, the *sequential depth* remains one. Basically, the procedure that we follow is to cover every ALU input port

in the data path with at least one controllable point while minimizing the total number of registers. The output ports are already covered by one observable point (MISR) by construction and therefore, there is no need to apply this procedure.

Assume there are n ALUs in the data path with m distinct input ports. Let l be the number of registers connected to the m ALU ports. Assuming that every ALU has two ports ($Left$ and $Right$), then $m = 2n$. We introduce an additional constraint (equation 4.3), since it is not desirable to cover both ports of an ALU with the same TPGR for obvious fault coverage reasons. We can formulate the problem formally as:

Constraints:

$$\sum_{i=1}^{l} Left_{i,j} * X_i \geq 1 \qquad j = 1, 2, ...m \tag{4.1}$$

$$\sum_{i=1}^{l} Right_{i,j} * X_i \geq 1 \qquad j = 1, 2, ...m \tag{4.2}$$

$$\sum_{i=1}^{l} (Right_{i,j} * Left_{i,j}) * X_i = 0 \qquad j = 1, 2, ...m \tag{4.3}$$

$$X_i = 0 \ or \ 1 \tag{4.4}$$

Where

$$Left_{i,j} = \begin{cases} 1 & \text{if } Reg\ i\ covers\ left\ port\ j \\ 0 & \text{otherwise} \end{cases}$$

and

$$Right_{i,j} = \begin{cases} 1 & \text{if } Reg\ i\ covers\ right\ port\ j \\ 0 & \text{otherwise} \end{cases}$$

Cost function to minimize

$$C = \sum_{j=1}^{l} X_j \tag{4.5}$$

We solve the above problem using a heuristic covering algorithm [Chri75]. However, we note that an *integer linear programming solution* would be efficient as well and very fast due to the small size of the problem. For our runing example of Figure 4.5(a), the initial test point selection phase reduced the number of controllable registers by 6. That is a reduction of 61.53 % of the original number of controllable registers.

4.5 Test Tradeoffs Illustration Using an Example

As stated earlier, one of the features of our *testable allocation* is its ability to tradeoff design and test quality. In what follows, we describe our methodology.

The test point selection described above reduces the number of test registers in the data path. However, we can further reduce the number of test registers using a set of test metrics introduced by [ChPa91a]. The idea is to remove an observable register at the output port of a TFB if the faults can be sensitized through intermediate modules [ChPa91a]. Of course, this will increase the sequential depth in the circuit. In order to illustrate our tradeoff scheme, we use the DFG of the TRIGO example, shown in Figure 4.17. The TRIGO DFG computes the Taylor series for the trigonometric functions $sin(t/T)$ and $cos(t/T)$ concurrently. All the operations for this example exist in our library ({+},{*},{/}, and also some of their combinations. The test hardware overhead was significantly reduced (by 17.96 %) after applying our test point selections. We discuss next some important aspects of our tradeoff model concerning tradeoffs coverage.

We have experimented with the fault coverage of several design styles for the trigo example versus the number of test patterns as shown in Figure 4.18. We notice that there is a measurable difference in fault coverage quality between the *BILBO* and the *test point selection option*, in favor of the *BILBO*. This difference is of about 2% but it should be measured against the area increase required by the *BILBO* designs when making tradeoff decisions.

We also note another interesting tradeoff measure that exists between *test area overhead* and *test time* or *fault coverage*. To be specific, our tradeoff scheme has the capability to "inject" observable points (MISRs) during the design process for the benefit of increasing the fault coverage (or reducing the number of test patterns). Thus, we can generate designs between the *BILBO* and the *test point selection* options by performing tradeoffs between test area overhead and fault coverage. This tradeoff situation is illustrated in Figure 4.18 for the trigo circuit. By injecting an observable register we can "break" a chain of multiplier-adder in the circuit. This chain is used to bypass test patterns from a controllable point (multiplier input) to an observable point (adder output). Breaking such chains improves the fault coverage of the circuit at the expense of increasing the circuit overhead; however, this increase is not very large. These tradeoffs are depicted in Figure 4.18 by shifting the test pattern/fault coverage curve of *the test points selection option* to the "right" depending on the test register injections we cause. We actually illustrate in Figure 4.18 three fault coverage/test pattern curves corresponding to one, two and three test register injections. From the fault coverage plots of Figure 4.18, we can derive tradeoff relationships between test hardware and fault coverage/test time. This relationship is shown in Figure 4.19 which depicts the normalized test hardware overhead versus the fault coverage assuming constant test patterns for each curve. The data points in each curve were derived by injecting a test register from one data point to the next one.

Coming back to the running example of the previous section, the final solution

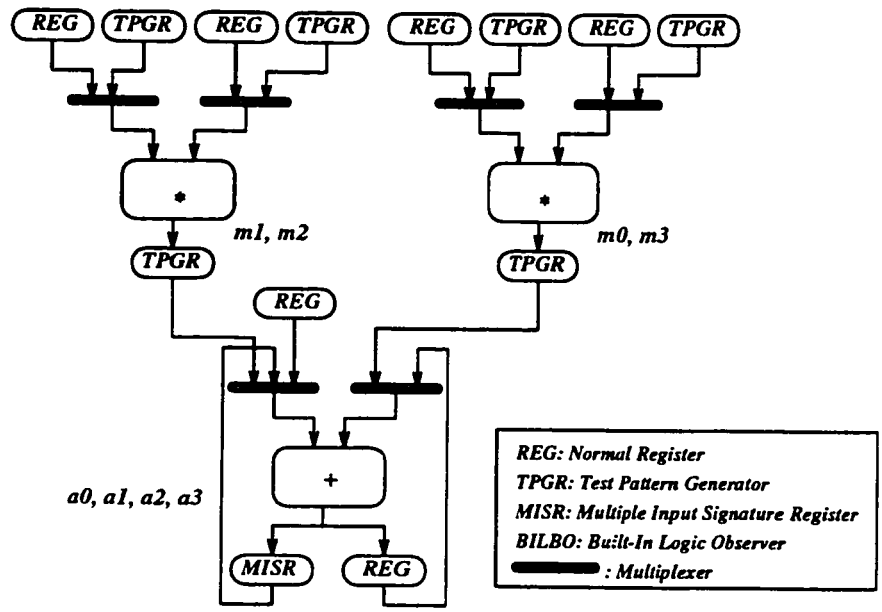


Figure 4.12: Biquad final solution

for Figure 4.5(a) is shown in Figure 4.12. We have the following comments:

- The tradeoff scheme reduced the number of observable points by 3, a reduction of 75 % of the original number of observable points.
- The designs generated by our method provide an interesting feature. The first design, based on no test point selection, has the highest fault coverage at the expense of larger area. The second and third designs, which we obtained after applying our test point selection and tradeoff schemes, present a good tradeoff or balance between area and test quality (fault coverage).

4.6 Results

We implemented the described allocation and tradeoff scheme on a Sun SPARC station IPC. The method is very fast and all reported results are produced in at most 1.13 CPU minutes (wave filter) including scheduling time. In what follows, we describe some of the results that we have attempted.

We validate our methods using two sets of data. In the first set we use two examples, a second order differential equation from HAL [PaKn89] and an example from FACET [TsSi86]. Results are shown in Table 4.1 and Table 4.2, respectively. We compare with [Avra91, Pang88, TsSi86, PaKn89] based on RALLOC [Avra91] cost model. We note that the designs generated by [Pang88, PaKn89, TsSi86] were modified by [Avra91] using CBILBOs in order to make them testable. As the results clearly show, the designs generated by our method, while fully testable, have better quality and also have a lower test overhead. This is a good indication that when testability is considered up front, designs are of a lower cost. The data path cost in this case was defined [Avra91] as:

$Cost = 20{*}BILBOs{+}35{*}CBILBOS{+}MuxIn{+}ControlSignals{+}Interconnect.$

We account for two more factors in the cost formula, the cost of TPGR's and MISR's defined respectively as: $14 * TPGR + 16 * MISR$. These costs factors are dependent upon the technology and implementation used.

The second set of data consists of two relatively large benchmark examples, *the fifth order elliptical wave filter example* and the *AR filter*. For the wave filter , first popularized by [PaKn89], we derive four testable designs based on four different schedules in 17, 19, 21 and 28 time steps. We assume multicyle operations in case of the multiplication while we assume additions take just one cycle. We show the detailed results for this example in terms of components and number and types of test points in Table 4.4. Table 4.3 shows the overhead using BILBO designs, and the overhead after applying our tradeoff scheme. The overhead was computed with respect to the overall area of the datapath which includes the area of ALUs, registers (normal and test registers) and multiplexers. In the case of 17 time steps, we notice a substantial overhead reduction, from to 23.22 % to 2.18 %. As illustrated in Figure 4.14, the datapath area decreases substantially after applying both tradeoff phases iteration due to the reduction in test overhead. The sequential depth was one in the case of BILBO option as well after the initial

System	ALUs	Test Registers Types				Mux In	Inter-Connect	Cntl Sig.	Cost
		BILBO	CBILBO	TPGR	MISR				
Splicer	(+)(-)(>) (*)(*)	1	5	0	0	17	34	23	269
HAL	(+)(-)(>) (*)(*)	1	4	0	0	19	35	24	238
RALLOC	(+)(-)(>) (*)(*)	4	1	0	0	22	38	27	202
Ours	(+*)(*->) (*+)	0	0	4	1	15	28	21	136

Table 4.1: Designs Comparisons for the HAL Differential Equation example

System	ALUs	Test Registers Types				Mux In	Inter-Connect	Cntl Sig.	Cost
		BILBO	CBILBO	TPGR	MISR				
Facet	(/)(-&+)(*|+)	3	5	0	0	15	31	23	304
Splicer	(/)(-&+)(*|+)	4	3	0	0	11	26	18	240
HAL	(/)(-&+) (*|+)	4	1	0	0	13	23	17	168
RALLOC	(+/|)(-&+) (*+)	4	1	0	0	16	29	21	181
Ours	(+/)(-&+) (*|+)	0	0	3	1	10	21	16	105

Table 4.2: Designs Comparison for the FACET Example

selection, while it increased to two after the final selection. We show the fault coverage versus the number of random test patterns in Figure 4.13, which indicates that we attain a high fault coverage while reducing the number of test registers.

The AR Filter was used initially by [Jain89] and used later for comparison purposes. We derived two schedules using time steps 8 and 10 assuming that all operations execute in one cycle. We show detailed results for this example in Table 4.5. Table 4.3 illustrates the overhead using BILBO designs, then the overhead after applying our tradeoff. Figure 4.16 shows the decrease in datapath area which was attained after our initial and final test point selection. We note that the sequential depth was one in case of the BILBO option and initial selection, but increased to two after the final selection. The fault coverage for this example versus the number of random patterns is shown in Figure 4.15.

Design example	Initial Overhead	Final Overhead
AR Filter, T = 8	20.21 %	5.63 %
AR Filter, T = 10	25.23 %	6.08 %
Wave Filter, T = 17	22.13 %	3.94 %
Wave Filter, T = 19	23.22 %	2.18 %
Wave Filter, T = 21	27.88 %	6.74 %
Wave Filter, T = 28	44.90 %	9.26 %

Table 4.3: Test overhead before and after tradeoffs

C_Step	Mode	ALUs	Test Registers Types			# Reg	Mux In
			BILBO	TPGR	MISR		
17	Normal	3 (*), 3 (+)	0	0	0	15	32
	BILBO	3 (*), 3 (+)	6	3	0	15	32
	Initial Selection	3 (*), 3 (+)	2	3	4	15	32
	Final Selection	3 (*), 3 (+)	2	3	2	15	32
19	Normal	2 (*), 2 (+)	0	0	0	14	29
	BILBO	2 (*), 2 (+)	4	5	0	14	29
	Initial Selection	2 (*), 2 (+)	1	2	3	14	29
	Final Selection	2 (*), 2 (+)	1	2	1	14	29
21	Normal	(*), 2 (+)	0	0	0	11	31
	BILBO	(*), 2 (+)	3	3	0	11	31
	Initial Selection	(*), 2 (+)	1	2	4	11	31
	Final Selection	(*), 2 (+)	1	2	2	11	31
28	Normal	(*), (+)	0	0	0	17	32
	BILBO	(*), (+)	5	2	0	17	32
	Initial Selection	(*), (+)	0	2	2	17	32
	Final Selection	(*), (+)	0	2	2	17	32

Table 4.4: Test results summary from the wave filter

C_Step	Mode	ALUs	Test Registers Types			# Reg	Mux In
			BILBO	TPGR	MISR		
8	Normal	3 (*), 3 (+)	0	0	0	16	31
	BILBO	3 (*), 3 (+)	6	8	0	16	31
	Initial Selection	3 (*), 3 (+)	2	2	4	16	31
	Final Selection	3 (*), 3 (+)	0	4	4	16	31
10	Normal	2 (*), 2 (+)	0	0	0	14	28
	BILBO	2 (*), 2 (+)	4	5	0	14	28
	Initial Selection	2 (*), 2 (+)	2	1	2	14	28
	Final Selection	2 (*), 2 (+)	1	2	2	14	28

Table 4.5: Test results summary from the AR Filter filter

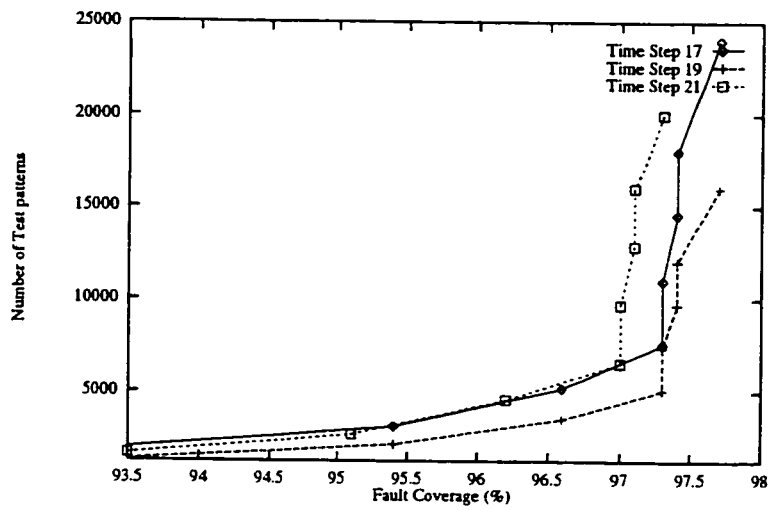


Figure 4.13: Fault coverage for the wave filter example

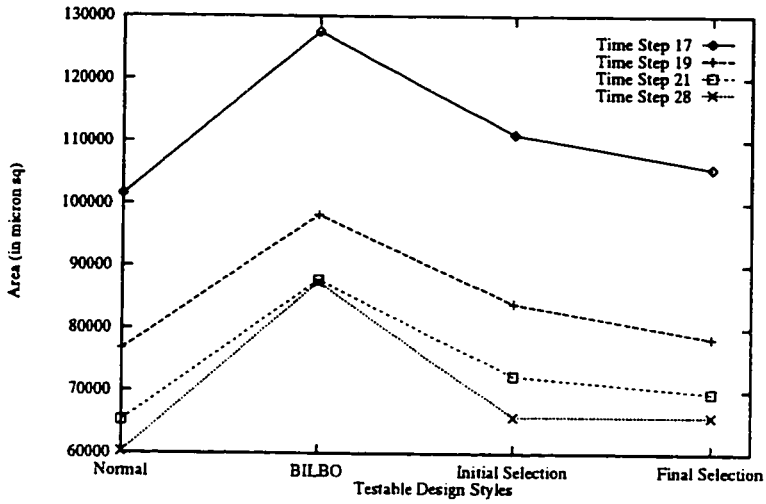


Figure 4.14: Area decrease in the wave filter after test points selection and trade-offs

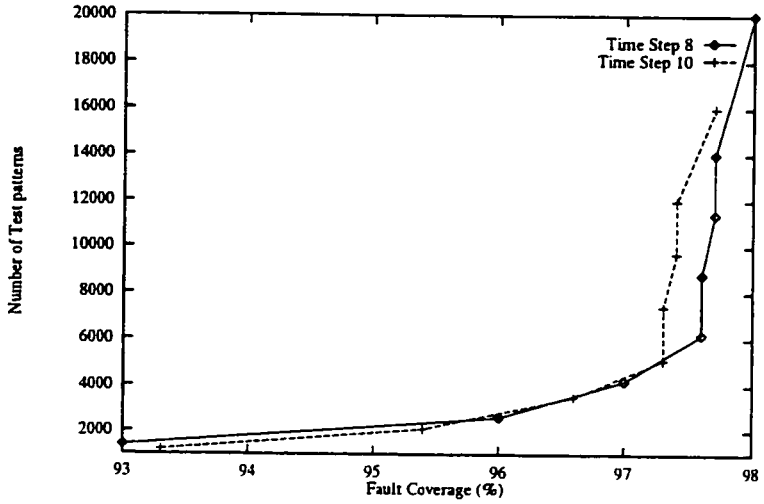


Figure 4.15: Fault coverage for the AR filter example

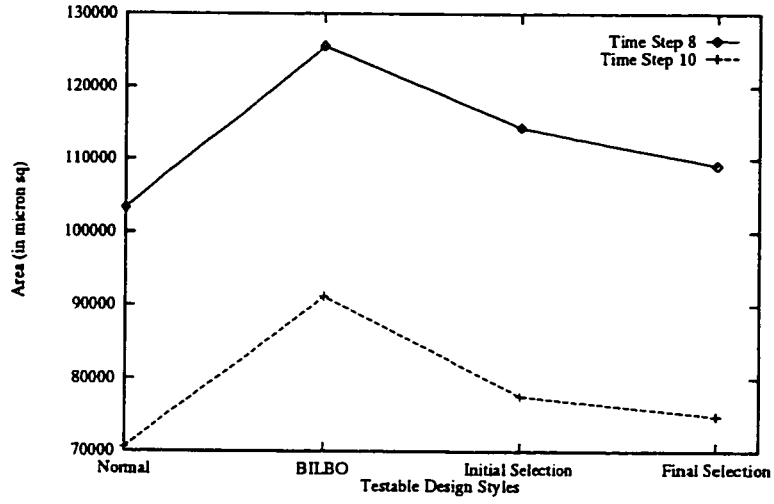


Figure 4.16: Area decrease in the AR filter after test points selection and trade-offs

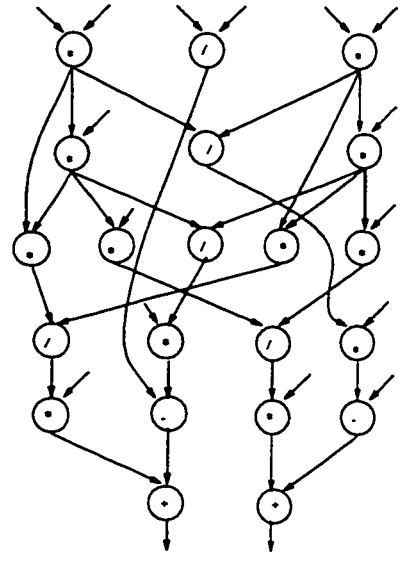


Figure 4.17: Scheduled DFG for the trigo example

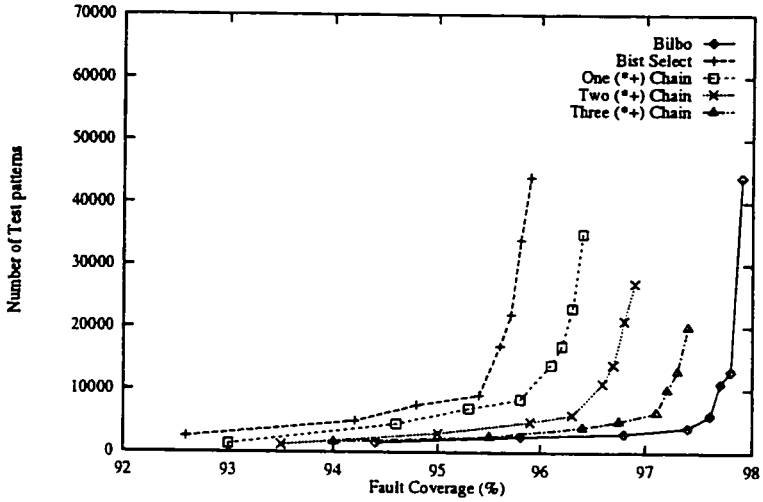


Figure 4.18: Fault coverage for the trigo example after injecting one, two, and three additional test points

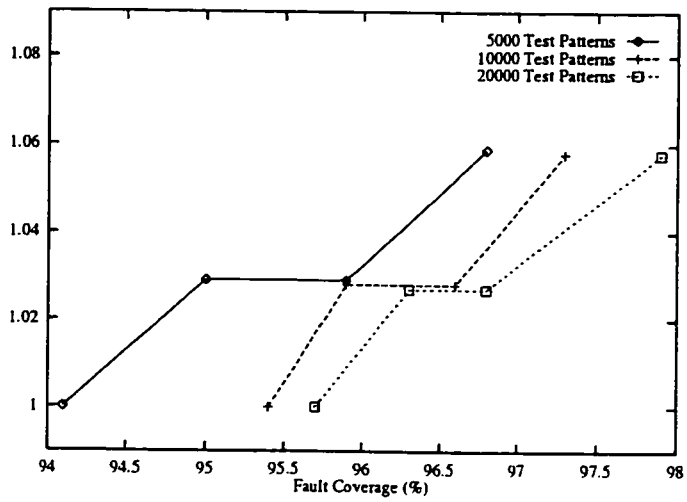


Figure 4.19: Relative relationship between test overhead and test time for the trigo example

Chapter 5

Datapath Reallocation

The previous chapters discussed the high-level synthesis task in general and the allocation task in coordination with testing in particular. As mentioned, the allocation task is fairly complicated and solutions are derived under a set of constraints. The constraints may lead to a good solution within the restricted space; however, better solutions are possible by conducting further redesign improvement and space exploration.

In this chapter, we introduce a methodology for redesign in high-level synthesis at the allocation level. The redesign process attempts to improve the design cost without changing the schedule timing while taking into consideration area, delay and floorplanning.

5.1 Introduction

5.1.1 Problem Definition and Motivation

Given a register-transfer level (RTL) description of a data path, the purpose of the redesign task is to improve the data path cost while taking into consideration various design issues including area, performance and floorplanning. The redesign task, a post-synthesis process, iteratively modifies the design while improving its cost. The motivation for the redesign approach is twofold:

1. *Design improvement:* The non-optimal nature of the synthesis process leaves a lot of room for improvement. By considering placement and routing, in addition to component cost, the design modifications becomes more effective and more realistic.

2. *Datapath reusability:* It may be desireable to reuse the old data path in order to generate an alternate structure under a different technology. Thus, the designer may want to reuse the old structure to generate a new one, optimized under a different cost function. For example, the original data path could be optimized under area while the new one will be optimized under delay.

In either case, the reallocation phase will require a *transformational rip-and-rebind* approach based on an iterative improvement method which is an efficient way to solve computationally complex problems. The method starts with a design relatively good, derived by a synthesis method, and applies a set of transformations so that to improve the design cost. The process will stop once no further improvements are possible. The cornerstone of our redesign method are two elements:

1. A set of transformations that are used as a *redesign* tool.
2. A *layout* and *delay* models that can act as a driving engine for the redesign process.

5.1.2 Related Research

Redesign in high level synthesis was discussed in Fasolt [Knap89] where a manual approach was used to modify the schedule and to modify the result of the allocation by manually inserting and/or removing registers and other components. The system was able to go back and repair the design once the user is done with the modifications. A later version of Fasolt [Knap91] used a layout model to automatically drive the choice of design transformations. In STAR [TsHs92], the allocation method is split in two phases. In the first phase, an initial datapath is generated. The datapath binding quality is evaluated in the second phase and modifications are made to improve the design quality. Recently, [KrNe92] proposed an allocation method similar to STAR. The method is iterative but it uses a random generator to choose the transformations to be applied. In conclusion,

none of the above methods apply the transformations in a systematic or guided fashion.

5.2 Reallocation Approach

As mentioned previously, our approach is based on a set of transformations and strategies. In what follows, we discuss the design transformations in general and present briefly the transformations that we will adopt during the datapath reallocation.

5.2.1 Background

The design of digital circuits is usually described at various levels of abstractions. The three main levels of the design hierarchy are the *behavioral*, the *structural* and the *layout* level. The design behavior can be described by a design language such as VHDL or internally represented by a data flow graph description. RTL design structures are described using a netlist of RTL components such as registers, functional units and multiplexers. Finally, the layout of a design is described by means of the geometries of the individual primitive cells. Other design levels such as the gate level are of course widely used in the design hierarchy.

Figure 5.1 shows the three design spaces corresponding to the behavioral, RTL and layout levels and the mappings from one space to another by means of scheduling, allocation and layout algorithms applied at different design levels. Informally speaking, a *transformation* consists of a set of rules that, subject to certain preconditions, modify a given design A into another design A', in the same design space as A, with A' being functionally equivalent to A.

In general, we can distinguish among the following transformation classes:

- *Behavioral transformations* which produce functionally equivalent but behaviorally different data flow representations. For example, applying the distributive property on the expression $E = a * (b + c)$ changes the data flow

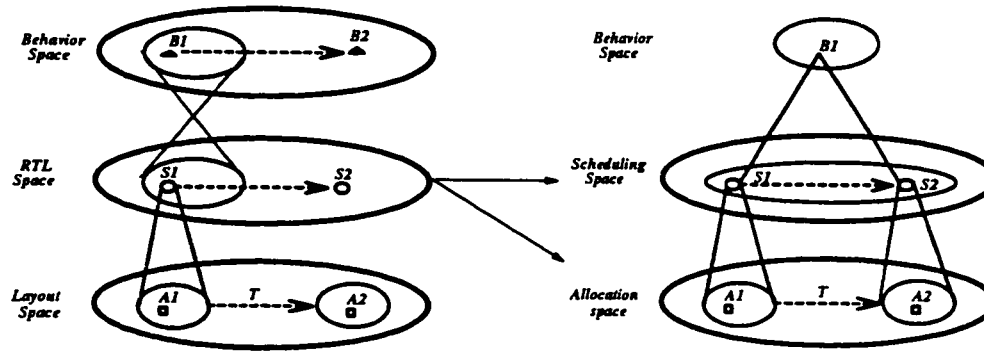


Figure 5.1: Design and Test Spaces

behavior of E into that of $E' = ab + ac$, where E' is obviously functionally equivalent to E.

- *Structural transformations* which modify an RTL design, but do not change its behavior. For example, merging an adder and a subtractor of a datapath into an ALU capable of performing both operations is a structural transformation.

The structural transformations can be further classified into the following two categories:

- *Rescheduling transformations,* which change the timing of the datapath structure for example, inserting a register in a datapath *may,* under some conditions, prolong the schedule by one time step.
- *Reallocation transformations,* those which change the datapath structure and binding but not its schedule.

We concentrate on performing restricted exploration of the RTL space, i.e., the allocation space, by means of reallocation transformations. Thus, preserving not only the design behavior but also the design schedule. One of the advantages of the reallocation process is that the allocation space is much smaller and more manageable than the scheduling space. However, the disadvantage is that it is not as flexible.

5.2.2 Transformations

The *reallocation transformations* correspond to moves of operation and store variables of the datapath structure, while keeping the schedule fixed. The transformations are characterized by a:

1. *Transformation rule* which describes a move in the design space.
2. *Prerequisite condition* which refers to the conditions that must be satisfied before the move is applied.
3. *Transformation effect* which is the modification resulting on the datapath from the move.

A transformation is *valid* if it can be applied without violating its prerequisites. We discuss next a set of basic transformations which we will use during reallocation. Although it is not difficult to show that these transformations are correct, a formal proof of correctness will not be given here. However, it should be noted that more complex reallocation transformations can be defined as sequences of these basic transformations assuming that their prerequisites are valid. Further, we note that each of the transformations has an *inverse* transformation, for example, the inverse of *Insert* is *Eliminate*.

Operation variable move

This transformation moves operation OV_a scheduled at time step $T(OV_a)$ from ALU_i to ALU_j. We indicate this transformation by $[MOVE : OV_a \in ALU_i \rightarrow ALU_j]$. The prerequisite is that there is no resource conflict for this move, meaning that ALU_j is not performing any operation during $T(OV_a)$. The effect is redistribution of functions between ALU_i and ALU_j along with a connection effect.[1]

[1]This regards the connections between ALU/registers and MUXes in the data path.

Store variable move

This transformation moves variable SV_a whose life span is $L(SV_a)$ from register REG_i to Register REG_j. We show this transformation by $[MOVE : SV_a \in REG_i \rightarrow REG_j]$. The prerequisite is that there is no variable in REG_j whose life span overlaps $L(SV_a)$. There is again a connection effect for this move.

Operation variable swap

This transformation exchanges operation OV_a in time step $T(OV_a)$ of ALU_i by operation OV_b in time step $T(OV_b)$ of ALU_j. We show this transformation by $[SWAP : OV_a \in ALU_i \leftrightarrow OV_b \in ALU_j]$. The prerequisite is $T(OV_a) = T(OV_b)$. Again, the effect is an exchange of functions between ALU_i and ALU_j and a connection effect as well.

Store variable swap

This transformation exchanges store variable SV_a with life-span $L(SV_a)$ in REG_i by variable SV_b with life-span $L(SV_b)$ in REG_j. We indicate this transformation by $[SWAP : SV_a \in REG_i \leftrightarrow SV_b \in REG_j]$. The prerequisite is that no conflict in life spans in both destination registers. The effect is an exchange of variables between REG_i and REG_j and a connection effect.

ALU insert

This transformation inserts an ALU in the datapath. This can be used when an operation or store variable move (or swap), respectively, can not be applied. This transformation may occur when there is a time conflict on $T(OV_a)$ with some operation in ALU_j.

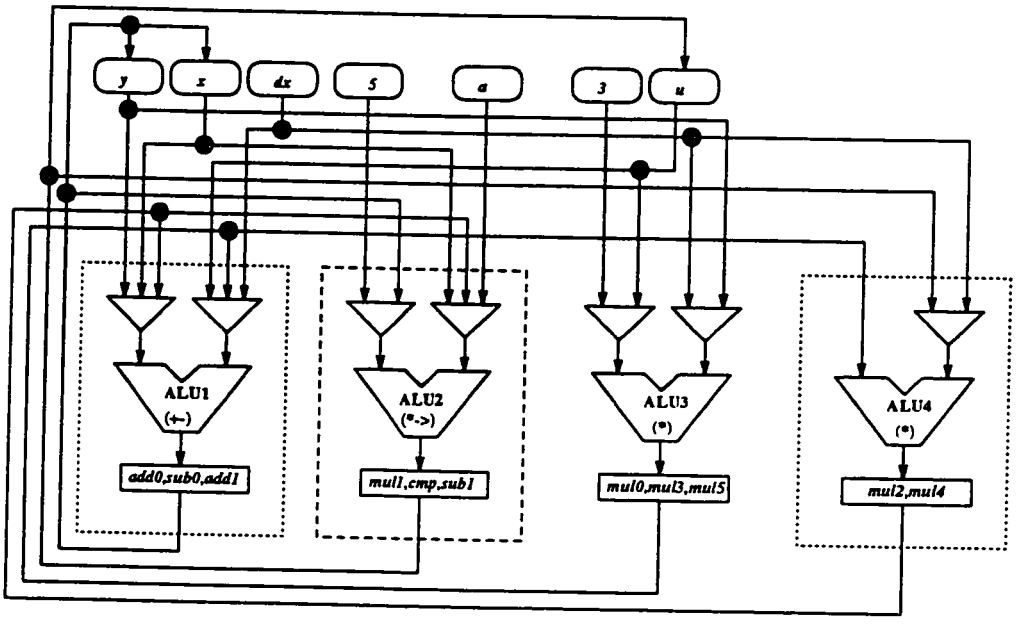


Figure 5.2: The Hal example before reallocation

Register insert

This transformation inserts a register in the datapath. This transformation may occur when $L(SV_a)$ overlaps with some variable life-spans in REG_j. This transformation makes the move out of ALU_i or REG_i valid (satisfying transformation prerequisites) although the destination is changed.

Eliminate ALU or register

This transformation reduces an ALU or register if they carry no variables as a result of prior move or swap transformations. This is basically a cleanup transformation.

T	Name	Op	Reg1	LeftIn	RightIn
1					
2	add0	+		x	dx
3	sub0	-		mul2	u
4	add1	+		y	mul5

ALU1

T	Name	Op	Reg2	LeftIn	RightIn
1	mul1	*		5	x
2					
3	cmp	>		add0	a
4	sub1	-		sub0	mul4

ALU2

Figure 5.3: A partial grid like representation for the differential equation datapath

5.3 Design Representation

5.3.1 Structural model

The allocation phase maps a DFG node V_a into two distinct variables, namely an *operation* variable OV_a and a *store* variable SV_a, where OV_a and SV_A are produced and stored, respectively, in an ALU and register pair of the datapath. We use the same structural model which we introduced in chapter 4. Thus, variables will be stored in registers at the third layer in the TFBs.

5.3.2 Grid representation

We use a two dimensional grid, similar to the one introduced by [DeNe89]; however, we add register and interconnect information. Each vertical slice corresponds to a processing unit (ALU) operation set, registers set associated with the ALU variables, and the inputs at the ALU ports. The inputs at the ALU ports are collapsed to form multiplexers. We note the following remarks:

- The schedule time is fixed during allocation. Therefore, the ALUs operations can only slide horizontally to other ALUs whenever possible. It is obvious that in such a case no improvement in datapath cost can occur unless *the datapath functionality is changed.*
- The lower bound on number and types of ALUs are determined during scheduling. The same remark applies to the number of registers. This provides with a stopping criteria in both cases.

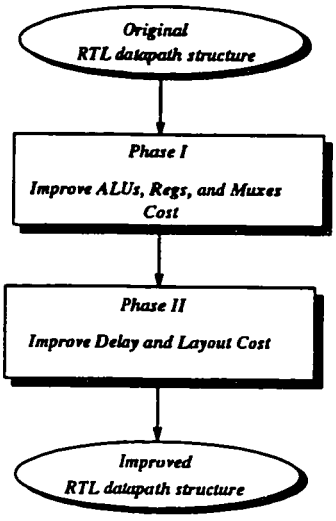


Figure 5.4: Reallocation Approach

In the next sections, we should discuss all these issues and elaborate on the strategies we used. Further, we would like to develop a guided mechanism for the selection process.

5.4 Datapath Reallocation

As noted in the previous section, the transformations are associated with two design modifications. *First*, a local redistribution of operations and/or register variables. *Second*, a limited connection change in the original datapath. We remark further that the change in component cost, including the MUX cost, due to the transformations can be easily computed from the library. The connection cost can be more accurately accounted for by invoking the layout estimator. The question that arises is how to apply the transformations in a systematic way. A straightforward method to attack this problem is to apply these transformations in a hill-climbing fashion leading to an incremental cost reduction transformation by transformation. An annealing method based on individual transformations is

another alternative. The disadvantage of these methods is that they are not well guided and thus time consuming. We notice that an improvement in datapath cost may require applying a set of transformations. Thus, in our approach we will take a more global view by selecting not one-by-one, but sequences of transformations. The evaluation of the strategies is based on well guided cost estimation choices. We discuss next for both reallocation phases along with the strategies associated with each.

The datapath is a weighted sum of various components such as ALUs, registers and multiplexers or buses. We separate our reallocation method into two phases, as shown in Figure 5.4. *The first phase* aims at exploring modifications in ALUs cost by altering their functionalities, if possible. We then explore any possible improvement in registers and multiplexers cost. *The second phase* attempts to reduce the delay and layout cost by using a layout estimator as a driving engine while keeping the datapath functionality fixed. The rationale is: 1) not to destroy whatever improvement phase I has accomplished; 2) not to reduce the number of registers or multiplexers on the account of the layout cost of the data path. There is another important motivation and that is to split the problem into smaller subproblems; thus, helping to achieve a good solution in a much better time. We measure the RTL design cost by the total area of the RTL components (ALUs, Registers, MUXes) obtained from a component library. Although this metric maybe adequate for some designs, we use a layout estimator to provide a much more realistic cost evaluation of our designs.

One of the characteristics of the synthesis process is the interaction with the user. Thus, the user or the designer should be given some control over the optimization process and the guiding mechanism. The guiding process is illustrated by various issues. Firstly, the designer has the option to specify the design library. Secondly, the designer can specify the desired area and/or delay of the design.

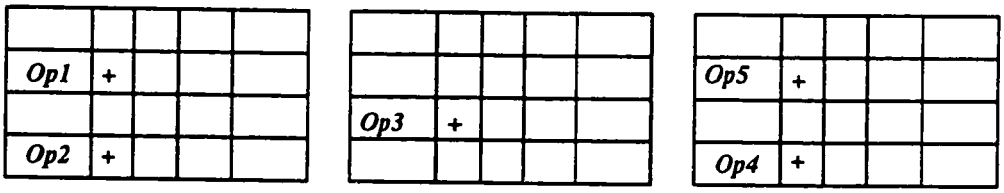


Figure 5.5: Simple example

5.5 Phase I: Components Reallocation

5.5.1 ALUs reallocation

Given a datapath, the lower bound on the number of ALUs is predetermined by the schedule. If this number is similar to the one in the datapath, then no improvement in ALUs type is possible. If no improvement in area is possible, it is still possible to selectively exchange the ALUs with slower or faster ones in order to improve the datapath delay. But this will be covered later in coordination with delay estimation. The strategy in this phase is to fix the number and types of ALUs and proceed to other components.

Assuming that the datapath under reallocation has multifunctional ALUs, then we distinguish among the following four reallocation cases:

1. From *single function* ALUs to *single function* ALUs.
2. From *single function* ALUs to *multi-function* ALUs.
3. From *multi-function* ALUs to *single function* ALUs.
4. From *multi-function* ALUs to *multi-function* ALUs.

Each case of the above four cases will be treated separately in the subsequent sections.

Reallocation from single functions ALUs to single functions ALUs

The reallocation process in this case will not change the binding but rather reduce the number of ALUs. The reason in this case for the large number of ALUs is a bad allocation method *or* an allocation optimized under a different technology library.

Let A_1, A_2, and A_3 be three ALUs in a given datapath such as in Figure 5.5. From the schedule, we notice that the maximum number of concurrent adders is 2. Therefore, we know that we can reduce the number of adders in the datapath to two and that will be our stopping criteria.

Definition 1 *Define the horizontal mobility, $M(Op)$, of a given DFG operation, assigned to ALU_i, and scheduled at time step j as:*

$$M(Op) = \sum_{allALUs} [ALU_i \text{ is not busy at time step } j]$$

For the example shown in Figure 5.5, $M(Op1) = 1$ while $M(Op3) = 2$.

Definition 2 *An ALU is reduceable if* $\forall$ *operation* $Op_j \in ALU$, $M(Op_j) \neq 0$.

In the above example shown in Figure 5.5, $M(Op_3) \neq 0 \Rightarrow A_2$ is reduceable. The same applies to A_1 where $M(Op_1) = 1$ and $M(Op_2) = 1 \Rightarrow A_1$ is reduceable as well.

We note that if one ALU is reduceable, then there exists at least one more ALU which is reduceable. To solve this problem, we pick a reduceable ALU with the most number of operations and apply the move transformation.

From multi function to single function ALUs

Given a data path which consists of multifunctional ALUs, it can be transformed into one with only single function ALUs. The motivation is to reduce the problem to the one described in the previous section, and it is accomplished as follows.

We split every ALU in the datapath into its individual single functional units, using the *ALU split* transformation. For example, if an ALU has the operation set $+ - *$, we split it into three ALUs: $+$, $-$, and $*$. Next, we apply the first procedure (single to single) in order to optimize the area and/or number of ALUs.

Reallocation from multi functions ALUs to multi functions ALUs

In a multifunctional ALU, the most costly operation dominates the overall ALU cost. Thus, it is useful to develop a strategy with the aim of reducing the number of such costly operations from the datapath. Note that the same strategy will apply to reduce cheaper operations from the ALUs if there is no room to improve the cost by transformations of more expensive operations.

Consider a datapath consisting of $ALU_1, ..., ALU_n$. Let O_k denote an operation type with $k = 1, ..., m$ where m is the number of operation types in the data path. Let $f(O_k)$ denote the *frequency* of the operation O_k, i.e., the number of times O_k appears in the ALUs of the datapath.

The basic property that characterizes the constituent transformations of *Reduce* is that they all have the same *source* ALU. Informally, the strategy is as follows:

- *Reduce Strategy:* Removing one operation of type O_k from an ALU. We show this by the notation: $Reduce(ALU_i, O_k)$. To achieve this strategy we have to move all operation variables of type O_k from ALU_i to other ALUs. For $Reduce(ALU_i, O_k)$ to be possible there must exist at least one sequence of valid moves that make up this strategy.

The motivation of $Reduce(ALU_i, O_k)$ is to reduce the frequency of O_k, $f(O_k)$, by one or more. For our example of Figure 5.2, $f(*) = 3$, $f(+) = 1$, $f(-) = 2$. Clearly, after $Reduce(ALU_2, *)$ in Figure 5.2, we have $f(*) = 2$. In this example, we used $Reduce(ALU_2, *)$ since $*$ is the most expensive operator and dominates the cost of the datapath components. To achieve this strategy, we applied $[MOVE : mul1 \in ALU_2 \rightarrow ALU_4]$. Let us assume that in this example multiplication is

three times more costly than an addition, subtraction, or comparison. Then the result of this strategy is a 20% cost reduction on component (ALU) cost.

Thus, our reallocation approach aims at reducing the frequency of the most costly operation in the datapath, by applying the Reduce strategy for this operation repeatedly, as much as possible. This approach can continue and apply to other operations, in decremental order of their costs. This works with $*$ and $+$ operations because $Cost(*) > Cost(+)$. However, there is a question about choosing between $+$ and $-$ for the Reduce strategy because their costs are about the same. We handle these questions by introducing the *move table and procedures* with respect to an operation type and which select a given strategy.

A. Move tables and procedures

The move table is an effective data structure that captures all move/swap transformations concerning O_k in an efficient way to form the Reduce strategies. For our example data path, the Move Tables with respect to operations $*$ and $+$ are shown in Figure 5.6(a) and 5.6(b), respectively. The rows correspond to the time steps T_i and the columns to ALU_j. Every entry (T_i, ALU_j) of a Move Table consists of either a set of integers $a, b, c, ...$ or a blank. The integers denote destination ALUs for moving operation type O_k, such as a $*$, from ALU_j, and these moves are valid in time step T_i. The blank entries mean no moves are valid in that time step.

Thus, in Figure 5.6(a), entry (T_2, ALU_3) is 2 meaning that the destination ALU is ALU_2, i.e., the following move is valid: $[MOVE : mul3 \in ALU3 \rightarrow ALU_2]$. Some entries in the Move Table are accented. The reason is that those destinations are associated with a Swap rather than a Move. For example, entry (T_3, ALU_3) is $2'$, meaning the following transformation is valid: $[SWAP : mul5 \in ALU_3 \leftrightarrow cmp \in ALU_2]$. In Figure 5.6(b), the entry (T_4, ALU_1) is $2', 3, 4$. This means either of the three transformations with respect to $+$ from ALU_1 are valid. In effect, every integer entry in the Move Table represents a move or swap transformation

in symbolic form.

Given this structure of the Move Table for O_k, we can compose the strategy *Reduce*(ALU_i) by the following simple procedure:

- For every integer entry in a time step under ALU_i, and for every step, combine a single integer entry in each time step with a single integer entry in every other time step. The result is all possible move/swap sequences that compose *Reduce*(ALU_i).

Thus in Figure 5.6(b), the strategy *Reduce*($ALU_1, +$) is achieved by the following three move/swap sequences written symbolically in AND/OR form:

2&(2′ OR 3 OR 4) = (2&2′) OR (2&3) OR (2&4)

Also, in Figure 5.6(a), the strategy *Reduce*($ALU_3, *$) is achieved by the sequence of moves 4&2&2′.

Although there are very few sequences in the above examples that compose each *Reduce*(ALU_i) strategy, it may appear that in the worst case there may be very many such sequences. In general, one may have a growth of the order of $(n-t)^t$ where t is the number of time steps and n the number of ALUs. However, this will happen only when dealing with *sparse* datapath charts, and this is not the case with designs that have gone through an allocation process.

An interesting property of the Move Table is that in every row the non blank entry sets are identical to each other. For example, entries under ALU_2 and ALU_3 in time step 2 or 3. The reason is these entry sets indicate the "empty slots" of the datapath chart that an operation, say $*$, can move. Since ALU_2 and ALU_3 both do $*$ in time step 2, then $*$ can move into the same slots. This property significantly reduces the time for building the Move Table. We have the following comments.

- *Comment 1:* We note that in the Move Table of Figure 5.6(a) the column under ALU_1 is all blank. The reason is that we do not want to move $*$ from the other ALUs into ALU_1 because such a transformation will increase the data path cost by creating a new multiplier.

- *Comment 2:* Reducing the frequency of an operator say $*$ not only reduces the number of ALUs containing $*$, but it may even reduce the actual number of ALUs.

- *Comment 3:* The notion of Reduce can be extended to apply not only to a single operator but also to an operation set, for example, if $O_k = \{+, -\}$ then $Reduce(ALU_i, O_k)$ refers to removing $\{+, -\}$ from ALU_i. We remark that when this applies to two single function ALUs, ALU_i and ALU_j, containing only $+$ and $-$ respectively, then $Reduce(ALU_i, O_k)$, if possible, will modify ALU_j into a multifunctional, $+, -$, whereas ALU_i will in effect be eliminated. In this case, $Reduce(ALU_i, O_k)$, has the effect of *merging* two single function ALUs into one multifunctional ALU, which could reduce the cost. The point here is that by properly choosing the set O_k in the Reduce strategy, we can have several different effects on the datapath while reducing the cost.

Procedure 1: The following reallocation procedure is used to reduce the ALU cost of a given datapath based on the Reduce strategy for operati on set type O_k.

1. Form the Move Table of *current-datapath.*

2. Generate $Reduce(ALU_i, O_k)$ for $i = 1, ...n$.

3. Compute the cost of *current-datapath* for the strategy in question.

4. For $i = 1, ..., n$ select a strategy with lowest cost.

5. Perform the selected strategy generating *new-datapath.*

6. Go to step 1.

The above procedure terminates when in some *new-datapath* no Reduce strategy is possible (step 2). If there are n ALUs, then clearly there could be $n - 1$ iterations, at worst.

We illustrate this procedure for our running example of Figure 5.2. Using the Move Table of Figure 5.6(a), the $Reduce(ALU_i, *)$ strategy, $i = 2, 3, 4$ results in three datapath charts, respectively shown in Figure 5.7 a,b,c. Of the three strategies the one with the least cost is $Reduce(ALU_2, *)$ with new $f(*) = 2$.

T	ALU1	ALU2	ALU3	ALU4
1		4	4	
2			2	2
3			2'	2'
4				

(a)

T	ALU1	ALU2	ALU3	ALU4
1				
2	2			
3				
4	2',3,4			

(b)

Figure 5.6: Move tables for the runing example

T	ALU1	ALU2	ALU3	ALU4
1			mul0	mul1
2	add0		mul3	mul2
3	sub0	cmp	mul5	mul4
4	add1	sub1		

(a)

T	ALU1	ALU2	ALU3	ALU4
1			mul0	mul1
2	add0	mul3		mul2
3	sub0	mul5	cmp	mul4
4	add1	sub1		

(b)

T	ALU1	ALU2	ALU3	ALU4
1		mul1	mul0	
2	add0	mul2	mul3	
3	sub0	mul4	mul5	cmp
4	add1	sub1		

(c)

Figure 5.7: Data Path charts for our runing example

However, the Reduce strategy on $*$ is no longer possible. We will now consider the earlier question, how would we choose among the operations $\{+, -, cmp\}$ assuming, as it is usually the case, that they have almost the same cost.

To handle this problem, it is reasonable to choose the operation with the largest frequency for the one to apply the next Reduce. In the example of Figure 5.7(a), we have $f(+) = 1, f(-) = 2, f(cmp) = 1$. Thus the $-$ is the most frequent operation to apply Reduce, i.e., to reduce $f(-)$ by one. So we apply $Reduce(ALU_i, -)$ with $i = 1, 2, 3, 4$ using the Move Table for $-$, Figure 5.8(a), based on the new datapath chart. Thus $Reduce(ALU_1, -)$ and $Reduce(ALU_2, -)$, respectively, can

T	ALU1	ALU2	ALU3	ALU4
1				
2				
3	2'			
4		1'		

(a)

T	ALU1	ALU2	ALU3	ALU4
1			mul0	mul1
2	add0		mul3	mul2
3	cmp	sub0	mul5	mul4
4	add1	sub1		

(b)

T	ALU1	ALU2	ALU3	ALU4
1			mul0	mul1
2	add0		mul3	mul2
3	sub0	cmp	mul5	mul4
4	sub1	add1		

(c)

Figure 5.8: (a) Move table for $-$ using Figure 5(a), (b) Data path after reduce ALU1, (c) After reduce ALU2.

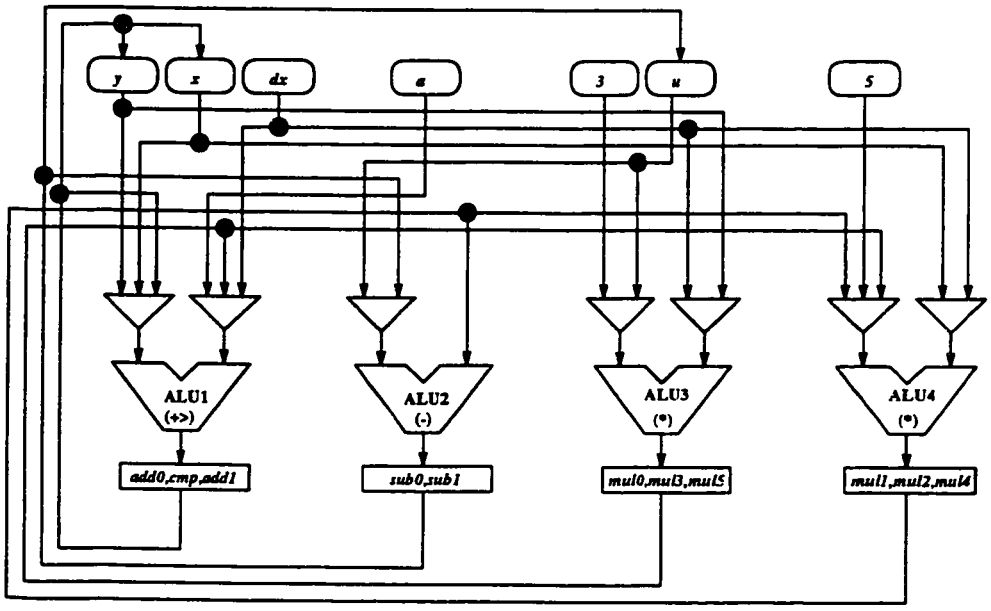


Figure 5.9: The Hal example after reallocation

be effected by the symbolically represented swaps, $2'$ and $1'$, respectively. Their corresponding new datapath charts are shown in Figure 5.8(b) and (c) respectively. Note that the Reduce strategy should reduce the frequency of the operation it applies and this disqualifies several other possible destination moves of $-$. Of the two possible Reduce above, both making $f(-) = 1$, the one which does not increase $f(+)$ is chosen, namely $Reduce(ALU_1, -)$.

The above example leads to the following general procedure for reallocation a datapath, outlined below.

Procedure 2:

1. Begin with the most costly operator, O_k.
2. $Reduce(ALU_i, O_k)$, $i = 1, ..., n$ by applying Procedure 1.
3. Choose, on the basis of cost, one $Reduce_i$

4. Form new datapath and Move Tables

5. If the other operators do not have similar cost (i.e., $+, -, >$), then goto 1.

6. Compute frequencies for the rest of operations.

7. Choose most frequent operator, say O_{k+1}

8. $Reduce(ALU_i, O_{k+1})$,

9. Go to step 3

This procedure terminates when all the Reduce strategies have applied to all operations in the data path, and each Reduce has terminated by Procedure 1.

From single function to multi function ALUs

This case is a special case of the above case.

5.5.2 Registers Reallocation

The next step in the reallocation process, before proceeding to phase II, is to explore possibilities in registers improvement using *swap* and *move* transformations. The transformations will not affect the datapath functionality since they are applied only to ALUs of the same type. The number of optimum number of registers in this case can be obtained from the schedule.

In order to reduce the number of registers, we have to move or swap operations and their variables in the same time. This is done as follows:

- Move a variable from one register to another one using transformation *store variable move*. The precondition for this move is that no life span conflict exists in the destination register.

To achieve the above goal, we proceed as follows. We construct the register grid (Figure 5.10) and look for the available "windows" or empty slots in the destination register. Such windows provide an opportunity to accommodate variables

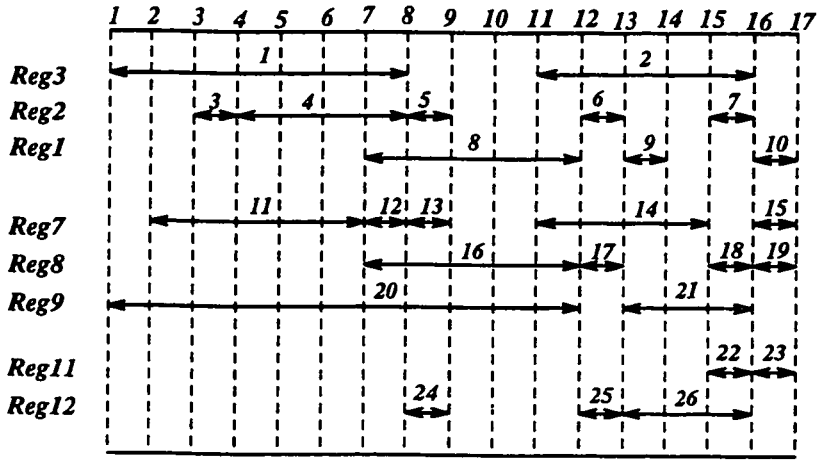


Figure 5.10: Registers Reallocation: initial solution from wave

Input: Grid for a given datapath
Output: Grid with reduced number of registers

Construct a list of registers corresponding to ALU's that have the same functionality. For every list do:

1. Set $j = 1$
2. Count the number of available slots or windows for every register i, $i = j, ..., n$. Let $Count_i$ be this number.
3. Sort all the registers from $i = j, ..., n$ in decreasing order, based on $Count_i$.
4. For all registers from $i = j+1, ..., n$, apply the transformation $[MOVE : SV_a \in REG_i \rightarrow REG_j]$, $\forall a \in REG_i$.
5. Increment j. If j $\geq$ n, then done. Otherwise go to step 2.

Figure 5.11: Registers Reallocation Algorithm

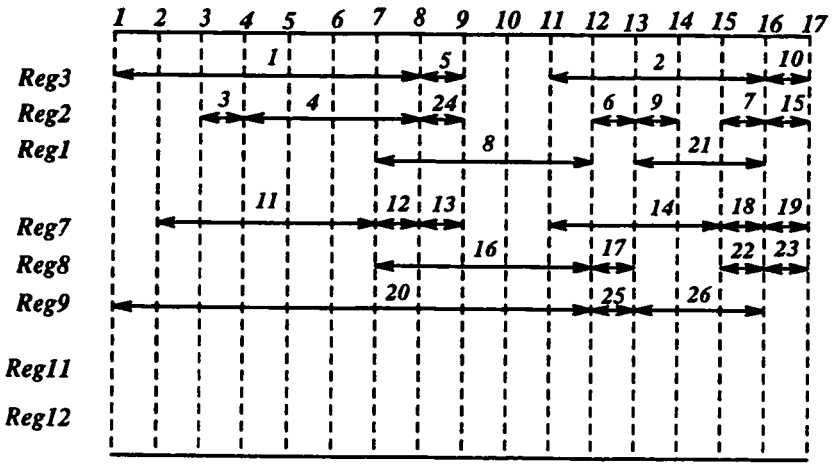


Figure 5.12: Registers Reallocation: final solution from wave after reallocation

in other registers. The basic idea is then to fill, in a systematic way, as many of these windows as possible so that to create empty registers. The empty registers will be later deleted using transformation *eliminate register*. We note that in our example, the reallocation process reduced the number of registers by two (Figure 5.12), resulting in six registers. The register reallocation algorithm is shown in Figure 5.11.

5.6 Phase II: Reallocation with Layout Consideration

Once the components cost has been improved, phase II improves the interconnect area using a layout estimator. Our strategy in this phase is based on the following considerations:

- *Fixed schedule*: This forces to move or swap operations in the same control steps.
- *Fixed functionality*: We may only move or swap operations without altering the data path functionality since we do not want to degrade the result obtained in the first phase of reallocation.

- *Layout estimation critique*: At each iteration, the layout estimator can provide the interconnect cost which may account till up of 75% of the chip area. The final decision will be made based on the largests cost gain.

The algorithm for phase II has been summarized below:

1. *Step 0*: Use layout estimator to compute layout cost of the initial design (*dpcost*). Sort operation types based on the cost of their operators. Sort the ALUs based on their connectivity factor. Set $i = 1$ and $j = 1$.
2. *Step 1*: Consider ALU_i and find the operations of type j and their corresponding control steps e.g. O_k and t.
3. *Step 2*: Check control step t in $ALU_{i+1}, \cdots, ALU_n$. If it contains an operation of type j, call the layout estimator to compute *dpcost* after applying the *exchange* transformation.
4. *Step 3*: Evaluate the new data path cost. If there is improvement , accept that transformation, otherwise stop.
5. *Step 4*: Compute $j = j + 1$. If $j > k$ stop, otherwise go to step 1.
6. *Step 5*: Compute $i = i + 1$. If $i > n$ stop, otherwise go to step 1.

The algorithm will terminate in $n * k$ iterations, where n and k are the total number of ALUs and operation types, respectively.

5.7 Results

The described reallocation scheme was implemented along with the layout estimator as a post-synthesis tool to our synthesis system [PaCH91b]. We used the wave filter benchmark example [PaKn87] to further illustrate our method. We generated two schedules using time steps 17 and 19. The initial solution was allocated using a heuristic method [PaCH91b]. Thus, the reallocation approach makes sense to derive improvements. The results for the wave filter are shown in Figure 5.13 and 5.14. The first graph (Figure 5.13) shows the improvement in the solution after executing the *Reduce* strategies. The first data points indicate the design

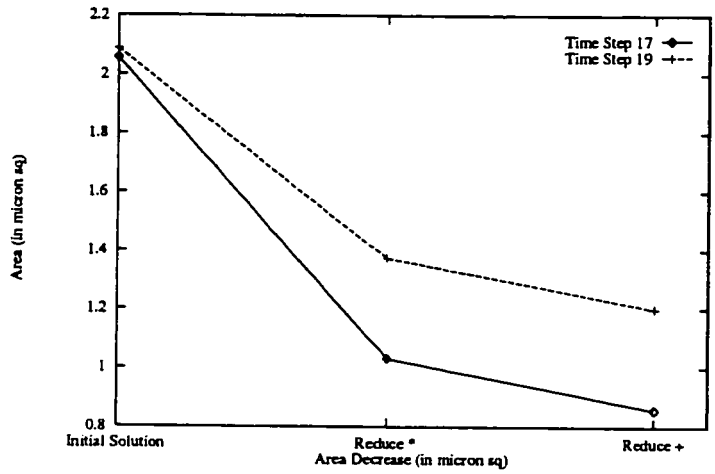


Figure 5.13: Results from wave filter after applying Reduce strategies (Components area only)

initial area generated by the synthesis system. The second data points show the solution after applying the strategy $Reduce(ALU_i), *)$, while the last data points show the area for the final solution after applying the strategy $Reduce(ALUi, +)$.

The layout area was computed and taken into consideration in Phase II, and the results are shown in Figure 5.14 for time step 17 and 19 respectively. The First data points indicate the initial solution with interconnect consideration generated by the synthesis system, while the second data points indicate the area after applying phase I. The third data points show the improvement after applying phase II.

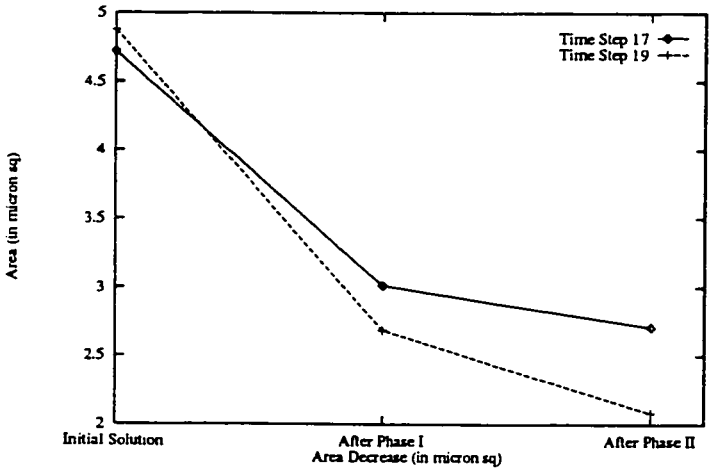


Figure 5.14: Results from wave filter after Phase I and Phase II

Chapter 6

VHDL Generator: A Bridge to Lower Level Synthesis

The previous chapters described some of the synthesis algorithms in our prototype synthesis for testability system, SYNTEST. In specific, we described the *allocation* and the *reallocation* methods. We also described the *test-points* selection method that SYNTEST uses in order to reduce the design test overhead. In this chapter, we describe the *VHDL generator* which provides a bridge to lower level synthesis tools. Finally, we put it all together by describing the synthesis process using an example in a journey from behavior to silicon.

6.1 VHDL: Quick Overview

VHDL, short for VHSIC (Very High Speed IC) Hardware Description Language, is a formal language for hardware description, standardized by IEEE in 1987. VHDL usually describes a design using one or more modules, each described using a declaration part (*entity*) and an *architecture* part. The declaration part declares the module as a black box, identified by a name, and an interface to the outside world (input/output ports). The architecture part describes the module implementation, and contains various concurrent processes; the simplest of which is a simple assignment statement. A process could be a complex one, containing a declaration part, a sensitivity list that sepcifies the signals the process is sensitive to, and a statement part in which all statements are executed sequentially in the order they appear. VHDL supports a timing model as well by associating

a waveform with signals, and by using constructs such as the wait statements. Design entities can also use global declarations in the architecture to establish communication among various processes as well as declarations from external packages and libraries.

6.2 VHDL: A High-Level Synthesis Perspective

VHDL is a "multilevel" simulation language, which can be described at the *behavioral*, *data flow* or *structural* levels. The data flow description is usually used for logic synthesis and will not be addressed in this work. The VHDL language has become a standard for describing, designing and simulation of IC circuits. Thus, it has many semantics suitable for simulation rather than for synthesis. In what follows, we describe and identify the VHDL features that we accept for high-level synthesis in SYNTEST. The input to high-level synthesis is a behavioral level description, leading to a structural one, in the form of a state-machine description (FSMD). We use SYNTEST in order to transform a behavioral description into a structural one by:

1. Verifying the syntax, the semantics, and the validity of the original VHDL description.
2. Transforming the VHDL description into a consistent control data flow format.
3. Maximizing the concurrency using scheduling since only synchronous hardware is allowed in high-level synthesis.
4. Mapping and optimizing the original behavior to RTL hardware components that implement the intended behavior, through datapath allocation.
5. Extracting a finite state machine controller description for the datapath.

In what follows, we discuss the VHDL constructs handled in SYNTEST.

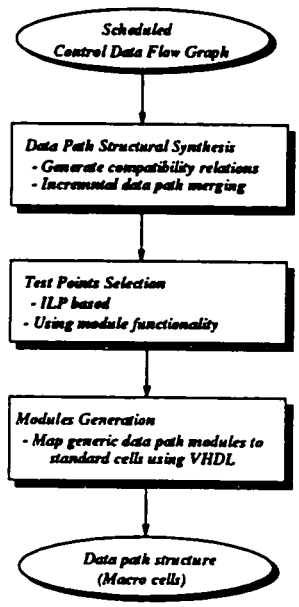


Figure 6.1: From behavior to silicon: Process

Design Entities

Design entities represent a well defined design component with input and output ports. The entity has an *entity declaration* part which is used to define the interface between a given design entity and the outside world, and an *architectural body* that describes the behavior to be synthesized and/or simulated.

We use the entity declaration statement at the highest level of the VHDL model in order to define the hardware model's input, output and bi-directional signals. The final structural view, after synthesis, has the same original signals in addition to a *reset* signal, the system *clock* signal, and the *scanin* and *scanout* signals in case of the test options. Generics are ignored at this level.

Subprograms

Subprograms define algorithms for computing values and include *functions* and *procedures*. In SYNTEST, they are handled by flattening each subprogram call

using in-line expansion and synthesizing one copy of the necessary hardware for each cell. This may create problems if the subroutines are large. A solution in here would be to synthesize each subroutine independently and creates passing mechanisms for passing both control and data into the synthesized hardware. However, the latter approach is not taken in SYNTEST. Packages are handled in the same way.

Types

Data typing is characterized by the number of bits, the type (boolean, integer, ...) and data representation (signed, unsigned, 2's complement, ...). Data typing allows the compiler to detect semantics errors and allows for consistency check. However, overloading data types may burden the synthesizer for no reason. Thus, we allow integer, bit, and bit vectors in SYNTEST. In case of integer, a 4 bit data path width is assumed unless the user has used the *range* construct in which case the bit width is computed. For arrays, we allow all the above types. No composite types, record types, or access types are allowed in the current version of SYNTEST.

File Types

Files are not intended to represent real hardware and thus have no place in high-level synthesis, and thus ignored by SYNTEST.

Signals and variables

VHDL is a language based on simulation semantics. Signals represent data carriers that are assigned values at discrete points of time. In SYNTEST, we map VHDL signals to storage elements or physical wires. Signals are allowed in SYNTEST except for resolved ones. Signals, variables declarations and assignment are allowed as well.

Operators

SYNTEST supports the following logical, arithmetic, and relational operators: *and*, *or*, *not*, *xor*, +, −, *, /, <, >, =. Additional operators can be handled as long as they are introduced in the SYNTEST internal library. The VHDL operators are mapped by the SYNTEST allocator to RTL circuits.

Expressions

An expression is a formula that defines the computation of a value. SYNTEST supports arithmetic expression evaluation.

Wait Statements

Wait statements are asynchronous by nature while high-level synthesis is synchronous. The reason is that the behavior is expressed in terms of register transfers for each clock cycle in the design. Thus, a fundamental friction exists between the simulation semantics of the wait statement and the actual behavior. This implies that the wait statement cannot be precisely synthesized in high-level synthesis.

Assertion Statements

Assertion statements are ignored during high-level synthesis in SYNTEST.

Conditional Statements

If-then-else and *case* constructs are fully supported in SYNTEST. If the case statement is too complex, then the controller description will be much more complex.

Loop, Next, Return and Exit Statements

Loop statements are handled in SYNTEST by transforming the loop into a condition. The *next* and *exit* statements are synthesized as branches in the FSM

transition. However, the target will differ in this case since in case of the next, the target is the first statement in the loop while in the exit statement the target is the state to which the control would return following normal completion of the appropriate loop statement. The return statement assigns a value to a register which is copied after flattening the function body.

Concurrent Statements

Only *processes* are handled in SYNTEST. The current version handles just a single process due to modeling restrictions in high-level synthesis, as described shortly.

Blocks

Blocks are structural in nature and thus are not considered to be within the domain of high-level synthesis.

Process Statements

Process statements are the fundamental behavioral feature of VHDL and consequently supported by SYNTEST. Process statements allow the designer to describe the model's operation by means of algorithmic, sequential statements. The sensitivity list is ignored though and the body of the process is mapped to a datapath and a controller description.

Optimization

SYNTEST supports the following optimization transformations:

1. Copy and constant propagation.
2. Dead-code elimination.
3. In-line expansion of subroutines.

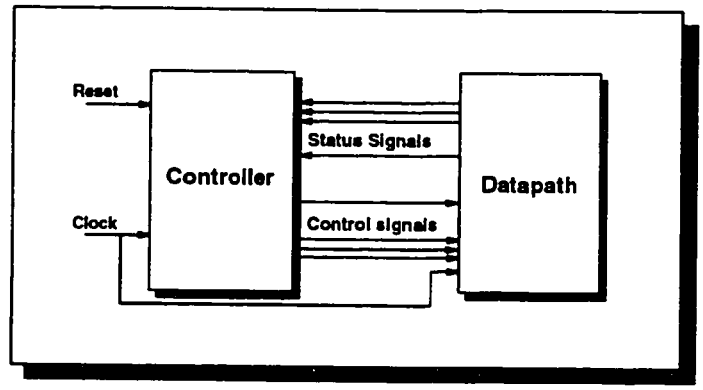


Figure 6.2: SYNTEST Design Model

6.3 Describing The Output

SYNTEST supports various output formats including BDS (OCTTOOLS), VHDL, and a net-list description. The VHDL outputs are geared toward some specific synthesis tools requirements, in specific toward the COMPASS and the ALLIANCE tools. In what follows, we describe the datapath and controller VHDL description in SYNTEST.

6.3.1 Data Path Description

The data path in SYNTEST is described using a mixed behavioral and structural description. Components are declared and then instantiated using signals in order to create a structural description for the data path. The data path components are, in turn, described in one of two formats: *behaviorally*, geared toward COMPASS Design Automation Tools, and in a *data flow format*, geared toward the ALLIANCE tools.

Components in SYNTEST consist of functional units, registers, multiplexers, constants, port assignments and signals to connect all the previous components. The data path components are described next.

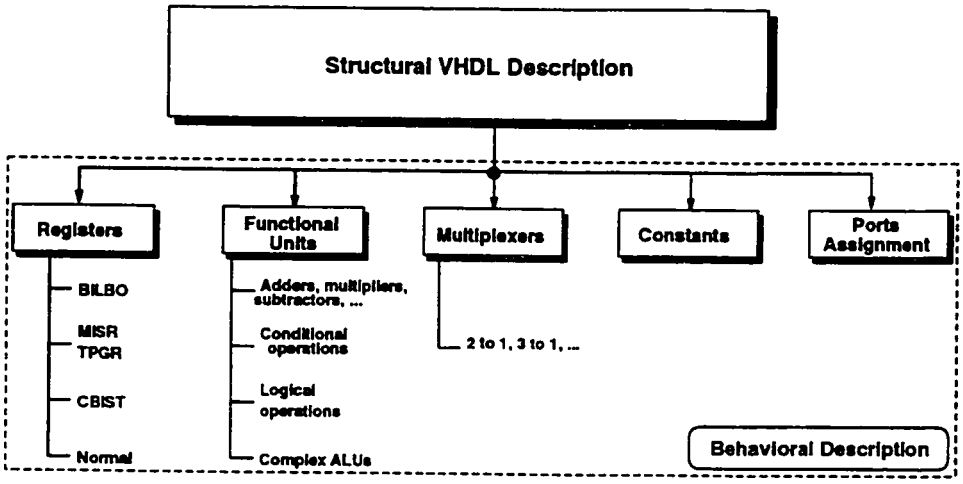


Figure 6.3: Data Path Description in VHDL

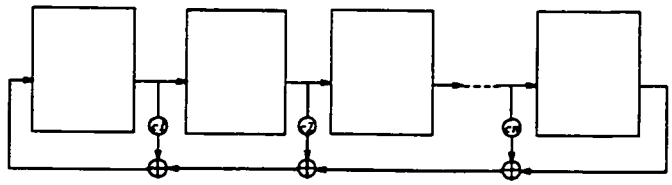


Figure 6.4: Linear Feedback Shift Register

Functional Units

Functional units in SYNTEST could be single or multifunctional. In the latter case, they require additional control lines in order to select the necessary function. The data registers are assumed to be of a constant width; therefore, the output of an $n \times n$ multiplication is an n-bit value, with the highest n bits being truncated. A comparator has one bit output, status signal, which is fed subsequently to the controller.

Normal Registers

We use the term normal registers to indicate registers which are idle during test mode. Registers are positive edge triggered, based on a D flip-flop implementation,

with an *enable* signal. All values in the registers can only be changed at the rising edge of the system clock and stored if the enable signal is high.

Test Registers

Test registers are used during test mode in order to test the data path components. A test register could be an MISR, TPGR, Circular BIST, or a BILBO. The function of each register is described at the behavioral level, based on Autonomous Linear Feedback Shift Register (LFSR). An LFSR has the canonical form shown in Figure 6.4. In this Figure, c_i is a binary constant that implies a connection exists if $c_i = 1$, while $c_i = 0$ implies that no connection exists. The LFSR is usually represented using a *generating function* with the denominator denoted by a polynomial, $P(x) = 1 + c_1x + c_2x^2 + \cdots + c_nx^n$. The polynomial is usually referred to as the characteristic polynomial of the LFSR and it characterizes the LFSR during signature analysis and test patterns generation.

A test register in SYNTEST can act as a normal register during the system normal execution mode. Furthermore, all test registers can act as shift registers in order to bring all test registers to a known initial state, and to shift out the final test signature after the test has been completed. A TPGR has a test patterns generation mode that generates pseudorandom test patterns during testing while an MISR register has an additional signature analysis mode that compresses the test responses. A BILBO has all the above functions, and can act as both an MISR and a TPGR, but in different test sessions. Figure 6.5(a) shows a BILBO schematic while Figure 6.5(b) shows a blow-up for a cell. The BILBO operation modes are shown in Figure 6.5(c).

In order to implement the test registers, it is imperative to specify the feedback connection, c_i for the LFSR characteristic polynomial. Thus, SYNTEST accepts the polynomial coefficients for input and constructs the feedback function. Note that it is the designer responsibility to select a good characteristic polynomial and

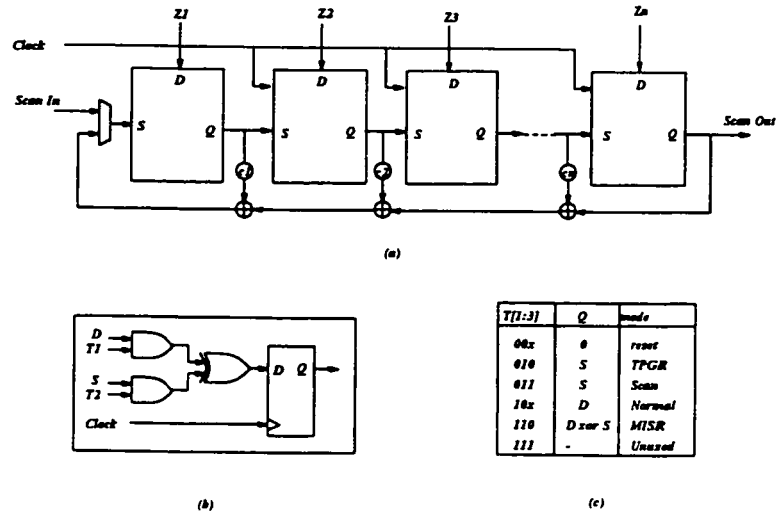


Figure 6.5: BILBO register: (a) Schematic; (b) Cell description; (c) Modes of operations

no effort will be made to check the "wisdom" of such a choice.

Multiplexers

Multiplexers in SYNTEST are described using case statements. It is interesting to note in here the difference between synthesis and simulation interpretation of constructs. For example, COMPASS Design Automation always infers a latch in the case of the OTHERS branch in a case statement.

Constants

Constants are described using standard functions from COMPASS Design Automation and then stored in registers for later use.

6.3.2 Controller Synthesis

The data path controller is described using a FSMD using a control table which specifies the next state and a set of control signals as a function of the present

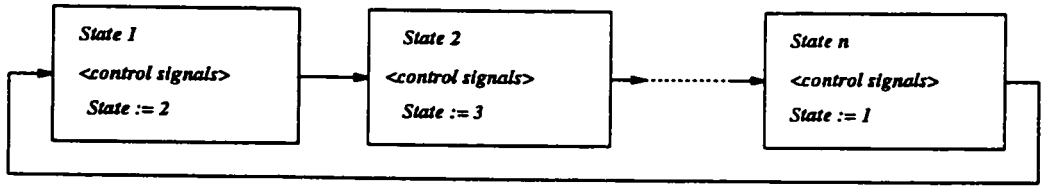


Figure 6.6: Ring State Transition Diagram

state. The present state is usually encoded as binary values $p_k \cdots p_1 p_0$ where $k \leq \lceil log_2 \rceil - 1$ and m is the number of states. The next states are encoded as binary values $r_k \cdots r_1 r_0$. Each output signal, c_i, controls a functional unit, a storage element, or an interconnect component in the data path. The controller model assumes a synchronous system and can be implemented using a PLA model, or random logic.

The control unit consists of three register types: *state registers* which keep track of the current state and decides on the next state; *output registers* which hold the control signals to avoid possible race conditions between data path and controller during execution, and *status registers* which maintain condition values. There are three important stages, necessary for the controller generation:

- The determination of the controller states.
- The determination of the next state transition.
- The determination of the control signals for each state.

The first two issues are easily determined from the scheduler. Thus, each time step is mapped to a controller state. We determine the next state based on the next contiguous control step, as implied by the FSMD, except in the case of loops and branches. The last stage is determined incrementally, and was described within the context of cost in chapter 4, and will be described more shortly.

Controller Synthesis: An Incremental Approach

As we have discussed in Chapter 4, data path allocation in SYNTEST is an incremental process. Thus, an initial data path (IDP) is generated based on a one-to-one mapping. All the control information, including the time step during which control signals are activated, are stored in the IDP. Control is defined by producing a set of control signals, activated during the execution of the operations, assigned to state S_i. When mutual exclusive operations are encountered and merged, the data path stores the corresponding signals and later trigger the necessary branch. There are three types of signals in SYNTEST: register load signals or enables, multiplexers select signals, and ALU select function signals. Note that only multifunction ALUs have a function select.

Each time two TFBs are chosen for merging purposes, the control signals are updated. As the TFBs are merged, multiplexers are introduced along with more complex ALUs. The resulting data path has less registers but more multiplexers and possibly more complex ALUs. Thus, it is easy to introduce again the loss and gain functions in the context of controller in order to explore tradeoffs during data path and controller design, during allocation.

Controller Types

Depending on whether the initial behavior has conditional statements, two control styles arise. We describe both models next.

The first controller style results from a behavior without any conditional statements. The controller in this case is straightforward, in the form of a ring state diagram (Figure 6.6). We define a state as a set of control signals, activated when the state is being visited. A state diagram is a graph with states as vertices and transition arcs as its edges. A ring state diagram is a state transition where all the states are a continuous sequence and the last state has the first state for next state.

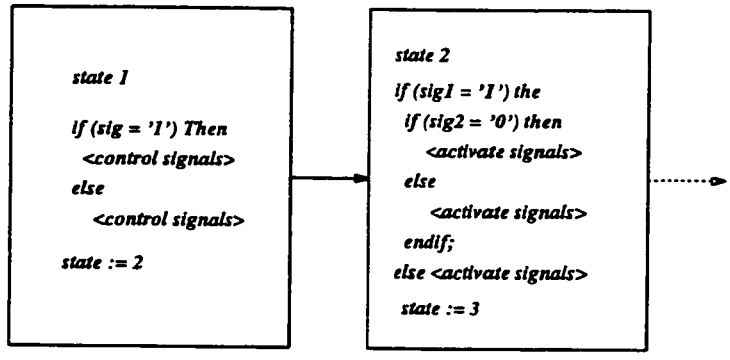


Figure 6.7: Controller State Diagram with Conditional Branches

In the case of conditional branches, each branch is represented by a separate set of states. If a state has only one condition value alive, then an if-then-else statement is used. Otherwise, a case statement is used (Figure 6.7).

We note the following aspects in the final controller design and which are rather due to synthesis aspects in COMPASS:

- The initialization of all the control signals before execution, at the asynchronous reset. Otherwise, the logic synthesis tool infers latches to store the initial value of the signals; thus, duplicating the controller logic.
- The number of cases in the controller is restricted to 14 as indicated by the *range* statement imposed on the *this_state* variable. The reason is due to a restriction by COMPASS in this regard. ALLIANCE cannot handle the process statement and thus cannot synthesize the controller part.
- ALLIANCE requires the specification of ground and power lines in the port specification of *entities*. This is not the case with COMPASS.
- Finally, we include COMPASS standard packages in the header of every file. These include an arithmetic and a component library.

6.4 From Concept to Silicon: An Example

6.4.1 The Differential Equation Example

The HAL differential equation example solves the following second order differential equation: $y'' + 3xy' + 3y = 0$. The example was first introduced by Pauline in [PaKn87] and used later as a benchmark example for high-level synthesis. We chose this specific example for two reasons. The first reason is the research community familiarity with its specifics. The second reason is the example simplicity which allows us to illustrate the design process with clarity and without much complexity.

6.4.2 Design capture and scheduling

The very first phase involved in the design process is to capture the initial behavior using a hardware description language (VHDL) which can be done easily using a text-editor.[1] The VHDL description for the differential equation example is shown in Figure 6.8. The VHDL code is parsed next and an intermediate code is generated in order to be passed to the scheduler. This allows for more flexibility in the design capture since the designer may decide to build a different parser or use a different language and then interface it to SYNTEST.

The next step is to schedule the code sequence generated by the parser in order to exploit any parallelism that may exist. The constraints on time and resources are specified next. Thus, the example will be scheduled in four time steps where every time step correspond to 100 ns. Furthermore, constraints are added on resources availability. Thus, we assume that only two adders, two multipliers, and one subtractor are available in the library. This will affect the operations concurrency in the schedule. The resulting schedule is shown in Figure 6.9.

[1] We could have also used the DFG Generator to capture the behavior graphically and then generate a VHDL description.

```
ENTITY diffeq IS
   PORT (Xinport:  in integer;
         Xoutport:  out integer;
         DXport:  in integer;
         Aport:  in integer;
         Yinport:  in integer;
         Youtport:  out integer;
         Uinport:  in integer;
         Uoutport:  out integer);
END diffeq;
ARCHITECTURE diffeq OF diffeq IS
   BEGIN
         PROCESS (Aport, DXport, Xinport, Yinport, Uinport)
         variable x_var,y_var,u_var, a_var, dx_var:  integer;
         variable x1, y1, t1,t2,t3,t4,t5,t6:  integer;
         BEGIN
               x_var := Xinport;
               a_var := Aport;
               dx_var := DXport;
               y_var := Yinport;
               u_var := Uinport;
               while (x_var < a_var) loop
                     t1 := u_var * dx_var;
                     t2 := 3 * x_var;
                     t3 := 3 * y_var;
                     t4 := t1 * t2;
                     t5 := dx_var * t3;
                     t6 := u_var - t4;
                     u_var := t6 - t5;
                     y1 := u_var * dx_var;
                     y_var := y_var + y1;
                     x_var := x_var + dx_var;
               end loop;
               Xoutport <= x_var;
               Youtport <= y_var;
               Uoutport <= u_var;
         END PROCESS;
  END diffeq;
```

Figure 6.8: HAL Differential Example in VHDL

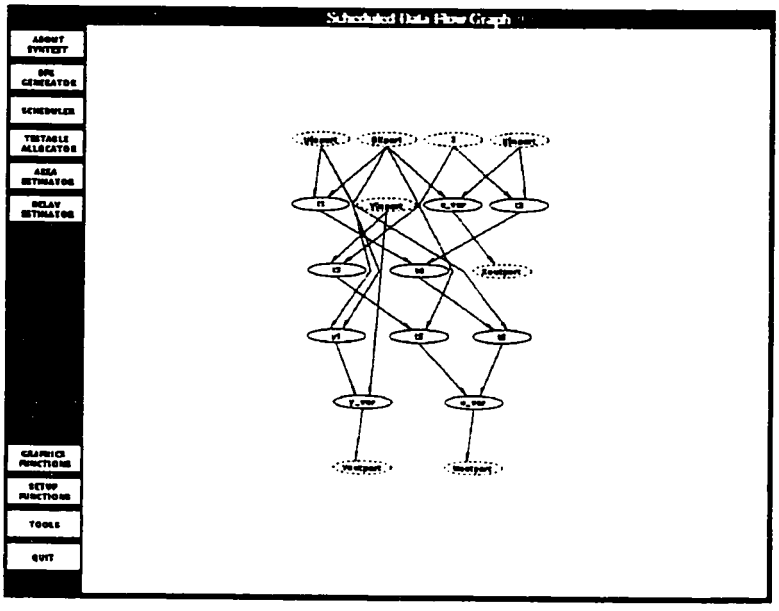


Figure 6.9: Differential equation schedule

6.4.3 Testable allocation and test points selection

In the next step, the allocator is executed in order to allocate hardware resources for our example. The allocation process is more interesting since many design styles and solutions possibilities can be explored using the various *design and test options* available in the allocator. The example is allocated with all design optimization options invoked. The test options are turned off at this stage. Once the allocator is ran, we notice that it cannot meet the scheduler constraints in terms of number of multipliers and thus the final solution requires three multipliers instead. The reason is due to testability considerations since variables ($t3$, $t5$), ($t1$, $t4$), and ($t2$, $t4$) cannot share the same multiplier since this will create a self-loop; thus, hindering the data path testability.

The same remark applies to the two variables $t6$ and u_var, resulting in two subtractors in this case instead of one. It should be noted that the self-loop restriction can be relaxed resulting in an optimum solution in terms of ALU types

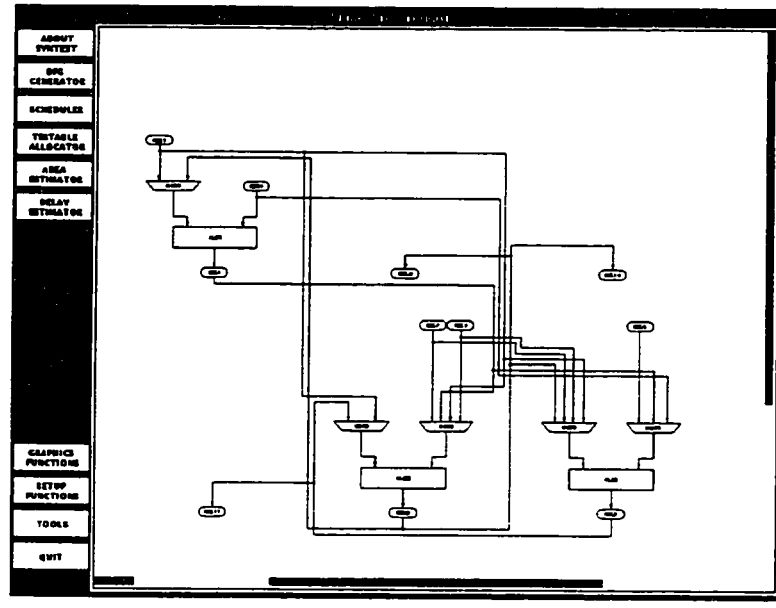


Figure 6.10: Differential equation data path

though the design may not be testable. Runing the allocator again using the test options will result in designs with test points selected; however, each design will have a different cost depending on the option used. The most expensive design would be based on the BILBO or strictly testable design with an overall cost of 83,292 micron squares followed by the test points selection scheme which uses the ILP formulation described in chapter 4, resulting in a cost of 81,392 micron squares. Finally, the test points selection with module functionality which is based on the previous approach but removes additional test points based on [ChPa91a] results in a cost of 74,360 micron squares. In the last case, registers *reg3, reg4, reg5, reg7* were configured as controllables (TPGRs) while register *reg2* was configured as observable (MISR). The design area for a normal design, without test considerations, was 69,844 micron squares, a distant last among all other design costs. The design critical path was 441 *ns*.

Option	Area (λ^2)	Routing Factor	# Standard Cells	Transistors Number
Normal	3,505,230	68%	309	4403
BILBO	5,628,720	75%	536	6739

Table 6.1: Design comparisons for the Differential Equation example after using COMPASS

6.4.4 Synthesizing The Output

The last stage in the design process is to synthesize the chip generated by SYNTEST at a lower level of abstraction using a synthesis tool, such as COMPASS. This is easily accomplished by synthesizing the root structural file of Figure 6.11 resulting in the structure shown in Figure 6.17 which was captured graphically from COMPASS. During the synthesis process, COMPASS ASIC Synthesizer flattens the components until the leaf behavioral components are reached. Figures 6.18, 6.20 and 6.21 show examples of typical RTL behavioral descriptions in SYNTEST for a register, four-to-one multiplexer, and a complex ALU, respectively. These leaf components are synthesized individually and then connected together as shown in Figure 6.22. We show a small portion of the controller in Figure 6.24 for illustration purposes and show the chip final layout in Figure 6.25 which was derived using COMPASS Chip Assistant.

```
Library ieee;
Library COMPASS_LIB;
USE ieee.STD_Logic_1164.all;
USE COMPASS_LIB.COMPASS.ALL;

ENTITY DIFFEQ_struct IS
   PORT (
        Uinport_port:  IN BIT_VECTOR(3 DOWNTO 0);
        DXport_port:  IN BIT_VECTOR(3 DOWNTO 0);
        Xinport_port:  IN BIT_VECTOR(3 DOWNTO 0);
        Yinport_port:  IN BIT_VECTOR(3 DOWNTO 0);
        Xoutport_port:  OUT BIT_VECTOR(3 DOWNTO 0);
        Youtport_port:  OUT BIT_VECTOR(3 DOWNTO 0);
        Uoutport_port:  OUT BIT_VECTOR(3 DOWNTO 0);
        RESET: IN BIT;
        CLOCK: IN BIT
        );
   END DIFFEQ_struct;
```

Figure 6.11: Top level chip description

```
Controller:  DIFFEQ_cont PORT MAP(
        load_reg1 => load_reg1_sig,
        load_reg2 => load_reg2_sig,
        load_reg3 => load_reg3_sig,
        load_reg4 => load_reg4_sig,
        load_reg5 => load_reg5_sig,
        load_reg6 => load_reg6_sig,
        load_reg7 => load_reg7_sig,
        load_reg8 => load_reg8_sig,
        load_reg9 => load_reg9_sig,
        load_reg10 => load_reg10_sig,
        load_reg11 => load_reg11_sig,
        mux_sel1 => mux_sel1_sig,
        mux_sel2 => mux_sel2_sig,
        mux_sel3 => mux_sel3_sig,
        mux_sel4 => mux_sel4_sig,
        mux_sel5 => mux_sel5_sig,
        mux_sel6 => mux_sel6_sig,
        alu_sel2 => alu_sel2_sig,
        alu_sel3 => alu_sel3_sig,
        sys_clk => clock,
        reset_sig => reset
);
```

Figure 6.12: Top level chip description: Controller

```
DataPath:  DIFFEQ_dp PORT MAP(
        mux_sel1 => mux_sel1_sig,
        mux_sel2 => mux_sel2_sig,
        mux_sel3 => mux_sel3_sig,
        mux_sel4 => mux_sel4_sig,
        mux_sel5 => mux_sel5_sig,
        mux_sel6 => mux_sel6_sig,
        alu_sel2 => alu_sel2_sig,
        alu_sel3 => alu_sel3_sig,
        Uinport_sig => Uinport_port,
        DXport_sig => DXport_port,
        Xinport_sig => Xinport_port,
        Yinport_sig => Yinport_port,
        Xoutport_sig => Xoutport_port,
        Youtport_sig => Youtport_port,
        Uoutport_sig => Uoutport_port,
        load_reg1 => load_reg1_sig,
        load_reg2 => load_reg2_sig,
        load_reg3 => load_reg3_sig,
        load_reg4 => load_reg4_sig,
        load_reg5 => load_reg5_sig,
        load_reg6 => load_reg6_sig,
        load_reg7 => load_reg7_sig,
        load_reg8 => load_reg8_sig,
        load_reg9 => load_reg9_sig,
        load_reg10 => load_reg10_sig,
        load_reg11 => load_reg11_sig,
        sys_clk => clock
);
```

Figure 6.13: Top level chip description: data path

```
SIGNAL reset_sig:  BIT;
SIGNAL Uinport_sig:  BIT_VECTOR(3 DOWNTO 0);
SIGNAL DXport_sig:  BIT_VECTOR(3 DOWNTO 0);
SIGNAL Xinport_sig:  BIT_VECTOR(3 DOWNTO 0);
SIGNAL Yinport_sig:  BIT_VECTOR(3 DOWNTO 0);
SIGNAL Xoutport_sig:  BIT_VECTOR(3 DOWNTO 0);
SIGNAL Youtport_sig:  BIT_VECTOR(3 DOWNTO 0);
SIGNAL Uoutport_sig:  BIT_VECTOR(3 DOWNTO 0);
SIGNAL load_reg1_sig:  BIT;
SIGNAL load_reg2_sig:  BIT;
SIGNAL load_reg3_sig:  BIT;
SIGNAL load_reg4_sig:  BIT;
SIGNAL load_reg5_sig:  BIT;
SIGNAL load_reg6_sig:  BIT;
SIGNAL load_reg7_sig:  BIT;
SIGNAL load_reg8_sig:  BIT;
SIGNAL load_reg9_sig:  BIT;
SIGNAL load_reg10_sig:  BIT;
SIGNAL load_reg11_sig:  BIT;
SIGNAL mux_sel1_sig:  BIT;
SIGNAL mux_sel2_sig:  BIT;
SIGNAL mux_sel3_sig:  BIT_VECTOR(1 DOWNTO 0);
SIGNAL mux_sel4_sig:  BIT_VECTOR(1 DOWNTO 0);
SIGNAL mux_sel5_sig:  BIT_VECTOR(1 DOWNTO 0);
SIGNAL mux_sel6_sig:  BIT_VECTOR(1 DOWNTO 0);
SIGNAL alu_sel2_sig:  BIT;
SIGNAL alu_sel3_sig:  BIT;
```

Figure 6.14: Signals connecting the data path and the controller components

```
ARCHITECTURE Controller OF DIFFEQ_cont IS
   BEGIN
   Process (RESET_SIG,SYS_CLK)
           VARIABLE this_state:  INTEGER range 0 to 13;
           VARIABLE reg_output_state:  BIT_VECTOR(0 TO 10);
           VARIABLE mux_output_state:  BIT_VECTOR(0 TO 9);
           VARIABLE alu_output_state:  BIT_VECTOR(0 TO 1);
           BEGIN
                IF (RESET_SIG = '1') THEN
                   this_state := 0;
                ELSE IF (SYS_CLK'EVENT AND SYS_CLK = '1') THEN
                        CASE this_state IS
                             WHEN 0 =>
                                  reg_output_state := B"11100000000";
                                  mux_output_state := B"1101101010";
                                  alu_output_state := B"00";
                                  this_state := 1;
                             WHEN 1 =>
                                  reg_output_state := B"11000000100";
                                  mux_output_state := B"0001110000";
                                  alu_output_state := B"00";
                                  this_state := 2;
                             WHEN 2 =>
                                  reg_output_state := B"11100000000";
                                  mux_output_state := B"1100010100";
                                  alu_output_state := B"01";
                                  this_state := 3;
                             WHEN OTHERS =>
                                  NULL;
                             END CASE;
                       load_reg1 <= reg_output_state(0);
                       alu_sel3 <= alu_output_state(0);
                       mux_sel1 <= mux_output_state(0);
                       mux_sel2 <= mux_output_state(1);
                       END IF;
                END IF;
           END PROCESS;
   END;
```

Figure 6.15: Partial controller behavioral description (Time step 3 is not shown)

```
ARCHITECTURE RTL_DataPath OF DIFFEQ_dp IS
BEGIN
     reg_1:  reg4
          PORT MAP(
               clock => sys_clk,
               input => out_alu1,
               output => out_reg1,
               load => load_reg1
               );
     cons_3:  cons4_3
           PORT MAP(
                output => const_3
                );
     mux4:  mux4_4to1
         PORT MAP(
              input1 => out_reg1,
              input2 => out_reg2,
              input3 => out_reg7,
              input4 => out_reg8,
              output => out_mux4,
              Sel => mux_sel4 );
     alu3:  alu4_73 -- function:  *+-
         PORT MAP (
              operand1 => out_mux5 (3 DOWNTO 0),
              operand2 => out_mux6 (3 DOWNTO 0),
              Sel => alu_sel3,
              result => out_alu3 (3 DOWNTO 0)
              );
     END;
```

Figure 6.16: Partial data path description (some components along with components and signals declarations are not shown)

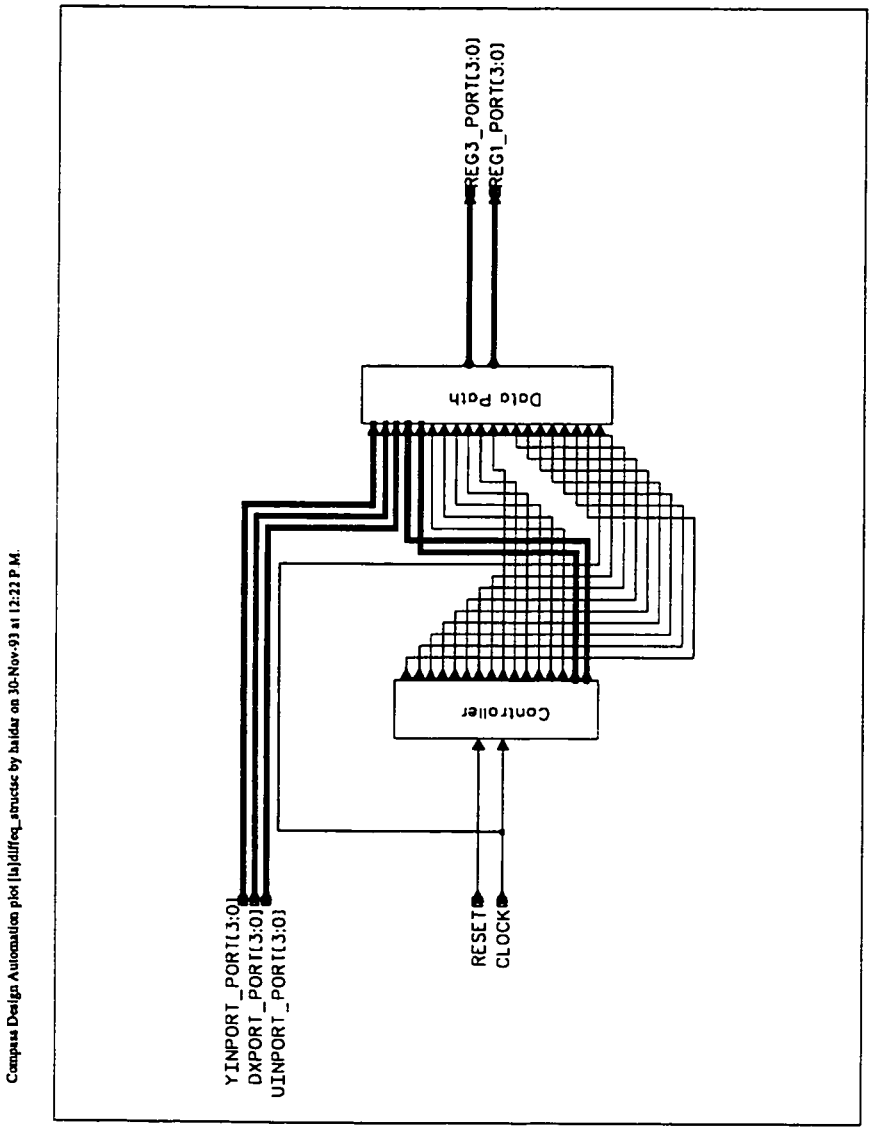


Figure 6.17: Chip structural view in COMPASS

```
ENTITY reg4 IS
   PORT(
   input:  IN BIT_VECTOR(3 DOWNTO 0);
   output:  OUT BIT_VECTOR(3 DOWNTO 0);
   load:  IN BIT;
   clock:  IN BIT
   );
END reg4;

ARCHITECTURE reg4_behavior of reg4 IS
   begin
        process (clock)
                begin
                      IF (clock'event AND clock = '1') THEN
                         IF (load = '1') THEN
                            output <= input;
                         END IF;
                      END IF;
                end process;
   end reg4_behavior;
```

Figure 6.18: Register behavioral description in VHDL

```
ENTITY bilbo4 IS
   PORT(Input:  IN BIT_VECTOR(3 DOWNTO 0);
   Output:  OUT BIT_VECTOR(3 DOWNTO 0);
   mode:  IN BIT_VECTOR(2 DOWNTO 0);
   load, clock, ScanIn:  IN BIT;
   ScanOut:  OUT BIT);
END bilbo4;

ARCHITECTURE bilbo4_behavior of bilbo4 IS
   begin
   process (clock)
   VARIABLE temp :  BIT_VECTOR(3 DOWNTO 0);
   VARIABLE feedback :  BIT;
   VARIABLE i :  INTEGER;
     begin
       IF (clock'event AND clock = '1') THEN
          IF (mode(2 downto 1) = "00") THEN
             temp := "0000";
          ELSIF (load = '1' AND mode(2 downto 1) = "10") THEN
             temp := input;
          ELSIF (mode = "011") THEN --- Shift mode
             Scanout <= temp(0);
             FOR i IN 0 to Input'length-2 LOOP
                 temp(i) := temp(i+1);
             END LOOP;
             temp(Input'length-1) := ScanIn;
          ELSIF (mode = "110") THEN --- MISR mode
             feedback := Input(0) XOR Input(2);
             FOR i IN 0 to Input'length-2 LOOP
                 temp(i) := temp(i+1) XOR input(i);
             END LOOP;
             temp(Input'length-1) := feedback XOR input(input'length-1);
          ELSIF (mode = "010") THEN --- TPGR mode
             feedback := Input(0) XOR Input(2);
             FOR i IN 0 to Input'length-2 LOOP
                 temp(i) := temp(i+1);
             END LOOP;
             temp(input'length-1) := feedback;
          END IF;
          Output <= temp;
       END IF;
     END process;
   END bilbo4_behavior;
```

Figure 6.19: BILBO behavioral description in VHDL

```
ENTITY mux4_4to1 IS
   PORT(
   input1:  IN BIT_VECTOR(3 DOWNTO 0);
   input2:  IN BIT_VECTOR(3 DOWNTO 0);
   input3:  IN BIT_VECTOR(3 DOWNTO 0);
   input4:  IN BIT_VECTOR(3 DOWNTO 0);
   Sel:  IN BIT_VECTOR(1 DOWNTO 0);
   output:  OUT BIT_VECTOR(3 DOWNTO 0)
   );
END mux4_4to1;

ARCHITECTURE mux4_4to1_behavior OF mux4_4to1 IS
   begin
   process(input1,input2,input3,input4,Sel)
   begin
        CASE sel IS
             WHEN "00"=> output
                      <= input1;
             WHEN "01" => output
                       <= input2;
             WHEN "10" => output
                       <= input3;
             WHEN "11" => output
                       <= input4;
             END CASE;
        end process;
END mux4_4to1_behavior;
```

Figure 6.20: Four-to-one behavioral multiplexer description in VHDL

```
ENTITY alu4_73 IS
   PORT(
   operand1:  IN BIT_VECTOR(3 DOWNTO 0);
   operand2:  IN BIT_VECTOR(3 DOWNTO 0);
   Sel:  IN BIT_VECTOR(1 DOWNTO 0);
   result:  OUT BIT_VECTOR(3 DOWNTO 0)
   );
END alu4_73;

ARCHITECTURE alu4_73_behavior OF alu4_73 IS
   BEGIN
   PROCESS (operand1, operand2, Sel)
      VARIABLE temp:  BIT_VECTOR(7 DOWNTO 0);
         BEGIN
               CASE Sel IS
                     WHEN "00" =>
                              temp := operand1 * operand2;
                              result <= temp (3 DOWNTO 0);
                     WHEN "01" =>
                              result <= operand1 + operand2;
                     WHEN "10"=>
                             result <= operand1 - operand2;
                     WHEN OTHERS => NULL;
               END CASE;
         END PROCESS;
END alu4_73_behavior;
```

Figure 6.21: A complex ALU behavior in VHDL

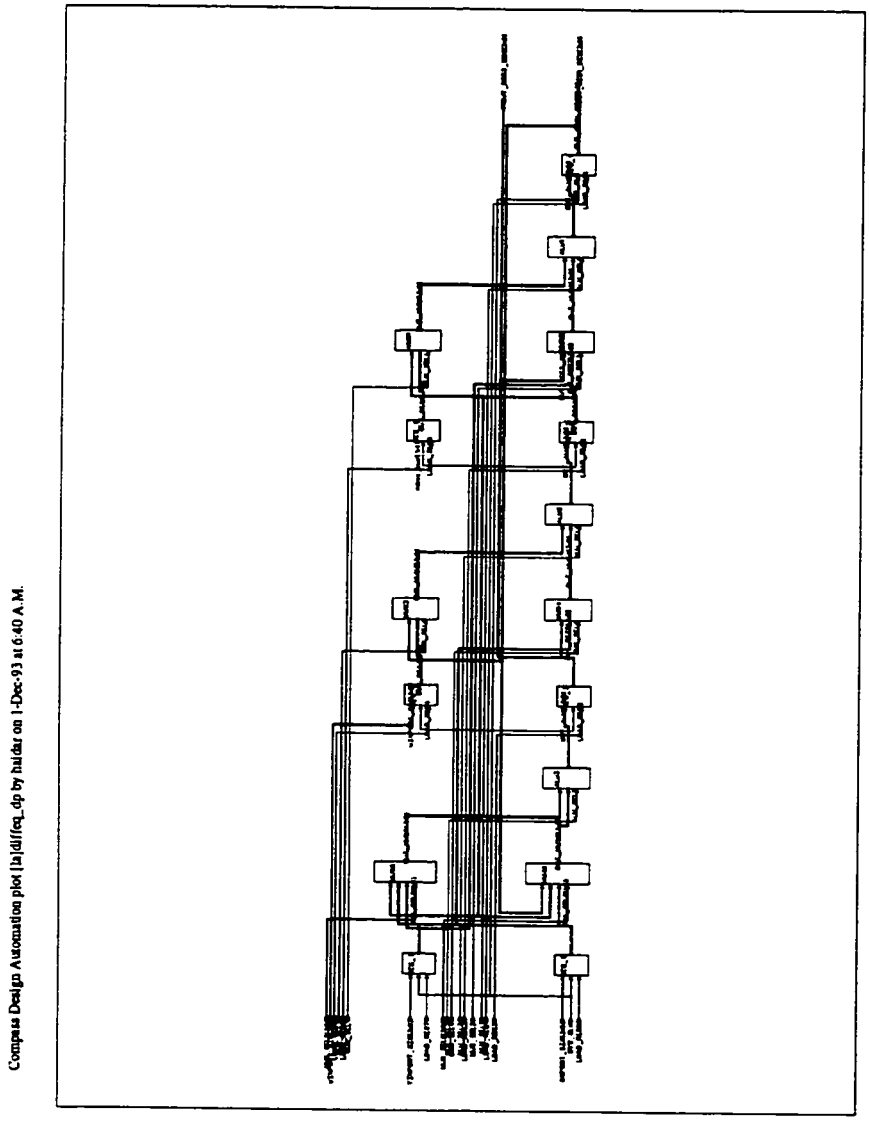


Figure 6.22: Data path structural view using COMPASS

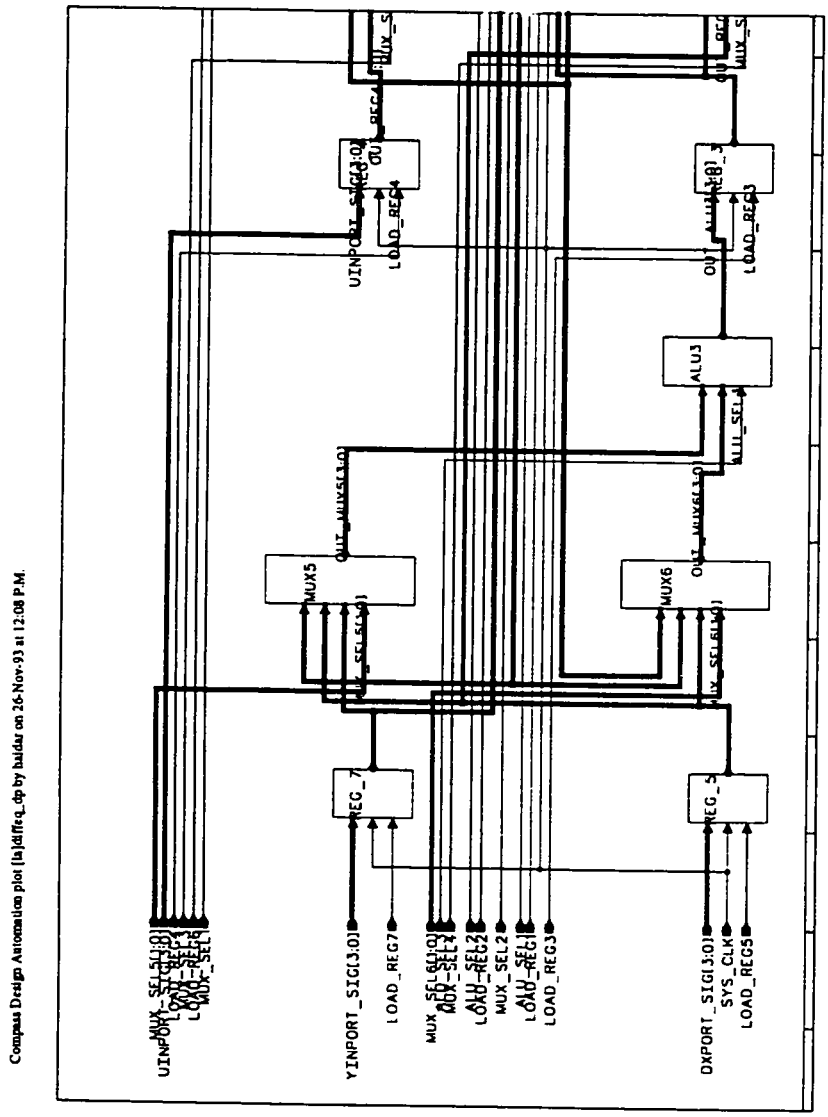


Figure 6.23: Partial Data path structural view using COMPASS

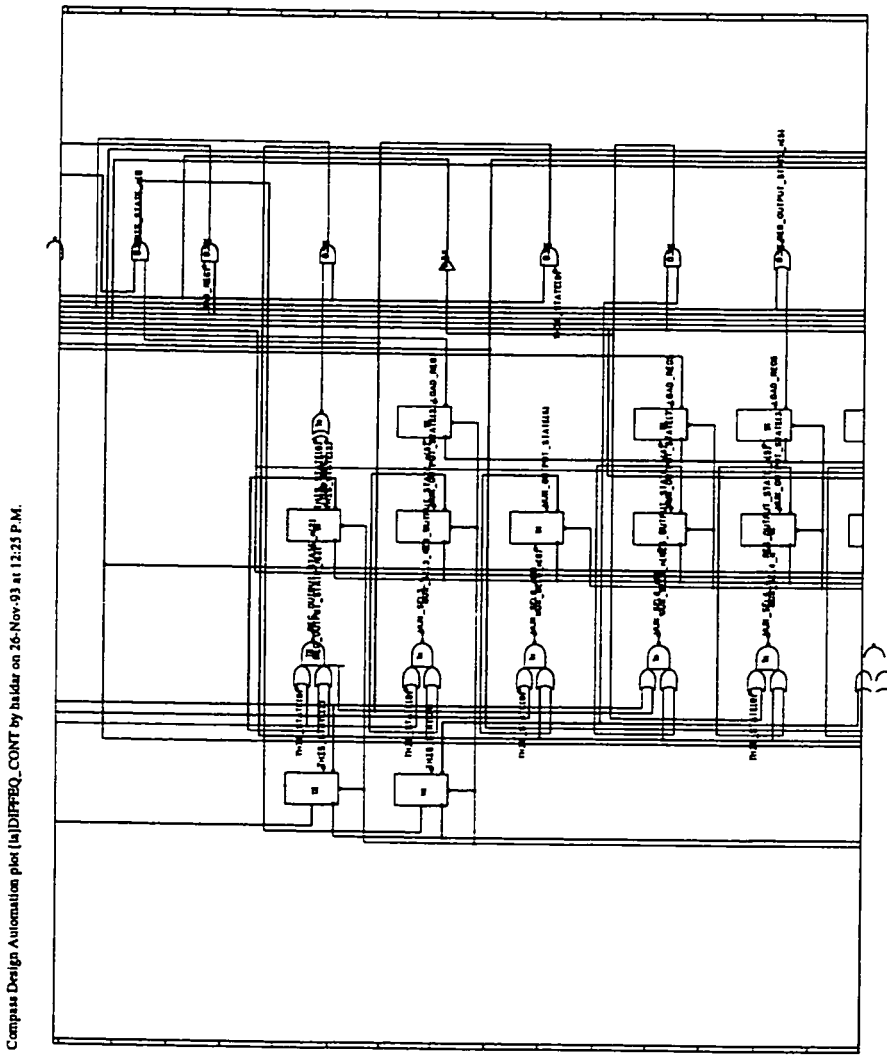


Figure 6.24: Partial controller view using COMPASS

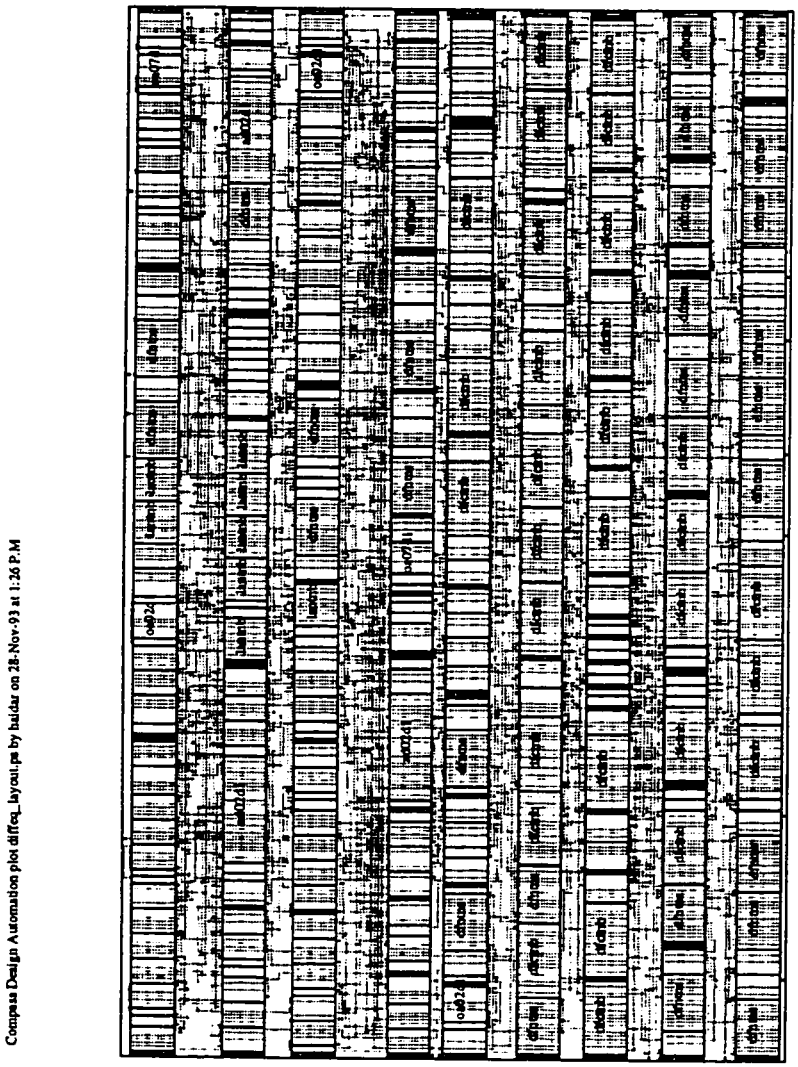


Figure 6.25: Chip final layout

Chapter 7

Conclusion and Future Work

In response to the industry need for tools to bridge the gap between design and test at the system level on one hand, and between high level and logic synthesis on the other hand, we have developed and implemented SYNTEST. In this chapter, we put our research into perspective and detail any future extensions.

7.1 Resource Allocation and Reallocation Techniques in High Level Synthesis with Testability Constraints

As we have pointed out in Chapter , the main question that we had to deal with was how to devise a methodology to incorporate synthesis and testability in a unified environment. In what follows, we summarize the results of this research in terms of methodology as well as in terms of implementation.

1. *A unified environment for synthesis with test*

   As a result of this research, we have completed the implementation of a prototype synthesis for test system. We have released two versions of the software, supported by documentation and user manuals. The current version of the system (version 2.0) is implemented in C and runs under Unix. Current work is underway to port the system into other platforms.

2. *A method for testable data path allocation*

   The testable data path allocator is the main driving engine in SYNTEST. The tool is based on the BIST methodology and is supported by design and test cost estimation tools. The tool also supports various aspects such as conditionals, pipelining, and controller cost estimation.

3. *A method for data path reallocation*

   The reallocation method we proposed is a novel technique which has become a necessity due to the rapid change in technology. The reallocation method was implemented but has not been incorporated in SYNTEST as of the date of this manuscript. Several designs were attempted and results are very promising.

4. *A link to lower level synthesis*

   A synthesis tool is as good as its practicality. In order to make the technology transfer process a smooth one, we have used VHDL along with commercial tools. The approach is useful for two reasons. The first one is due to the fact that we are using an automatic programming approach to make the transition from behavior to structure; thus, minimizing the risk of errors during such process. The other reason is that this allows the designer to get realistic designs based on commercial library vendors that conform to industry standards. This also allows for a realistic evaluation, verification and validation of the designs before fabrication.

7.2 Future Research

In the previous chapters we have discussed our research along with the resulting tools and system. In what follows we discuss possible future directions and extensions envisioned as:

1. With the shrinking of design rules and the decrease in cost for silicon area, performance has become more critical. It is true that, in general, by decreasing the silicon area, the chip performance is improved. However, a direct approach considering placement and floorplanning estimation during allocation could be a good alternative. This could be accomplished either by consulting a floorplanner during allocation or through back-annotation, using a commercial tool.

2. The reallocation phase can be modified to incorporate performance consideration during Phase I. The advantage over allocation would be the capability to consider a mixture of components that accomplish the same function. For example, we can consider various types of adders such as fast adders (big area) inserted in the critical path while slower ones (small area) inserted off the critical path.

3. Finally, data path reallocation could be considered in conjunction with testability. This would be done in two ways. The first by inserting components (registers, muxes) in a given data path in order to improve its fault coverage while keeping a tight balance with the design area and performance. The second approach would be by considering designs already testable (does not require insertion) and improving its fault coverage and performance through the modification of the registers test attributes.

Appendix A

Publications

- H. Harmanani, C. Papachristou, "An Improved Method for RTL Synthesis with Testability Trade-Offs,' in *Proceedings of the International Conference on Computer Aided Design,* November 1993.
- C. Papachristou, H. Harmanani, "A Method for RTL Synthesis with Testability Trade-Offs," *Proceedings of the TECHON*, September 1993.
- C. Papachristou, H. Harmanani and M. Nourani, "An Approach for Redesigning in Data Path Synthesis," in *Proceedings of the 30th Design Automation Conference*, June 1993.
- H. Harmanani, C. Papachristou, S. Chiu and M. Nourani, "SYNTEST: An Environment for System-Level Design for Test," in *Proceedings of the European Design Automation Conference*, September 1992.
- C. Papachristou, S. Chiu, H. Harmanani, "A Data Path Synthesis Method for Self-Testable Designs," *Proceedings of the 28th Design Automation Conference*, June 1991, pp. 378-384.
- C. Papachristou, S. Chiu, H. Harmanani,"SYNTEST: a method for high-level SYNthesis with self-TESTability," *Proceedings of the International Conference on Computer Design,* October 1991, pp. 458-462.
- C. Papachristou, S. Chiu, H. Harmanani, "Allocation Algorithms for Self-Testable Designs," *Poster Presentation, The European Design Automation Conference*, February 1991, Amsterdam, Holland.

Bibliography

[DAC90] "Testing Strategies for the 1990s," *Panel Discussion*, Design Automation Conference, 1990.

[AbBr85] M. Abadir, M. A. Breuer, "A Knowledge-Based System for Designing Testable VLSI Chips," *IEEE Design & Test*, pp. 56-68, August 1985.

[AbBF90] M. Abramovici, M. Breuer, A. Friedman, *Digital Systems Testing and Testable Designs*, Computer Science Press, 1990.

[Agra91] V. Agrawal, Plenary Speech, *International Conference on Computer Design*, October 1991.

[Avra91] L. Avra, "Allocation and Assignment in High-Level Synthesis for Self-Testable Data Paths," *Proceedings of the International Test Conference*, pp. 463-472, 1991.

[Been90] F. Beenker, R. Dekker, R. Stans, M. Van Der Star, "Implementing Macro Test in Silicon Compiler Design," IEEE Design and Test, April 1990, pp. 41-51.

[ChBu88] X. Chen and M. Bushnell, " A Module Area Estimator for VLSI Layout," in *Proc. 25th Design Automation Conference*, July 1988, pp. 54-59.

[ChAg90] K. Cheng, V. Agrawal,"Synthesis of Testable Finite State Machines," *ISCAS-90*, 1990.

[ChPa91a] S. Chiu, C. Papachristou, "A Design for Testability Scheme with Applications to Data Path Synthesis," *Proceedings of the 28th Design Automation Conference*, June 1991, pp. 271-277.

[ChPa91b] S. Chiu, C. Papachristou, "A Built Self-Test Approach for Minimizing Hardware Overhead," *Proceedings of the International Conference on Computer Design*, October 1991, pp. 282-285.

[Chri75] N. Christofides, *Graph Theory: An Algorithmic Approach*, Academic Press, New York, 1975.

[CaST91] R. Camposano, L. Saunders, R. Tabet, "VHDL as Input for High-Level Synthesis," *IEEE Design & Test of Computers*, pp. 43-49, March 1991.

[Catt89] F. Catthoor, J. Van Sas, L. Inze, H. De Man, "A Testing Strategy for Multiprocessor Architecture," IEEE Design and Test, April 1989, pp. 18-34.

[Drag89] M. Dragomrrecky, et al. "High-Level Graphical User Interface Management in the FACE Synthesis Environment," *Proceedings of the 26th Design Automation Conference*, June 1989, pp. 549-554.

[DeMi90] G. De Micheli, D. Ku, Frederic Mailhot, T. Tuong, "The Olypmpus Synthesis System for Digital Design," Technical Report, Stanford University.

[DeNe89] S. Devadas, R. Newton, "Algorithms For Hardware Allocation In Data Path Synthesis," *IEEE Transaction on CAD*, pp. 768-781, July 1989.

[EiWi77] E.B. Eichelberger, T.W. Williams, "A logic design structure for LSI testing," *Proceedings of the 14th Design Automation Conference*, pp. 462-468, June 1977.

[FuKa91] M. Fujita, T. Kakuda, " A Redesign Approach to High-Level Synthesis," The 1991 High-Level Synthesis Workshop.

[GaKu83] D. Gajski, R. Kuhn, Guest Editors' Introduction: New VLSI Tools, IEEE Computer, 6 (12):11-14, 12 1983.

Silicon Compilation, Addisson-Wesley, 1988.

[Gajs88] D. Gajski (editor), *Silicon Compilation,* Addisson-Wesley, 1988.

[Gajs92] D. Gajski, N. Dutt, A. Wu, S. Lin, *High-Level Synthesis: Introductrion to Chip and System Design*, Kluwer Academic Publishers, 1992.

[GaJo79] M. Garey, D. S. Johnson, *Computer and Intractability*, W. H. Freeman, 1979.

[GeEl88] C. Gebotys, M. Elmasri, "VLSI Design Synthesis with Testability," *Proceedings of the 25th Design Automation Conference*, June 1988, pp. 16-21.

[HaPa83] L. Hafer, A.C. Parker, "A Formal Method for the Specification, Analysis, and Design of Register- Transfer Level Digital Logic," *IEEE Transactions on Computer-Aided Design*, 1(1983), pp. 4-18.

[Harm91] H. Harmanani, *Testable Data Path Synthesis with Integer Linear Programming Allocation*, M.S. Thesis, Department of Computer Engineering and Science, Case Western Reserve University, 1991.

[HPCN92] H. Harmanani, C. Papachristou, S. Chiu and M. Nourani, "SYNTEST: An Environment for System-Level Design for Test," in *Proceedings of the European Design Automation Conference*, September 1992.

[HaPa93] H. Harmanani, C. Papachristou, "An Improved Method for RTL Synthesis with Testability Trade-Offs,' *Proceedings of the International Conference on Computer-Aided Design*, November 1993.

[HuPe87] C.L. Hudson, G.D. Peterson, "Parallel Self-Test With Pseudo-Random Test Patterns," *Proceedings of the International Test Conference*, pp. 954-971, Sept. 1987.

[John89] B. Johnson, *Design and Analysis of Fault Tolerant Digital Systems*, Addison-Wesley, 1989.

[Jain89] 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.

[KoMZ79] B. Koenemann, J. Mucha and G. Zwiehoff, "Built-In Logic Block Observation Techniques," *Proceedings of the International Test Conference* , pp. 37-41, Oct. 1979.

[Knap89] D. Knapp, "Manual Rescheduling and Incremental Repair of Register-Level Datapaths," *Proceedings of the ICCAD*, pp. 58-61, 1989.

[Knap91] D. Knapp, "Datapath Optimization Using Feedback," *Proceedings of the EDAC*, pp. 129-134, 1991.

[KrNe92] G. Krishnamoorthy, J. Nestor, "Data Path Allocation using an Extended Binding Model," *Proceedings of the 29th Design Automation Conference*, pp. 279-284, 1992.

[KuPa87] F. Kurdahi, A. Parker, "REAL: A Program for Register Allocation," *Proceedings of the 24th Design Automation Conference*, pp. 210-215, 1987.

[KuPa89] F. J. Kurdahi and A. C. Parker, " Techniques for Area Estimation of VLSI Layouts," in *IEEE Trans. on CAD*, January 1989, pp. 81-92.

[LaRu71] B. Landman and R. Russo, " On a pin versus block relationship for partitioning of logic graphs," in *IEEE Trans. Comput.*, vol. C-20, Dec. 1971, pp. 1469-1478.

[LeHL89] J. Lee, Y. Hsu, Y. Lin, "A New Integer Linear Programming Formulation For the Scheduling Problem in Data Path Synthesis," *Proceedings of the International Conference on Computer Aided Design*, 1989, pp. 20-23

[Lee88] T. Lee, W. Wolf, N. Jha, J. Acken, "Behavioral Synthesis for Easy Testability in Data Path Allocation," *International Conference on Computer Design*, 1992, pp. 29-32.

[LeSp90] J. Leenstra, L. Spaanenburg, "Hierarchical Test Assembly for Macro Based VLSI Design," *Proceedings of the International Test Conference*, pp. 520-529, 1990.

[LiGa88] J. Lis, D. Gajski, "Synthesis from VHDL," *International Conference on Computer Design*, 1988, pp. 378-381

[Marw86] P. Marwedel, "A New Synthesis Algorithm for the MIMOLA Software System," *Proceedings of the 23rd Design Automation Conference*, June 1986, pp. 271-277.

[McCl85] E.J. McCluskey, "Built-In Self-Test Techniques," *IEEE Design and Test*, Vol. 2, No. 2, pp. 21-28, April 1985.

[McCl86] E.J. McCluskey,*Logic Design Principles*, Prentice-Hall, Englewood Cliffs, N.J., 1986.

[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.

[MuSJ92] A. Majumdar, K. Saluja, R. Jain, "Incorporating Testability Considerations in High-Level Synthesis," *Proceedings of Fault Tolerants Computing*, pp. 272-279, 1992.

[MuHa88] B. Murry, J. Hayes, "Hierarchical Test Generation Using Precomputed Tests for Modules," *Proceedings of the International Test Conference*, pp. 221-229, 1988.

[NCR89] *The NCR ASIC Data Book*, 1989.

[NoPa92] M. Nourani, C. Papachristou, "Move Frame Scheduling and Mixed Scheduling-Allocation for the Automated Synthesis of Digital Systems," *Proceedings 29th Design Automation Conference*, June 1992.

[NoPa92] M. Nourani and C. A. Papachristou, " A Deterministic Layout Estimation Algorithm for RTL Datapaths," *Proceedings of the 30th Design Automation Conference*, June 1993.

[OrGa86] A. Orailoglu, D. Gajski, "Flow Graph Representation," *Proceedings of the 23rd Design Automation Conference*, June 1986, pp. 503-509.

[Pang88] B.M. Pangrle, "Splicer: A Heuristic Approach to Connectivity Binding," *Proceedings of the 25th Design Automation Conference*, June 1988, pp. 536-541.

[PaKo90] C. Papachristou, H. Konuk, "A Linear Program Driven Scheduling and Allocation Method Followed by an Interconnect Optimization Algorithm," *Proceedings of the 27th Design Automation Conference*, June 1990, pp. 77-83.

[PaCH91] C. Papachristou, S. Chiu, H. Harmanani, "A Data Path Synthesis Method for Self-Testable Designs," *Proceedings of the 28th Design Automation Conference*, June 1991, pp. 378-384.

[PaCH91b] C. Papachristou, S. Chiu, H. Harmanani,"SYNTEST: a method for high-level SYNthesis with self-TESTability," *Proceedings of the International Conference on Computer Design*, October 1991, pp. 458-462.

[PaPM86] A.C. Parker, J. Pizarro, M. Mlinar, "MAHA: A Program for Data Path Synthesis," *Proc. 23rd Design Automation Conf.*, pp. 461 - 466, June 1986.

[PaKn87] P. Paulin, J.P. Knight, "Forced-directed scheduling in automatic data path synthesis," *Proceedings of the 24th Design Automation Conference*, pp. 195-202, June 1987.

[PaKn89] P. Paulin, J.P. Knight, "Forced-Directed Scheduling for the Behavioral Synthesis of ASIC's", *IEEE Transactions on Computer Aided Design*, Vol. 8, pp. 661-679, June 1989.

[Raba88] J. Rabaey, H. DeMan, J. Vanhoof, F. Cathoor, "Cathedral II: A Synthesis System for Multiprocessor DSP," *In Silicon Compilation*, pp. 311-360, 1988.

[Sas91] J. Sas, F. Catthoor, P. Vandeput, F. Rossaert, H. De Man, "Automated Test pattern Generation for the Cathedral-II/2nd Architectural Synthesis Environment," *Proceedings of The EDAC,* pp. 208-213, 1991.

[SuKi90] C. Su, C. Kime, "Multiple Path Sensitization for Hierarchical Circuit Testing," *Proceedings of the International Test Conference,* pp. 152-161, 1990.

[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.

[TsHs92] F. Tsai, C. Hsu, "STAR: An Automatic Data Path Allocator," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Volume 11, No. 9, pp. 1053-1064, September 1992.

[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.

[Ueda85] K. Ueda, H. Kitazawa et al., " Chip Floor Plan for Hierarchical VLSI Layout Design," in *IEEE Trans. on CAD*, January 1985, pp. 12-22.

[WePa91] J. Weng and A. C. Parker, "3-D Scheduling: High-Level Synthesis with Floorplanning," *Proceedings of the DAC*, 1992, pp. 668-673.

[WiPa83] T. Williams, K. Parker, "Design For Testability — A Survey", *Proceedings of The IEEE*, Volume 71, Number 1, January 83, pp. 98-112.

Index

observability, 13
strictly structurally testable, 43

ADAM, 20, 23
Advanced Design AutoMation System, 20
ALAP, 31
ALLIANCE, 29
ASAP, 31

BILBO, 26, 40, 41, 64, 100
BIST, 26
Built-In Logic Block Observer, 40

CATREE, 21
CBILBO, 41, 63
COMPASS, 29, 30
COMPASS Chip Assistant, 30
COMPASS Logic Assistant, 30
concurrent built-in logic-block observation, 41
conditional, 57
Controllability, 13
controller, 30, 56
critical path, 37

Data path allocation, 9
delay estimator, 34
description, 3
Design Automation Assistant, 19
Design For Testability, 12
DFG generator, 30
DFT, 12, 23, 40

FACET, 62
Facet, 19
FDS, 16
forced directed scheduling, 16
freedom, 20

GDSII, 30

HAL, 16, 62
High-level synthesis, 3, 5, 6

IDP, 50
integer linear programming, 32

layout estimator, 34
Left Edge Algorithm, 21
LFSR, 40, 100
life span, 46
life time, 46
Linear Feedback Shift Register, 40
Logic level synthesis, 3
loosely structurally testable, 43

MAG, 50
MAHA, 20

MFS, 31
MISR, 32, 40
Modified Automatic Hardware Allocator, 20
module allocation graph, 32, 48
Move Frame Scheduling, 31
Multiple Input Signature Registers, 40

non-structurally testable designs, 43

Output Randomness measure, 34

Pipelining, 56

RALLOC, 22, 62
random patterns, 40
REAL, 21
reallocation, 72
redesign, 71

Scheduling, 6
self-adjacency, 40
sequential depth, 43, 45, 59
sequential path, 39
silicon compiler, 2
specification, 3
SYNTEST, 26
synthesis, 2
System level synthesis, 2

TDES, 23
technology library, 48
Technology mapping, 3
Test Cost Estimation Tool, 33
Test Pattern Generation Registers, 40
testable allocator, 32
Testable data path allocation, 45
Testable Design Expert System, 23
Testable Functional Block, 26, 42
TFB, 26
TPGR, 32, 40
tradeoffs, 5, 13, 61
Transparency measure, 33

validation, 3
verification, 3
VHDL, 93
VHDL ASIC Synthesizer, 30
VHDL Synthesis System, 18
VHSIC (Very High Speed IC) Hardware Description Language, 93
VLSI, 23

wave filter, 63

X-Window, 34