Copyright © 1988, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

TECHNIQUES FOR OPTIMIZATION-BASED SYNTHESIS OF DIGITAL SYSTEMS

by

Srinivas Devadas

Memorandum No. UCB/ERL M88/54

12 August 1988

TECHNIQUES FOR OPTIMIZATION-BASED SYNTHESIS OF DIGITAL SYSTEMS

Srinivas Devadas

ABSTRACT

A prototype framework which can automatically synthesize integrated circuits from behavioral descriptions has been developed. The system is characterized by the use of optimization tools at every level of the synthesis process to enhance the quality of the designs. The system proceeds through the various steps of hardware allocation, control specification, state assignment and combinational logic synthesis to produce optimized gate-level specifications. Formal verification and test strategies have been incorporated into the system.

Given a data flow specification, simulated-annealing-based algorithms which find a globally optimal placement of micro-instructions, thus producing an optimal datapath configuration, have been developed. Algorithms for state assignment of finite state machines targeted toward multi-level logic implementations have been proposed. A connection between multi-level and multiple-valued Boolean minimization has been established. Algorithms for Boolean decomposition, based on multiple-valued Boolean minimization, have been developed which decompose a PLA into a set of smaller interconnected PLAs such that the overall area of the resulting logic network is minimized. An efficient algorithm has been proposed for the verification of the equivalence of two sequential circuit descriptions at the same or differing levels of abstraction, namely at the register-transfer (RT) level, the State Transition Graph level or the logic level. An efficient deterministic sequential test pattern generation algorithm, effective for mid-sized sequential circuits, has been developed. This algorithm can be used in conjunction with an Incomplete Scan Design approach to test generation for large sequential circuits. Finally, the relationship between sequential logic synthesis has been explored and a connection between state assignment and the testability of a sequential machine has been established.

Dedicated to Lord Vishnu

ACKNOWLEDGEMENTS

I am indebted to my advisor Prof. Richard Newton for his friendship, guidance and unfailing support during my years at Berkeley. But for his enthusiasm and the inspiration he provided, much of this work would not have been possible.

I have greatly benefited due to my association and interaction with Professors Robert Brayton, Donald Pederson and Alberto Sangiovanni-Vincentelli. I thank them for their continuing interest in my research projects. I have also benefited from many discussions with Prof. Carlo Séquin of the Computer Science Department. Prof. Alberto Grünbaum of the Math Dept. was kind enough to serve on my thesis committee and read my dissertation.

I have had the good fortune of meeting many wonderful people at Berkeley. In particular, I would like to thank Tony Ma, my long time friend, research associate and office-mate, for things too numerous to enumerate.

Jeffrey Burns, George Jacob, Kartikeya Mayaram and Fabio Romeo have been, and will always be, good friends. I am especially grateful to Jeff and Karti for providing support during some difficult times.

I've had lots of great tennis games (the ones I won) with Bob Brayton, George Jacob, Theo Kelessoglou, Vedat Milor, Gordon Jacobs and Tom Quarles. I hope my constant aiming at Tom when he is near the net will not deter him from playing with me in the future !

Suresh Krishna, Kinson Ho, Su Tang and I have spent many fun-filled evenings in San Francisco and elsewhere. We remain good friends despite their constantly needling me about my car, my driving, my radio station, my choice of movies and...

Watson Chan, Randy Cieslak, Wayne Christopher, Abhijit Ghosh, Pramod Jain, Chuck Kring, Vijay Madisetti, Rick Spickelmier, Ruey-sing Wei and Albert Wang are a few of the great guys I've been lucky enough to meet over the years. Randy and Wayne, frequent dinner companions, introduced me to several nice restaurants around town. I thank Watson for putting up with me in his and Tony's apartment during my last two months at Berkeley.

I express my gratitude for the financial support that made this research possible. I acknowledge support from the Semiconductor Research Corporation in the summer of 1985 and the Digital Equipment Corporation in the summer of 1986, as well as the VAX-based computing environment I used to develop my experimental systems. I was supported in part by the Defense Advanced Research Projects Agency under contract N00039-86-R-0365 and in part by a grant from AT&T Bell Laboratories. While the work on testing described in this dissertation is not exactly what was intended when the AT&T-supported phase of the project began, their encouragement and support is gratefully acknowledged.

TABLE OF CONTENTS

CHAPTER 1

Introduction

A considerable research effort has focused on automating the integrated circuit (IC) design process over the past few years (e.g. [1] [2] [3] [4]). A variety of Computer-Aided Design (CAD) tools for the logic [5] [6] and physical design [7] of integrated circuits have been developed. It is clear that an integrated set of computer design aids coupled with an unified approach to data management is essential for VLSI design. To this end, research has focused on IC *synthesis systems* [8] i.e. systems which can automatically generate mask-level layout of integrated circuit chips from high-level, programming-language-like specifications. In this chapter, the necessity for, and the characteristics and requirements of, synthesis systems are presented and previous work in this area is reviewed.

1.1 Need for Synthesis Systems

There are integrated circuit applications, such as speech synthesis, bandwidth compression and recognition, modems and digital data transmission and digital control systems, where in order to achieve a complete and efficient integration of the system functions, it is necessary to design special-purpose chips which are to perform only a single task. Unfortunately, even though this can yield enormous savings in the size of the system and its power consumption, the design cost in both time and money can often be prohibitive. In addition, there are not many designers who have the expertise to design these system chips, which often require knowledge in both analog and digital circuit design as well as digital signal processing and computer architecture.

Semi-custom design techniques such as gate-arrays [9] and standard cells [10] offer an environment where faster turnaround can be guaranteed by design tools which can place and route complex functions in a short time. However, the task of logic design, i.e. specifying the gates and the interconnections which implement a certain behavior of the system, may still consume a large amount of design time.

Application-Specific Integrated Circuit (ASIC) *synthesis systems* have been proposed as a solution to the problem of automatic integrated circuit generation from a high-level behavioral or algorithmic description of the functions of the system to be implemented (e.g. [11] [12] [13] [8]). The demand for and use of ASIC synthesis systems is increasing at a rapid rate in the IC industry today.

1.2 The Level of the Input Description

The level of the input description to these synthesis systems varies – some systems require a relatively low register-transfer level description of the design, where information about required hardware resources (e.g. buses, arithmetic units) is explicitly available. For example, if the design in question is a pipelined datapath of a computer, the pipeline schedule has to be explicit in the input description. The overall structure of the circuit, i.e. the interconnection of the various modules can thus be inferred in a straightforward way from the description. Several *silicon compilers* have been developed (e.g. MACPITTS [13]), which automatically generate mask layout beginning from a register-transfer (RT) description of a datapath or a finite state machine (FSM) controller. These silicon compilers typically proceed through phases involving logic extraction, optimization and layout synthesis.

Behavioral synthesis systems, like the CMU-DA system [14] [1], begin from higher level algorithmic specifications where the behavior rather than the structure of the design is specified. The task of a behavioral synthesis system is thus more complicated than the typical silicon compiler task – a phase which *allocates* hardware resources and decides their spatial and temporal delineation (Section 1.3.1), thus specifying the structure of the design, precedes

the logic design and physical design phases. Complete synthesis systems also incorporate vitally important verification and test strategies. For example, synthesis and verification share a common methodology in the USC Design Automation System [15] [16].

It is important to note that the same language can be used in the context of specifying an input to a synthesis system at either of these two levels. For example, languages like ISPS [17] and BDS [18] are used for the functional specification (behavior) of a design or the register-transfer level specification of a design. Constructs in these languages are interpreted differently depending on the level of specification – e.g. variables in the language may be interpreted as existing registers or *values* which are to be stored and allocated in registers.

1.3 Inputs to Behavioral Synthesis Systems

Given that the input description specifies the behavior of the IC, different kinds of inputs to a synthesis system are possible. Possible inputs are purely architectural descriptions of the instruction sets of a general-purpose computers (with no information about the implementation). Another possibility would be software programs describing algorithms which are to be implemented in hardware. In either of these two cases, the task confronting the synthesis system is the same – datapaths executing these descriptions optimally, with associated control, have to be synthesized. However, the complexity in performing these tasks varies significantly, especially in the hardware resource allocation phase.

General-purpose computer descriptions are typically very detailed and and involve extensive bit manipulations. The instruction fetch-decode-execute loop takes only a few cycles and is usually constrained to be highly sequential. A lot of attention to detail must be paid in synthesizing from these descriptions and they tend to be uninteresting from a global optimization point of view. The decisions that can be taken during hardware allocation are tightly constrained by the input description – e.g. given the instruction set of a I-8080 microprocessor, there is virtually no parallelism between arithmetic operations and a second ALU in a hardware implementation would be useless. Control logic, on the other hand, is quite

complicated in large general-purpose computers.

One can also synthesize specialized processors which are designed to execute a given software description. For example, given a string hash table procedure or a MOSFET model evaluation routine, the goal would be to synthesize a datapath which would execute this program optimally. These descriptions are typically less detailed, less structured and not as constrained as compared to general-purpose sequential computer descriptions. Also, the associated control for these special-purpose machines is usually quite simple. Since parallelism in these programs is usually not explicitly specified, it must be extracted. A lot of potential for global optimization exists in synthesizing from these specifications – the allocation step (Section 1.3.1) involves many decisions and trade-offs. For example, the datapath executing a typical MOSFET model evaluation routine can use one to four ALUs. Depending on the execution speed and chip area constraints the optimal number of ALUs can be found during the resource allocation process.

1.3.1 Constraint Specification

Aside from the behavioral specification, the synthesis system is generally given constraints by the user which the final circuit implementation must satisfy. These constraints are typically constraints on the chip area (physical constraints) or on execution speed (delay constraints) of the resulting datapath, but may involve more complex considerations.

Depending on the needs of the user, the system may receive constraints of varying degrees and the effect of these constraints may be felt in different phases of the synthesis process. For example, a specific constraint on the number of ALUs or registers that the datapath can use affects the hardware allocation phase. Broader constraints, such as a constraint on the final chip area or a constraint on the clock period, are passed downward through all phases of design.

The specification of these constraints has traditionally been separated from the behavioral description. Constraints are typically tacked onto descriptions just before synthesis

and are rarely formalized. A unified framework allowing constraint-driven synthesis (propagating constraints across all synthesis tools) necessitates the development of a language supporting constraint specification across different levels of abstraction, namely behavioral, register-transfer and logic levels. Research, focusing on the development of such a language, is currently underway in various places, including Berkeley.

1.4 Phases in Behavioral Synthesis Systems

1.4.1 Hardware Allocation

The hardware allocation phase in behavioral synthesis, first described in [14], generates a datapath which can implement all the data transfers required by the original specification. This step involves many decisions. For example, in a bus-style design, decisions involving both the number and the interconnection of buses, arithmetic units and registers have to be made. In the synthesis of pipelined computer datapaths, pipeline schedules satisfying required execution speed constraints must be found. A pipeline synthesis procedure was first published in [19].

A wide variety of tradeoffs between execution speed and hardware resource cost of the synthesized datapath have to be explored in the allocation phase. Serial and parallel implementations of input data flow descriptions can result in vastly different datapath configurations. For example, a datapath, A, might be twice as fast as datapath B, but it might occupy three times the area.

1.4.2 Control Synthesis

After the hardware resource decisions have been taken and a datapath which implements the required data transfers has been synthesized, the associated control which, in conjunction with the datapath can execute the behavioral specification must be synthesized. Control can take the form of micro-code to be stored in ROM or RAM, a sequencer, or a FSM controller [20].

The specification of the control logic is easily derived given the original specification and the synthesized datapath. In some synthesis systems (e.g. [14]), micro-instruction scheduling decisions may be made at this stage, i.e. after datapath synthesis/hardware allocation. In other allocators like EMUCS [21], control specification is regarded as a by-product of data path synthesis and no major decisions are taken.

Optimization of control logic, be it in the form of a micro-coded ROM or PLA-based FSM, is a critical and difficult task. Micro-code compaction algorithms (e.g. [22]), logic optimization (see [23] for references), state assignment (e.g. [24] [25] [26]), input and output encoding algorithms (e.g. [26] [27] [28]) are necessary in synthesis systems for optimal control synthesis.

1.4.3 Verification

It is important in this kind of environment to be able to verify that the optimization tools have not introduced any design errors during the synthesis process. The goal would be to formally verify that the generated layout implements the behavioral specification and satisfies the imposed constraints.

After the hardware allocation phase, logic verification tools (e.g. [29] [30]) can be used to verify equivalence of machine descriptions down to the gate/flip-flop level. This problem is NP-complete but a few algorithms have been shown to be practical even for large designs [29] [31] [32] [33]. Circuit and function extraction tools can be used to re-extract these gate/flip-flop descriptions from the synthesized layout to feed the logic verification tools. Timing

verifiers (e.g. [34]) can check that the constraints on the delay of the chip have been satisfied.

1.4.4 Testing

After chip fabrication, a test strategy is required to ascertain functionality correctness. Testing can be a difficult task especially for unstructured sequential designs – combinational designs are easier to test. The testing task can be alleviated by using built in self-test (BIST) [35] techniques or using a constrained design style like Scan Design [36]. BIST involves adding extra logic to the various modules, in an effort to make the design easily testable. Scan Design, which has been widely adopted, makes all the sequential elements observable and controllable from the outside and transforms the sequential testing problem into a combinational logic testing problem.

Unfortunately, both these procedures have a substantial area penalty associated with them – a design using BIST or Scan Design may be 10-20% larger than the same design not using either [37]. Efficient test generation algorithms for sequential circuits are thus very attractive. Some algorithms have been proposed for sequential test generation in the past (e.g. [38] [39]).

1.5 Related Work

Several synthesis systems operating from behavioral specifications have been developed in recent years. The CMU-DA [14] [1], Arsenic [40], Flamel [41] [42] and the USC Design automation project [15] [16] are typical examples and are reviewed here.

1.5.1 The CMU-DA System

One of the earliest behavioral synthesis systems was the CMU-DA system [14], which began from the CMU RT-CAD project [43]. This system has evolved over the years and incorporates many computer-aided design tools for synthesizing circuits from high-level specifications [1] [21] [44] [45] [46] [47]. The basic design methodology underlying the CMU-DA system is shown in Figure 1.1.

The design methodology is hierarchical. Complexity is handled by describing a design at higher levels of abstraction which eliminates unnecessary detail, particularly during early phases of a design. In addition to taking a hierarchical approach, this methodology employs the concept of a *design space*. Given a set of metrics which characterize the quality of a design, the design space is defined as a Euclidean space whose coordinate axes correspond to the metrics. Alternative designs are represented by points in the design space. Through the study of a design space, the designer can efficiently direct his use of the design aids to produce better designs.

The design process begins with a behavioral specification of the digital system of to be designed in a hardware description language ISPS [48] [17]. Transformation heuristics similar to those used in compiler design are used to optimize the original description.

The next step in the design process is synthesizing from the behavioral description a structure in terms of physical modules which will implement the required behavior. This step is partitioned into datapath and control synthesis.

Datapath synthesis is further separated into an allocation and a module binding step. Many datapath allocators have been developed, employing various algorithms (e.g. [49] [50] [51] [52] [21] [46]). for generating the register-transfer structure. Different *design styles* can be employed, e.g. distributed, bus and pipeline style.

Control synthesis involves *control allocation* which produces an control engine evoking the datapath devices in an order consistent with the behavioral description and *module binding* which results in an implementation of the control engine which can be a ROM or a PLA.

While the ISP language is useful for describing the desired behavior of a system, such a description implies a certain data flow, through the use of assignment constraints, and a certain control flow, through the use of loops, conditionals and procedures. These flows can impact the resultant cost/performance characteristics of the design. In order to remove the bias which is inherent in an ISP description, the ISP description is transformed into an

alternative representation called the *Value Trace* or *VT* [53] [54]. A Value Trace is a connected graph made up of a series of blocks called VT-bodies [54]. Each VT-body represents a control environment in the ISP description which has a single entry point. The datapath and control allocators use the VT as their input rather than the ISP description.

This system operates from a module data base which is created using logic and layout optimization tools. Verification and test strategies are currently not incorporated into the system – however some of the optimizing transformations used in the system have been proved

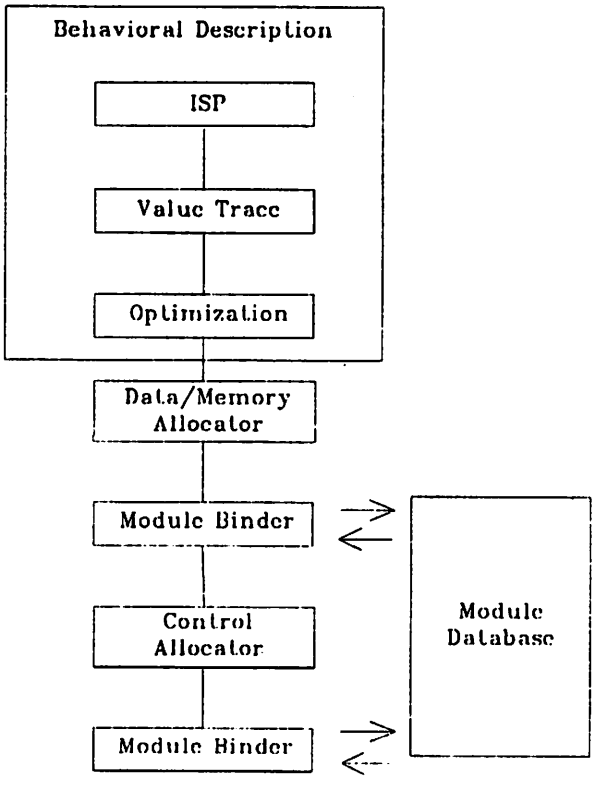


Fig. 1.1 CMU-DA design methodology

to preserve behavioral equivalence [55].

1.5.2 Arsenic

Synthesis in Arsenic, was originally developed at the University of Illinois [40] and continued at the University of California, Irvine [56], uses three steps across four levels of design, namely, algorithmic, register-transfer, abstract-cell and layout levels.

Arsenic begins synthesis with control step allocation, dividing the input specification into micro-instructions (MIs) at first, tentatively, so as to obtain maximum parallelism. The allocation of hardware is done for one MI at a time simultaneously with control step allocation.

The model of the control unit consists of a ROM and a sequencer.

1.5.3 Flamel

Flamel [41] [42] is a high-level hardware compiler which produces a hardware implementation of a given program minimizing the execution time of the implementation while meeting a user supplied constraint on the area of the hardware implementation. An overall picture of its operation is shown in Figure 1.2.

Flamel produces a datapath and a finite state machine controller. targeted for a bit-slice architecture. It performs a set of local block-level transforms on the input description in order to obtain maximum parallelism while meeting a resource (rather than a cost) bound. Depending on the user supplied area constraint, different kinds of time/area tradeoffs can be achieved.

1.5.4 The USC Design Automation Project

ADAM is a design automation system developed at USC, first described in [16], incorporating custom layout tools, and expert system for the design of testable circuits and a knowledge-based expert synthesis subsystem.

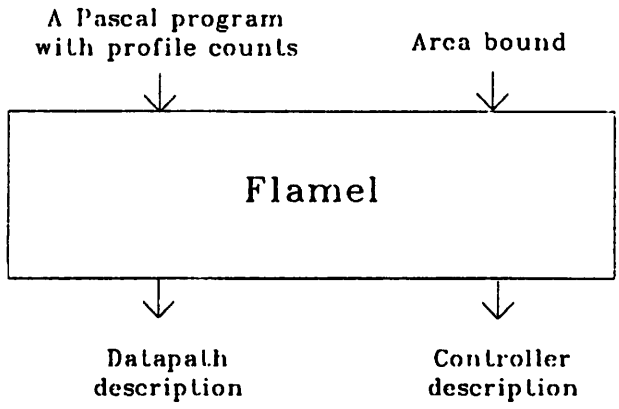


Fig. 1.2 Flamel's operation

The custom layout tools include a CMOS silicon compiler. The synthesis subsystem incorporates a clocking scheme and pipeline synthesizer [57] [19] and a datapath synthesis program [58].

Synthesis and verification share a common methodology in this system [15]. Verification is performed by verifying parts of the design which have been specified and synthesizing the missing elements.

1.6 Summary

In this chapter, provided an introduction to the problem of synthesizing integrated circuit chips from behavioral descriptions has been provided and some previous work in this area has been reviewed. In the next chapter, an overview of the behavioral synthesis system developed as part of this research will be presented. In the following chapters, a detailed description of the algorithms used in each phase of synthesis as well as the verification and test strategies incorporated into the synthesis system will be given.

CHAPTER 2

Overview

The behavioral synthesis system described in this chapter is the focus of this dissertation. A prototype framework which can synthesize automatically integrated circuits from behavioral descriptions has been developed and is illustrated in Figure 2.1. A number of tools for datapath and control synthesis have been developed – they have been incorporated into this system. Efficient verification algorithms have been developed to verify across the optimization tools in the synthesis pipeline. A strategy for testing the synthesized circuit has been developed. The relationship between logic synthesis and testability has been explored and a synthesis procedure to ensure fully testable sequential machines has been developed.

At this time, this system can be used for the synthesis of general-purpose computers, given instruction set specifications, or for the synthesis of specialized processors executing given algorithmic descriptions. Test cases described in this dissertation include a MOSFET model evaluation co-processor for a hardware simulation engine and digital signal processors. The system is characterized by the use of optimization tools at every level of the synthesis process to enhance the quality of the designs. These tools give the designer the ability to explore the complex tradeoffs in the design space and can be used to synthesize in different design styles. Rather than using a fixed library and incorporating module binding steps in synthesis, all logic blocks (ALUs, random logic and control) are assumed custom-designed using existing logic synthesis tools [6] [23] and existing layout generators [59] [60] [61] [62]. These blocks would then be placed and routed using a macro-cell layout system [63].

The input description, the tools developed for the datapath and control synthesis phases and some of the custom layout tools that have been used to complete the prototype system are

described below. The verification and test strategies that have been developed and incor-

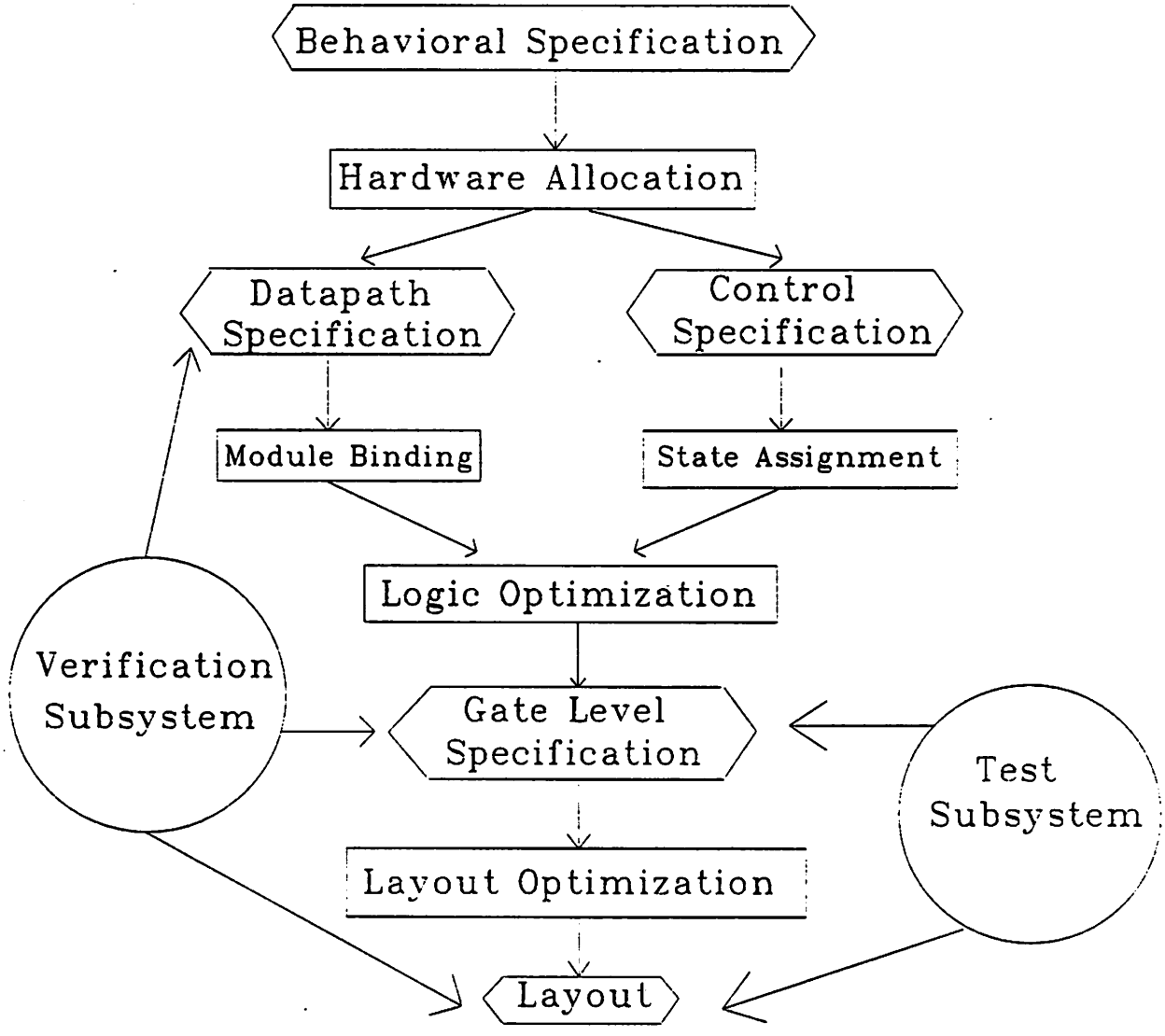


Fig. 2.1 Flowchart of the Behavioral Synthesis System

porated into the system are also described,

2.1 Input Description

General-purpose computers have traditionally been specified in hardware description languages like AHPL [64], DDL [65] ISPS [17] and BDS [18], which are specifically designed for this purpose. More recently, a language called VHDL has become popular and appears to be an important new direction [66]. On the other hand, tasks like string hash table lookups are very easily written in programming languages like C [67] or PASCAL [68]. The input description to the system described here can be in either of these two forms – a BDS description of a computer or a C program implementing an algorithm.

This input description is translated into a intermediate textual description which consists of two parts. The first part is a data-flow description which specifies all the required data transfers between the program values. Some of these data transfers may be mutually exclusive due to conditional clauses in the input description. These relationships are specified in the data-flow description, but the control signals associated with these relationships are not. The second part of the description, which represents the control function, has this information. The partitioning of the description into separate control and data-flow is a common approach to the problem. The CMU-DA system was one of the first to take such an approach [14].

The translation to the intermediate description incorporates some optimizing transformations like dead code elimination and loop unwinding.

2.2 Datapath Synthesis

Given the intermediate description which specifies all the data transfers required, all precedence constraints between the various operations are extracted. Parallelism and sequentiality

is explicitly introduced into the description following this analysis.

2.2.1 Hardware Allocation

The hardware allocation step in datapath synthesis can be formulated as *a two-dimensional placement problem of micro-instructions in space and time.* The problem solved is to synthesize a datapath corresponding to the input data-flow specification such that a given, arbitrary function, $f(T, C)$ of execution time T and hardware cost C, is minimized. The hardware costs are the sum total of the costs associated with registers, arithmetic units, buses and links in the datapath, based on required layout areas for placement and wiring. A given placement of micro-instructions corresponds to a unique datapath with a certain hardware cost and execution speed. Optimal conditional resource sharing is achieved by solving a constrained placement problem where disjoint instructions are allowed to occupy the same spatial and temporal location. Given a data-flow specification, simulated-annealing-based [69] algorithms have been developed, which find a globally optimal placement of micro-instructions, thus producing an optimal datapath configuration [70] [71].

Tradeoffs between execution speed and hardware cost of resulting datapaths are achieved by specifying different cost functions to the synthesis program. Thus, a variety of datapaths can be synthesized for any given input, with different time/area tradeoffs.

There are three main differences between this approach and others taken in the past [21] [58] [72] [41]. First, all the allocation subproblems, namely, arithmetic unit, register and interconnect allocation are tackled simultaneously, rather than sequentially or iteratively. Second, the optimization is completely global in nature, the entire sequence is optimized, and therefore the entire datapath. Third, I have used a probabilistic hill-climbing algorithm [73],

simulated annealing, which can avoid the traps of locally-minimum solutions.

2.2.2 Pipeline Synthesis

Pipeline synthesis entails both placement of the micro-instruction sequence as well as partitioning the sequence into phases. Hardware resources, like arithmetic units, cannot be shared across phases in a pipeline. The number of phases in a pipeline thus affects both the throughput and the hardware resources required.

The algorithms used in hardware allocation have been extended to handle pipelines. A *partition* of micro-instructions across phases and a *placement* of micro-instructions within each phase is found so as to minimize $f(T, C)$, introduced earlier. Pipelining versus parallelism tradeoffs in the design space can be explored by varying the number of phases/partitions in order to vary throughput and hardware cost.

During synthesis, based on the required execution speed of the data path, decisions are made as to what the delay specifications of the different operations/micro-instructions should be. These specifications later become delay constraints on the logic synthesis and layout tools in the module generation subsystem.

2.3 Control Specification and Synthesis

In this system, control specification is a by-product of the datapath synthesis step. After the hardware allocation step produces the placement of micro-instructions, sequences of values for the different control signals to the ALUs and registers can be easily found. Mutually exclusive operations will be executed based on the values of some control signal, usually the status bits of an ALU. A controller is synthesized as a PLA based FSM or using multi-level combinational logic and flip-flops.

Optimal control logic synthesis is a difficult task necessitating the use of powerful optimization algorithms. The FSM controller is typically implemented using combinational logic and feedback registers. The combinational logic can be implemented as a microcoded

ROM, PLA or as a multi-level logic network. Efficient state assignment algorithms can substantially reduce controller area. After state encoding, the resulting combinational logic specification is optimized in different ways depending on the targeted implementation. The goal of the state assignment step is to find an encoding of internal states of the FSM so as to minimize the final area of the machine *after* combinational logic optimization.

2.3.1 State Assignment

All previous techniques for optimal state assignment have been targeted toward two-level logic or PLA implementations [26] [28]. Algorithms for state assignment of finite state machines targeted toward multi-level logic implementations [74] [75] have been developed. These algorithms find a state assignment of a FSM which minimizes an estimate of the area used by a multi-level implementation of the combinational logic. The estimate considered here is consistent with the estimate used by multi-level logic optimization algorithms [76] [77] [6] : the number of literals in a factored form of the logic. The algorithms heuristically minimize the number of literals in the resulting combinational logic network *after* multi-level logic optimization.

Multi-level logic optimization programs like MIS [6] and SOCRATES [77] primarily use algebraic techniques for factorizing and decomposing the Boolean equations by identifying common sub-expressions. Heuristics have been proposed based on *maximizing the number and size of common sub-expressions* and *minimizing the number of literals* which exist in the Boolean equations that describe the combinational logic part of the FSM after the states have been encoded but *before* logic optimization. The state assignment algorithms find pairs or clusters of states which, if kept minimally distant in the Boolean space representing the encoding, result in a large number of common sub-expressions in the Boolean network.

Literal counts averaging 20-40% less than other state assignment techniques have been obtained.

A good encoding of input and output signals in the controller can substantially reduce its area. For example, depending on the codes assigned to different ALU operations, the controller would require different areas after logic optimization. Input and output encoding algorithms for two-level [28] and multi-level logic [74] implementations are used in this system.

2.3.2 Multi-Level Logic Optimization

Research done over the past 30 years has resulted in to efficient methods for implementing combinational logic in optimal two-level form using Programmable Logic Arrays (PLAs). However, many logic blocks are inappropriate for this kind of implementation. For example, there exist functions whose minimum two-level representation has $2^n - 1$ product terms, where n is the number of primary inputs. In addition, even if a two-level representation contains a reasonable number of product terms, there are many cases in which a multi-level representation can be implemented in less area and generally as a much faster circuit.

Two basic methodologies have evolved for multi-level logic synthesis: 1) global re-structuring, where the logic functions are "factored" into an optimal multi-level form with little consideration of the form of the original description (e.g. [6] [78]; 2) peephole optimization, where local transformations are applied to the user-specified (or globally-optimized) logic function (e.g. [79] [80]).

Global re-structuring procedures have been shown to be crucially necessary in producing optimal designs. Factoring algorithms have been proposed [76] [6] which are effective in partitioning complex logic functions.

The factoring algorithms proposed in [76] [6] are primarily based on *algebraic* techniques. Boolean *factoring/division* techniques can achieve superior results. However, techniques proposed so far for Boolean factoring and multi-level Boolean minimization (e.g [81]) require very large amounts of CPU time.

Multi-level logic networks can be realized by standard cell or gate array implementations. For small-medium (< 50 product terms) sized two-level representations the PLA is a

very compact structure whose size is comparable (if not smaller) than a corresponding multi-level implementation. Topological optimization techniques like folding [82] can further reduce PLA area. A set of interconnected PLAs can thus exploit the layout compactness of PLAs without being constrained by the relative inflexibility of two-level logic structures.

A PLA can be decomposed into a set of interconnected PLAs which feed into one another. To perform this decomposition, factoring algorithms are required. Algorithms for *Boolean decomposition* have been developed, which decompose a PLA into a set of smaller interconnected PLAs such that the overall area of the resulting logic network, deemed to be the sum of the areas of the constituent PLAs, is minimized [83] [84].

The proposed algorithms are based on *multiple-valued* minimization. Given a PLA, a subset of inputs to the PLA is selected. This *selection* step incorporates a new algorithm which selects a set of inputs such that the cardinality of the multiple-valued cover, produced by representing all combinations of these inputs as different values of a single multiple-valued variable, is much smaller than the original binary cover cardinality. A relatively small size for the multiple-valued cover implies that the number of good Boolean factors contained in this subset of inputs is large. The different cube combinations given by this subset of inputs are *re-encoded* to satisfy the constraints given in the multiple-valued cover, thus producing a binary cover for the original PLA whose cardinality equals the multiple-valued cover cardinality. The re-encoding process incorporates a new encoding algorithm which minimizes the number of bits required to satisfy all or a subset of the constraints produced by multiple-valued minimization.

These algorithms have produced excellent results over a wide range of examples. Total delays and/or areas of resulting PLAs after Boolean decomposition are invariably smaller than the original PLAs. This approach exploits the layout compactness of PLA structures to produce small, fast multi-level logic implementations. Large PLAs have been reduced by factors

of 2-3 in size and delay [83].

2.4 Module Generation

Given the specification of a logic module, which may be an ALU, register file or random logic, the module generation subsystem generates a custom layout for the module. The logic synthesis tools used are, ESPRESSO, a two-level logic minimizer [85] and MIS, a multi-level logic optimization program [6]. Layout is generated after logic optimization in either standard cell, PLA or Gate Matrix styles. The layout optimization programs include TimberWolf, a standard cell placement program [61] and GENIE [86] a generalized array optimization program. Module generators are WOLFE [60], a standard cell place and route system which uses TimberWolf for placement and YACR [87] for routing, GEM [59] and OCTOPUS [62], Gate Matrix and PLA generators which use GENIE for topological optimization.

2.5 Place and Route

Placement and routing of the logic modules is performed by the MOSAICO layout system. MOSAICO [63] is an integrated macro-cell layout system with tools for multi-layer channel routing [88], power and ground routing, channel definition and ordering, and floor-planning and placement. Tools in MOSAICO run and generate symbolic layout views of the design. A spacing program takes the results after detailed routing in symbolic form and produces mask geometries while guaranteeing that design rules are satisfied. The OCT data manager is used to store the design at each stage of the layout process.

2.6 Verification

It is essential to be able to verify that the synthesized circuit implementation and the register-transfer level description actually represent the same machine. Several logic verification algorithms have been incorporated into the synthesis system for this purpose.

Many formal verification approaches have been taken to prove/disprove the equivalence of two combinational logic circuits, at the gate level and at differing levels (e.g. [89] [90] [91]

[92]). Only a few are practical for large circuits. Equivalence of combinational logic descriptions can be verified in the system described in this dissertation using the PROTEUS [29] logic verification package which has successfully verified large designs.

Very little work has gone into verifying sequential designs. An algorithm has been developed for the verification of the equivalence of two sequential circuit descriptions at the same or differing levels of abstraction, namely at the register-transfer (RT) level and the logic level [31]. The descriptions can represent general finite automata at the differing levels – a finite automaton can be described in an BDS-like language and its equivalence to a logic level implementation can be verified using my algorithm. Two logic level automata can be similarly verified for equivalence.

Previous approaches to sequential circuit verification have been restricted to verifying relatively simple descriptions with small amounts of memory. A new algorithm has been developed, which has been shown to be computationally efficient for much more complex circuits. The efficiency of this algorithm lies in the *exploitation of don't care* information derivable from the RTL or logic-level description (e.g invalid input and output sequences) during the verification process. Using efficient cube enumeration procedures at the logic level, I have been able to verify the equivalence of finite automata with a large number of states in small amounts of CPU time.

A two-phase enumeration-simulation algorithm has also been developed for verifying the equivalence of two logic level finite automata with the same or differing number of latches, given reset states or transfer sequences for the finite automata. This algorithm is as efficient as the general approach for verifying sequential machines described at different levels, but is much less memory-intensive. Using this algorithm, I have verified the equivalence

of finite state machines with more than 2500 states.

2.7 Testing

After the circuit has been synthesized, a test strategy is required to check that the chip satisfies its specification. A set of test sequences has to be found to pinpoint possible *faults* in the fabricated circuit, which is typically sequential in nature.

Test generation for sequential circuits is a difficult task. A popular approach to solving this problem is to make all the memory elements controllable and observable, i.e. Complete Scan Design [36] [37]. Scan Design approaches have been successfully used to reduce the complexity of the problem of test generation for sequential circuits by transforming it into one of combinational test generation, which is considerably less difficult. The design rules of Scan Designs also constrain the sequential circuits to be synchronous so that the normal operation of the sequential circuit is free of races and hazards. However, there are situations where the cost in terms of area and performance of Complete Scan Design is unaffordable. In addition, even though the general sequential testing problem is very difficult, there may be cases where test generation can be effective. Simply making all the memory elements scannable in a sequential circuit, without even first investigating how difficult the problem of generating tests for it is, could unduly incur unnecessary area cost.

Several approaches [93] [94] [39] [95] [96] [97] have been taken in the past to solve the problem of test generation for sequential circuits. They are either extensions to the classical D-Algorithm or based on random techniques [94] [96]. When the number of states of the circuit is large and the tests demand long input sequences, they can be quite ineffective for test generation. This is because no *a priori* knowledge of the length of the test sequence is available. In the extended D-Algorithm methods, a large amount of effort may be wasted in trying to find short sequence tests for faults that require long ones. Random testing techniques are based on continuous simulations and grading of test vectors according to simulation results. They can be very time consuming for difficult faults that have only a few long test sequences.

The new approach developed to test pattern generation [98] [99] [100] for sequential machines represents a significant departure from previous methods. First, using algorithms based on *state space enumeration* and information contained in a partial State Transition Graph (STG) of the machine, test sequences are generated to detect a large number of faults in the circuit. Then, an algorithm identifies a *minimal subset of memory elements* which if made scannable will result in easy detection of all remaining irredundant but difficult-to-detect faults. The identification of this subset is performed by analyzing the connectivity of the STG of the machine. Detection of all irredundant faults is guaranteed as in the Complete Scan Design case, but at much less area and performance cost, since much fewer lines, if any, need to be made observable.

2.8 Relationship between Logic Synthesis and Testing

The relationship between combinational logic synthesis and test generation is well known – it has been comprehensively reviewed in [101] and [102]. In [81], a synthesis procedure which guaranteed fully testable irredundant combinational logic circuits was proposed. The tests for all single stuck-at faults in the combinational logic circuit are obtained as a by-product of the optimization procedure [81]. Equally intimate relationships between the more complicated problems of sequential circuit synthesis and test generation have been envisioned.

A synthesis and optimization procedure has been developed [103], which beginning from a State Transition Graph description of a Moore or Mealy finite automaton produces a 100% testable logic-level implementation of the machine. The test sequences for all single stuck-at faults in the machine can be derived using test generation algorithms on the combinational logic blocks of the machine.

I can show that a strong relationship exists between state assignment, logic optimization and testability of a sequential machine. A procedure of constrained state assignment and combinational logic optimization can ensure 100% testability for both Moore and Mealy finite state machines. Results obtained on benchmark examples show that the area penalties

incurred due to the constraints imposed during state coding and logic optimization are small. The performance of the resulting circuit is usually *better* than a unconstrained design (This is because one of the constraints imposed requires combinational logic partitioning in the machine).

2.9 Limitations and Future Work

While the current system and the optimization tools in the system, described in the previous sections, represent significant advances and improvements in various areas, some limitations exist and should be addressed in the future.

A major limitation of datapath synthesis approaches, including the approach developed in this system, has to do with data-dependent loop exits. When loops are static and the number of iterations is known in advance, allocation algorithms can find an optimal schedule of operations that minimizes the hardware resource cost or a schedule that maximizes execution speed (or a combination of the two). However, allocation algorithms cannot solve the problem of finding an optimal schedule when the number of iterations in a loop is a variable, whose value is dependent on data inputs.

The state assignment algorithms presented in this dissertation produce significantly better results than other techniques, but are restricted in the sense that they model only the simple multi-level optimization of common cube extraction. Taking into account more complicated multi-level optimizations like common kernel extraction could improve the quality of results obtained. However, early approaches that use a kernel-only optimization strategy [104] have had limited success.

The algorithms for Boolean decomposition produce high-quality results and execute within reasonable CPU times for small to medium-sized circuits. Many steps in the algorithms have worst-case exponential time complexities. For large circuits, the CPU time requirements can become exorbitant. Also, the algorithms are presently restricted to begin from a two-level representation of a logic function. Developing fast algorithms for Boolean

decomposition, generalized to operate on multi-level circuits, represents a major challenge in combinational logic synthesis.

The deterministic sequential test generation algorithm developed [99] is considerably faster than algorithms proposed in the past. For datapaths, which are sequential circuits with a large number of latches, a State Transition Graph description is too large to store/generate and too cumbersome to manipulate. Different representations that are less memory intensive and which facilitate efficient state justification can speed up the test generation process.

The synthesis procedure of constrained state assignment that ensures fully and easily testable sequential machines is the first of its kind. Optimal combinational logic synthesis can ensure irredundant circuits without any area penalty. Optimal sequential synthesis procedures for fully testable non-scan sequential machines have not been proposed, at the time of this writing. Developing such procedures, represents a major theoretical challenge in this area.

2.10 Organization of this Dissertation

This dissertation is organized as follows. In Chapter 3, algorithms for hardware allocation for automatic datapath synthesis from behavioral descriptions are described. Problems of pipelined datapath synthesis are addressed. Control synthesis is the subject of Chapter 4 and 5. State assignment techniques for finite state machine controllers targeted toward multi-level logic implementations are described in Chapter 4. New algorithms for Boolean decomposition in multi-level logic optimization are described in Chapter 5. The verification and test subsystems in the behavioral synthesis system are described in Chapters 6 and 7 respectively. In Chapter 8, an optimization procedure which begins from a State Transition Graph description of a finite state machine and synthesizes a fully testable logic-level sequential machine is presented.

CHAPTER 3

Automated Datapath Synthesis

3.1 Introduction

The goal of the datapath synthesis step in a behavioral synthesis system is to produce register-transfer (RT) level hardware designs from an architectural description of a computer or to produce an RT design which implements a given program described in a high-level language in hardware. Significant effort has gone into the development of techniques for automated datapath synthesis (e.g. [21] [41] [58]) in recent years. However, even now, effective and versatile procedures are not available.

Initial work to tackle this problem included the development of a mathematical model for the datapath [49] to describe the conditions and relationships to be satisfied. Mixed integer-linear programming techniques were used. Unfortunately, even for very small specifications, the cost of generating a design exploded rapidly.

The expert system approach was taken in the DAA [45] [46] system. Design rules were collected, and based on these design rules, a rule-based data memory allocator was developed. As is the case with most rule-based techniques, only local optimization was possible and extensive changes could not be made to the input description to attain a globally optimal solution. Similar problems afflicted the allocators described and implemented in [47] [44]. Global optimization steps have been introduced into the expert system approach [105], but DAA has been used mainly to synthesize general-purpose computer datapaths where very little parallelism exists, and therefore little room for optimization during allocation.

A more global algorithmic approach to the allocation problem was first taken in [50] [52]. FACET is a automatic datapath synthesis program which minimizes the number of storage

elements, data operators and interconnection units. However, FACET performs these steps *sequentially* and independently of the following task(s). It thus does not provide for flexible area/time tradeoffs during allocation.

The USC MAHA system [58] uses critical path determination to perform hardware allocation. The heuristics used to guide scheduling are based on the concept of the *freedom* of an operation. A force-directed scheduling approach to hardware allocation has been taken in [72]. The optimization step is global and uses heuristics based on predecessor and successor *forces* on an operation. The different heuristics used in both these scheduling algorithms [58] [72] may result in locally-minimum solutions.

Synthesizing from arbitrary software descriptions (e.g. a MOSFET model evaluator in a circuit simulator) rather than the instruction-set specification of general-purpose computers offers considerably more room for optimization. As mentioned earlier, general-purpose computers tend to be rather uninteresting from an optimization point of view. Other than Trickey's work [41] [42] and synthesis of digital signal processor datapaths [19] [58], very little attention has been paid to the specialized processor synthesis problem.

In [42], the problem of extracting parallelism from a program and providing a maximal schedule of the operations in the program, while meeting a user-specified bound on each kind of processing unit(s), was addressed. In this chapter, I am concerned with the more general problem of *hardware allocation*, where the decisions on the number of processing units, storage elements and their interconnections are made; the scheduling problem is only a small part of the overall hardware allocation process.

In this chapter, new algorithms for the *simultaneous cost/resource constrained allocation of registers, arithmetic units and interconnect* in a datapath are presented. These algorithms operate under a wide variety of user-specified constraints on hardware resources and costs. There are three main differences between this approach and others taken in the past (e.g. [21] [58] [41]). First, all the allocation subproblems, namely, arithmetic unit, register and intercon-

nect allocation are tackled simultaneously, rather than sequentially or iteratively. Second, the optimization is global in nature, rather than the local optimizations of some previous approaches (e.g. [21]). The entire sequence is optimized and therefore the entire datapath. Third, a probabilistic hill-climbing algorithm [73], simulated annealing, which can avoid the traps of locally-minimum solutions, has been used. While simulated annealing is a general approach to combinatorial optimization, the key to its successful use in solving the hardware allocation problem has been the development of a multivariate formulation of this problem, along with a robust cost function, annealing schedule and an appropriate move set.

The hardware allocation problem in automatic datapath synthesis can be formulated as a *two-dimensional placement problem of micro-instructions in space and time.* The problem solved is to synthesize a datapath corresponding to the input data flow specification such that a given, arbitrary function of execution time and hardware cost, $f(T, C)$, is minimized. The hardware costs are the sum total of the costs associated with registers, arithmetic units, buses and links in the datapath, based on required layout areas for placement and wiring. A given placement of micro-instructions corresponds to a unique datapath with a certain hardware cost and execution speed. *Optimal conditional resource sharing is achieved by solving a constrained two-dimensional placement problem* where *disjoint* instructions are allowed to occupy the same spatial and temporal location. Given a data flow specification, algorithms are presented which find an optimal placement of micro-instructions, thus determining the spatial and temporal delineation of resources and producing an optimal datapath configuration.

The datapath synthesis problem is formulated here as one of two-dimensional placement of micro-instructions and modifications to incorporate conditional resource sharing are presented. Given this formulation, a simulated-annealing-based approach to solve the allocation problem is presented in Section 3.3. These algorithms are extended to handle looping constructs present in general software programs in Section 3.4. Results and illustrative examples, including the synthesis of a specialized processor datapath for MOSFET model evalua-

tion, are presented in Section 3.5. Extensions to synthesize some forms of pipelined datapaths are described in Section 3.6.

3.2 The Hardware Allocation Problem

In this section, the algorithms used in the allocation process are described. These algorithms take the architectural description of the machine or a software program and automatically synthesize the datapath corresponding to that description under specified hardware constraints and costs. In Section 3.2.1, some definitions are given. The input description used is described in Section 3.2.2. Basic allocation problems are described in Section 3.2.3, and the formulation of the datapath synthesis problem as a placement problem is presented in Sections 3.2.4-5. The cost function for this placement problem is described in Section 3.2.6. In Section 3.2.7, the placement formulation is extended to incorporate conditional resource sharing.

3.2.1 Basic Definitions

A micro-instruction is deemed to have two coordinates, a **spatial coordinate** and a **temporal coordinate**. The spatial coordinate corresponds to the arithmetic unit that the micro-instruction is executed on. The temporal coordinate corresponds to the clock cycle that the micro-instruction begins to be executed. A **space-time slot** corresponds to a spatial and temporal coordinate pair. A **time slot** corresponds to a temporal coordinate and all spatial coordinates. A **space slot** corresponds to a spatial coordinate and all temporal coordinates. Note that a micro-instruction may take multiple clock cycles to execute. It will be assumed for the purposes of this section that a micro-instruction executes in one clock cycle. However, this constraint will be relaxed in Section 3.4.

Conditional clauses may exist in the input description. Conditional clauses specify mutual exclusion relationships between micro-instructions. If during a pass through a code sequence, the execution of micro-instruction A implies that micro-instruction B will not be executed in the same pass, then A and B are deemed to be **mutually exclusive**. Two micro-

instructions can occupy the same space-time slot if and only if they are mutually exclusive.

A **two-dimensional placement** of instructions specifies spatial and temporal coordinates for each micro-instruction.

The datapath that is synthesized is a clocked sequential circuit, with an associated finite state machine controller. It is assumed that all ALU ports are latched.

3.2.2 Input Description

The behavioral description to be synthesized from can be a description of the instruction set of a computer or the description of an algorithm in a subset of C. In either case, the description is converted into a code sequence where parallelism, sequentiality and mutual exclusion are explicitly stated. During this transformation, various compiler-like optimization techniques (e.g. dead code elimination, constant folding) are used. The code sequence produced only has information about the data transfers required between program values. The control signals which initiate these data transfers are not explicitly stated. This control signal information is used only when the specification of the state machine controller for the datapath has to be derived.

The serial blocks are due to the dependences associated with any description. Mutual exclusion is a result of the conditional clauses in the input description. An example of an input sequence is shown in Figure 3.1, with *serial*, *parallel* and *disjoint* blocks, which are the means of representing sequentiality, parallelism and mutual exclusion respectively. Each operation is represented in a lisp-like syntax given by (*op* oper1 oper2 .. operN result), where *op* is any arithmetic or Boolean operator. An additional kind of block is the *implic* block. Parallelism or sequentiality are not explicitly stated in the *implic* block – they are automatically derived by checking for data dependences prior to hardware allocation.

The INITIAL and FINAL declarations imply that the following variables are live in the beginning and the end of the sequence respectively. The SYMMETRIC declaration

```
(serial
  (parallel
    (add x1 y1 z1)
    (add x2 y2 z2)
  )
  (parallel
    (mult z1 y3 z3)
    (minus z2 y4 z4)
  )
  (disjoint
    (divide z3 x3 z5)
    (divide z4 x4 z5)
  )
)
INITIAL x1 x2 y1 y2
FINAL z5
SYMMETRIC mult add
```

Fig. 3.1 Input Description

enumerates all the operations whose operands are interchangeable.

3.2.3 Basic Allocation Problems

The hardware allocation process consists of a variety of subproblems. Register allocation deals with allocating variables in the given description to a minimum number of registers. Arithmetic unit allocation entails scheduling operations on a minimum number of ALUs, meeting a cost or an execution time constraint. During the allocation, an optimal grouping of arithmetic operators within each ALU is also found. For instance, one might have two ALUs, one performing arithmetic operations and the other performing Boolean operations. Typically, one would like each of the ALUs to perform disjoint sets of operations, but this is not always possible. Lastly, interconnect allocation deals with implementing the sets of data transfers required in each time frame and allocating buses and links or multiplexor and de-multiplexor connections in the datapath.

The basic tradeoff in hardware allocation is between serial and parallel implementations of data flow descriptions. Given an input code sequence, one can synthesize a maximally parallel datapath which is expensive in terms of hardware resource cost and uses a large number of registers and arithmetic units. On the other hand, one can synthesize a cheap, serial datapath with a single ALU, which is likely to take a lot longer to execute the same task. Hardware resource cost used in this context generally represents the layout area required to implement the different modules in the datapath after placement and wiring issues have been taken into account. Depending on the user's objective function, the optimal datapath configuration will lie somewhere between these two extremes. Thus, the allocation process must trade off hardware resource cost against the execution time of the code sequence in an effort to find an optimal solution.

3.2.4 A subproblem

First, a subproblem in the allocation process is defined and solved.

Given a code sequence with singly-assigned variables and precedence constraints between operations, assign the code operations to M ALUs so a given, arbitrary function of the number of registers required, N_r, and the execution time, T, $f(T, N_r)$, is minimized.

Since the datapath is a clocked sequential circuit, a maximally parallel description would use lots of registers, but would execute the fastest. A completely serial description would require a minimal number of registers, but would be slow. An algorithm based on clique partitioning presented in [50], optimizes the number of registers *with a fixed code sequence schedule*, while the goal here is to find the optimal sequence exploiting *the extra degree of freedom* of being able to change the schedule.

Given a code sequence the lifetimes of all the variables can be calculated. The *lifetime* of a singly assigned variable is the duration between its assignment and last use. The number of registers required would be proportional to the overlap of the live periods of the singly-assigned variables, or to put it differently, the number of registers required is the *maximal*

density of variable lifetimes across the entire sequence. This is illustrated in Figure 3.2.

Disjoint variables are those whose lifetimes do not overlap. The allocation of registers to singly-assigned variables entails finding the best possible grouping of disjoint variables in sets so as to minimize the number of sets.

There is freedom in the ordering of the code operations as long the precedence constraints are not violated and the constraint on the number of processing units is satisfied. Given a code sequence exploiting this freedom can result in a smaller set of registers being required. This is illustrated in Figure 3.3. In Figure 3.3(a), an example code sequence being executed on a single ALU is shown. Without changing the order of the operations in the code sequence, the minimum number of registers required is 4, as shown in Figure 3.3(b). Allowing re-ordering of operations within the sequence produces a 3 register solution in Figure 3.3(c).

Finding the optimal ordering of operations within a sequence, so as to allocate a minimum set of registers, reduces to the *PLA multiple folding problem.* The goal is to try to find an ordering of the rows (which correspond to the code operations) under certain ordering constraints (constraints due to dependences and processors) such that the maximum number of disjoint columns (each column corresponds to the lifetime of a variable) can be coalesced (the

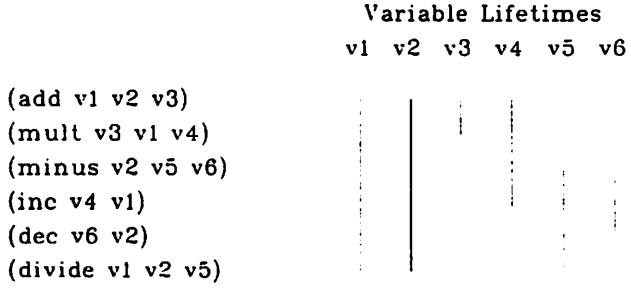


Fig. 3.2 Densities of Variable Lifetimes

```
v1 = v2 + v3
v4 = v2 - v3
v5 = v1 * v2
v6 = v4 and v3
v7 = v5 or v6
```

(a) Code sequence

```
R1 = R2 + R3
R4 = R2 - R3
R1 = R1 * R2
R4 = R4 and R3
R4 = R1 or R4
```

(b) Register allocation without re-ordering

```
R1 = R2 + R3
R1 = R1 * R2
R2 = R2 - R3
R2 = R2 and R3
R3 = R1 or R2
```

(c) Register allocation with re-ordering
Fig. 3.3

maximal number of variables can be merged). As mentioned earlier, there is a tradeoff between minimizing the execution time (the number of rows) and minimizing the number of registers (the number of columns). Therefore, in the general case of minimizing a function of execution time, T, and the number of registers, N_r, i.e. $f(T, N_r)$, an attempt is made to find an *optimal aspect ratio* of the PLA.

The PLA folding problem has been effectively solved using graph heuristics [106], simulated annealing [86] and exact branch and bound techniques [107]. These techniques can be used to solve the problem of register allocation as well. However, this formulation is merely representative of one part of the entire datapath synthesis process. An extended formulation is

now presented.

3.2.5 Formulation of the Entire Data Path Synthesis problem

The approach to synthesize a datapath described here is to give a general procedure which minimizes a given, arbitrary function of execution time and hardware cost. The entire cost of a datapath can be represented as:

$$C = p1 * (N_{alu}) + p2 * (T_{exec}) + p3 * (N_{reg}) + p4 * (N_{bus})$$

where N_{alu}, N_{reg} and N_{bus} are the number of arithmetic units, registers and buses used in the design, respectively, and T_{exec} is the time required by the implementation to execute the given code sequence. The costs of the ALUs, registers and interconnect, given by the parameters $p1$ through $p4$, can be estimated taking into account layout area, placement and wiring issues. A procedure which minimizes C under constraints would synthesize an optimal datapath.

This problem can now be formulated as a *placement* problem of code operations in two dimensions, that of space and time. A given spatial and temporal placement of code operations represents a datapath and has a unique cost C. A two-dimensional grid where each vertical slice corresponds to a processing unit/ALU and each horizontal slice corresponds to a time slot is constructed, as shown in Figure 3.4. Code operations are placed in grid locations corresponding to a ALU and a time slot, under precedence constraints, due to the dependences associated between them. *Nets* connect the occurrences of variables in the code operation and also connect variables to arithmetic units in corresponding slots. The internal position of the variable in the code operation is also specified, e.g. in a binary ADD a variable can be in the first or second positions for a given configuration.

The execution time is related directly to the number of occupied horizontal time slots. The horizontal time slots may be of different widths, the widths would be proportional to the delays corresponding to the code operations occupying that slot. The issue of operations having different associated delays is elaborated on in Section 3.4.

SPACE/TIME	ALU1	ALU2	ALU3
TIME1	(add x1 y1 z1)	(mult x2 y2 z2)	(equal x3 z3)
TIME2	(minus z1 x2 k1)	(divide z2 x1 k2)	
TIME3	(or k1 z2 l1)		(inc k2 l2)

Fig. 3.4 2-Dimensional Grid of code operations

The number of processing units is equal to the number of occupied vertical space slots. The operations that a given processing unit has to perform depends on the operators occupying the grid locations in its corresponding vertical space slice. A processing unit may be simply perform an increment operation, or may be a complex floating point unit capable of multiply, add and divide operations. Thus, the formulation takes into account *the grouping of arithmetic operators* into processing units.

The number of registers required to realize the variables is related to the maximum density of nets across the entire grid. This is because the *extent* of the nets connecting occurrences of a variable is a representation of the lifetime of the variable. Given a maximum density of lifetimes M, using the Left Edge Algorithm (used widely in channel routing [108]), the variables can be coalesced into M registers.

The interconnect relationship to the physical entities of nets and code operations is more difficult to formulate. Obviously the number of registers and ALUs is related weakly to the number of interconnections required. Other measures of interconnect complexity can be obtained – the number of links required can be related to the stagger of nets in this formulation.

The *stagger* of a net is defined as the number of different space slots that the net connects to, minus one. If a variable is used to store input/output data for more than one ALU, it will exist in more than one space slot and the stagger of the net corresponding to the variable will be non-zero. The more staggered a net, the more the number of ALUs the variable and

eventually the register, feeds into. To minimize the number of links, one could minimize the sum total of the stagger of all the nets. However, groups of variables may be coalesced into the same register. This register will then need to connect to all the ALUs that any of the variables connected to. Only variables which are disjoint can be coalesced into the same register. The sum total of net stagger does not model the effect of merged variables accurately. *The stagger of nets between disjoint variables* is a better indicator of interconnect complexity (number of links) at any stage. The net stagger is further refined by the position information of the variables within the code operation. The position information takes into account the fact that variables may be feeding into one or both ports of the ALU.

Another good measure of the number of buses required given a schedule is the maximum number of distinct sources and number of sinks in all the time slots (which is an indicator to the number of parallel data transfers required). So, even if all the registers have been allocated previously, the tradeoffs between execution time and interconnections can be made. In the general case, execution time can be traded for registers, processing units and interconnections.

A cost function must therefore be defined in terms of the above mentioned quantities. The problem is then to find a global placement of code operations in the space-time slots under the dependence constraints and a placement of variables within the code operations, which minimizes cost. Then, the variables can be coalesced into registers and the interconnections into buses.

Some variables, for instance arrays, are better stored in memory. If they are, accessing them potentially takes more cycles. There is a tradeoff between reducing the number of registers by allocating variables to memory locations and increasing the execution time. This tradeoff can be explored if necessary.

To solve this problem, various techniques for solving the placement problem can be employed. The goal is to find a placement which produces a global minimum for the function

$f(T, C)$. The use of simulated annealing has produced excellent results for integrated circuit cell placement problems [61], where a complex, non-linear objective function has to be minimized under constraints. Hence, simulated annealing was chosen to solve this particular placement problem. This simulated-annealing-based algorithm is described in Section 3.3.

3.2.6 The Cost Table

The specification of costs is vitally important. Given a complex cost function, the simulated-annealing-based algorithm can find near-optimal solutions, *for that cost function*, within reasonable amounts of CPU time [86]. Ideally, the hardware costs should reflect the exact layout area of of the datapath. Inaccurate costs can result in a datapath that is sub-optimal in terms of area, even though the placement produced is optimal for the specified cost function. While the areas of individual modules (e.g. registers, ALUs) can be estimated exactly or near-exactly, estimating routing area is much more difficult. In [109], the effects of incorrect estimation were presented and shown to be significant in the final result.

A cost table (Figure 3.5) specifies the cost of hardware resources and operators. It also specifies implicitly the parameters $p1$ through $p4$ in the cost function C (Section 3.2.5). The parameters $p1$, $p3$ and $p4$ are *area parameters*, while $p2$ is an *execution time parameter*. The area parameters reflect the layout area of the individual modules. The execution time parameter, $p2$, is a way of specifying whether a fast datapath or a relatively slow one is desired. A higher $p2$ implies a greater cost for execution time and will result in a faster datapath. These parameters are not necessarily constants, they are, in general, functions of the number of ALUs and/or registers and/or buses in the datapath. For the examples tried, this formulation

appears to work well, as is made clear later.

3.2.6.1 Register Costs

$p3$ is equal to the area of the library register to be used. It is a multiplying factor for the number of registers in the datapath. In the cost table of Figure 5, $p3$ is a function of the number of registers, in an effort to estimate routing area (Section 3.2.6.3). The first 5 registers cost 10 units each, the next 5 cost 15 units.

3.2.6.2 Costs of ALU operations

The cost of each arithmetic or Boolean operator should reflect the layout area to implement that operator. A complication arises when attempting to optimally group operators within ALUs. Given that the ALU is to be implemented using combinational logic, the area required by a set of operators is generally, *not* equal to the sum of the areas required to implement each operator separately. A case in point is an ALU implementing addition and subtraction. This ALU would be only slightly larger than an ALU implementing only addition or only subtraction, not twice the size. Thus, ALU costs cannot be calculated using simple additive relationships.

This problem is alleviated by defining costs not only for each operator for small sets of operators as well. A *multiply* operator may have a cost of 100 units, a *divide* a cost of 200 units, and an ALU performing *multiply* and *divide* may be deemed to have a cost of 210 units depending on library-specific information. Given an arbitrary set of operators, the program checks to see if costs have been specified for any subset of operators before adding costs up

```
# cost of different operations in a ALU
ALU
add 50
sub 50
fadd 100
mult 250
add sub 60

# register costs
REGISTER
# starting from register 1, each register has cost 10 units
1 10
# starting from  register 5, each register has cost 15 units
5 15

# execution time
EXECUTION
1 50
50 50

# interconnect, buses and links
BUS
1 100
3 150

LINK
1 5
100 10
```

Fig. 3.5 Example Cost File

for the single operators.

3.2.6.3 Estimating Interconnect Area

The areas of the individual modules can be estimated accurately and included in the cost table. The number of links and buses can be estimated closely, as described he number of links and buses can be estimated closely as described in the previous section. The area for a link/bus is to be used as parameter $p4$. This area is typically a complex function of the number of registers and ALUs in the datapath. Assuming that $p4$ is a constant, i.e. that interconnect area is a linear function of the number of links/buses can be quite inaccurate [109].

The approach used relies on empirical estimations of routing area. For example, given a layout style, the increase in routing area (not total area) due to incremental additions of registers and associated links is evaluated and this cost is added to the link and register costs. The link and register costs then become piecewise-linear functions. Data points over a range of numbers of ALUs and registers in a datapath are obtained. The number of data points required to obtain exact accuracy is, unfortunately, infinite. However, with a reasonable small number of data points, one can do better than a linear approximation on the number of links. In the cost table of Figure 3.5, register and interconnect costs are modeled as piecewise-linear functions and ALU costs are modeled as linear functions.

Accurate estimation of routing area remains largely an unsolved problem [109]. It is clear that the total area of a datapath is a highly non-linear function of the number of ALUs, links and registers. Given this complex function, or a good approximation of this function, the simulated-annealing-based algorithm described in the next section, obtains high-quality solutions.

3.2.7 Conditional Resource Sharing

Conditional clauses can result in mutually exclusive or disjoint statements. For example, the statements in the THEN and ELSE clauses of an IF statement are disjoint. *Disjoint statements can exist on top of each other on the same space-time slot.* The algorithm takes into account mutual exclusion and finds a optimal schedule for the code sequence with an arbitrary number of conditional clauses.

Placing operations on the same space-time slot amounts to conditional resource sharing. Many forms of conditional resource sharing are possible. The co-existence of two ADD operations on the same space-time slot implies that the two operations are sharing an adder since they are mutually exclusive. If two operations sharing a common variable exist on the same location, a register will be shared by the two disjoint operations, and it will store information dependent on conditional clauses.

The problem thus becomes a placement problem with constraints on what statements that can exist on the same time and space coordinates.

Disjoint blocks may be arbitrarily nested in the code sequence. Initially, before the optimization, mutual exclusion relationships between each pair of operations in the given code sequence is found, and this information is exploited. For example, given:

```
(disjoint
    s_1
    (disjoint
        s_2
        s_3
    )
)
```

s_1 is deemed to be disjoint from both s_2 and s_3 and s_2 is disjoint from s_3.

3.3 A Simulated Annealing Based Solution

3.3.1 Introduction

Simulated annealing, proposed by Kirkpatrick *et. al* [69], has proved to be an effective solution to the cell placement problem in LSI layouts [61]. Its basic feature is that it allows *hill climbing moves* [73] in exploring the configuration space of the optimization problem, specified in such a manner that the minimum of the objective function is to be found. The probability of accepting these hill climbing moves is controlled by a parameter analogous to temperature in the physical annealing process and this parameter decreases gradually as the annealing process proceeds. The simulated annealing algorithm can be used for combinatorial optimization problems specified by a finite set of states and a cost function defined on all the states. The algorithm randomly generates a new state or configuration and the new state is accepted or rejected according to a random acceptance rule governed by the parameter analogous to temperature in the physical annealing process. The basic algorithm proceeds as fol-

lows:

```
T = T0 ;
X = Starting_Configuration ;
while( "cost is changing" ) {
   for( "a certain number of times" ) {
      Generate_New_State( j ) ;
      if( accept(c(j), c(X), T) ) {
         X = j ;
      }
   }
   T = update(T) ;
}
```

Whether or not a new state is accepted is determined by the function **accept()**:

```
accept( c( j ), c( i ), T ) {
   change_in_cost = c( j ) - c( i ) ;
   if ( change_in_cost < 0 ) return(1) ;
   else {
      Y = exp( - change_in_cost/T ) ;
      R = random(0, 1) ;
      if ( R < Y ) return(1) ;
      else return(0) ;
   }
}
```

This basic algorithm forms the core of the new approach. The parameter T is analogous to the temperature in the physical annealing process. At every temperature point, a number of random moves are generated. The number of moves generated is a parameter which can be controlled by the user; it affects the quality of the solution profoundly. Theoretical results exist that simulated annealing asymptotically approaches the global optimum of the configuration space [73]. This approach has been shown to work well for a variety of combinatorial optimization problems. However, it provides its best advantage over more conventional heuristic approaches when the function it is to minimize is complex and non-linear and the search space is constrained.

While the probabilistic core is quite straighforward, the two most important things in any simulated-annealing-based algorithm are the *generation of new states* (**Generate_New_State()**) during the annealing process and the *cost function* (**c()**) to be

optimized for. The generation of states and the cost function together determine the quality of solutions which can be obtained.

These two aspects of the simulated-annealing-based algorithm for the allocation problem are described in detail below.

3.3.2 Generating New States

For the hardware allocation problem, new states are generated during the annealing process in three different ways:

(1) Interchanging two code operations.

(2) Displacing a code operation from one location to another.

(3) Interchanging the variables in a symmetric operation (e.g. ADD).

Moves (1) and (2) must satisfy certain constraints, namely that the precedence constraints between operations cannot be violated by such a move, and operations on the same space-time slot must be disjoint. Examples of interchanges and displacement of operations in illustrated in Figure 3.6.

The generation of states proceeds as follows:

Generate_New_State_1():

(1) The operations are numbered from 1 to N_{op}, the number of operations. Two numbers R_i and R_j are randomly generated, such that $1 \leq R_i \leq N_{op}$, $1 \leq R_j \leq (N_{op} \times RAT1)$, where $RAT1$ is the ratio of displacements to interchanges (typically 5).

(2) If $R_j \leq N_{op}$, an interchange of the two operations R_i and R_j is tried. If the interchange violates any constraint, and R_i/R_j (in that order) happens to have a symmetric operator, the variables in R_i/R_j are interchanged.

(3) If $R_j > N_{op}$, a new location for the first operation is randomly generated and the operation is displaced to the new location, if the displacement does not violate the before-

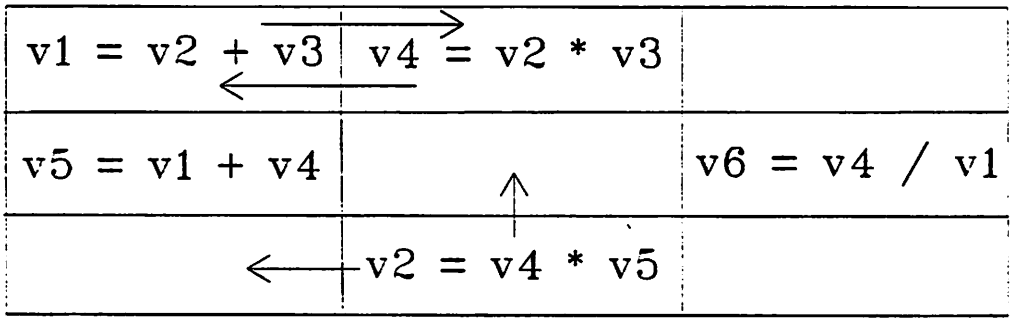


Fig. 3.6 Interchanges and Displacements During Annealing

mentioned constraints.

During the end of the annealing process (at low temperatures), the generation of states takes a different form so as to generate states which are more likely to be accepted.

Generate_New_State_2():

(1) Identical to Step 1 of **Generate_New_State_1()**.

(2) If $R_j \leq N_{op}$, an interchange between R_i and the operation immediately to the left or right of R_i is tried, in randomly generated order. If one direction fails, the other is tried. If both fail, a variable interchange in R_i is tried.

(3) If $R_j > N_{op}$, a displacement of R_i to the immediate left or right in the same time slot, immediately ahead or behind in the same space slot is tried in randomly generated order.

3.3.3 The Cost Function

The cost function used should be representative of both the hardware and execution time cost function C (Section 3.2) to be optimized.

The total execution time required for the entire sequence is one part of the cost function. In the general case, the execution time may be weighted by the frequency of code kernels within a code sequence. A *kernel* in a code sequence has the property that no operation outside the kernel is executed in between operations contained within the kernel. Given a large code sequence, parts of the sequence (kernels) may have higher execution time weights associated with them because they are more frequently used. The weighted *spread* (the time of execution of the last operation in the kernel – the time of execution of the first operation) of kernels can be calculated, given a schedule for the entire code sequence.

The number of registers required in hardware is given by the maximum density of nets (which connect occurrences of variables) across all the time slots. The number of registers required is part of the cost function.

For each space slot, the sum of the costs of all the distinct operators (or operator sets) required is found. The sum of all these costs is the processor cost constituent of the cost function.

Interconnect cost is estimated by estimating the number of links and buses required in hardware. The stagger of nets between disjoint variables is good indicator of link costs. The number of buses required is estimated by calculating the maximum number of distinct sources and number of sinks in all the time slots, since this is a good indication of the number of

parallel data transfers required.

3.3.4 Hardware Resource Constraints

Hardware resource constraints, (e.g. limits on the number of ALUs or registers) can easily be incorporated into the simulated-annealing-based algorithm by penalizing configurations which violate any of these constraints. A penalty is added to the cost of such a intermediate configuration and is sufficiently high so as to ensure that the final solution satisfies all the constraints.

3.3.5 Execution Time Constraints

A bound on the time required by the datapath to execute the code sequence, or parts of the code sequence, may be given. This constraint is incorporated using a penalty function approach, as in the case of constraints on hardware resources.

3.3.6 Stopping And Inner Loop Criteria

The number of states generated per temperature point is a certain integer multiple, MC, of the number of code operations. This number is user-specified and varies depending upon the amount of CPU time the user wants to spend. If a higher number of states, $MC \times N_{op}$, are generated, it is likely that a better solution will be obtained, but more CPU time will be expended. It has been experimentally determined that if MC is between 1-10, good results are obtained within reasonable amounts of CPU time. $MC \gg 10$ does not help much in terms of solution quality, while requiring large amounts of CPU time. $MC < 1$ usually results in low-quality solutions.

The temperature is lowered to a fraction (typically 0.90) of its original value after each temperature point. The annealing process terminates when the cost function has not changed in value for three temperature points. These numbers have been determined experimentally

and are consistent over a range of examples in producing good solutions.

3.4 Further Extensions

For the sake of clarity in presentation it has been implicitly assumed thus far that the operations in the input description have equal delays. However, in general, operations in a software program may have drastically different delays. For example, a 32-bit multiply may take more than 10 times the time required by an integer increment.

It is not difficult to generalize the formulation of the datapath synthesis problem to handle operations with different delays. A generalized two-dimensional placement of operations is shown in Figure 3.7. The height of each operation is proportional to its delay. For example, the MULTIPLY has a delay which is 3 times the ADD. The placement now resembles a set of linked list of operations (one for each ALU), rather than the matrix of operations of Figure 3.4.

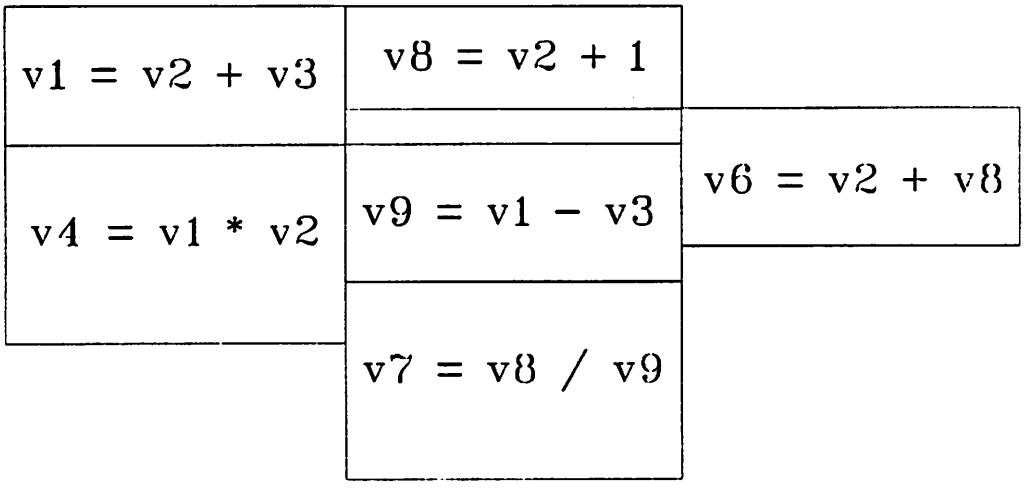


Fig. 3.7 Generalized Two-Dimensional Placement

The simulated-annealing-based algorithm for hardware allocation as described in Section 3.3 made no assumptions about the relative delays of operations. If operations have different delays, the highest common factor of all the different operation delays in the data flow description is calculated. This becomes the size of one time frame. Operations can occupy more than one time frame. During interchanges and displacements of operations in time or space, the time positions of the successors of the interchanged and displaced operands may also change. This is illustrated in Figure 3.8.

Loops are a succinct way of representing iteration in programming languages. It is important that an allocation algorithm be able to provide for loops in the input description. Current allocation algorithms and the algorithm presented here are restricted to handling loops whose iteration count can be statically determined. Data-dependent loop exits, which imply that the number of iterations of a loop is a variable, cannot be handled in an optimal way by the approach presented here and is a subject for future research.

One method of dealing with loops is to treat each loop as a single operation with delay equal to the number of iterations times the delay of each iteration. This single operation is

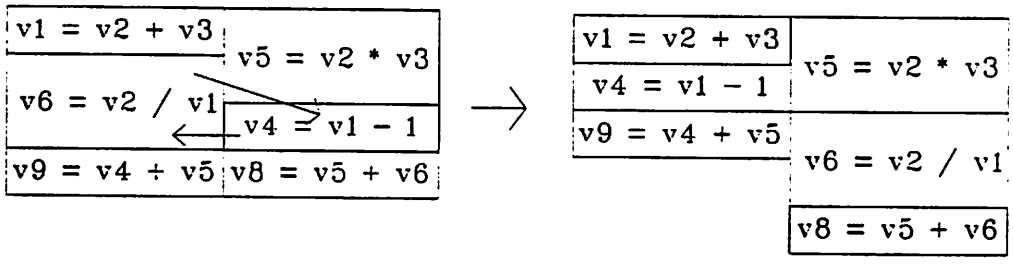


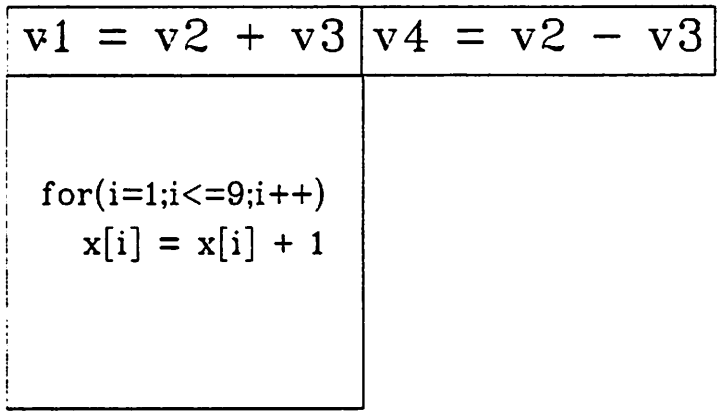
Fig. 3.8 Before (left) and after interchange

scheduled just like other basic operations. However, the problem with this approach is that all the iterations of a loop are always scheduled serially on a single ALU. It may be beneficial to schedule iterations in parallel on different ALUs.

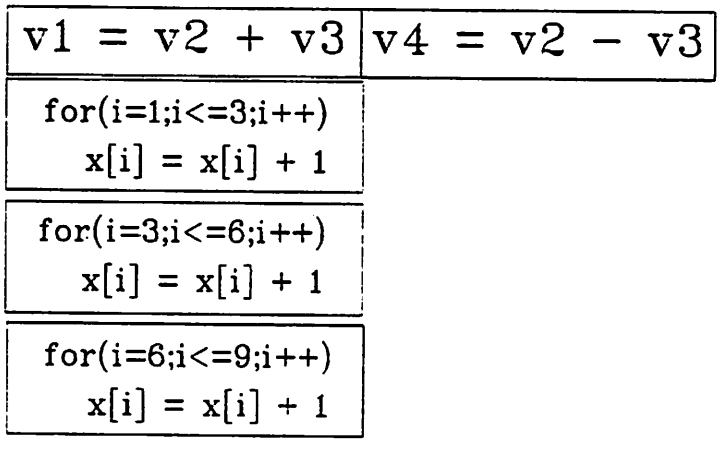
Another method of dealing with loops in the input description is *full unwinding* [21]. In full unwinding, all the iterations in a loop are expanded into a number of operations. The number of operations after unwinding will be proportional to the number of iterations in the loop. These operations can be scheduled independently and may be executed in parallel if the precedence constraints between them are not violated. This method exploits all the degrees of freedom present in scheduling iterations of loops separately. However, given a loop with a large number of iterations, full unwinding is not always feasible.

My solution to this problem is what I call *dynamic partial unwinding* of loops during the annealing process. Initially, all loops are represented as basic operations and their delays computed. However, they are *tagged*. During the annealing, a possible move (other than displacing tagged or untagged operations) is to split a tagged operation into two or more components. For example, a 10-iteration loop may be split (unwound) into two 5-iteration components. These components are also tagged. The components are scheduled separately and may be executed in parallel if no precedence constraints exist between them. However, this splitting does not preclude the possibility of the all the iterations of the loop being executed on the same ALU if that happens to be the best configuration. A possible scenario of loop splitting during the annealing is shown in Figure 3.9.

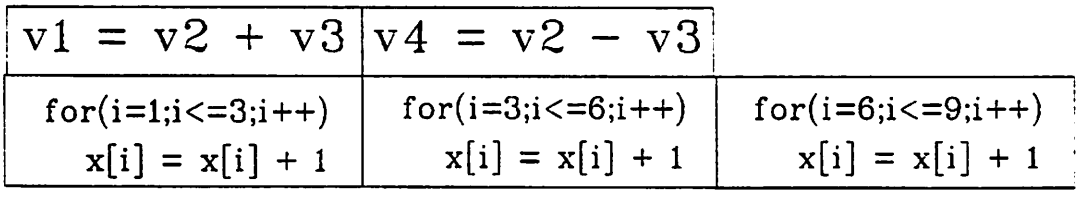
The components after splitting are tagged and may be further split up into subcomponents. The number of components a loop is split into (the degree of unwinding) and the level of splitting is specified initially by the user. If the number of components equals the number of loop iterations, then the result is full unwinding. If splitting is not allowed, then the loop is being treated as a basic operation.



(a) Initial placement



(b) Loop splitting



(c) Final placement
Fig. 3.9

Another extension is trading off delay and cost for single operations. For example, different adders may exist in the library with varying area costs and delays. A fast adder per-

forming 32-bit addition in 25ns may cost 10 units, a slower 40ns adder may cost only 5 units. The choice of the adder which minimizes the objective function, f, can be made during the annealing. A move during the annealing would be to change a fast adder into a slow one or vice versa. In general, more than two implementations with different cost-delay tradeoffs can exist for an operator.

3.5 Examples and Results

The code sequence in [50] is used as a first example to be synthesized, using the techiques described. The input file is shown in Figure 3.10. The *implic* block has been used for convenience, since data dependences between operations are quite complicated.

In the first run, (using the simulated-annealing-based algorithm) the costs of arithmetic operations were chosen to be ≥ 50 units, each register cost was chosen as 10 units, each link 10 units and execution cost per time slot was fixed at 5 units[1]. Execution speed was thus given a low priority in this run. The optimization produced a serial sequence shown in Figure 3.11(a), which needs eight cycles to execute. CPU time required for the simulated annealing run was 30 seconds on a VAX 11/8650 running ULTRIX. The datapath synthesized after bus allocation is shown in Figure 3.11(b). The minimal number of registers and interconnections have been used.

Bus allocation is done after the code operation placement using algorithms similar to [51]. However, *during the placement*, the amount of interconnect required is calculated at every stage and minimized as described earlier. It was assumed, while performing bus allocation, that the data transfers for every micro-instruction (*op* V_a V_b V_c) look as follows:

V_a->link->bus->link->*ALUin*1
V_b->link->bus->link->*ALUin*2
ALUout->link->bus->link->V_c

That is, value V_a passes via a link, a bus and another link to the first port of an ALU. Similar

1 These numbers were selected only as an example, but reflect my estimate of implementation cost in a simple

```
(implic
  ( add v1 v2 v3 )
  ( minus v3 v4 v5 )
  ( mult v3 v6 v7 )
  ( add v3 v5 v8 )
  ( add v1 v7 v9 )
  ( divide v10 v5 v11 )
  ( equal v3 v13 )
  ( equal v1 v12 )
  ( and v11 v8 v14 )
  ( or v12 v9 v15 )
  ( equal v14 v1 )
  ( equal v15 v2 )
)
INITIAL v1 v2 v4 v6 v10
FINAL v1 v2 v4 v6 v10
SYMMETRIC add mult or and
```

Fig. 3.10 Input File for example from [50]

transfers are represented by the other two clauses. The two input transfers to the ALU are required to occur in parallel. If in fact, one is allowed to make the two input transfers to an ALU in sequence, one can synthesize a datapath for this example with only one bus.

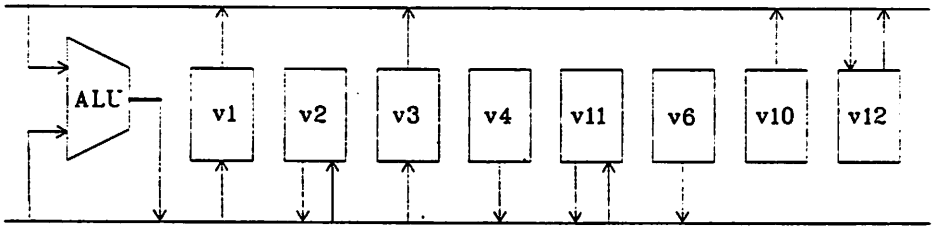
The freedom in being able to arrange symmetric operands in order to minimize interconnect has been exploited by the program. If that had not been done more links would have been required.

The placement of code operations produced by the program given a higher execution time cost than in the previous case, that of 50 units, is shown, in Figure 3.12(a). The register/ALU/interconnect cost was unaltered from the previous run. Note that the placement is such that operations in the two ALUs have *no* operators in common – an optimal grouping. The datapath corresponding to the code sequence in Figure 3.12(a) is shown in Figure 3.12(b), again with a bus-style design. The CPU time required for synthesis was 40 seconds on a

standard cell library [110].

(add v1 v2 v3)	(equal v1 v12)
(minus v3 v4 v11)	
(mult v3 v6 v2)	
(add v3 v11 v3)	
(add v1 v2 v2)	
(divide v10 v11 v11)	
(and v3 v11 v1)	
(or v12 v2 v2)	

(a) Code-sequence after 2-D placement



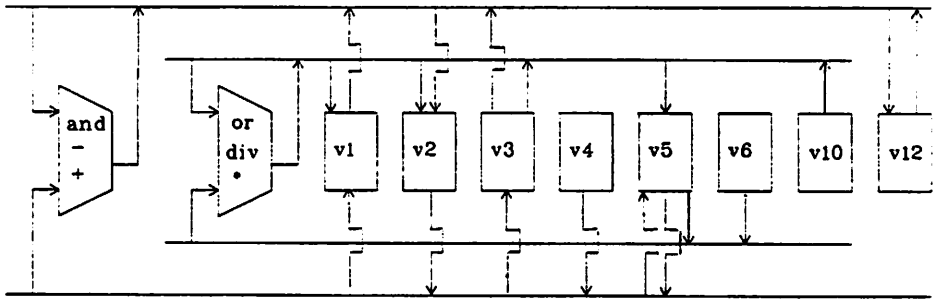
(b) Synthesized Bus-style Data-Path
Fig. 3.11

VAX 11/8650. For two micro-instructions in the same time slot, all the ALUin transfers are assumed to occur simultaneously, and all the ALUout transfers together. In the datapath shown four buses are required. If the constraint of simultaneous input/output transfers to all ALUs is relaxed, fewer buses will suffice. The finite state machine controller specification for the datapath is shown in Figure 3.12(c). A single input is required to start computations. The outputs are the load signals to the different links in the datapath. Some links are controlled by the same output.

Another small example, this time with conditional clauses in the input description, is shown in Figure 3.13. The input description is shown in Figure 3.13(a), the two-dimensional placement in Figure 3.13(b) and a multiplexor-style datapath, which takes 5 or 6 cycles to execute the description depending on what conditions are asserted, is shown in Figure 3.13(c).

(add v1 v2 v3)	(equal v1 v12)	
(minus v3 v4 v5)		(mult v3 v6 v2)
(add v3 v5 v3)		(divide v10 v5 v5)
(and v1 v2 v2)		(or v3 v5 v1)
(add v12 v2 v2)		

(a) Code-sequence after 2-D placement



(b) Synthesized Bus-Style Data-Path

0 s0	s0 NOP NOP	00000000000000000000
1 s0	s1 NOP NOP	00000000000000000000
– s1	s2 AND NOP	01000000100110000010
– s2	s3 SUB MUL	00101100000001100111
– s3	s4 ADD DIV	00001011001000011011
– s4	s5 AND OR	11010100000100001011
– s5	s6 ADD NOP	00010000010100000010

(c) Finite State Machine Controller

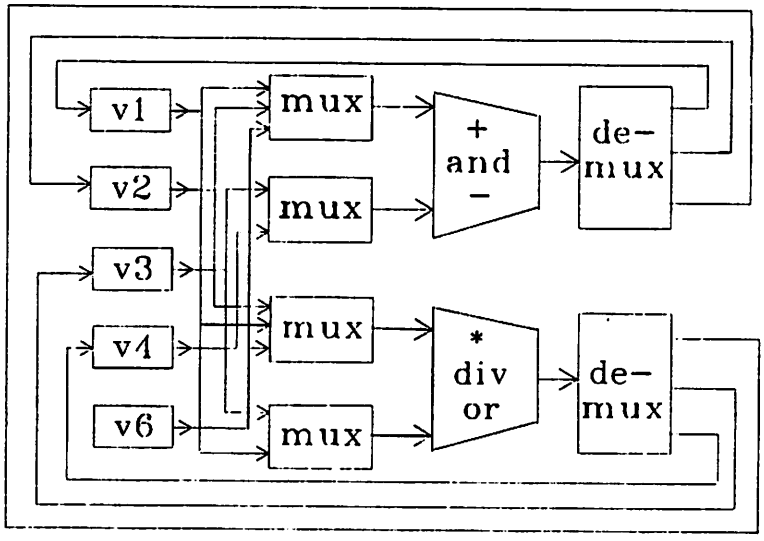
Fig. 3.12

```
(serial
   (parallel
      (add v2 v3 v1) (divide v2 v3 v4)
   )
   (disjoint
      (add v1 v4 v6) (minus v1 v4 v6)
   )
   (disjoint
      (mult v6 v3 v7)
      (serial (divide v6 v3 v8) (mult v8 v2 v7))
   )
   (parallel
      (and v7 v4 v9) (or v7 v1 v10)
   )
)
```

(a) Input Description

(add v2 v3 v1)	(divide v2 v3 v4)
[(add v1 v4 v6) (minus v1 v4 v6)]	
	[(mult v6 v3 v6) (divide v3 v6 v3)]
	(mult v2 v3 v6)
(and v6 v4 v2)	(or v6 v1 v3)

(b) 3-Dimensional Placement



(c) Multiplexor-Style Data Path

Fig. 3.13

A larger example is a MOSFET model evaluation routine implementing the DC part of the Schichman-Hodges [111] or SPICE level-1 MOSFET model for MNA circuit simulation. The goal, as before, was to synthesize the datapath of a specialized processor executing the software description optimally under different cost constraints. The inputs to the processor are the MOSFET node-to-ground voltages and device model parameters and the outputs are the currents, equivalent conductances and their derivatives as needed by the companion model. The datapaths generated in this example could be used as co-processors for model evaluation in a hardware simulation engine [112].

The software description initially consisted of about 150 lines of C code. This was converted into about 300 lines of input to the synthesis program. A total of 228 possible operations existed in the input description (some of them mutually exclusive). The operators used were all floating point source and target – add, minus, divide, multiply, minimum, maximum, compare. Using different hardware and execution time costs, three different datapaths were synthesized.

The first datapath generated was a serial implementation with a single ALU; the second and third have two ALUs. The execution speeds of the datapaths (normalized to the serial datapath), the number of registers, buses and links in the datapath, estimated areas of the datapaths (normalized to the serial datapath) and CPU times in minutes for synthesis on an VAX 11/8650 running ULTRIX are summarized in Table 3.1. The ALUs in Datapaths 2 and 3 execute different sets of operations. In Datapath 3, both ALUs perform multiplication/division as well as addition and subtraction. In Datapath 2, only ALU1 performs multiplication/division. The datapaths are shown in Figure 3.14. This large example illustrates how the algorithms described in this chapter can be used to effectively explore tradeoffs in the design space.

MOSFET evaluation entails filling in a matrix of currents and conductances – the matrix is assumed to be stored in memory. This would be the case if the datapaths are to be used as co-processors for a hardware simulation engine.

DP	execution time	#reg	#bus	#link	estimated area	CPU time
1	1.0	21	2 + 1*	54	1.0	10.1m
2	0.65	21	4 + 1*	66	1.7	9.2m
3	0.54	21	4 + 1*	77	2.5	11.2m

* memory bus

Table 3.1 MOSFET model datapath statistics

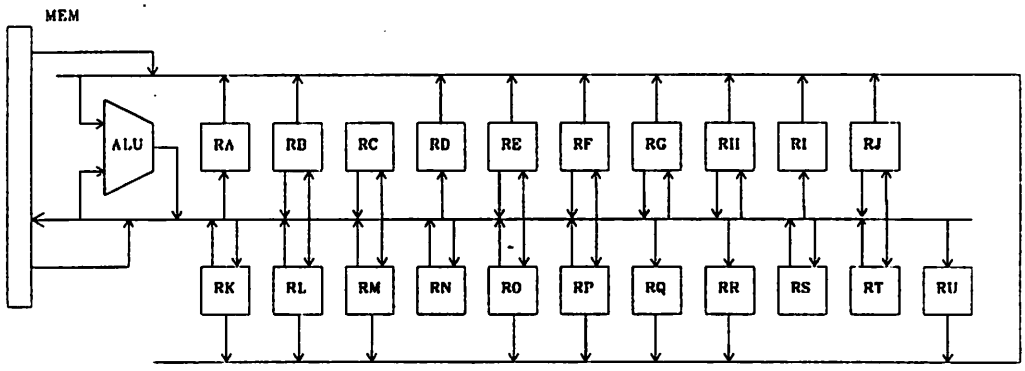
The C program implementing the MOSFET model evaluation routine is included as Appendix A. The intermediate textual description produced from the C program is also included in Appendix A.

3.6 Synthesizing Pipelined Datapaths

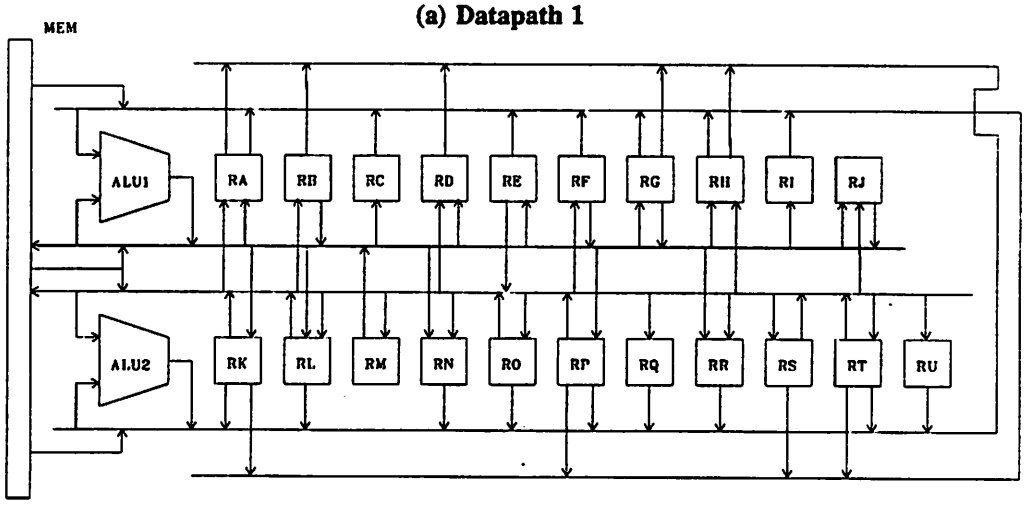
Pipelining is an essential feature of the computers being designed today [113]. Pipelining implies overlapping of multiple tasks – each computation task is partitioned into subtasks and each subtask is executed in a clock cycle. Consecutive tasks are initiated at some intervals called the *latency* of the pipeline, which are integral multiples of a clock cycle.

Given an input data flow specification, pipeline synthesis involves splitting the data flow graph into stages (phases or partitions), with constraints on the number of stages and stage delays, so as to optimize for execution time and/or hardware cost. Engineering solutions to pipeline scheduling given fixed hardware resources have been published [114] [115]. A pipeline synthesis procedure based on scheduling algorithms was first published in [19].

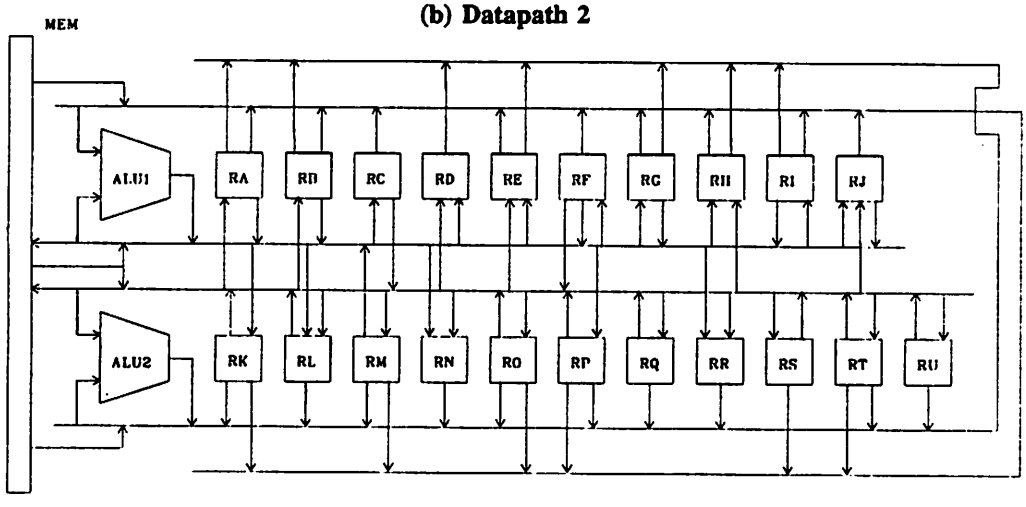
SEHWA [19] generates datapaths from data flow graphs along with a clocking scheme which overlaps execution of tasks. SEHWA estimates the cost of a pipeline based on the number of processing units of each type and the number of latches required in the hardware implementation. It has been used to synthesize clocking schemes for general-purpose computers with fetch-decode-execute pipelines [57] and pipelined digital signal processors.



(a) Datapath 1



(b) Datapath 2



(c) Datapath 3

Fig. 3.14

The goal here is to solve a more general pipeline synthesis problem, where register/latch, arithmetic operator and interconnect cost is taken into account during the pipelining. To this end, the hardware allocation algorithms presented in Sections 3.2-3.4 have been extended to to able to synthesize pipelines.

3.6.1 Extensions for Pipeline Synthesis

Hardware resources cannot be shared across pipeline stages. For example, given a two-stage pipeline, after pipeline setup, the micro-operations in both stages will have to be simultaneously performed on each clock cycle (albeit on different input streams) and will therefore need distinct computational units.

Pipeline synthesis involves partitioning the input data flow description into a number of pipeline stages and scheduling micro-operations within each stage meeting a cost or an execution time constraint. The problem solved is to synthesize a pipelined datapath, given a constraint on the maximum delay for each stage, while minimizing a user-specified function of hardware resource cost, C, and throughput of the pipeline, E, namely, $f(E, C)$.

The following modifications were made to the simulated-annealing-based hardware allocation algorithm to synthesize pipelined datapaths.

(1) The algorithm begins with a serial pipeline schedule which does not violate the maximum stage delay constraint. This serial schedule is constructed by scheduling operations serially in a given stage and beginning another stage when the stage delay exceeds the maximum allowed value. Given a partition, hardware costs are calculated as before, treating every partition as a separate, two-dimensional placement and adding up all the hardware costs of each partition. This step is needed because hardware resources cannot be shared across the phases.

(2) Moves are then generated during the annealing as described in Section 3.3.2, interchanging and displacing operations, both within a stage as well as across adjacent stages. The moves are such that the precedence constraints between operations are not violated.

However, the maximum stage delay limit may be violated by a move. These violations are allowed in intermediate solutions but are penalized to ensure that they do not appear in the final result. Operations in the last phase may be displaced to a previously empty following phase, increasing the number of phases. The number of phases may also decrease during the annealing.

(3) Any event that prevents a pipeline from operating at the maximum possible rate is called *resynchronization*. It is assumed that each resynchronization delays the next task until the first initiation clock cycle after the completion of the current task. The throughput, E, of the pipeline is measured using the number of stages, k, the delay of the stages, d_i, and the expected resynchronization rate, ρ, using the equation shown below, which is similar to those derived in [19].

$$E \propto 1/(\ 1 + (\ \underset{i}{MAX}\ (\ d_i\ )\ .\ k - 1\ )\ \rho\ )$$

The tradeoff between delay and cost for single operations (Section 3.4) can also be made while synthesizing pipelined datapaths.

3.6.2 Examples

A example of pipelining a data flow specification with is illustrated in Figure 3.15. In Figure 3.15(a), the unpipelined data flow specification is given. The tradeoffs for the adders and multipliers specified as (cost, delay) number sets are given in Figure 3.15(b). Given these tradeoffs, along with a maximum stage delay limit of 100 ns, 20 ns latch delay and a latency of 2, the program was asked to find the cheapest possible schedule with a maximum of 6 stages. The schedule synthesized is shown in Figure 3.15(b). $+_f$ denotes a fast adder and $+_s$ a slow adder (similar subscripts apply to multiply). Both kinds of adders and multipliers have been used to maximum advantage. Since the latency is 2, resources can be shared across stages 1 and 2, 3 and 4, 5 and 6 so two $+_s$, one $+_f$, three $*_s$ and one $*_f$ unit(s) are required adding up to a total cost of 12.5 units. The multiplier in stages 5-6 has to be a $*_f$ unit since **a4** has to be computed after computing **z4** in stage 6. The CPU time required to synthesize

this pipeline schedule was 2 minutes on the VAX 11/8650.

The second example is the MOSFET model evaluator of Section 3.5. The datapath synthesized for a two-stage pipeline with latency 1 is shown in Figure 3.16. The statistics of this datapath are compared with those of datapaths 1 and 3 (Figure 3.14) in Table 3.2. Datapath 3

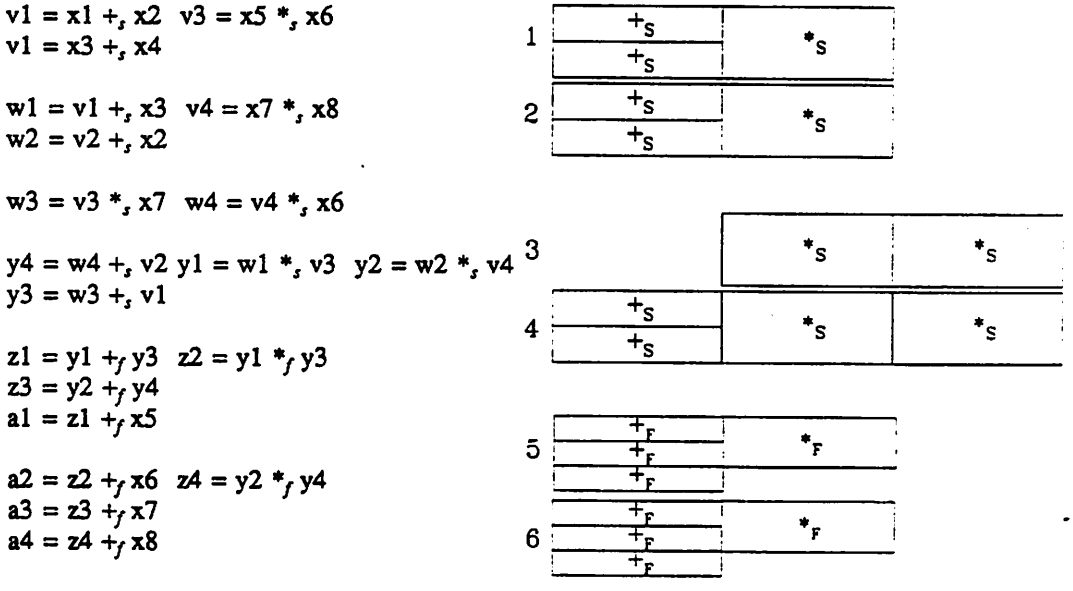
---

v1 = x1 + x2 v2 = x3 + x4 v3 = x5 * x6 v4 = x7 * x8
w1 = v1 + x3 w2 = v2 + x2 w3 = v3 + x7 w4 = v4 + x6
y1 = w1 + v3 y2 = w2 + v4 y3 = w3 + v1 y4 = w4 + v2
z1 = y1 + y3 z2 = y1 * y3 z3 = y2 + y4 z4 = y2 * y4
a1 = z1 + x5 a2 = z2 + x6 a3 = z3 + x7 a4 = z4 + x8

(a) Input specification

OPERATOR	Cost	Delay
$+_s$	1.0	40ns
$+_f$	1.5	25ns
$*_s$	2.0	80ns
$*_f$	3.0	50ns

(b) Cost-Delay tradeoffs



(c) Synthesized pipeline schedule

Fig. 3.15

---

is a parallel implementation of the MOSFET model routine with 2 ALUs, whereas Datapath 4 is a pipelined implementation with 2 stages (each with a single ALU). Datapath 4 has higher throughput (assuming no resynchronization) but is slightly larger in area. The links shown in dotted lines in Figure 16 correspond to data transfers occurring from the registers in the first

DP	execution time	#reg	#bus	#link	estimated area	CPU time
1	1.0	21	2 + 1*	54	1.0	10.1m
3	0.54	21	4 + 1*	77	2.5	11.2m
4	0.50**	24	4 + 1*	70	2.6	13.1m

* memory bus
** signifies throughput rather than execution time

Table 3.2 Serial, Parallel and Pipelined datapath statistics

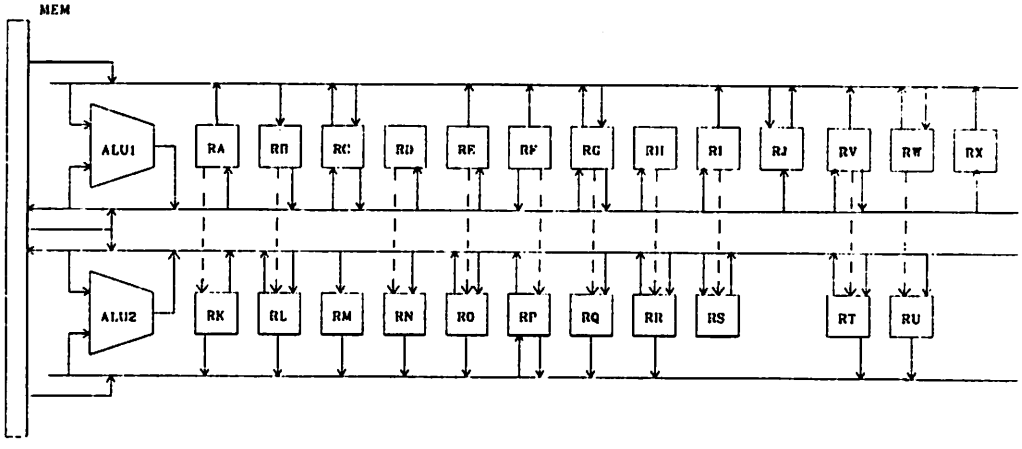


Fig. 3.16 Datapath 4

pipeline stage to registers in the second pipeline stage.

3.7 Conclusions

In this chapter, a novel method for synthesizing datapaths from behavioral descriptions has been presented. The entire allocation process in datapath synthesis has been formulated as a two-dimensional placement problem of micro-instructions in space and time. This formulation allows simultaneous cost-constrained allocation of registers, arithmetic units, interconnect (buses and links) while trading off hardware cost against execution speed. A simulated-annealing-based solution to the datapath synthesis problem which has achieved excellent results has been presented. Unlike previous approaches, this approach can operate under a wide variety of user-specified constraints on hardware resources and costs. Finally, this simulated-annealing-based approach has been extended to synthesize pipelined datapaths.

CHAPTER 4

Control Synthesis: State Assignment

4.1 Introduction

The hardware allocation process in datapath synthesis produces a register-transfer level structural specification of the circuit. The registers, arithmetic units and the interconnections implementing the data transfers between them are now specified. The next step of *control synthesis* is to synthesize a finite state machine (FSM) controller which in conjunction with the datapath can execute the given behavioral input description.

Control synthesis involves three steps as illustrated by Figure 2.1. First, a specification of the FSM controller is derived from the input description and the datapath specification. In the behavioral synthesis system described here, the entire *control specification* is a by-product of the datapath synthesis step. Given the two-dimensional placement of micro-operations, the specification of the finite state machine controller is easily derived. This derivation step is described in Section 4.3 after some basic definitions are given in Section 4.2.

The FSM controller can be implemented using combinational logic and feedback registers as shown in Figure 4.1. The registers store the internal states of the FSM. Traditionally, the combinational logic has been implemented using micro-coded Read Only Memories (ROMs) or Programmable Logic Arrays (PLAs). The specification of the PLA or ROM is dependent upon the encoding of internal states in the FSM. The process of encoding these states, called *state assignment*, is the second step in control synthesis. New techniques for state assignment to produce minimum-area FSM implementations are presented in Section 4.4.

More recently, FSMs have been implemented using *multi-level* combinational logic. Multi-level implementations of logic functions can be substantially smaller and faster than

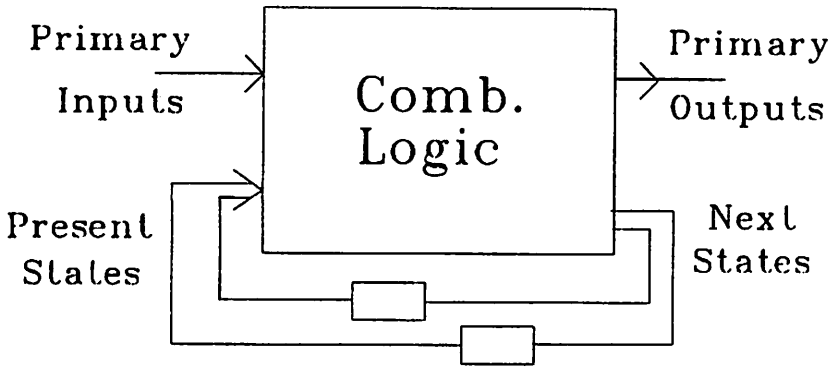


Fig. 4.1 Finite State Machine Implementation

corresponding PLA (two-level) implementations. In both cases, complex optimization strategies are required for area-efficient and/or time-efficient implementations. The final step in control synthesis, *logic optimization*, incorporates these techniques.

4.2 Preliminaries

In this section, some basic definitions are given for terms that will be used in the sequel. The object being defined appears in bold type.

A **variable** is a symbol representing a single coordinate of the Boolean space (e.g. a). A **literal** is a variable or its negation (e.g. a or $\overline{a}$). A **cube** is a set C of literals such that $x \in C$ implies $\overline{x} \notin C$ (e.g., $\{a,b,\overline{c}\}$ is a cube, and $\{a,\overline{a}\}$ is not a cube). A cube represents the conjunction of its literals. The trivial cubes, written 0 and 1, represent the Boolean functions 0 and 1 respectively. An **expression** is a set f of cubes. For example, $\{\{a\},\{b,\overline{c}\}\}$ is an expression consisting of the two cubes $\{a\}$ and $\{b,\overline{c}\}$. An expression represents the disjunction of its cubes.

A Finite State Machine (FSM) is represented by two equivalent structures:

(1) Its **State Transition Graph** $G(V, E, W(E))$ where V is the set of vertices corresponding to the set of states S, where $\|S\|=N_s$ is the cardinality of the set of states of the FSM, an edge (v_i, v_j) joins v_i to v_j if there is a primary input that causes the FSM to evolve from state v_i to state v_j, and $W(E)$ is a set of labels attached to each edge, each label carrying the information of the value of the input that caused that transition and the values of the primary outputs corresponding to that transition.

(2) Its **State Transition Table** $T(I, S, O)$ where I is the set of inputs, S is the set of states as above, and O is the set of outputs. It is assumed that the primary inputs and outputs of the FSM are in Boolean form. A row of the table corresponds to an edge in the State Transition Graph. The table has as many rows as edges of State Graph and as many columns as

$$N_i + N_o + 2$$

where N_i is the number of bits used to encode the inputs, N_o is the the number of bits used to encode the outputs, and 2 refers to the present state and the next state. The matrix has Boolean entries for the inputs and outputs and "symbolic" entries for the columns corresponding to the present and the next states, carrying the name of the present state and of the next state respectively. The rows of the matrix are divided into two fields: the first field contains the input pattern and the names of the present state, the second field contains the output pattern and the names of the next state. Note that the input pattern may contain don't care entries.

The number of bits required to encode the N_s states in the machine is denoted N_b, $N_b \geq \lceil \log(N_s) \rceil$. The number of encoding bits used varies – using a larger N_b can result is a smaller logic implementation.

4.3 Control Specification

The inputs to this phase of synthesis are the structural specification of the datapath and the two-dimensional placement of operations. The output of this phase is the State Transition Table of a finite state machine which controls the datapath so as to execute the given behavioral description.

A FSM can be specified by a State Transition Table or State Transition Graph. Every time slot in the two-dimensional placement corresponds to a state in the State Transition Table of the FSM. The FSM is specified as a Mealy machine, whose outputs are a function of the inputs as well as the internal state of the machine. The outputs of the FSM are the load signals to the different links in the datapath as well as the selection signals to the different ALUs. Status signals from the ALU are possible inputs to the FSM. External inputs may also exist. Depending on the input combinations, different transitions occur in the FSM.

An example of a State Transition Table of a FSM controller is given in Figure 4.2(c). This specification has been derived from the two-dimensional placement of operations and datapath derived earlier and shown in Figure 3.13. These figures have been reproduced as Figure 4.2(a) and (b) below. In this case, the FSM is implemented as a Mealy machine to establish a one-to-one correspondence between the states of the FSM and the time slots in the two-dimensional placement. The edges in the State Transition Table can then be enumerated. Each edge asserts the outputs required by the micro-instructions in its time slot. For example, the first edge in the State Transition Table of Figure 4.2(c) asserts the load signals required for the execution of the two micro-operations in the first time slot of the two-dimensional placement. A single external input to start computations was assumed in this case. The selection signals to the two ALUs have been specified symbolically (e.g. NOP stands for no operation, ADD for addition, SUB for minus). Binary codes are assigned to both the states and these symbolic outputs so as to minimize controller area in the next step of synthesis. Since, conditional branches existed in the original input description, some of the states have more than one

fanout edge. For example, two edges with different input and output combinations fan out from state *st1* to state *st2*. The outputs of these edges will cause different micro-operations to be performed by the datapath. Depending on the input control signal (the 2nd input in this case) one and only one of the sets of micro-operations will be performed.

4.4 State Assignment

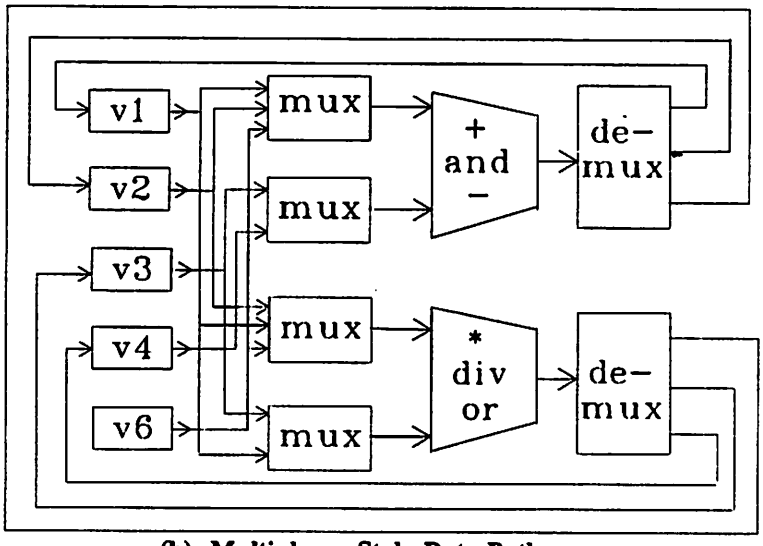
4.4.1 Introduction

Given the State Transition Table of the FSM controller, the machine can be implemented using combinational logic and feedback registers as shown in Figure 4.1. Binary codes are assigned to the internal states and symbolic inputs/outputs of the FSM to provide a specification for the combinational logic, which can then be implemented as a PLA or a multi-level logic network. State assignment and input/output encoding can profoundly affect controller area. The goal of this step is to find an encoding of states, inputs and outputs of the FSM such that the area of the FSM *after* combinational logic optimization is minimized. This is illustrated in Figure 4.3.

If the FSM is being implemented by a PLA then, after state assignment, the PLA is minimized using two-level logic minimization programs like ESPRESSO [23]. The number of product terms in the optimized PLA is significantly affected by the encoding of internal states in the FSM. This is illustrated in Figures 4.4 and 4.5. An example FSM specification is shown in Figure 4.4(a). The assignment of codes to the states of the FSM given in Figure 4.4(b) produces a PLA with eight product terms in Figure 4.4(c) after logic minimization. A different assignment of codes, given in Figure 4.5(a), produces a six product term PLA (Figure

(add v2 v3 v1)	(divide v2 v3 v4)
[(add v1 v4 v6) (minus v1 v4 v6)]	
	[(mult v6 v3 v6) (divide v3 v6 v3)]
	(mult v2 v3 v6)
(and v6 v4 v2)	(or v6 v1 v3)

(a) 2-Dimensional Placement



(b) Multiplexor-Style Data Path

```
0- st0  st0 NOP NOP 000000
1- st0  st1 ADD DIV 111100
-0 st1  st2 ADD NOP 100101
-1 st1  st2 ADD NOP 100101
-0 st2  st3 NOP DIV 001001
-1 st2  st3 NOP DIV 001001
-- st3  st4 AND OR  111101
```

(c) State Transition Table of FSM Controller

Fig. 4.2

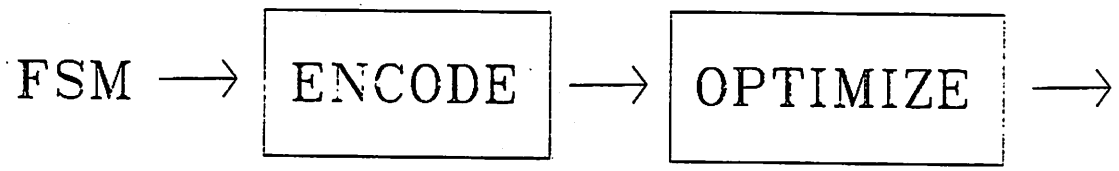


Fig. 4.3 Encoding and Logic Optimization

```
000 st0  st0 0101
100 st0  st1 0101
010 st0  st3 0101
--1 st0  st0 1010
100 st1  st1 0101
0-0 st1  st0 0101
110 st1  st2 0101
--1 st1  st0 1010
110 st2  st2 0101
100 st2  st1 0101
```

(a) Example FSM

```
st0 -> 01   st1 -> 10
st2 -> 00   st3 -> 11
```

(b) State Assignment

```
010 01  10 0000
100 -0  10 0000
--1 10  01 1010
--1 01  01 1010
100 0-  10 0101
0-0 10  01 0101
0-0 01  01 0101
1-0 -0  00 0101
```

(c) Minimized PLA implementation

Fig. 4.4

st0 -> 00 st1 -> 01
st2 -> 11 st3 -> 10

(a) State Assignment

```
010 00  10 0000
110 -1  10 0000
--1 0-  00 1010
100 0-  01 0101
0-0 0-  00 0101
1-0 -1  01 0101
```

(b) Minimized PLA implementation
Fig. 4.5

4.5(b)).

4.4.2 Need for new techniques of State Assignment

Previous work in *automatic* FSM state assignment has been directed at the minimization of the number of product terms in a sum-of-products form of the combinational logic [24] [25] [116] [117] [26] [118] [28] and, hence, the results obtained are relevant for the cases where the combinational logic is implemented using Programmable Logic Arrays (PLAs). In practice, most large FSMs cannot be synthesized as a single PLA for performance reasons – multi-level logic implementations are generally used for smaller delays or smaller areas (or both). Results using manual state assignment have shown that existing automatic state assignment techniques are inadequate for producing optimal multi-level logic implementations [119]. This is illustrated in Figure 4.6-8. Using a state assignment program targeted toward PLA implementations the states of the FSM in Figure 4.6 are given the codes in Figure 4.7(a). This encoding produces a six product term PLA (Figure 4.7(b)) after two-level logic minimization. After multi-level logic optimization, the resulting network contains 16 gates and is shown in Figure 4.7(c). A different assignment of codes (Figure 4.8(a)) produces a larger PLA with seven product terms (Figure 4.8(b)), but a smaller multi-level logic network with 15 gates (Figure 4.8(c)). This example illustrates the need for state assignment techniques targeted

```
-0 st0   st0 0
11 st0   st0 0
01 st0   st1 -
0- st1   st1 1
11 st1   st0 0
10 st1   st2 1
1- st2   st2 1
00 st2   st1 1
01 st2   st3 1
0- st3   st3 1
11 st3   st2 1
```

Fig. 4.6 Example FSM

toward a different objective, namely, optimal *multi-level* implementations of FSM combinational logic.

4.4.3 State Assignment for Multi-level Logic Implementations

In the following sections, a strategy is presented for finding a state assignment of a FSM which minimizes an estimate of the area used by a multi-level implementation of the combinational logic[2]. The estimate considered here is consistent with the estimate used by multi-level logic optimization algorithms [76] [6] [77]: the *number of literals in a factored form* of the logic. Algorithms have been developed, which produce a state assignment that heuristically minimizes the number of literals in the resulting combinational logic network *after* multi-level logic optimization.

Multi-level logic optimization programs like MIS [6] and SOCRATES [77] primarily use algebraic techniques for factorizing and decomposing the Boolean equations by identifying common sub-expressions. The heuristics developed are based on *maximizing the number and size of common sub-expressions* and *minimizing the number of literals* in the Boolean equations that describe the combinational logic part of the FSM, after the states have been

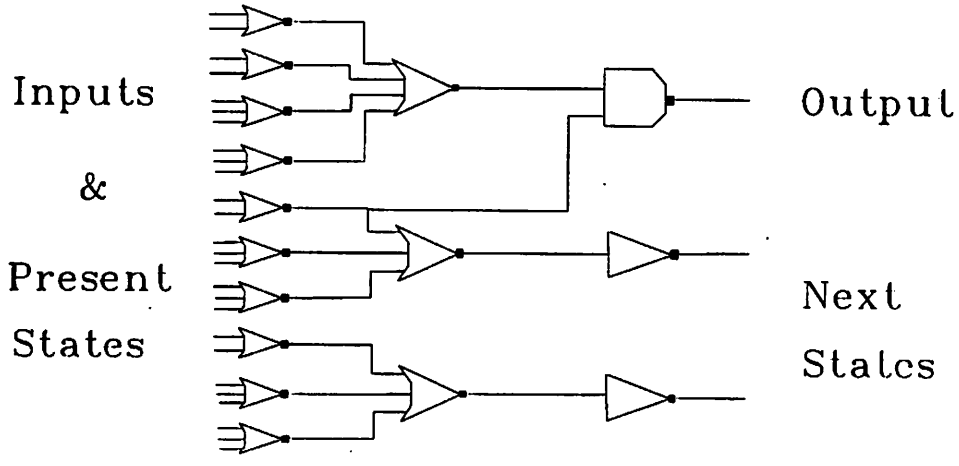
2 Note that this work does not consider logic delay as an explicit factor in the optimization.

st0 -> 00 st1 -> 01
st2 -> 11 st3 -> 10

(a) State Assignment

```
10 -1   10 0
11 1-   01 0
0- 10   10 1
01 0-   01 1
-0 -1   01 1
-1 1-   10 1
```

(b) Minimized PLA implementation



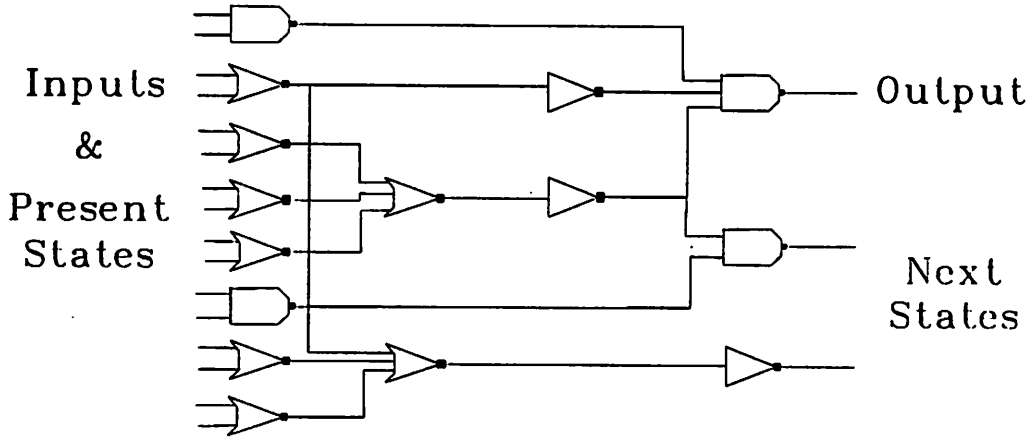
(c) Optimized Multi-level Implementation
Fig. 4.7

st0 -> 00 st1 -> 10
st2 -> 01 st3 -> 11

(a) State Assignment

10 10 01 1
0- 11 01 0
01 -- 10 0
1- 01 01 1
0- 1- 10 1
-1 -1 01 1
0- -1 10 1

(b) Minimized PLA implementation



(c) Optimized Multi-level implementation
Fig. 4.8

encoded, but *before* logic optimization. The state assignment algorithms find pairs or clusters of states which, if kept minimally distant in the Boolean space representing the encoding, result in a large number of common sub-expressions in the Boolean network.

I have obtained results over a wide range of benchmarks which illustrate the efficacy of these techniques. Literal counts averaging 20-40% less than other state assignment techniques have been obtained. Some erroneous results using these algorithms, which have been imple-

mented in the program MUSTANG, have been published in the literature recently [104]. The errors were later corrected. MUSTANG outperforms random assignment (best of 10 runs) by 27% over a wide range of examples.

In Section 4.4.4, the nature of the problem is described and the basic approach followed to obtain a good state assignment is presented. In Section 4.4.5, two algorithms are presented. The embedding algorithm used is described in Section 4.4.6. Results on the benchmark examples are presented in Section 4.4.7.

4.4.4 The Basic Approach

The state assignment problem consists of assigning a string of bits (a code) to each of the states the machine might take, so that no two states have the same code. After a code has been assigned, the FSM can be implemented trivially once the storage elements (flip-flops) have been chosen, with a PLA. For example, assume that the storage elements are D flip-flops (one per bit). Then, each edge (v_i,v_j) of the State Transition Graph or row of the State Transition Table, corresponds to a product term, with the input part represented by the bits specified in the label $w((v_i,v_j))$ for the primary input and the bits forming the code for v_i (the present state), and the output part represented by the bits forming the code for v_j and the bits specified in $w((v_i,v_j))$ for the primary outputs. As mentioned earlier, this representation of the FSM can be optimized using a two-level logic minimizer as ESPRESSO [23] to reduce the number of product terms needed to implement the logic function. Of course, different encoding of the states yield different logic functions as illustrated by Figure 4.4-5. It is of great interest to assign codes to states so that the final optimized PLA has as few product terms as possible. Algorithms have been proposed that solve this problem by using a symbolic optimization step to determine a set of constraints on the encoding to guarantee that certain product terms could be eliminated in the final implementation [26] [28]. However, in some cases, the size of the PLA remains too large to satisfy timing or area constraints. In this case, a multi-level implementation of the logic is a better choice. The two-level logic description may be

then mapped into a multi-level implementation by factoring and decomposing the logic functions corresponding to the outputs. According to the particular target technology (e.g. CMOS standard cells, CMOS static gates laid out in the gate-matrix style or Weinberger arrays), a decomposition and factorization will be more effective than others.

A number of algorithms have been proposed that perform this step effectively (e.g. [76] [6] [77]). These algorithms represent the logic to be implemented as a *Boolean network*, i.e., a directed graph where each node corresponds to a logic function with *one output* and an arc is provided between two nodes if the output of one function is an input of the other. Because the output of each node is unique, a node and an output are in one-to-one correspondence.

In principle, these algorithms should use a cost function that depends on the final implementation technology. However, due to the many different target technologies used, it is very difficult to identify a single meaningful cost function that could be optimized effectively. Thus, an estimate for the final area is generally used. An estimate that has been used successfully in many cases is the number of literals in a factored form of the logic function. Then, *the optimal state assignment problem can be formulated as the problem of assigning codes to the states so that the total number of literals in the factored form of the logic function is minimized.*

It is certainly difficult to devise an exact measure of how many literals a particular state assignment will yield after multi-level logic optimization has been carried out, because of the great complexity of the algorithms used for this purpose [6].

The key point in the proposed algorithms for the state assignment problem is the model used to predict the results obtained by the multi-level logic optimizer after the encoding has been performed. In this case, I focused on the operations of MIS [6], the Berkeley logic optimizer.

The algorithms in MIS [6] can be classified in two categories: algebraic and Boolean methods. It is very difficult to model the optimization achieved by MIS with the use of

Boolean methods, while it is feasible to predict at least some of the operations that the algebraic division algorithms use to minimize the logic.

Among the several algebraic optimization algorithms used by MIS are:

(1) Factoring of logic equations.

(2) Common sub-expression identification.

(3) Common cube extraction.

These three techniques are illustrated in Figure 4.9. The latter two techniques are *algebraic division* techniques; expressions are divided by common cubes or sub-expressions in order to produce smaller expressions with new intermediate variables. Common cube extraction is actually a subset of common sub-expression identification – a sub-expression may be a single cube.

The algorithm presented in this dissertation tries to *maximize the number of common cubes* that can be found by the logic optimization algorithms in the encoded two-level network. Maximizing the number of common cubes results in a large number of good factors that can be extracted during optimization to produce a reduced literal multi-level

---

Factoring:
ace + bce + de -> ((a + b) c + d) e

Common sub-expression identification:
ace + bce + de -> *s*ce + de
ade + bde + af -> *s*de + af
s = a + b

Common cube identification:
ace + bce + de -> *t*c + *u*c + de
ade + bde + af -> *t*d + *u*d + af
t = be *u* = ae

Fig. 4.9 Factoring, Common sub-expression and Common cube identification

---

representation.

There are two basic processes behind the influence of state assignment on the number of common cubes in the encoded State Transition Table (STT), a two-level representation, which is the starting point for multi-level logic optimization.

To begin, consider the second field (the present state field) in the STT of the machine shown in Figure 4.6. If the states *st*0 and *st*2 are assigned codes of distance N_d, then the lines of the next state *st*1 will have a common cube with $N_b - N_d$ literals (due to edges 3 and 8 in the STT). Similar relationships exist between other sets of states.

Now consider the third field (the next state field) of distance N_d. In this case, the present state *st*1 becomes a common cube for $N_b - N_d$ next state lines *whatever its code is* (due to edges 5 and 6 in the STT). The number of literals in the common cube is, of course, N_b. Again, similar relationships exist between other sets of states in the machine.

The input and output spaces (the first and fourth fields) also have an influence on the number of common cubes after encoding. If two different input combinations, i_1 and i_2, produce the same next state from different or same present states, then there is a common cube corresponding to $i_1 \cap i_2$ in the input space. Similarly, outputs asserted by different present states have common cubes corresponding to their intersections.

Given any machine, there are a large set of relationships between state encoding and the number/size of common cubes in the network prior to logic optimization. The reduction in literal count or "gains" that can be obtained by coding a given pair of states with close codes, so single/multiple occurrences of common cubes can be extracted, can be estimated. Given these gains for each pair of states, one can attempt to find an encoding which maximizes the overall gain.

There arises a complication in gain estimation. First, while the number of literals in the common cubes can be found exactly, the number of *occurrences* of these cubes in the logic function depends on the encoding of the next states. In the example above, assume that *st*0

was assigned *111* and *st*1 was assigned *110*. There is a common cube *11* (with 2 literals) for the next state lines but the number of occurrences of this common cube depends on the number of 1's in the code of *st*4 (which is not known at this time). This problem is alleviated by treating the gains as *relative* merits rather than absolute and using an average-case analysis (see Section 4.4.5.1).

It should be noted that these statically-computed gains interact. Extracting some common cubes can *increase the number of logic levels* (to the outputs) of other common cubes and can also decrease the gain in extracting them. For instance, a sequence of two cube extractions on a two-level network can produce a three or a four-level network. Statically computing gains and maximizing the number of common cubes works because, given a particular encoding, the optimal *sequence* of cube extractions to produce a minimal-literal multi-level network can be found by the logic optimizer. The goal then is to find an encoding that maximizes the number of common cubes in the initial two-level network.

The approach used is to build a graph $G_M(V,E_M,W(E_M))$ where V, the set of nodes in G_M, has a one-to-one correspondence with the states of the finite state machine, E_M is a complete set of edges, i.e., every node is connected to every other node, and $W(E_M)$ represents the gains that can be achieved by coding the states joined by the corresponding arc as close as possible. These gains are statically and independently computed by enumerating the different relationships between the input, state and output spaces.

Then, the states are encoded, using this graph to provide the cost of an assignment of a state to a vertex of the Boolean hypercube.

A critical part of this approach is the generation of $W(E_M)$. I have experimented with two algorithms: one assigns the weights to the edges by taking into consideration the second and fourth fields of the State Transition Table, and is henceforth called *fanout-oriented*. The second algorithm assigns weights to the edges by taking into consideration the first and third fields and is henceforth called *fanin-oriented*.

The fanout-oriented algorithm attempts to *maximize the size* of the most frequently occurring common cubes in the encoded machine prior to optimization. The fanin-oriented algorithm attempts to *maximize the number of occurrences* of the largest common cubes in the encoded machine prior to optimization. These two algorithms are based on the two different processes behind the influence of state assignment on the number of common cubes in the network described earlier.

4.4.5 Algorithms for Graph Construction

In this section, both a fanout-oriented algorithm and a fanin-oriented algorithm are presented, which define a set of weights for the undirected graph $G_M(V,E,W(E_M))$, introduced earlier. The weights represent a set of closeness criteria for the states in the machine which reflect on the number of common cubes in the encoded machine prior to optimization. Both these algorithms have a time and space complexity polynomial in the number of inputs, outputs and states in the machine to be encoded. In the sequel, the two algorithms are described and analyzed.

4.4.5.1 A Fanout Oriented Algorithm

This algorithm works on the output and the fanout of each state. Present states which assert similar outputs and produce similar sets of next states are given high edge weights (and eventually close codes) so as to *maximize the size of common cubes* in the output and next state lines.

The algorithm proceeds as follows:

(1) Construct a complete graph $G_M(V,E_M,W(E_M))$, with the edge weight set, $W(E_M)$ empty. For each output, all the edges, $W(E)$, in the State Transition Graph G, are scanned to identify the nodes which assert that output. N_o sets of weighted nodes which assert each output are constructed. If a node asserts the same output more than once it has a correspondingly larger weight in the set.

(2) For each next state, sets of present states producing that next state are found (N_s sets are constructed).

The pseudo-code below illustrates these steps of the procedure. *nw* stores the weight of the nodes in each of the different sets.

```
for( i = 1 ; i ≤ N_o ; i = i + 1 ) {
  foreach( edges e(v_k, v_l) ∈ G ) {
    if ( W(e).output[i] is 1 ) {
      O_SET_i = O_SET_i ∪ v_k ;
      nw(O_SET_i, v_k) = nw(O_SET_i, v_k) + 1 ;
    }
  }
}
foreach( edges e(v_k, v_l) ∈ G ) {
  NS_SET_l = NS_SET_l ∪ v_k ;
  nw(NS_SET_l, v_k) = nw(NS_SET_l, v_k) + 1 ;
}
```

---

(3) Using these N_o *O_SET* and N_s *NS_SET* sets of nodes, $W(E_M)$ is constructed. The edge weight, *we*, is equal to the multiplication of the weights of the two nodes corresponding to the edge across all the sets. The weights corresponding to the next state sets have a multiplicative factor equal to the half the number of encoding bits, $N_b/2$. The reasoning behind the use of a multiplicative factor is given at the end of the section. The pseudo-code for the calculation of *we* is shown below.

```
foreach( (v_k, v_l) ∈ G_M ) {
  for( i = 1 ; i ≤ N_s ; i = i + 1 )
    we(e_M(v_k,v_l)) = we(e_M(v_k,v_l)) + nw(NS_SET_i, v_k) * nw(NS_SET_i, v_l) ;
  we(e_M(v_k,v_l)) = we(e_M(v_k,v_l)) * N_b/2 ;
  for( i = 1 ; i ≤ N_o ; i = i + 1 )
    we(e_M(v_k,v_l)) = we(e_M(v_k,v_l)) + nw(O_SET_i, v_k) * nw(O_SET_i, v_l) ;
}
```

The first step of the fanout-oriented algorithm entails enumerating the relationships between the present states and the output space. If two different present states assert an output, it is possible to extract a common cube corresponding to the intersection of the two state codes. By constructing the N_o different output sets and counting the number of times a pair of states occurs together in each output set, the algorithm effectively computes the number of occurrences of the common cube $X \cap Y$, for all states X and Y.

In the second step, the next states produced by each pair of present states are compared. A state pair which produces the same next state has an associated common cube corresponding to the pair-wise intersection. The number of occurrences of this common cube is dependent on the number of 1s in the code of the next state and therefore cannot be estimated exactly (unlike in the first step). It is assumed that the average number of 1s in a state's code is $N_b/2$. Since one is concerned with relative rather than absolute merits, the approximation that each common cube occurs in $N_b/2$ next state lines is a good one. Thus, a multiplying factor of $N_b/2$ is used in the second step. Ideally, this factor should be a function of the encoding and not a constant for all state pairs.

Given the number of occurrences of different common cubes in the machine, this algorithm assigns weights so as to maximize the size of the most frequently occurring cubes.

The graph generated by the fanout-oriented algorithm for the example FSM of Figure 4.6 is shown in Figure 4.10. The output set corresponding to the single output is $(st0^2, st1^3, st3^2)$, where the superscripts denote the weights $nw()$ for each state in the set. The next state sets are:

$$st0 \rightarrow (st0^2, st1^1)$$

$$st1 \rightarrow (st0^1, st1^1, st2^1)$$

$$st2 \rightarrow (st1^1, st2^1, st3^1)$$

$$st3 \rightarrow (st2^1, st3^1)$$

The weight of the edge between the states $st2$ and $st3$ with $N_b = 2$ is

$(1 \times 1 + 1 \times 1) \times N_b/2 + 3 \times 2 = 8$. Similarly, the other edge weights can be calculated.

4.4.5.2 A Fanin-Oriented Algorithm

The algorithm described above ignored the input space of the finite state machine. The algorithm works well for FSMs with a large number of outputs and small number of inputs. However, the number of input and output variables could both be quite large. In this section, a fanin-oriented algorithm is described, which operates on the input and fanin for each state. Next states which are produced by similar inputs and similar sets of present states are given high edge weights (and eventually close codes) so as to *maximize the number of common cubes* in the next state lines.

The algorithm proceeds as follows:

(1) The graph G_M is constructed. N_s sets of weighted next states which fan out from each present state in G are constructed as shown below. *nw* stores the weight of each node in all the sets.

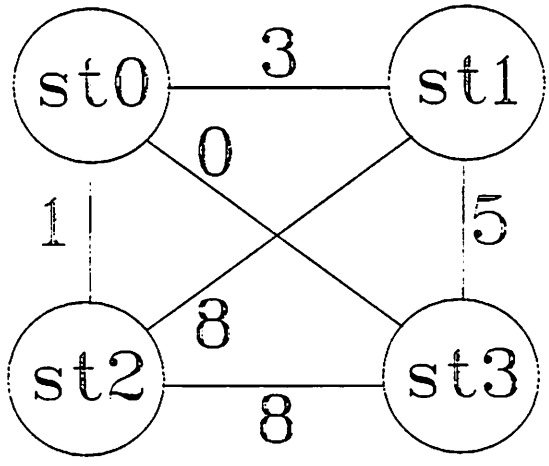


Fig. 4.10 Graph produced by fanout-oriented algorithm

```
foreach( edge e(v_k, v_l) ∈ G ) {
  PS_SET_k = PS_SET_k ∪ v_l ;
  nw(PS_SET_k, v_l) = nw(PS_SET_k, v_l) + 1 ;
}
```

(2) For each input, sets of next states are identified which are produced when the input is 1 and when the input is 0. $2*N_i$ such sets are constructed as shown below.

```
for( i = 1 ; i ≤N_i ; i = i + 1 ) {
  foreach( edge e(v_k, v_l) ∈ G ) {
    if ( W(e).input[i] is 1 ) {
      I_SET^ON_i = I_SET^ON_i ∪ v_l ;
      nw(I_SET^ON_i, v_l) = nw(I_SET^ON_i, v_l) + 1 ;
    }
    if ( W(e).input[i] is 0 ) {
      I_SET^OFF_i = I_SET^OFF_i ∪ v_l ;
      nw(I_SET^OFF_i, v_l) = nw(I_SET^OFF_i, v_l) + 1 ;
    }
  }
}
```

(3) The weights on the edges in the graph, *we*, are found using the N_i I_SET^{ON}, N_i I_SET^{OFF} and N_s PS_SET sets of nodes as illustrated in the pseudo-code below. Between each pair of nodes in G_M, an edge with weight equal to the multiplication of the weights of the two nodes across all the present state sets (scaled by N_b) and all the input sets is added.

```
foreach( (v_k, v_l) ∈ G_M ) {
  for( i = 1 ; i ≤ N_s ; i = i + 1 )
    we(e_M(v_k,v_l)) = we(e_M(v_k,v_l)) + nw(PS_SET_i, v_k) * nw(PS_SET_i, v_l) ;
  we(e_M(v_k,v_l)) = we(e_M(v_k,v_l)) * N_b ;
  for( i = 1 ; i ≤ N_i ; i = i + 1 ) {
    we(e_M(v_k,v_l)) = we(e_M(v_k,v_l)) + nw(I_SET^ON_i, v_k) * nw(I_SET^ON_i, v_l) ;
    we(e_M(v_k,v_l)) = we(e_M(v_k,v_l)) + nw(I_SET^OFF_i, v_k) * nw(I_SET^OFF_i, v_l) ;
  }
}
```

The first step of the fanin-oriented algorithm entails enumerating the relationships between the input and next state space. A next state produced by two different input combinations i_1 and i_2 has a common cube $i_1 \cap i_2$. The size of this cube can be found. By constructing the $2 \times N_i$ different input sets and counting the number of times a pair of states occurs together in each input set, the algorithm computes similarity relationships between all next state pairs in terms of the inputs. Giving next state pairs that are produced by similar inputs high edge weights will result in maximizing the number of occurrences of the largest common input cubes in the next state lines.

In the second step, the present states producing each pair of next states are compared. If two different next states are produced by the same present state, the state is common to some next state lines. The number of occurrences of this common cube is dependent on the intersection of the two next state codes. To maximize the number of occurrences of these cubes, next state pairs which have many common present states are given correspondingly high edge weights. Since each of these cubes have N_b literals (as opposed to a single literal for a single input), a multiplying factor of N_b is used while combining the weights computed in the two steps.

Given the sizes of the different common cubes in the machine, this algorithm assigns weights so as to maximize the number of occurrences of these cubes.

The graph generated by the fanin-oriented algorithm for the example FSM of Figure 4.6 is shown in Figure 4.11. As can be seen, the weights of the edges in the graph are different from those generated by the fanout-oriented algorithm (Figure 4.10). Here the input sets are:

$$i_1(0) \rightarrow (st1^3, st3^2)$$

$$i_1(1) \rightarrow (st0^2, st2^3)$$

$$i_2(0) \rightarrow (st0^1, st1^1, st2^1)$$

$$i_2(1) \rightarrow (st0^2, st1^1, st2^1, st3^1)$$

The present state sets are:

$$st0 \rightarrow (st0^2, st1^1)$$

$$st1 \rightarrow (st0^1, st1^1, st2^1)$$

$$st2 \rightarrow (st1^1, st2^1, st3^1)$$

$$st3 \rightarrow (st2^1)$$

The weight of the edge between $st0$ and $st1$ for $N_b = 2$ is $(1 \times 1 + 2 \times 1) + N_b \times (2 \times 1 + 1 \times 1) = 9$. The other edge weights are calculated in a simi-

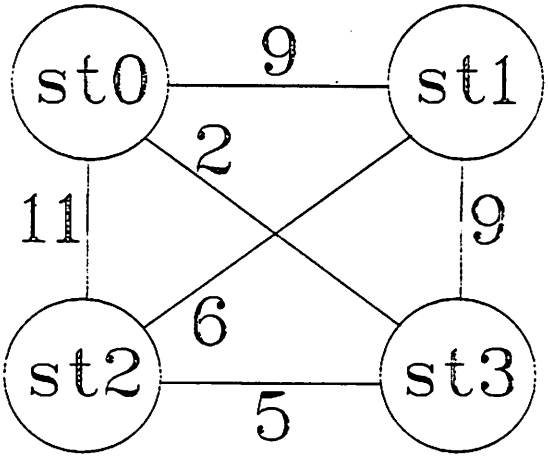


Fig. 4.11 Graph produced by fanin-oriented algorithm

lar fashion.

4.4.6 The Embedding Algorithm

The algorithms presented above generate a graph and a set of weights, like the graphs of Figure 5 and Figure 6, to guide the state encoding process. The problem now is to assign the actual codes to states according to the analysis performed by the fanin and the fanout-oriented algorithms. This problem is a classical combinatorial optimization problem called *graph embedding* [120]. Here, G_M has to be embedded in a Boolean hypercube so that the adjacency relations identified by G_M are satisfied in an optimal way. Unfortunately, this problem is NP-complete and there is little hope to solve it exactly in an efficient way. Thus, a heuristic approach to this embedding problem is used, that has given satisfactory results for the state assignment problem dealt with here.

The heuristic algorithm is called *wedge clustering*. This algorithm is used to assign codes to the nodes in G_M to minimize

$$\sum_{i=1}^{N_s} \sum_{j=i+1}^{N_s} we(e_M(v_i, v_j)) * dist(\ enc(\ v_i\ ),\ enc(\ v_j\ )\ )$$

where the v_k are the vertices in G_M, $we(e_M(v_k, v_l))$ is the weight of the edge, e, between vertices v_k and v_l, $enc(\ v_k\ )$ is the encoding of vertex v_k. The function $dist()$ returns the distance between two binary codes.

The graphs generated by the fanout and fanin-oriented algorithms have a certain structure associated with them, especially for large machines. In these graphs, typically small groups of states exist that are strongly connected internally (edges between states in the same group have high weights) but weakly connected externally (edges between states not in the same cluster have low weights). The embedding heuristic has been tailored to meet the requirements of this particular problem. The heuristic exploits the nature of the graph by attempting to identify strongly connected clusters and assigning states within each cluster with uni-distant codes.

The embedding algorithm proceeds as follows. Clusters of nodes with the cardinality of the cluster no greater than $N_b + 1$ and consisting of edges of maximum total weight are identified in G_M. Given G_M, the identification of these clusters is as follows – A node, $v_1 \in G_M$, with the maximum sum of weights of any N_b connected edges is identified. The N_b nodes, $y_1, y_2, .. y_{N_b}$ which correspond to the N_b edges from v_1 and v_1 itself are assigned minimally distant codes from the pool of unassigned codes (v_1 may have been assigned already, so may the other y_i). A maximum of N_b nodes are chosen so the y_i can be (possibly) assigned uni-distant codes from v_1. After the assignment, v_1 and all the edges connected to v_1 have been deleted from G_M and the node selection/code assignment process is repeated till all the nodes are assigned codes. The pseudo-code below illustrates the procedure:

$GG = G_M$
while(GG is not empty) {
  Select $v_1 \in GG$, $y_i \in GG$ so $\sum_{i=1}^{N_b} we(e_M(v_1, y_i))$ is maximum ;
  assign the y_i and v_1 minimally distant codes from unassigned codes ;
  $GG = GG - v_1$;
}

The following optimality result about the embedding heuristic can be proven. For convenience in notation, assume that $we(y_i, y_i) = 0 \; \forall \; i \quad \exists \; y_i \in G_M$.

Theorem 4.1: At a given iteration, if the N_b states $y_1, y_2, .. y_{N_b}$ are given uni-distant codes from the selected state v_1 and if

$$we(e_M(v_1, y_i)) \geq we(e_M(y_i, y_j)) + we(e_M(y_i, y_k)) \quad 1 \leq i, j, k \leq N_b, \; i \neq j \neq k \qquad (4.1)$$

then *the assignment is optimum* for this cluster of $N_b + 1$ states, i.e.

$$\sum_{i=1}^{N_b} we(e_M(v_1, y_i)) * dist(enc(v_1), enc(y_i)) + \sum_{i=1}^{N_b} \sum_{j=i+1}^{N_b} we(e_M(y_i, y_j)) * dist(enc(y_i), enc(y_j))$$

is minimum.

Proof: We have to prove that no assignment of codes to states can produce a cost which is

less than the cost, $C(v_1)$, produced by assigning the N_b states with uni-distant codes from v_1. We have:

$$C(v_1) = \sum_{i=1}^{N_b} we(e_M(v_1, y_i)) + 2 * \sum_{i=1}^{N_b} \sum_{j=i+1}^{N_b} we(e_M(y_i, y_j))$$

since the $y_1, ..y_{N_b}$ have uni-distant codes from v_1 and therefore are distance-2 from each other. To decrease the cost, the distances between the codes of the y_i have to be reduced from 2 to 1. This can only be done at the expense of an increase in the distance between some of the y_i and v_1 from 1 to 2. There are three possible ways of doing so. First, any state y_S can be selected from $y_1, ..y_{N_b}$ and code the rest of the y_i and v_1 can be encoded with uni-distant codes from y_S. Without loss of generality, assume y_1 was selected. We know, since v_1 was selected initially, that

$$\sum_{i=1}^{N_b} we(e_M(v_1, y_i)) \geq \sum_{j=1}^{N_b} we(e_M(y_k, y_j)) + we(e_M(y_k, v_1)) \quad \forall\ k \tag{4.2}$$

Using (2) above, it can easily be shown that:

$$C(y_1) = \sum_{i=2}^{N_b} we(e_M(y_S, y_i)) + we(y_1, v_1))$$

$$+ 2 * \sum_{i=2}^{N_b} \sum_{j=i+1}^{N_b} we(e_M(y_i, y_j)) + 2 * \sum_{i=2}^{N_b} we(e_M(v_1, y_i)) \geq C(v_1)$$

and similarly for $C(y_2), .. C(y_{N_b})$.

The second alternative in assigning codes is to select a y_S and assign it a code which is uni-distant from two other y_i (Only two y_i can be chosen since the y_i are distance-2 from each other). This code will be distance-3 from the unselected y_i and will be distance-2 from v_1. Without loss of generality, assume that y_1 was selected and assigned a uni-distant code from y_2 and y_3. We have:

$$C(y_1) = \sum_{i=2}^{N_b} we(e_M(v_1, y_i)) + \sum_{i=2}^{3} we(e_M(y_1, y_i)) + 2 * we(e_M(v_1, y_1)) + 2 * we(e_M(y_2, y_3)) +$$

$$2 * \sum_{i=4}^{N_b} \sum_{j=i+1}^{N_b} we(e_M(y_i, y_j)) + 2 * \sum_{i=2}^{3} \sum_{j=i+1}^{N_b} we(e_M(y_i, y_j)) + 3 * \sum_{i=4}^{N_b} we(e_M(y_1, y_i))$$

Expanding $C(v_1)$, we have:

$$C(v_1) = \sum_{i=2}^{N_b} we(e_M(v_1, y_i)) + 2 * \sum_{i=2}^{3} we(e_M(y_1, y_i)) + we(e_M(v_1, y_1)) + 2 * we(e_M(y_2, y_3)) +$$

$$2 * \sum_{i=4}^{N_b} \sum_{j=i+1}^{N_b} we(e_M(y_i, y_j)) + 2 * \sum_{i=2}^{3} \sum_{j=i+1}^{N_b} we(e_M(y_i, y_j)) + 2 * \sum_{i=4}^{N_b} we(e_M(y_1, y_i))$$

Canceling terms from $C(y_1)$ and using relation (1), shows that $C(y_1) \geq C(v_1)$.

The third alternative in assigning codes is to select a state y_S and $2 < p < N_b - 1$ states from the remaining y_i and make these p states uni-distant from y_S. y_S will be uni-distant from v_1, the p states will be distance-2 from v_1 and will be distance-2 from each other. If $p \leq 2$, then one is back to the second alternative (or worse) which is non-optimal. Similarly, $p = N_b - 1$ brings us back to the first alternative which is non-optimal. Assuming y_1 and $y_2, .. y_{p+1}$ are selected, we have:

$$C(y_1) = we(e_M(v_1, y_1)) + 2 * \sum_{i=2}^{p+1} we(e_M(v_1, y_i)) + \sum_{i=p+2}^{N_b} we(e_M(v_1, y_i)) +$$

$$\sum_{i=2}^{p+1} we(e_M(y_1, y_i)) + 2 * \sum_{i=2}^{N_b} \sum_{j=i+1}^{N_b} we(e_M(y_i, y_j))$$

Expanding $C(v_1)$, we have:

$$C(v_1) = we(e_M(v_1, y_1)) + \sum_{i=2}^{p+1} we(e_M(v_1, y_i)) + \sum_{i=p+2}^{N_b} we(e_M(v_1, y_i)) +$$

$$2 * \sum_{i=2}^{p+1} we(e_M(y_1, y_i)) + 2 * \sum_{i=2}^{N_b} \sum_{j=i+1}^{N_b} we(e_M(y_i, y_j))$$

Canceling terms from $C(y_1)$ and using equation (4.1), shows that $C(y_1) \geq C(v_1)$. In the general case, more than one y_S, each with an associated set of states from the remaining y_i, may be selected and each set made uni-distant from y_S. The proof for this case is more involved but follows in a similar way as for the previous case, expanding $C(v_1)$ and using equation (4.1). Q.E.D.

Thus, the heuristic is optimal for a graph satisfying equation (4.1) at each iteration of the embedding, if sets of minimally distant codes can be found. It produces good (though

perhaps sub-optimal) solutions for graphs satisfying

$$we(e_M(v_1, y_i)) \geq RAT * we(e_M(y_i, y_j)) + we(e_M(y_i, y_k)) \quad 1 \leq i, j, k \leq N_b, \; i \neq j \neq k$$

where RAT is close to 1. This, coupled with the fact that typical graphs produced by the fanout and fanin-oriented algorithms have strongly connected clusters of states, makes the embedding algorithm eminently suitable for our purpose.

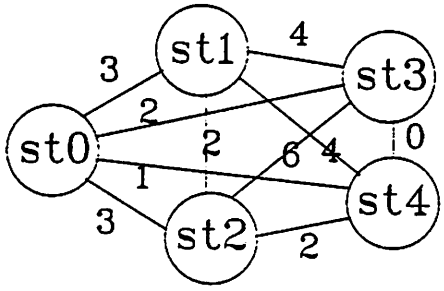
The algorithm is quite fast and has a worst-case time complexity of $O(N_s^2 (log(N_s) + N_b))$. Initially, the $N_s - 1$ fanout edges from *each* of the N_s states are sorted in decreasing order of weights which takes $O(N_s^2 log(N_s))$ time. The embedding itself may require a maximum of $N_s - N_b$ iterations. This is because in the first iteration, $N_b + 1$ states are encoded and in the worst case only one state is encoded in following iterations. To select a state, with maximum weight of any N_b connected edges, can be accomplished in $O(N_s N_b)$ time, giving an overall time complexity of $O(N_s^2 (log(N_s) + N_b))$.

The embedding algorithm is illustrated in Figure 4.12 using a small example with 5 states, to be encoded using 3 bits. Initially, the node corresponding to state *st3* is selected – it is the node with the maximum set of any 3 edge weights. The states corresponding to these three edges are *st0*, *st1* and *st2*. The three states are given codes uni-distant from *st3*. *st3* and its edges are deleted from the graph. The selection process continues, picking *st1* from the modified graph and encoding *st4*. This completes the encoding.

4.4.7 Results

I have run 20 benchmark examples (which have been obtained from various university and industrial sources) representing a wide range of finite automata on different state assignment programs as well as on the algorithms described in the previous sections. The size statistics of the examples are given in Table 4.1, with the minimum possible encoding for each FSM indicated under the column #enc.

The results obtained via random state assignment (RANDOM-A and RANDOM-B), using the state assignment program KISS (KISS), the fanout-oriented algorithm (MUST-P) and

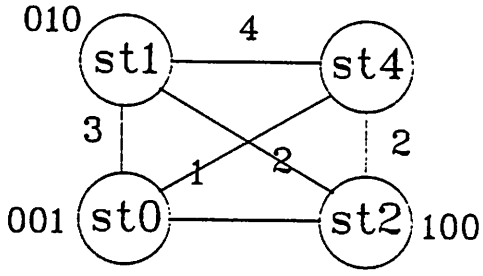


$N_B = 3$

st3 (st0, st1, st2)

st3 → 000 st0 → 001

st1 → 010 st2 → 100



st1 (st0, st2, st4)

st4 → 110

Fig. 4.12 An Embedding Example

EXAMPLE	#inp	#out	#states	#enc
bbara	4	2	10	4
bbsse	7	7	16	4
bbtas	2	2	6	3
cse	7	7	16	4
dk15x	3	5	4	2
dk16x	2	3	27	5
keyb	7	2	19	5
lion	2	1	4	2
lion9	2	1	9	4
mark1	5	16	14	4
mc	3	5	4	2
modulo12	1	1	11	4
planet	7	19	48	6
s1	8	6	20	5
s1a	8	6	20	5
scf	27	56	128	7
shiftreg	1	1	8	3
tav	4	4	4	2
tbk.min	6	3	16	4
train11	2	1	10	4

Table 4.1 Statistics of benchmark examples

the fanin-oriented algorithm (MUST-N) for multi-level implementations are summarized in Table 4.2 and 4.3. The number of literals after running through two optimization scripts in the multi-level logic synthesis tool MIS [6] are given for each of the state assignment techniques. The literal counts of Table 4.2 were obtained using a short optimization script and those of Table 4.3 using a much longer optimization script (which produces better results).

The literal counts under RANDOM-A were obtained using a statistical average of 10 different random state assignments (using different starting seeds) on each example. RANDOM-B was the best result obtained in the different runs. RANDOM-B is significantly better than RANDOM-A especially for the smaller examples. MUSTANG is the best result

EXAMPLE	RANDOM-A		RANDOM-B		KISS		MUST-P	MUST-N	MUSTANG
	#lit	RAT	#lit	RAT	#lit	RAT	#lit	#lit	#lit
bbara	120	1.48	91	1.12	103	1.27	81	108	81
bbsse	214	1.48	190	1.31	145	1.01	144	177	144
bbtas	37	1.15	26	0.81	34	1.06	51	32	32
cse	405	1.33	339	1.11	264	0.87	304	319	304
dk15x	122	1.17	109	1.04	91	0.87	104	128	104
dk16x	553	1.59	516	1.49	411	1.18	383	346	346
keyb	810	2.45	663	2.00	474	1.43	495	330	330
lion	20	1.11	18	1.00	21	1.16	18	22	18
lion9	61	3.05	52	2.60	37	1.85	25	20	20
mark1	112	1.28	89	1.02	114	1.31	130	87	87
mc	40	1.11	37	1.02	43	1.19	37	36	36
modulo12	43	1.19	40	1.11	49	1.36	40	36	36
planet	1063	1.24	1012	1.18	869	1.01	1033	854	854
s1	852	4.26	805	4.02	690	3.45	639	200	200
s1a	649	4.0	583	3.59	382	2.35	514	162	162
scf	1674	1.31	1596	1.25	1441	1.13	1390	1274	1274
shiftreg	37	18.5	32	16.0	8	4.00	2	8	2
tav	25	1.04	24	1.00	24	1.00	25	24	24
tbk.min	540	1.12	532	1.10	563	1.16	515	482	482
train11	67	1.34	53	1.06	46	0.92	50	55	50
TOTAL	**7444**	**1.62**	**6807**	**1.48**	**5809**	**1.26**			**4586**

#lit : Number of literals after multi-level logic optimization using fast script
RAT: Ratio to number of literals produced by MUSTANG using fast script

Table 4.2 Results obtained using fast script

produced by either the fanout or the fanin-oriented algorithm for each given example.

MUSTANG can be constrained to use any number of encoding bits greater than or equal to the minimum. For all examples, MUSTANG was run using the minimum possible bit encoding. Minimum bit encoding has been found to be uniformly good for multi-level logic implementations. KISS typically uses a 1-3 bits more than the minimum encoding length. The time required by MUSTANG for encoding these benchmarks varied between 0.1 CPU seconds for the small examples to 100 CPU seconds for the largest example, scf, on a VAX 11/8650.

The algorithms developed have achieved the goal of producing encodings which produce minimal area implementations after multi-level logic optimization as illustrated in Tables

EXAMPLE	RANDOM-A		RANDOM-B		KISS		MUST-P	MUST-N	MUSTANG
	#lit	RAT	#lit	RAT	#lit	RAT	#lit	#lit	#lit
bbara	95	1.39	76	1.11	79	1.16	68	75	68
bbsse	153	1.29	131	1.11	118	1.00	118	144	118
bbtas	32	1.45	24	1.09	28	1.27	41	22	22
cse	273	1.24	240	1.09	203	0.92	224	220	220
dk15x	109	1.18	94	1.02	85	0.92	92	108	92
dk16x	406	1.40	394	1.35	315	1.08	326	290	290
keyb	369	1.75	311	1.48	213	1.01	320	210	210
lion	19	1.18	16	1.00	16	1.00	16	16	16
lion9	55	2.75	43	2.15	36	1.80	22	20	20
mark1	102	1.22	99	1.19	99	1.19	115	83	83
mc	39	1.08	37	1.02	42	1.16	37	36	36
modulo12	41	1.24	36	1.09	47	1.42	38	33	33
planet	697	1.23	654	1.16	547	0.97	597	563	563
s1	424	2.65	354	2.21	352	2.20	376	160	160
s1a	363	2.57	337	2.39	258	1.82	307	141	141
scf	939	1.10	922	1.08	861	1.01	881	852	852
shiftreg	36	18.0	24	12.0	8	4.0	2	8	2
tav	23	0.95	22	0.91	22	0.91	24	24	24
tbk.min	355	1.19	342	1.15	381	1.28	348	297	297
train11	64	1.30	54	1.10	44	0.90	49	55	49
TOTAL	4594	1.39	4210	1.27	3754	1.14			3296

#lit : Number of literals after multi-level logic optimization using long script
RAT: Ratio to number of literals produced by MUSTANG using long script

Table 4.3 Results obtained using long script

4.2 and 4.3. The literal counts obtained by MUSTANG are on the average 30% better than random state assignment and 20% better than KISS. In some cases, the fanout-oriented algorithm does better than the fanin-oriented algorithm, when ignoring the common subexpressions in the input space is a good approximation.

MUSTANG does comparatively better than random assignment or KISS in the shorter optimization script case than in the more complex optimization script. This is to be expected since MUSTANG models only the common cube extraction process in multi-level logic optimization. In the short optimization script, cube factors dominate in the reduction of the size of the network. More complicated factors, not modeled by MUSTANG, come into play in the complex optimization script.

EXAMPLE	RANDOM-B		KISS		MUSTANG-N	
	#lit	#gate	#lit	#gate	#lit	#gate
cse	240	115	203	95	220	105
dk16x	394	175	315	143	290	124
keyb	311	158	213	112	210	112
planet	654	290	547	249	563	267
s1	354	174	352	173	160	93
s1a	337	169	258	131	141	83
scf	922	445	861	401	852	393
tbk.min	342	170	381	169	297	130

Table 4.4 Results after long script and technology mapping

For any given example, the literal counts obtained using MUSTANG and short or long optimization scripts are comparatively *closer* than using KISS or random assignment. For example, in **bbara**, using random assignment produces 120 literals after quick optimization versus 76 after a long optimization (on an average). The corresponding numbers for MUST-P are 81 and 68 respectively. MUSTANG eases the job of the multi-level logic optimizer, by providing a large number of easily detectable factors in the network before optimization – a short script can produce good results. Also, the time taken by MIS to optimize MUSTANG encoded examples is significantly shorter (20-40%) than to optimize examples encoded using different techniques. Again, this is because a large number of easily detectable factors exist in the pre-optimized network.

Recently, a paper was published in the Design Automation Conference [104], which reported erroneous results for MUSTANG in comparison with random assignment. The results were corrected during the presentation at the 1988 DAC. Over a wide range of examples (including others from those presented here), it has been determined that MUSTANG, on an average, outperforms random assignment (the best of 10 runs) by 25%.

Both MUSTANG and MIS optimize for literal counts rather than the number of gates in the network. MIS may produce arbitrarily complex gates in an optimized network. In many

cases, these gates have to be *mapped* to a specific technology library. It is worthwhile to see if the gains in literal counts do produce networks with fewer gates. The number of gates in the benchmark examples after intensive logic optimization and technology mapping are given in Table 4.4 for the different state assignment techniques. Only the largest examples are shown, the small examples had insignificant numbers of complex gates.

4.5 Conclusions

All previous work in automatic state assignment has been targeted toward two-level logic implementations of finite state machines. Multi-level logic implementations can be substantially faster and/or smaller than corresponding two-level implementations. The need for new techniques of state assignment directly targeting multi-level logic implementations has been shown and algorithms have been developed for this purpose. As compared to existing techniques, significant reductions in literal counts, averaging 20-40%, have been obtained on benchmark examples. Some erroneous results using these algorithms, which have been implemented in the program MUSTANG, have been published in the literature recently [104]. It has been shown that MUSTANG can consistently outperform random assignment over a wide range of examples.

CHAPTER 5

Combinational Logic Optimization

5.1 Introduction

Combinational logic circuits are extensively used in ASIC chips – in datapaths to realize arithmetic and Boolean operations and in finite state machine (FSM) controllers. Combinational logic circuitry can occupy a very significant fraction of area in an IC. The optimization of combinational circuits for speed and area is thus a very real and important problem.

Research over the past 30 years has resulted in efficient methods for implementing combinational logic in optimal two-level form using Programmable Logic Arrays (PLAs). However, many logic blocks are inappropriate for this kind of implementation. For example, there exist functions whose minimum two-level representation has 2^{n-1} product terms, where n is the number of primary inputs. In addition, even if a two-level representation contains a reasonable number of product terms, there are many cases in which a multi-level representation can be implemented in less area and generally as a much faster circuit. Two-level logic representations can be viewed as special cases of more general multi-level representations.

Optimal multi-level logic synthesis is a known difficult problem which has been studied since the 1950's. In recent years, an increasing level of research has become apparent in multi-level logic synthesis. One of the first of the modern developments is the Logic Synthesis system (LSS) [79] [121] at IBM, which has as a target technology a variety of gate arrays and standard cells. The Yorktown Silicon Compiler [78], which automatically synthesizes and lays out CMOS dynamic logic is based completely on multi-level logic and has Domino CMOS logic as its primary target technology. The SOCRATES system [77] is a multi-level logic synthesis system which uses gate arrays and standard cells, and is one of the

earliest to emphasize timing performance. The recently developed MIS system [6], is targeted toward area and timing optimization and uses algorithms which easily support a variety of target technologies.

For multi-level design, two basic methodologies have evolved: 1) global re-structuring, where the logic functions are "factored" into an optimal multi-level form with little consideration of the form of the original description (e.g. [6] [78]; 2) peephole optimization, where local transformations are applied to the user-specified (or globally-optimized) logic function (e.g. [79] [80]).

Global re-structuring procedures have been shown to be crucially necessary in producing optimal designs. Factoring algorithms have been proposed [76] [6] which are effective in partitioning complex logic functions.

The factoring algorithms proposed in [76] [6] are primarily based on algebraic techniques. Boolean factoring/division techniques can achieve superior results. However, techniques proposed so far for Boolean factoring and multi-level Boolean minimization (e.g [81]) require large amounts of CPU time.

Multi-level logic networks can be realized by standard cell or gate array implementations. For small-medium (< 50 product terms) sized two-level representations the PLA is a very compact structure whose size is comparable (if not smaller) than a corresponding multi-level implementation. Topological optimization techniques like folding [82] can further reduce PLA area. A set of interconnected PLAs can thus exploit both the layout compactness of PLAs without being constrained by the relative inflexibility of two-level logic structures.

A PLA can be decomposed into a set of interconnected PLAs which feed into one another. To perform this decomposition, factoring algorithms are required. In this chapter, algorithms for *Boolean decomposition* are presented, which decompose a PLA into a set of smaller interconnected PLAs such that the overall area of the resulting logic network, deemed to be the sum of the areas of the constituent PLAs, is minimized.

The algorithms are based on *multiple-valued* Boolean minimization [83]. Given a PLA, a subset of inputs to the PLA is selected. This *selection* step incorporates a new algorithm which selects a set of inputs such that the cardinality of the multiple-valued cover, produced by representing all combinations of these inputs as different values of a single multiple-valued variable, is much smaller than the original binary cover cardinality. A relatively small size for the multiple-valued cover implies that the number of good Boolean factors contained in this subset of inputs is large. Selecting different sets of inputs in the given logic function results in different multiple-valued cover cardinalities. Given a constraint on the number of inputs to be selected, the algorithm identifies a subset of inputs which when represented by a single multiple-valued variable results in the maximum reduction of cover cardinality.

Next, the different cube combinations given by this subset of inputs are *re-encoded* to satisfy the constraints given by the multiple-valued cover, thus producing a binary cover for the original PLA whose cardinality equals the multiple-valued cover cardinality. The number of distinct constraints specified by the multiple-valued cover affects the number of bits required to satisfy these constraints, which in turn affects the areas of the resulting PLAs. This problem of optimal constrained encoding, i.e. satisfying a set of constraints by a minimum code-length is a complex combinatorial optimization problem. Optimal constrained encoding has been formulated as a *constraint ordering* problem. The re-encoding process incorporates a new encoding algorithm which heuristically minimizes the number of bits required to satisfy all or a subset of the constraints produced by multiple-valued minimization.

These algorithms have produced excellent results over a wide range of examples. Total delays and areas of resulting PLAs after Boolean decomposition are invariably smaller than those for the original PLAs. This approach exploits the layout compactness of PLA structures to produce small, fast multi-level logic implementations. Large PLAs have been reduced by factors of 2-3 in size and delay [83].

In Section 5.2, basic definitions and notations used are given. The overall strategy is outlined using a simple example in Section 5.3. The input selection algorithm is described in Section 5.4. The algorithm used to re-encode the different input combinations, while minimizing the code-length, is described in Section 5.5. Results on several benchmark examples are given in Section 5.6.

5.2 Preliminaries

A **multiple-valued variable** m^p can take on values 0, 1 .. p–1. In particular, a binary-valued variable m^2 can take on values of 0 or 1. If the superscript is omitted, it is assumed to be 2.

Let p_i for i=1 $\cdots$ n be positive integers representing the number of values for each of n variables. Define the set $P_i \equiv \{0, .. , p_i\text{–}1\}$ for $i = 1 \cdots n$ which represents the p_i values that variable i may assume, and define $B_j \equiv \{0, 1, *\}$ which represent the value of the function. A **multiple-valued input, binary-valued output function** with k outputs, $P_{n \times k}$, (hereafter known as a **multiple-valued function**) is a mapping

$$P_{n \times k} : P_1 \times P_2 \cdots \times P_n \rightarrow B_1 \times .. \times B_k$$

The function is said to have n multiple-valued inputs, $m_i^{P_i}$, k binary-valued outputs, and variable i is said to take on p_i possible values.

The **cover** of a multiple-valued function, $P_{n \times k}$ is a collection of **cubes** or **implicants** $\{c_i, 1 \leq i \leq |P|\}$ where:

$$c_i = (c_{i1}, c_{i2}, .. c_{in}, b_{i1} .. b_{ik})$$

$|P|$ is the number of cubes in the cover. The first n entries represent the **input part** of the cube, and the next k entries represent the **output part** of the cube. (In the sequel, when no confusion can arise, only the input part of a cube will be listed). $c_{ij} \subseteq P_j$ is a collection of values of $m_j^{P_j}$ that appear in cube c_i.

A bit-vector representation of length p_j will be used for $c_{ij} = [\, c_{ij}(0), .. , c_{ij}(p_i - 1) \,]$, where:

$$c_{ij}(k) = 1 \text{ if } k \in c_{ij} \text{ else } 0$$

The inputs to a binary-valued logic function will be denoted x_i.

The area of a logic function, $P_{n \times k}$ is defined to be $(2n + k) \times |P|$.

5.3 Overall Strategy for PLA Decomposition

The goal is to decompose a given PLA into two smaller PLAs such that one PLA feeds into the second and the sum total of the areas of the two PLAs is minimum. This transformation is illustrated in Figure 5.1.

It should be noted that (1) a primary input may feed into both PLA I and II and (2) only a subset of the inputs of the original PLA feed into the decomposing PLA I.

Consider the PLA specification of Figure 5.2(a). This specification is optimal, with minimum cardinality. Inputs 1, 2 and 3 are selected. Eight distinct combinations of these inputs exist in the PLA description corresponding to eight minterms. These inputs now are re-encoded, i.e. assigned different binary codes. The binary codes assigned to the various input combinations are listed in Figure 5.2(b). Now, replacing each input combination by its distinct binary code results in the PLA specification shown in Figure 5.2(c). Minimizing these two PLA specifications, namely the encoding PLA (Figure 5.2(b)) and the re-encoded PLA

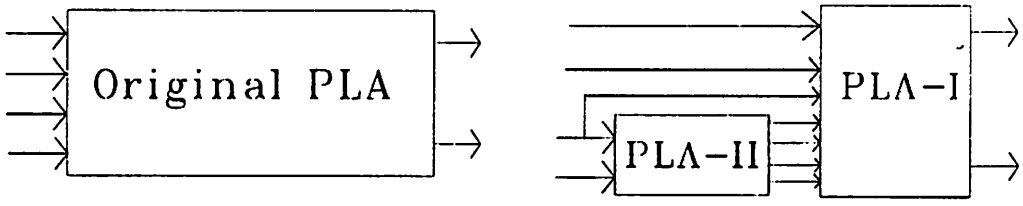


Fig. 5.1 PLA Decomposition

(Figure 5.2(c)), produces the results in Figure 5.2(d). The cardinality of the re-encoded PLA is one-quarter that of the original. The original PLA can be realized in smaller area by PLA-II feeding into the decomposed PLA-I as shown in Figure 5.1.

Thus, two steps have to be performed in PLA decomposition. First, a subset of inputs to the original PLA have to be selected. Next, these inputs have to be re-encoded. *Re-encoding* a set of inputs is a process that replaces each distinct combination of these inputs (which occur in the original PLA description) by a distinct binary pattern. The length of these binary patterns may be less than, equal to, or greater than the number of inputs being re-encoded. The only constraint on the pattern length (the number of re-encoding bits), is due to the fact that no binary pattern may be assigned to more than one input combination. If for a set of inputs of cardinality n_i, $N_d \leq 2^{n_i}$ distinct input combinations exist, the number of re-encoding bits required is $N_b \geq \log(N_d)$.

The final goal of both the selection and re-encoding steps is to minimize the number of product terms in the decomposed PLA description. Now, given a selection of inputs, a *lower bound* on the number of product terms in the decomposed PLA (PLA-II) exists even for an optimal re-encoding. The goal of the selection step is to find a subset of inputs which has the least lower bound so as to obtain a maximal area decrease via decomposition. The re-encoding step can be formulated as a multiple-valued minimization problem followed by a constrained encoding problem (The state assignment problem was similarly formulated in [26] previously). The lower bound on the number of product terms in the decomposed PLA, given a selection of inputs, can be found via multiple-valued minimization. Also, this lower bound can always be achieved by re-encoding. However, the number of encoding bits required, N_b, can vary greatly. Finding a minimum N_b during re-encoding is a complex combinatorial optimization problem.

The formulation of re-encoding as a multiple-valued minimization followed by a constrained encoding problem is illustrated in Figure 5.3. The PLA specification of Figure 5.2(a)

```
100 00 1
010 00 1
001 00 1
111 00 1
000 10 1
110 10 1
101 10 1
011 10 1
000 01 1
110 01 1
101 01 1
011 01 1
100 11 1
010 11 1
001 11 1
111 11 1
```

(a) Original PLA

```
100 -> 111
010 -> 110
001 -> 101
111 -> 100
000 -> 011
110 -> 010
101 -> 001
011 -> 000
```

(b) Re-encoding

```
111 00 1
110 00 1
101 00 1
100 00 1
011 10 1
010 10 1
001 10 1
000 10 1
011 01 1
010 01 1
001 01 1
000 01 1
111 11 1
110 11 1
101 11 1
100 11 1
```

(c) Re-encoded PLA

```
111 100
010 100
001 100
100 100
--0 010
-0- 001
```

```
100-- 1
0101- 1
001-- 1
111-0 1
```

(d) Minimized PLAs II & I

Fig. 5.2 Example decomposition

is reproduced in Figure 5.3(a). This time two inputs, namely inputs 1 and 2 are selected. A

new cover where each of the distinct combinations of these inputs is represented by a value of a single multiple-valued variable, M, is shown in Figure 5.3(c). This variable takes on four different values. The values correspond to the four distinct input combinations and are shown in Figure 5.3(b). The multiple-valued cover of Figure 5.3(c) is minimized to produce the cover of Figure 5.3(d). The multiple-valued cover has half the cardinality of the original PLA. The constraints to be satisfied by an encoding to produce a binary cover with the same cardinality are enumerated in Figure 5.2(e). These constraints are all the distinct combinations existing in the multiple-valued part of the cover of Figure 5.3(d).

Given the constraints obtained from the minimized multiple-valued cover, an encoding of input combinations can be found, which will produce a binary cover for the decomposed PLA of the same cardinality as the cardinality of the minimized multiple-valued cover (Figure 5.3(d)). In fact, an encoding can be found such that each of the implicants in the multiple-valued cover can be represented by a single implicant in the binary cover. The encoding has to the satisfy the *constraint relation* imposed by the multiple-valued implicants [26].

For this description, each distinct input combination to be encoded is called a *node*. Each node represents a value of the multiple-valued variable, M. A node is said to exist in a constraint (which corresponds to a multiple-valued implicant) if the implicant takes on the value of the node (in the notation used, this means that the bit position corresponding to the node contains a 1). For any given constraint, C, the *group face*, C_F, for C, is the smallest cube containing the codes of each node in C. The constraint relation imposed is that each group face, C_F is disjoint (does not intersect) any codes of all other nodes not in C [26].

In the example above, the encoding for the input combinations in Figure 5.2(b) satisfies the constraints of Figure 5.3(e). The group face corresponding to the first constraint 1100 is 0-[3] (= 01 $\cup$ 00) which does not intersect the codes of the other two nodes, namely 10 and

3 The symbol - is used to represent a *don't care* condition for the variable.

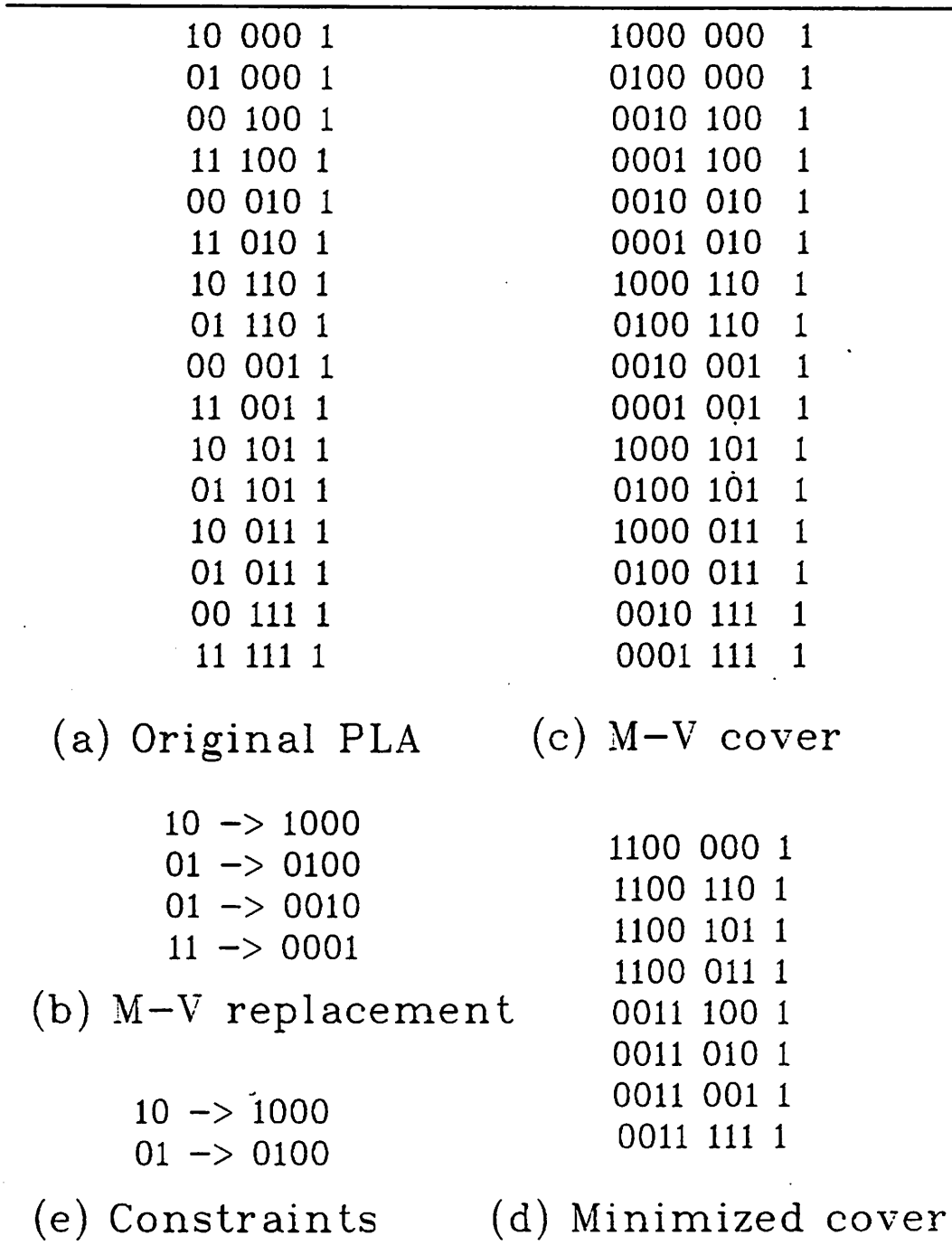


Fig. 5.3 Re-encoding via multiple-valued minimization

5.4 Selection

The goal of the selection process is to identify a subset of inputs, SI, $|SI| = N_s$, in the given PLA which when *re-encoded* will reduce the cardinality of the binary cover to a maximum extent (more than any other subset of N_s inputs).

One approach is to identify these inputs by exhaustive search. For each subset of inputs, all distinct input combinations can be replaced by a multiple-valued variable and the cover minimized. The subset of inputs which produces the smallest multiple-valued cover cardinality can be selected.

There are two problems with the above-mentioned approach. The number of possible subsets of inputs is exponentially related to the number of inputs to the PLA. Also, the evaluation of a selection can be very time consuming, since it involves performing an optimal or near-optimal multiple-valued minimization. Thus, this approach is infeasible for anything but the smallest PLAs (about 8 inputs).

The approach used to computing the cover cardinality after re-encoding is to *estimate* (rather than evaluate) the cardinality of the optimal multiple-valued cover produced by replacing all distinct combinations of the selected inputs by a single multiple-valued variable. Each distinct combination becomes a different value of the multiple-valued variable. The cardinality of the minimum cover of the new multiple-valued input, binary output logic function is the minimum cardinality that can be achieved by re-encoding the selected inputs.

Informally, the selection algorithm proceeds as follows:

(1) All, or a subset of inputs, are selected, and each distinct input combination in the original PLA is represented by a value of a multiple-valued variable.

(2) A complete, undirected graph where each vertex in the graph is an input to the PLA is constructed. The graph has weighted edges whose weights are all initially set to zero.

(3) The multiple-valued cover is minimized using a multiple-valued logic minimizer like ESPRESSO-MV [85].

(4) Next, the multiple-valued cover is examined to see what values, i.e. what distinct input combinations, have been merged together. Each implicant in the multiple-valued cover contains one or more values which correspond to the original binary input combinations. For each implicant, the set of binary input combinations which have merged to form this implicant is found. The inputs which *prevent* this set of binary input combinations from merging in the original binary cover are found for each implicant.

(5) Given the set of inputs for each implicant, the edge weights in the graph are modified. Essentially, a weight directly proportional to the number of merged input combinations in the implicant and inversely proportional to the number of inputs preventing this merge from occurring in the original binary cover is added to the edge between each pair of inputs in the set.

(6) Given the graph with weighted edges, a set of vertices (inputs) of cardinality N_s with the maximum sum of weights of interconnecting edges is found. This set of inputs is the selected set.

A more formal description of the selection algorithm follows. A binary cover $P_{n \times k}$ is given. (If the cover is minimal all $L = |P|$ cubes $c_i = (c_{i1}, c_{i2}, \cdots c_{in})$ are distinct). A new function, $P'_{1 \times k}$ is constructed, whose input is a single multiple-valued variable, $m^{|P|}$. Each cube $g_i \in P'_{1 \times k}$ corresponds to a value of the variable $m_i^{|P|}$

$$g_i(j) = 1 \text{ if } j = i \text{ else } 0\ ,\ \ 1 \le j \le L$$

Thus, a function with n binary-valued inputs, $P_{n \times k}$ is converted into a function with a single multiple-valued input, $P'_{1 \times k}$. $P'_{1 \times k}$ is then minimized to produce $P''_{1 \times k}$.

I will illustrate this transformation with an example. A binary cover $P_{3 \times 3}$ and the resulting functions $P'_{1 \times 3}$ and $P''_{1 \times 3}$ are shown in Figure 5.4.

We have a graph $G(V, E, W(E))$ where $V = \{v_i : v_i \leftrightarrow x_i\}$ are the vertices in the graph. x_i, $i = 1 \,..\, n$ are the n inputs to the PLA. The edges in graph $E = \{v_i, v_j\} \forall i, j \ \ i>j$; the graph is complete and undirected. Initially the weights of the edges $W(E) = 0$.

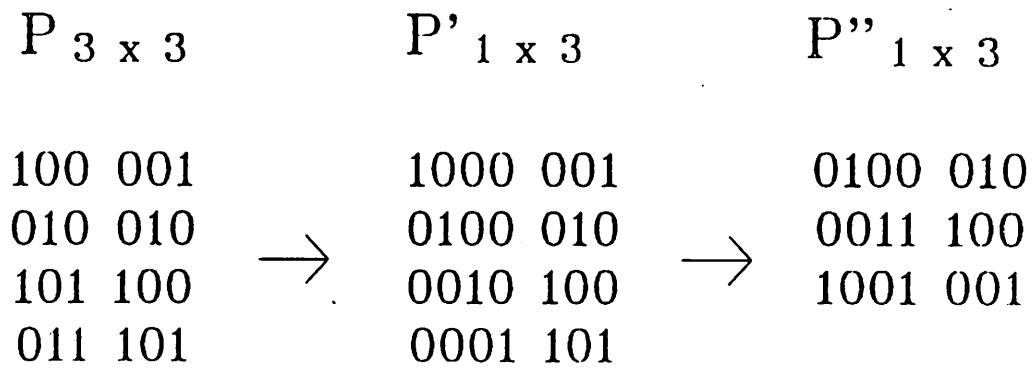


Fig. 5.4 Selection

The pseudo-code below illustrates the calculation of weights of the edges in the graph $W(E)$.

```
calculate-edge-weights()
{
  foreach ( cube g_i ∈ P''_{1 x k} ) {
    C_i = {c_j : g_i(j) = 1} , 1 ≤ i ≤ |P| ;
    X_i = {x_i : ∃ c_j, c_k ∈ C_i, c_ji ≠ c_ki} ;

    w_i = |C_i| / (|X_i| - 1) ;
    foreach ( x, y ∈ X_i, x ≠ y ) {
      w(x, y) = w(x, y) + w_i ;
    }
  }
}
```

C_i is the set of input combinations in the original binary cover which merge to form the implicant m_i in the minimized multiple-valued cover. X_i is the set of input combinations which prevent this merging in the original cover. For example, two binary implicants c_i, $c_j \in P_{n \times k}$ which differ in two (or more) bit positions, r and s, i.e. $c_{ir} \neq c_{jr}$ and $c_{is} \neq c_{js}$ cannot merge due

to inputs x_r and x_s. If any input takes on different values in the cubes in C_i, it is included in X_i. The weight of the edge between each pair of inputs in X_i is incremented by w_i.

The rationale for the calculation of w_i is as follows. If the inputs in the set X_i are selected, then C_i cubes in the binary cover will merge into a single multiple-valued implicant. Since the goal is maximum cardinality decrease the edges between these inputs are weighted correspondingly. However, since one typically wants to select a fixed number of inputs, the cardinality of X_i is also a factor. The weight is inversely proportional to the number of inputs in X_i. A single input cannot prevent a merging of cubes by itself in a binary cover, so the cardinality of $|X_i| \geq 2$. What one requires is the largest possible reduction in the cardinality of the cover (given by $|C_i|$ for each implicant) selecting the smallest number of inputs (given by $|X_i|$).

Given the graph $G(V, E, W(E))$ a set of N_s vertices, $SI = \{v_k\}$ (inputs) are selected such that:

$$\sum_{i=1}^{N_s} \sum_{j=i+1}^{N_s} w(s[i], s[j]) \quad s[i], s[j] \in SI$$

is maximized.

This selection of inputs is performed by exhaustive search. Typically, the number of inputs, N_i, to the logic functions encountered in practice, is ≤ 100. The number of inputs selected, N_s, varies between 2–8. An exhaustive search algorithm has a complexity of $O(N_i^{N_s})$ and is feasible. A heuristic polynomial-time algorithm that will iteratively select N'_s ($< N_s$) inputs at a time until N_s inputs are selected can be used if N_s is large.

This selection algorithm has been experimentally shown to be a good heuristic for a large class of logic functions (see Section 5.5). In addition, I can prove that the algorithm produces an optimal selection for a certain class of functions.

Theorem 5.1: Given an optimal binary cover, $P_{n \times k}$, the cardinality of the optimal multiple-valued cover produced by replacing all distinct combinations of any $l \leq k$ inputs by a single

multiple-valued variable is $\geq \frac{|P|}{2^{l-1}}$.

Proof (By contradiction): Let us assume that the multiple-valued cover has a cardinality $M < \frac{|P|}{2^{l-1}}$. It is true that any single-output function with r inputs can be represented by a cover of cardinality $\leq 2^{r-1}$. Therefore, each of the implicants of the multiple-valued cover can at most represent 2^{l-1} binary implicants. This means that the multiple-valued cover can be transformed into an equivalent binary cover of cardinality $M \times 2^{l-1} < |P|$, implying that the original binary cover was non-optimal. Q.E.D.

Theorem 5.2: Given a set of functions, F, each with n input variables $X = x_1, x_2, .. x_n$ of the form $F_i = (x_1 \oplus x_2 .. \oplus x_k) \cdot G_{i1}(x_{k+1}, .. x_n) + \overline{(x_1 \oplus x_2 .. \oplus x_k)} \cdot G_{i2}(x_{k+1}, .. x_n)$ $\exists$ $G_{i1} \cap G_{i2} = \phi$ $\forall$ i with $k \geq 2$, the optimal selection for k inputs is $x_1, .. x_k$ and will be produced by the selection algorithm.

Proof: It is easy to see that selecting $x_1, .. x_k$ will result in a multiple-valued cover cardinality $\frac{|F|}{2^{k-1}}$, since $|x_1 \oplus x_2 .. \oplus x_k| = |\overline{x_1 \oplus x_2 .. \oplus x_k}| = 2^{k-1}$. This is the maximum possible for any selection of k inputs by Theorem 5.2. We now have to prove that the graph, G, constructed by the selection algorithm has the maximum sum of weights of edges between inputs $x_1, .. x_k$ for any k out the n inputs to F. Call the multiple-valued cover produced by selecting all $x_i \in X$, F_m. Obviously $|F_m| \leq \frac{|F|}{2^{k-1}}$. Each of the implicants in F_m have been prevented from forming in the original binary cover by every pair of inputs $x_i, x_j \exists 1 \leq i, j \leq k$, and possibly other pairs of inputs $x_y, x_z \exists y > k$ *or* $z > k$. Therefore the clique, $C \in G$, comprising the inputs $x_1, .. x_k$ has *each* of its edges incremented by w_i (in the inner loop of function **calculate-edge-weights**()) for *every* implicant $g_i \in F_m$. Weights of other edges in graph not in C may also be incremented by w_i, but in every iteration of the outer loop in function **calculate-edge-weights**(), $\sum_{a=1}^{k} \sum_{b=a+1}^{k} w(x_a, x_b)$ is incremented by $k \times k-1 \times w_i$. We immediately have:

$$\sum_{a=1}^{k}\sum_{b=a+1}^{k} w(x_a, x_b) \geq \sum_{q=1}^{k}\sum_{r=q+1}^{k} w(s[q], s[r]) \;\forall\; s[q], s[r] \in X$$

Q.E.D.

5.5 Encoding

The selection process identifies a subset of inputs, *SI*. The different binary combinations of the input variables in *SI* are now re-encoded to minimize the number of product terms in the original PLA, $P_{n \times k}$.

First, the input cubes, $c_i \in P_{n \times k}$, are separated to form (u_i, s_i), where u_i represents that part of c_i corresponding to the unselected binary-valued inputs, and s_i represents the part of c_i corresponding to the selected binary-valued inputs, *SI*. The cubes c_i are made disjoint in s_i, i.e.

$$s_i \cap s_j = \phi \text{ if } s_i \neq s_j$$

to produce $c'_i \in P'_{n \times k}$. Note that $|P'| \geq |P|$.

Note that a trivial transformation $|P| \rightarrow |P'|$ exists, where each of the s_i's is a minterm. However, this may result in a very large $|P'| \gg |P|$, ($|P'| \leq 2^{|SI|} \times |P|$), decreasing computational efficiency ($|SI|$ is the number of selected inputs). Thus, one would like to find a transformation which minimizes $|P'|$ while satisfying the s_i disjointness constraint. A simple strategy of splitting a s_i cube only if it intersects another cube s_j suffices to keep $|P'|$ from increasing exponentially with $|SI|$. The worst-case complexity is, however, exponential.

A description of the algorithms used in re-encoding follows. Let D be a set of *nodes* where each node corresponds to a distinct s_i. D is represented by a multiple-valued variable with $|D|$ values, $m^{|D|}$, each value corresponding to a node in D. Replacing the $s_i \in c'_i \in P'_{n \times k}$ with a multiple-valued variable, $m^{|D|}$, a new cover $Q_{(n-|SI|+1) \times k}$ is obtained, which has $n-|SI|$ binary-valued inputs and a single multiple-valued input.

Q is minimized using a multiple-valued logic minimizer (e.g. ESPRESSO-MV [23]) to produce $Q'_{(n-|SI|+1) \times k}$.

Define a *code matrix* $A \in \{0, 1\}^{|D| \times N_b}$ whose rows are the new encodings of the nodes in D. Define a *constraint matrix* $M_c \in \{0, 1\}^{|Q'| \times |D|}$,

$$M_c(i, j) = 1 \text{ if } m'_i(j) \in Q' = 1 \textit{ else } 0$$

In [26] it was shown that if the different values of m (each node in D) were given binary codes A, of some length N_b, satisfying the *constraint relation* imposed by M_c, a minimized binary cover $PF_{(n-|S|+N_b) \times k}$ with $n-|S|+N_b$ binary-valued inputs, would be such that $|PF| \leq |Q'|$.

Each row in M_c specifies a constraint on the binary codes, A. Given a row, i, in M_c, N_i is the group of nodes for which $M_c(i,j) = 1$. The *group face* for N_i is the smallest cube containing the codes of each node in N_i. The constraint relation imposed is that each group face, N_i is disjoint (does not intersect) any of the codes of all other nodes in D not in N_i [26].

The constrained encoding problem is to find a matrix A with the minimum number of columns N_b which satisfies the constraint matrix M_c. The number of columns in A directly affects the areas of the resulting PLAs after decomposition.

It was shown in [26] that:

(1) All duplicate rows in M_c may be discarded.

(2) All rows with a single 1 or all 1's can be discarded.

(3) Any row which is the bit-wise intersection of two or more rows can be discarded.

(4) M_c^T, the transpose of M_c, satisfies the constraint relation, M_c.

The approach used to solving the constrained encoding problem is different from [26] and [28]. In [26], a row-based encoding scheme has been proposed. This encoding scheme constructs A row by row. This row-based encoding scheme fails to be effective for large examples [28]. A column-based encoding scheme, which constructs A column by column was proposed in [28]. Here, the constraint matrix M_c is compacted using relation (3) above and A is constructed incrementally as M_c^T. If at any point a constraint is already satisfied by A, then

it is discarded. This is possible even if all relationships using relation (3) have been exploited. I prove a result stronger than relation (3) here and formulate the constraint compaction problem as an *constraint ordering* problem.

Theorem 5.3: Given a constraint matrix, M_c, $A = M_c^T$ satisfies all constraints which are obtained by bitwise-intersecting two or more rows in M_c or their (bitwise) complements in any combination.

Proof: Construct a constraint, C, which is the bitwise intersection of the rows or complement of rows in M_c in any combination. We assume without loss of generality that C is constructed using all the rows in M_c in true or complemented form. (If C is constructed using a subset of rows, then we can identify a matrix M'_c corresponding to this subset of rows. We then have to prove that $A' = M'^T_c$ satisfies C, where C has been constructed using all the rows of M'_c). A bit vector B specifies if C has been constructed using the true or complemented form of each row in M_c. That is, $B[i] = 1$ if row $M_c(i)$ has been used in true from and $B[i] = 0$ if $M_c(i)$ has been used in complemented form. Examining A, it is easy to see that the group face corresponding to the group of nodes specified by C is $F = B$. The codes of nodes not in the group given by C differ from F in at least one bit position. If the code of a node, i, not in the group, was the same as F then $B[i]$ would equal 1, and a contradiction exists. Since the codes of nodes not in the group differ from F, A satisfies C. Q.E.D.

As indicated by Theorem 5.3, given a set of n constraints, 3^{n-1} (corresponding to true, complemented and unused possibilities for each constraint) possible constraints may exist which can be derived from these n constraints. Thus, given a constraint matrix M_c, it is quite possible that a large fraction of the rows of M_c can be derived from a small fraction of remaining rows. An incremental, column-based encoding scheme like in [28] which only checks for relationships given by (3) will produce a non-optimal solution if the constraints which can be derived from other constraints are satisfied first.

For small n, the relationships between all constraints can be found. However, checking for relationships given by Theorem 5.3 between all constraints is not possible for large n. Constraint ordering is a viable alternative. Given a particular *ordering* of rows in M_c, A is constructed incrementally, column by column, as in [28]. Constraints which are already satisfied are discarded. The cost of a particular ordering is the number of columns, N'_b, in A. Then the ordering of rows in M_c is changed and A is re-constructed. It is possible that a different value for N'_b corresponds to the new A. The goal is find an ordering which minimizes the number of encoding bits required, N'_b.

A variety of search techniques can be used to find an optimal ordering which minimizes N'_b. I have applied simulated annealing and a constructive heuristic algorithm to solve this problem. I have found that the constructive heuristic algorithm produces results close to the results produced by the iterative simulated-annealing-based algorithm, but much more quickly. For small examples, where the optimal solution is known, this algorithm, which is described below has found the optimal ordering (the ordering which minimizes the number of encoding bits, N'_b).

The algorithm first determines the set of required constraints, which cannot be derived from all the remaining constraints. For each constraint, an encoding based on all the remaining constraints is constructed and a check is made to see if the encoding satisfies the constraint. The complexity of encoding construction is $O(N_c*N_n)$ where N_c is the number of constraints and N_n the number of nodes to be encoded. The complexity of checking if an encoding satisfies a constraint is $O(N_n*N_l)$, where N_n is as before and N_l is the length the encoding.

A column-based encoding which uses each of the required constraints is constructed. All remaining constraints satisfied by this encoding are discarded. If any constraints remain, a constraint is selected which when added to the required constraint set produces an encoding which satisfies a maximum number of unsatisfied constraints. The process is repeated until all constraints have been satisfied. The number of encoding bits required is equal to the number

of required constraints.

Thus, A has been determined by this procedure given a Q'. A PLA $PF_{(n-|S|+N'_b) \times k}$ such that $|PF| = |Q'|$ is produced. A second PLA $PS_{|S| \times N'_b}$, whose cubes ps_i are formed from the s_i corresponding to the nodes in D, and A is the encoding PLA. The input part of ps_i is the cube s_i and the output part is the i-th row of A. These two PLA's perform the function of the original PLA, $P_{n \times k}$.

```
find-optimal-ordering:
{
  R0 = φ
  foreach ( row i in Mc ) {
      delete row i from Mc ;
      if ( A' = M'c^T does not satisfy Mc)
          R0 = R0 ∪ Mc(i) ;
  }
  Mc = Mc − R0 ;
  A = R0^T ;
  R = R0 ;
  while (Mc is not empty) {
      foreach ( row i in Mc ) {
          if (A satisfies Mc(i) )
              delete row i from Mc ;
      }
      C = select-constraint( A, Mc ) ;
      delete the row corresponding to C from Mc ;
      R = R ∪ C ;
      A = R^T ;
  }
}
```

```
select-constraint( A0, M0):

{

  foreach ( row i in M0 ) {
      A = A0 ;
      append column M0(i)^T to A ;
      M is obtained by deleting row i from M0 ;
      num = 0 ;
      foreach ( row j in M ) {
          if ( A satisfies M(j) )
            . num = num + 1 ;
      }
      select b, the row with maximum num ;
  }
  return( Mc(b) ) ;
}
```

5.6 Results

Decomposition results obtained on a large set of benchmark PLAs are given in Table 5.1. The PLAs were all optimized using the logic minimizer ESPRESSO before decomposition. In the table, the number of inputs to the original PLA (#inp), the number of outputs (#out) and the number of product terms (#prod) after two-level minimization, the area of the PLA (orig. area) are given. The number of selected inputs in decomposition (# sel. inputs), the areas of the two resulting PLAs (area PLA-I and area PLA-II), the ratio of areas after and before decomposition (ratio) and the CPU time required for decomposition on a VAX 11/8800 are given. The numbers under the ratio column are calculated by dividing the total area of the two resulting PLAs by the area of the original PLA. As can be seen, large reductions in areas have been obtained (some numbers are as low as 0.2). A simple timing analysis indicated that the delays of the decomposed PLA pair were smaller than those of the original PLA for all the examples. The CPU times required for decomposition are quite reasonable. The examples **x7dn** and **seq** are very large and hence require significantly more time than the other examples. A large fraction of the CPU time is expended by multiple-valued minimization during

the selection and encoding phases.

For standard cell or gate-array implementations of logic, a good estimate of area is the number of literals (transistors) in the logic. The primary goal of the decomposition algorithms is to try to minimize the number of product terms in the resulting PLAs rather than the

EXAMPLE	#inp	#out	#prod	orig. area	# sel. inputs	area PLA-I	area PLA-II	ratio	CPU time
5xp1	7	10	65	1560	4	255	884	0.73	5.0
9sym	9	1	87	1653	4	240	700	0.57	5.0
Z5xp1	7	10	63	1512	4	272	918	0.78	5.3
add6	12	7	355	11005	4	121	3973	0.37	21.4
alu3	10	8	66	1848	4	180	1326	0.81	5.3
clpl	11	5	20	540	4	30	345	0.69	1.2
dist	8	5	121	2541	2	1479	860	0.92	6.8
duke2	22	29	86	6278	4	44	6468	1.03	12.3
f51m	8	8	76	1824	4	304	1178	0.81	3.8
in2	19	10	136	6528	4	91	6272	0.97	16.7
in4	32	20	212	17808	4	196	13904	0.79	95.7
in7	26	10	54	3348	4	60	3162	0.96	8.0
max128	7	22	82	2952	3	104	2508	0.88	22.3
misg	56	23	69	9315	4	154	7923	0.86	12.5
mp2d	14	14	31	1302	4	195	924	0.86	7.9
nxcpla1	9	23	41	1681	3	98	1568	0.99	5.4
radd	8	5	75	1575	4	121	475	0.38	1.5
rd53	5	3	31	403	3	63	156	0.54	0.2
rd73	7	3	127	2159	4	224	459	0.31	1.8
rd84	8	4	255	5100	4	192	848	0.20	4.1
root	8	5	57	1197	4	210	756	0.80	5.0
sao2	10	4	58	1392	4	224	1088	0.94	8.7
seq	41	35	334	39078	4	1332	27846	0.74	546.1
sqr6	6	12	49	1176	2	21	1248	1.08	7.6
sym10	10	1	210	4410	4	195	1311	0.34	9.2
vg2	25	8	110	6380	4	154	4928	0.79	10.7
x1dn	27	6	110	6600	4	99	3696	0.58	5.5
x2dn	82	56	104	22880	4	176	20520	0.90	52.6
x7dn	66	15	539	79233	8	1184	35045	0.45	944.2
z4ml	7	4	59	1062	3	56	384	0.41	0.8

Table 5.1 PLA Decomposition Results

EXAMPLE	Original PLA			Decomposed PLA		
	Initial #lit.	Final #lit.	CPU time	Initial #lit.	Final #lit.	CPU time
5xp1	347	152	76.4	165	104	18.5
9sym	609	201	403.6	229	122	106.5
Z5xp1	358	165	85.4	163	108	23.0
add6	2551	81	1345.7	155	80	16.3
alu3	347	95	32.5	135	95	33.6
clpl	75	19	1.9	29	19	1.0
countl	394	161	173.6	177	140	41.7
dist	874	402	609.9	507	401	507.4
duke2	993	505	25.5	962	491	26.4
f51m	395	109	65.9	176	101	22.8
radd	415	49	29.8	63	45	6.2
rd53	175	53	12.3	62	47	6.8
rd73	903	94	182.4	182	104	31.1
rd84	2451	126	917.7	188	127	48.7
root	383	159	110.4	194	137	76.5
sao2	496	154	88.5	221	173	56.3
sqr6	266	142	36.3	179	132	37.5
sym10	1470	302	1289.3	246	176	164.4
vg2	914	92	79.8	281	111	12.3
x1dn	1097	108	85.2	182	108	18.6
x2dn	564	218	326.8	297	218	67.3
z4ml	311	39	20.4	53	39	5.6

Table 5.2 Multi-level Optimization Results

number of literals. However, the decomposed PLAs serve as an excellent starting point for multi-level logic optimizers even if targeting toward standard cell or gate-array layout styles because a large number of good factors are identified during decomposition. This is illustrated in Table 5.2. For each example, the multi-level logic optimizer, MIS [6] was made to execute a standard optimization script with different starting points, namely, the original PLA and the two decomposed PLAs. For a set of examples run with the two different starting points, the number of literals before and after optimization and the CPU time required for optimization are given in Table 5.2. As can be seen better results were achieved in significantly faster time

using the decomposed PLA pair rather than the original PLA.

There are two examples (**duke2** and **sqr6**) where the total area of decomposed PLA pair is larger than the area of original PLA. This was because in both cases the number of encoding bits required to satisfy the constraints produced after multiple-valued minimization was much larger than the number of selected inputs. Although a respectable decrease in the number of product terms was achieved the increase in the number of inputs resulted in the area of the re-encoded PLA (PLA-II) becoming larger than the original. However, the re-encoded PLA has fewer literals (transistors) than the original PLA. In fact, if both the original PLA and the decomposed PLA are compacted using a folding program like GENIE [86], the area of the decomposed PLA pair becomes smaller than the original PLA (since the decomposed PLAs are sparser than the original). Also, even for these two examples, the results obtained by MIS using the decomposed PLA pair as the starting point were better than starting with the original PLA. Also, in both cases, the decomposed implementation is faster than the original PLA.

5.7 Conclusions

Multi-level implementations of logic can be substantially smaller and faster than corresponding two-level i.e. Programmable Logic Array (PLA) implementations. Work in the area of multi-level logic optimization has been concentrated primarily in the development of algebraic techniques for factoring and decomposing logic equations. In this paper, algorithms for Boolean decomposition have been presented, which decompose a PLA into a set of smaller, interconnected PLAs such that the overall area of the resulting logic network, deemed to be the sum of the areas of the constituent PLAs, is minimized.

These algorithms have produced excellent results over a wide range of examples. This approach exploits the layout compactness of PLA structures to produce small, fast multi-level logic implementations. Large PLAs have been reduced by factors of 2-3 in size and delay.

CHAPTER 6

Verification of Logic Circuits

6.1 Introduction

Logic verification refers to the Boolean equivalence check of two logic designs. The designs in question may be purely combinational logic or they may be sequential finite state machines. These designs may be described at different levels of abstraction – a combinational logic design may be represented by a schematic gate-level description or by a Boolean Truth Table; a sequential circuit may be described by a State Transition Graph, as an interconnection of gates and flip-flops or in a register-transfer (RT) level language (e.g. ISPS [17]).

Verifying the equivalence of logic circuit descriptions at differing levels of abstraction is an important problem and has many possible applications. For example, after the synthesis of a logic-level finite automaton from a higher level register-transfer description, it is essential to be able to verify that the optimization tools during synthesis have not introduced any design errors in the circuit and that the synthesized description and the original specification actually represent the same machine.

One approach to the general verification problem is exhaustive simulation. Unfortunately, the number of simulations required grows exponentially with the number of inputs for even a purely combinational logic circuit, and grows even faster for sequential circuits since all possible input vector *sequences* have to be simulated to prove equivalence. A different approach is to use *formal verification* techniques which are input pattern independent and can guarantee functional equivalence.

Many formal verification approaches have been taken to prove/disprove the equivalence of two combinational logic circuits, at the gate level and at differing levels [89] [30] [90] [92]

[122] [123] [124]. Some of these approaches, for example, [30] and [90] transform the verification problem into a testing problem. A package of programs called PROTEUS [29] incorporates several efficient algorithms for verifying combinational logic circuits. In particular, an algorithm called LOVER [29] in PROTEUS has successfully been used on combinational circuits with a large number of gates.

Boolean equivalence checks can also be performed at the switch level [124] [123]. Using *symbolic simulation* and heuristically efficient Boolean function manipulation algorithms, the program MOSSYM [124] performs equivalence checks between combinational logic descriptions at the switch level against a specification of Boolean equations.

Sequential circuit verification is a considerably more difficult problem, in the general case when there is no correspondence between the latches (states) of the two circuits[4]. The approaches taken to solve the sequential verification problem include the use of temporal logic [125] and PROLOG [126]. The use of temporal logic helps for asynchronous circuits [127] but is not necessary in the synchronous circuit case. Algorithms have been proposed for formally verifying the equivalence of two gate-level sequential circuit descriptions with differing numbers of latches using symbolic Boolean manipulation [128]. However because of the intractability of the problem, most of the approaches taken so far have been restricted to small to medium sized circuits with a small amount of memory elements (4-6 latches).

Algorithms for formally verifying the equivalence of two sequential machines described at the register-transfer, State Table or logic level have been developed as part of this research and have been incorporated in the verification subsystem of our behavioral synthesis system. By exploiting the don't care information available at the various levels of abstraction (e.g. invalid input and output sequences), the complexity of the verification problem has been reduced significantly and the equivalence of finite automata with more than a thousand gates and 256 states has been successfully verified in less than 10 CPU minutes on a VAX 11/8650

4 In the special case of a one-to-one correspondence between the two circuits, the problem reduces to a combinational circuit verification problem.

[31].

Given two logic-level finite automata and a reset state/transfer sequence for each machine, a two-phase enumeration-simulation verification algorithm efficient both in terms of memory and CPU time usage has been developed. I have used this approach for verifying the equivalence of logic finite automata with 17 latches and more than 2500 valid states [31].

This chapter is organized as follows – in Section 6.2, the formal definitions for the equivalence of two combinational or sequential logic designs are given. Combinational logic verification algorithms are briefly reviewed in Section 6.3. Algorithms developed for verifying sequential machines across differing levels of abstraction are presented in Section 6.4.

6.2 Definitions of Equivalence

6.2.1 Basic Definitions

In this section, a framework of concise definitions is provided for use in the remainder of the chapter. In the definitions, the object being defined appears in bold type.

Let $B = \{0,1\}$, $Y = \{0,1,2\}$. A **logic** (Boolean, switching) **function** ff in n input variables, $x_1,x_2,..x_n$, and m output variables, $y_1,y_2,..y_m$, is a function

$$ff: B^n \rightarrow Y^m$$

where $x = [x_1,..x_n] \in B^n$ is the **input** and $y = [y_1,..y_m] \in Y^m$ is the **output** of ff. B^n is the Boolean n-space associated with the function ff. Note that in addition to the usual values of 0 and 1, the outputs y_i may also take the **don't care value** 2 (or –). Such functions are called **incompletely specified** logic functions. A **completely specified function** f is a logic function taking values in $\{0,1\}^m$, i.e., all the values of the input map into 0 or 1 for all the components of f. For each component of an incompletely specified logic function ff, $ff_i, i = 1,..m$, one can define: the **ON-set**, X_i^{ON} (also denoted by $FF_i^{ON}(x)$) $\subset B^n$, the set of input values x such that $ff_i(x) = 1$, the **OFF-set**, X_i^{OFF} (also denoted by $FF_i^{OFF}(x)$), the set of values such that $ff_i(x) = 0$ and the **don't care** set X_i^{DC}, the set of values such that $ff_i(x) = 2$. A logic function with m=1

is called a **single-output** function, while $m>1$, it is called a **multiple-output** function.

The **complement** of a completely specified logic function f, called $\overline{f}$, is another completely specified logic function, such that its components, $\overline{f_1},..\overline{f_m}$, have their ON-sets equal to the OFF-sets of the corresponding components of f. The **intersection** of two completely specified logic functions, $f \cap g$, is defined to be the completely specified logic function h, whose components h_i, have ON-sets equal to the intersection of the ON-sets of the corresponding components of f and g. The **difference** between two completely specified logic functions, $h = f - g$, is a completely specified logic function h given by the intersection of f with the complement of g. The ON-sets of the components of h are the elements of the ON-sets of the corresponding components of f that *are not* in the ON-set of the corresponding components of g. The **union** of two completely specified logic functions is a completely specified logic function $h = f \cup g$, such that the ON-sets of the components of h, h_i, are the union of the ON-sets of f_i and g_i.

A completely specified logic function f is a **tautology**, written $f \equiv 1$, if the OFF-sets of all its components are empty. In other words, the outputs of f are 1 for all inputs. For example, for any completely specified function f, $f \cup \overline{f}$ is a tautology.

A **cube** in a Boolean n-space associated with a logic function, f, can be specified by its vertices and by an index indicating to which components of f it belongs. An input cube c is specified by a row vector $c = [c_1,..c_n]$ where each input variable takes on one of three values 0, 1 or 2 (or –). A 2 in the cube is a don't care input, which means that the input can take the values of either 0 or 1. For example, the cube 002 is equal to the union of the cubes 001 and 000. A cube with only 0 and 1 values of inputs is called a **minterm**. A cube, c, is defined to cover (contain) another cube, d, if each entry of c is equal to the corresponding

entry of d or is equal to 2.

6.2.2 Combinational Logic Design Equivalence

Combinational logic designs may be represented by two-level or multi-level logic functions or by gate-level circuits.

Two completely specified (single output) logic functions, f and g, are **Boolean equivalent** if and only if

$$f = 1 \Leftrightarrow g = 1. \tag{6.1}$$

Given two incompletely specified logic functions f and g, their ON and OFF-sets and the don't care set $F^{DC}(x)$ ($F^{DC}(x)$ could represent input combinations that cannot occur), f and g are Boolean equivalent under $F^{DC}(x)$ if and only if

$$(F^{ON}(x) \cap G^{OFF}(x)) \cup (F^{OFF}(x) \cap G^{ON}(x)) \subset F^{DC}(x). \tag{6.2}$$

Given two functions F and G and a Boolean function D representing the don't care inputs (e.g. inputs which cannot occur or for which equivalence is not required) checking the condition

$$D(x) \cup (F^{ON}(x) \cap G^{ON}(x)) \cup (F^{OFF}(x) \cap G^{OFF}(x)) = 1 \tag{6.3}$$

amounts to checking F and G for equivalence.

Given a gate-level circuit Y, every input combination evaluates the circuit to some known value, either a 0 or a 1. So, Y represents a completely specified logic function.

The Boolean equivalence of two gate-level circuits A and B implementing f, given that

$F^{DC}(x) = \phi$, can be verified by checking that $A^{ON} = B^{ON}$ or $A^{OFF} = B^{OFF}$.

6.2.3 Sequential Circuit Equivalence

A **finite automaton** (FA, finite state machine, FSM) consists of a finite set of states and a set of transitions from state to state that occur on input symbols chosen from an alphabet Σ. For each input symbol there is exactly one transition out of each state (possibly back to the state itself). One state, usually denoted q_0, is the initial state, from which the automaton starts. Some states are designated as final or accepting states.

A **deterministic finite automaton** (DFA) is formally denoted by a five-tuple (Q,Σ,δ,q_0,F), where Q is a finite set of *states*, Σ is a *finite input alphabet*, δ is the *transition function* mapping from $Q \times \Sigma$ to Q, q_0 is the *initial* state and $F \subset Q$ consists of states designated as *final* or *accepting* states. That is, $\delta(q,a)$ is a state for each state q and input symbol a.

A directed graph, called a **State Transition Graph** (STG) or State Transition Diagram, is associated with an DFA. A State Transition Graph was defined in Chapter 4, Section 4.2. The State Transition Graph of the DFA may also be equivalently represented by a State Transition Table.

The DFA accepts a string x if the sequence of transitions corresponding to the symbols of x leads from the start state to a final or accepting state. More formally, a string x is said to be **accepted** by a finite automaton $M = (Q,\Sigma,\delta,q_0,F)$ if $\delta(q_0,x) = p$ for some $p \in F$. The **language** accepted by M denoted $L(M)$, is the set $\{x|\delta(q_0,x) \in F\}$.

Finite automata with multiple outputs fall into two categories. The output is associated with the state in a **Moore** machine, and with the transition in a **Mealy** machine. A Moore machine is a six-tuple $(Q,\Sigma,\Delta,\delta,\lambda,q_0)$, where Q,Σ,δ and q_0 are as in the DFA. Δ is the *output alphabet* and λ is a mapping from Q to Δ giving the output associated with each state.

A Mealy machine is also a six-tuple $(Q,\Sigma,\Delta,\delta,\lambda,q_0)$, where all is as in the Moore machine except that λ maps $(Q \times \Sigma)$ to Δ giving the output associated with each transition.

Furthermore, a finite automaton may be described at the register-transfer (RT) or logic levels. In the former, the transitions occurring due to different input symbols (i.e. the λ and δ mappings) are specified in a high-level programming language-like description using control constructs. In the latter, the machine is described by an interconnection of logic gates and flip-flops. The logic gates implement the δ and λ mapping functions and the flip-flops (memory elements) store the state of the machine $q \in Q$. That is, the state of the machine is specified by the binary values stored in the flip-flops.

Here, only synchronous finite state machines are considered. Synchronous finite state machines have the property that the inputs to the machine are sampled only at integral multiples of a duration of time called a *clock period.* The outputs of the machine also change only at integral multiples of a clock period. It is possible to define the equivalence of sequential circuits with different or same clock periods.

Each sequential circuit, C, considered here has the following properties. The circuit may be described at the register-transfer, State Transition Graph (STG) or logic level. C is assumed to be completely specified, i.e. every transition from every possible state is specified.

(1) C has exactly one output.

(2) If C is implemented by logic gates and flip-flops, it will be assumed that logic gates are delayless and that latches have unit delays (one clock cycle).

(3) All primary inputs arrive simultaneously, after $\rho(C)$ cycles, for some integer $\rho(C)$. The input lines may change values only at time $k\rho(C)$ for $k = 1, 2 \ldots$ Thus the circuit is when-determinate [129].

If C is an m-input circuit, w a (possibly infinite) binary sequence, z an integer, and q_0 a state of C, then $C(w,z,q_0)$ denotes the output of C after z cycles, starting from state q_0, where the first m bits of w are input to C initially, followed by the second m bits of w, where in

general bits $(r-1)m+1$ through rm of w constitute the rth set of inputs to C.

We are given sequential circuits, C_1 and C_2 along with $\rho(C_1)$ and $\rho(C_2)$ and a start state (an initial assignment to the latches for a logic-level circuit, or an identified state in the given STG), q_1 and q_2 respectively.

The equivalence of C_1 and C_2 amounts to checking the condition

$$C_1(w, r\,\rho(C_1), q_1) = C_2(w, r\,\rho(C_2), q_2) \tag{6.4}$$

for all infinite binary sequences w, for all integers $r \geq 1$.

Equation 6.4 represents a very general form of equivalence checking between sequential circuits with different input sampling rates, $\rho(C)$. In fact, one can check a combinational logic circuit for equivalence with a sequential circuit. One circuit may be a 4-bit parallel combinational adder and the other a serial single-bit sequential implementation. In this case, $\rho(C_1) = 1$ and $\rho(C_2) = 4$.

6.2.4 Segmentation: Single-Output Cone Extraction

A decomposition method widely adopted in performing logic verification checks for multi-output circuits is *segmentation* [29]. Segmentation decomposes a multi-output circuit into many single output segments called *cone* circuits. Cone circuits are verified separately. In the sequel, unless otherwise specified, all circuits are assumed to be cone circuits.

6.3 Combinational Logic Verification Methods

In this section, approaches to verifying combinational logic designs are reviewed. Four different kinds of methods are described in the following subsections – verification by exhaustive simulation, verification using test generation techniques, verification using symbolic simulation and verification using enumeration-simulation techniques. Comparisons between some of these methods have been drawn using the logic verification framework called PROTEUS

and can be found in [29].

6.3.1 Verification by Exhaustive Simulation

Verification using exhaustive simulation entails applying all possible input combinations to the circuits, simulating them and checking to see if the output responses are identical. This can be carried out by a hardware accelerator or a software simulator.

Exhaustive simulation is impractical in most cases. Given a twenty input circuit the number of binary patterns to be simulated is over a million. Hardware accelerators can alleviate the problem to a certain extent, but more intelligent verification techniques are required for large circuits.

6.3.2 Verification using Testing Methods

The verification problem can be formulated as a redundancy identification problem as shown in Figure 6.1. The two circuits which are to be checked for equivalence A and B are connected by an exclusive-nor gate. Establishing the fact that F-stuck-at-1 is a redundant fault is equivalent to verifying the two designs for equivalence. If a test can be found to detect F-stuck-at-1 the test vector is an input combination which differentiates the two circuits.

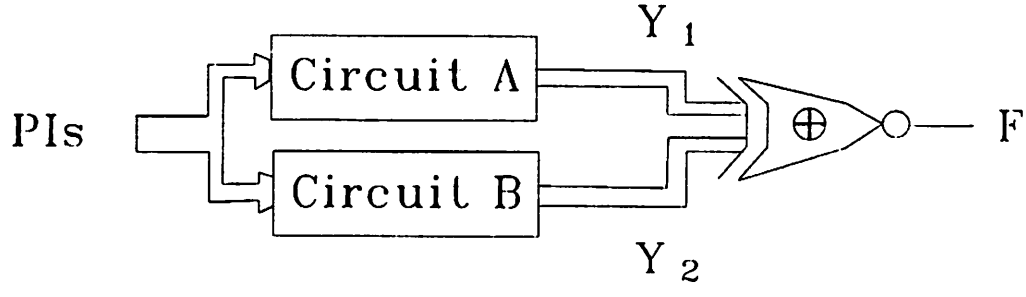


Fig. 6.1 Logic Verification via Redundancy Identification

Test pattern generation algorithms like the D-algorithm [130] and PODEM [131] can be used for identifying redundant faults in circuits and thus can be used to perform equivalence checks on logic circuits. VERIFY [90], a verification algorithm based on the D-algorithm finds counter-examples by performing line justification. Another algorithm called POVER [29] is based on the PODEM test generation algorithm. POVER is described in the next section.

6.3.2.1 Justification using PODEM - POVER

Given the composite circuit which has been constructed as shown in Figure 6.1, the backtrace algorithm of PODEM [131] can be used to perform line justification. In PODEM, given an output signal and a desired value on the output (F = 0), a path is traced from the signal to the primary inputs (PI) to obtain a PI assignment. This PI assignment is simulated to see if the desired value of the signal has been set up. If so, the procedure terminates. If the opposite value has been set, an opposite value is assigned to the PI and this value is propagated. If the signal remains unspecified, path tracing is repeated. The above procedure continues until either a successful PI assignment has been found (a counter-example has been found and the circuits are not equivalent) or all the PI assignments have been exhausted (the circuits are equivalent).

An efficient justification algorithm tries to justify an output value by setting a minimum subset of input values. In PODEM, justification of input combinations which are cubes rather than minterms is attempted in order to cover a large portion of the input space. For example,

justification could be attempted for an input combination, 1–1, representing four minterms.

6.3.3 Symbolic Verification

Algorithms for the symbolic manipulation of Boolean functions using a graphical representation have been proposed in [123] [124]. Verification is performed by extracting the Boolean functions from switch or gate-level circuits, representing them as directed, acyclic graphs and performing Boolean operations like *and*, *or* and *complement* to check for equivalence.

The data structure used resembles the binary decision diagram proposed by Lee [132] and Akers [133]. However, further restrictions are placed on the ordering of decision variables in the vertices. These restrictions enable the development of algorithms for manipulating the representations in a more efficient manner. The representation is in terms of *reduced* graphs and is a *canonical* form, i.e. every function has a unique representation.

The time complexities of the Boolean manipulation algorithms proposed in [124] are bounded by the product of the graph sizes for the functions being operated on. Complementing a function requires time proportional to the size of the function graph, while combining two functions with a binary operation (e.g. intersection, subtraction) requires at most time proportional to the product of the graph sizes. Since every function has a unique representation, checking for equivalence simply involves testing whether the two graphs match exactly. The identity check is performed by a graph isomorphism algorithm which requires time at most proportional to the sum of the two graph sizes.

The time required to validate logic designs is directly related to the size and number of the Boolean equations to be checked for equivalence. Verifying unstructured logic designs for equivalence is comparatively harder in this approach than verifying structured logic designs like ALUs. This is because the ALU can be specified by a compact set of Boolean equations (for example, the carry bit of an ALU can be compactly represented as a chain of exclusive-or's) whereas given a large unstructured control logic block, there may exist no compact Boolean representation for it. It then becomes very difficult to extract the Boolean functions

the logic block implements via symbolic simulation. The logic in a multiplier is much less structured than an ALU and has no compact Boolean representation. Hence, the multiplier is much more difficult to verify using symbolic simulation.

6.3.4 The LOVER Approach

LOVER (LOgic VERification) is an enumeration-simulation approach first proposed in [29] and is part of the PROTEUS system.

Let the two circuits to be checked for equivalence be A and B. First, a cube c from A^{ON} is generated (*enumerated*) and then *simulated* on B to check if B produces a 1 at the output. If so, the enumeration process continues to cubes from A^{ON}. If a 0 appears the circuits are not Boolean equivalent. If an x (unknown) appears c is split (cube-split) into smaller cubes and re-simulated until a known value appears at the output of B for each of the smaller cubes. Cube-splitting and simulation are implicitly exhaustive. The process continues until all the cubes from A^{ON} have been simulated. A similar process for A^{OFF} is then performed.

This framework does not specify which enumeration and simulation algorithm to use. The next section illustrates how justification algorithms (like the PODEM justification algorithm reviewed in Section 6.2.3.1) can be extended to become enumeration algorithms.

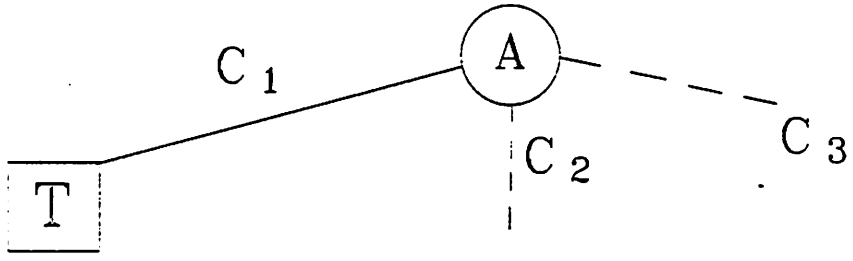
6.3.4.1 Enumeration as an Extension of Justification

In general, to perform equivalence checks, all values of input combinations that produce a 1 (0) at the output of a circuit A have to be found. That is, A^{ON} (A^{OFF}) has to be found. Of course, it is of interest to find the most compact representation of A^{ON} (A^{OFF}). If setting a *subset* of input values generates a 1 (0) at the output, it can be inferred that the input combinations produced by setting the remaining unset inputs in all possible ways will produce a 1 (0) at the output. Setting a small subset of input values to justify a 1 (0) to the output results in determining a large subspace of A^{ON} (A^{OFF}).

The difference between enumeration and justification is that in enumeration one is *not* satisfied when *a single* assignment to primary inputs which creates a 1 (0) at the output has been found. *All* possible input combinations have to be found. To make sure that the entire space is examined, backtracking at the decision points is performed recursively, beginning from the deepest decision point, and an alternative assignment is tried. The process continues until all decision alternatives are examined.

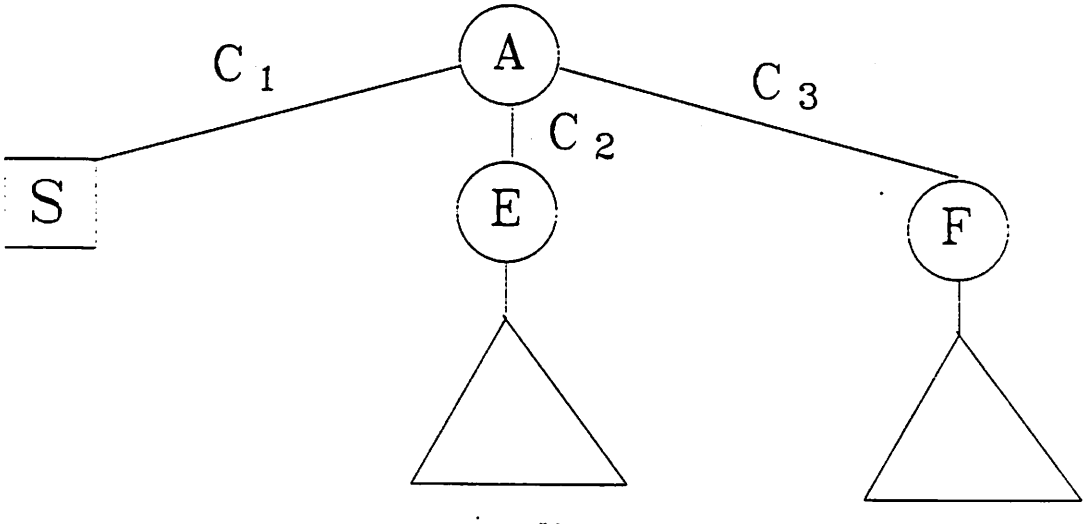
Thus, most justification algorithms, become enumeration algorithms after proper modification of the termination criterion. Consider a decision node in the decision tree shown in Figure 6.2(a) that a justification algorithm follows. Three choices are available at node A: c_i, i = 1, 2, 3. c_1 is chosen and eventually leads to a successful justification denoted by a "T" (Termination) leaf node. In a justification problem the goal is to reach a successful justification; the process backtracks and tries the next available choice only when the current choice of decision does not yield a solution. There is no need for the process to return to the current decision node when a solution can be found for the present choice. So c_2 and c_3 are not tried and the dashed lines reflect this. However, in the enumeration application, where the aim is to enumerate, implicitly and exhaustively, all the possible justifications, even when a successful justification is found, the process still needs to return and try all the remaining choices. This is illustrated by Figure 6.2(b) where both c_2 and c_3 are followed regardless of the outcome of c_1.

LOVER incorporates enumeration algorithms based on PODEM backtrace algorithm and



--- choices not tried in the justification problem

(a)



(b)

Fig. 6.2 Justification versus Enumeration

the VERIFY justification algorithm.

6.4 Sequential Logic Verification

6.4.1 Introduction

Much less work has gone into verifying sequential designs as compared to combinational designs. Sequential circuit verification is a considerably harder problem. If a one-to-one correspondence can be made between the latches/states of two sequential circuits, then the problem reduces to one of combinational logic verification, but this is not always possible.

6.4.1.1 Difficulties in Sequential Logic Verification

Verifying the equivalence of two combinational logic designs has been shown to be NP-complete [134]. This means that there is little hope of finding an algorithm whose running time is bounded by a polynomial in the number of inputs to the designs. Given two n-input logic designs, in the worst case, 2^n possible input combinations may have to be verified.

Sequential logic verification is even more difficult. A trivial algorithm based on Eq. 6.4, checking all possible input sequences for equivalence, would incur huge CPU time expenditure. The State Transition Graph of a finite automaton is a more compact representation of the machine. Checking the equivalence of two machines by checking the equivalence of their State Transition Graphs is a comparatively easier task than verification by Eq. 6.4. However, given a description of a machine in the RT or logic levels, the State Transition Graph of the machine or an equivalent representation has to be extracted from the description. Extraction from logic-level descriptions can be a very time and memory intensive operation.

In fact, given a logic-level sequential circuit with N_S latches and N_I inputs, up to 2^{N_S} possible states can exist for the machine (some of these states may not be reachable from the given reset state of the machine, in which case they become irrelevant during verification). Each of these states has 2^{N_I} edges fanning out of it, if the edge is represented by a minterm. The number of edges in the State Transition Graph of the sequential circuit thus is $O(2^{N_I+N_S})$. The State Transition Graph of the circuit has to be extracted from the logic description to check for equivalence. Even if the CPU time required to extract all the edges in the State Transition Graph (which is required for equivalence check) is affordable, memory

requirements for storing these edges may be too expensive.

6.4.1.2 The New Approach

The new approach that has been developed involves extracting the State Transition Graphs (STGs) of the two finite automata – from the register level or the gate level circuit. These two STGs are then checked for equivalence. While extracting the second STG from the logic-level circuit, the use of don't care information from the first STG enables us to reduce the number of states and the number of edges in the second STG. The number of states of a finite automaton grows exponentially with the number of latches in the circuit. However, for large machines, the number of states actually visited given the input sequences is typically a small fraction of the total number of possible states. This is especially true if a state assignment program [26] has been used in the synthesis process which minimizes combinational logic and may or may not produce a minimum bit encoding of the states. The use of invalid output sequence information and implicit cube enumeration (Section 6.4.2.2) on the combinational logic part of the gate level finite automaton enables us to detect these invalid states (actually decode the internal state encoding) thereby reducing the complexity in checking equivalence. Given a large number of states, the number of edges in the STG may be prohibitively large. Allowing only valid input sequences enables us to reduce the number of edges in the STG, again reducing the time required to check for equivalence. As opposed to the symbolic Boolean manipulation [124] techniques used in [128] and [92] for sequential circuit verification, modifications of backward justification algorithms are used on the combinational logic parts to *implicitly enumerate* the input combinations.

A two-phase enumeration-simulation algorithm to verify the equivalence of two logic-level finite automata given a reset state/transfer sequence for each machine has been developed. The State Transition Graph (STG) of one of the logic automata is enumerated using a generalized PODEM-based [131] enumeration algorithm for sequential circuits. That is, paths from the reset state in the STG are *dynamically enumerated* from the first logic automa-

ton and simulated on the second logic automaton. One of the attractive features of this approach is that the entire STG (all the edges) of the logic automaton does not have to be stored, merely *a single path* in the STG (whose length can be constrained), is stored. This verification algorithm is efficient both in terms of memory and CPU time usage.

The algorithms used in the extraction of STGs of Deterministic Finite Automata (also known as DFAs or Moore machines) described at the gate level exploiting don't care information are described in Section 6.4.2. The extraction of the STG from the register-transfer level finite automaton and the algorithm used for formally checking the equivalence of two DFAs represented by STG's are described in Section 6.4.3 & 6.4.4 respectively. The algorithms described for DFAs (Moore machine) in Section 6.4.2 are extended to Non-deterministic Finite Automata (NFAs or Mealy machines) machine) in Section 6.4.5. The enumeration-simulation approach for the verification of two logic-level finite automata with specified reset states is described in Section 6.4.6. Results for several examples for both approaches are given in Section 6.4.7.

6.4.2 Extraction of Moore Machine State Graphs

A general model for a Moore machine, given by a six-tuple (Q,I,O,NSL,OL,q_0), at the logic level is shown in Figure 6.3. The output combinational logic block, OL, performs the Q to O mapping. The next state logic block, NSL, generates the next state given the present state and input vector, performing the $Q \; X \; I$ to Q mapping and q_0 is the initial state. D latches or flip-flops constitute the memory elements. The two combinational logic blocks will henceforth be referred to as the OL and NSL blocks respectively. The Moore machine is constructed in such a fashion that the output is only a function of the present state and not a function of the present input vector. For a Moore machine we have

$$o_i = f_i(ps_1, ps_2, .., ps_{N_s}) \quad 1 \le i \le N_o$$

$$ns_i = g_i(ps_1, ps_2, .., ps_{N_s}, i_1, i_2, .., i_{N_i}) \quad 1 \le i \le N_s$$

where ps_j and ns_j denote the present state and next state values respectively and i_k denotes the

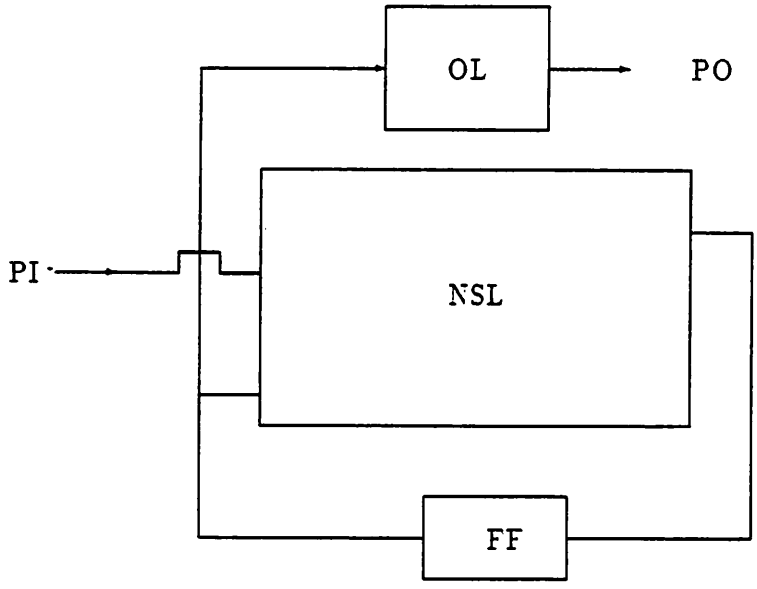


Fig. 6.3 General Moore Machine Model

input.

6.4.2.1 Extraction Task

Given a logic-level description consisting of gates and latches of a machine the goal is to extract the State Transition Graph of the corresponding DFA. This can be done in a number of ways.

One method which has been used in this context [128] [92] is *symbolic Boolean simulation* [123] of the combinational logic blocks in the circuit which expresses each output as a algebraic function (with Boolean operations) of the inputs. Given these algebraic functions, the edge transitions between two arbitrary states can be expressed as a conjunction of these functions and possibly their complements. Thus the equations corresponding to f_i and g_i can be extracted from the OL and NSL blocks of the machine and the edge transitions can be expressed as a function of the g_i and $\overline{g_i}$ and the output for each state can be found using f_i. Unfortunately, for combinational logic blocks with a large number of gates the size/length of

the equations becomes prohibitively large and the extraction process becomes inefficient. Also, before including an edge in the State Transition Graph, it has to be checked for satisfiability, which is an NP-complete operation [120].

Flattening the logic circuit is another alternative. Flattening involves reducing the combinational logic blocks into two level (PLA) form. Given a truth table for these combinational logic blocks the edges in the transition table can easily be found by inspection. Flattening however may require exponential CPU time and memory requirements and is not viable in many cases.

A third approach involves using backward justification algorithms to enumerate the ON and OFF sets of a logic function. Enumeration has been successfully used in the combinational logic verification problem [29]. A method for STG extraction, using an enumeration algorithm based on the PODEM justification algorithm, while exploiting don't care information is described in Section 6.4.2.4.

6.4.2.2 Implicit Cube Enumeration using PODEM

By modifying the termination condition of the justification algorithm used in PODEM (described in Section 6.3.2.1), both the ON-set and OFF-set can be implicitly, but exhaustively, enumerated. This is illustrated in Figures 6.4 and 6.5.

An example circuit with 5 inputs and a single output is shown in Figure 6.4. The decision tree used while enumerating the ON-set of the output is shown in Figure 6.5. In general, two decision trees are required: one for the ON-set verification and the other for the OFF-set.

Each node in the decision tree represents a primary input (PI) assignment. Initially, all primary inputs are assigned unknown values (corresponding to the node START in the decision tree of Figure 6.5).

Given an initial objective, i.e. to set a primary output line to a 1 or 0, a path is traced from the objective line backwards to a primary input to obtain a PI assignment. A 1 initial

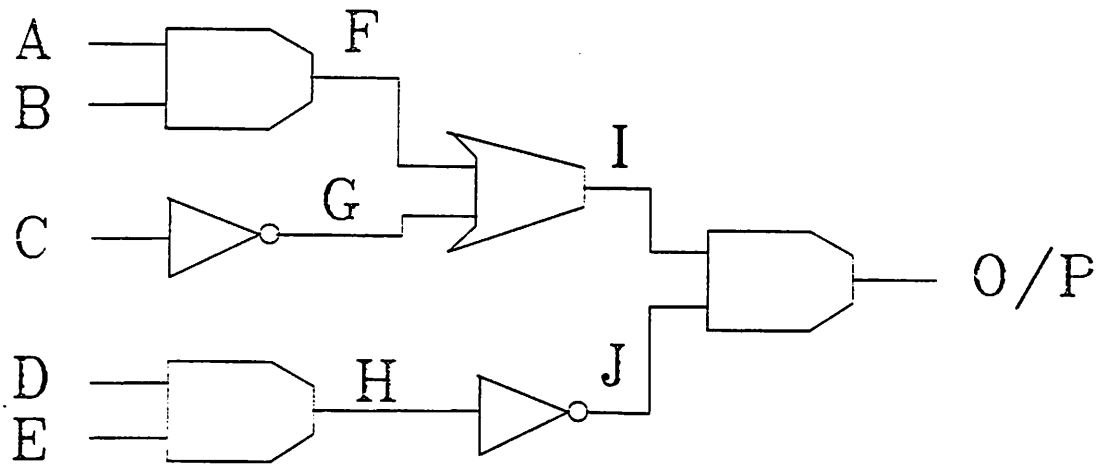


Fig. 6.4 An Example Circuit

objective corresponds to the enumeration of the ON-set and a 0 initial objective corresponds to the OFF-set enumeration. In this example, the objective was to set the primary output line to a 1. The first PI assignment was to set input C to 0 (Figure 6.5).

After each new PI assignment, the circuit is simulated using the current set of PI assignments to see if the value at the objective line has been set up. If not, the backtrace process continues. For example in Figure 6.5, after setting input C (to 0) the value of the primary output is unknown, so the backtrace process continues, selecting and setting input D (to 0).

If the desired value has been achieved, a cube in the corresponding set has been found. In the example, after D has been set, the desired value of the output (= 1) has been set up. The cube --00- has been enumerated in the ON-set of the circuit (Figure 6.5). If verifying

against another circuit, this cube would be simulated on the other circuit.

If the opposite value has been set up, the algorithm backtracks to the last PI assignment, tries the alternative value and flags the node to indicate that both assignment choices has been tried. If the alternative has already been tried, the node is removed and the backtrack process continues until an unflagged node with a possible alternative is reached. The backtrack process is also applied when a desired value has been set up at the objective line. After enumerating cube --00-, the algorithm backtracks to the last PI assignment, namely D, and sets it to a different value of 1 (Figure 6.5). This is different from PODEM in which the enumeration process terminates when the desired value is set up at the objective line. When the decision tree is found to be empty in the backtrack process, the total input space for the corresponding set has been implicitly, but exhaustively, enumerated.

6.4.2.3 Extraction Using Enumeration

The inputs to the logic-level extraction program are the combinational logic blocks OL and NSL and the State Transition Table, *STT*1, generated either from a RTL description, or another logic-level description. The output is a State Transition Table *STT*2. The extraction of a State Transition Table from an ISP-like RTL description is described in Section 6.4.3. The input and output sequence don't care information can be derived from *STT*1 and used in generating *STT*2. The following steps are performed during the extraction process.

(1) The ON-set and OFF-set is found for each output of the OL and NSL blocks. using implicit cube enumeration. C_A^{OFF} (C_A^{ON}) is denoted as the OFF-set (ON-set) for a line A.

(2) If there exist N_s latches in the logic description, i.e. N_s outputs to the NSL block, the number of states which can exist in the corresponding finite automaton is 2^{N_s}. However, given a set of *valid input sequences*, some of these states may never be reachable from the starting state q_0. For example, given a 4-bit encoding of states in a 9 state finite automaton, 7 (= 2^4 - 9) such states exist. These states which can never be reached from q_0 can be discarded, since one wishes to verify the outputs of the two machines under the valid input set alone. Finding

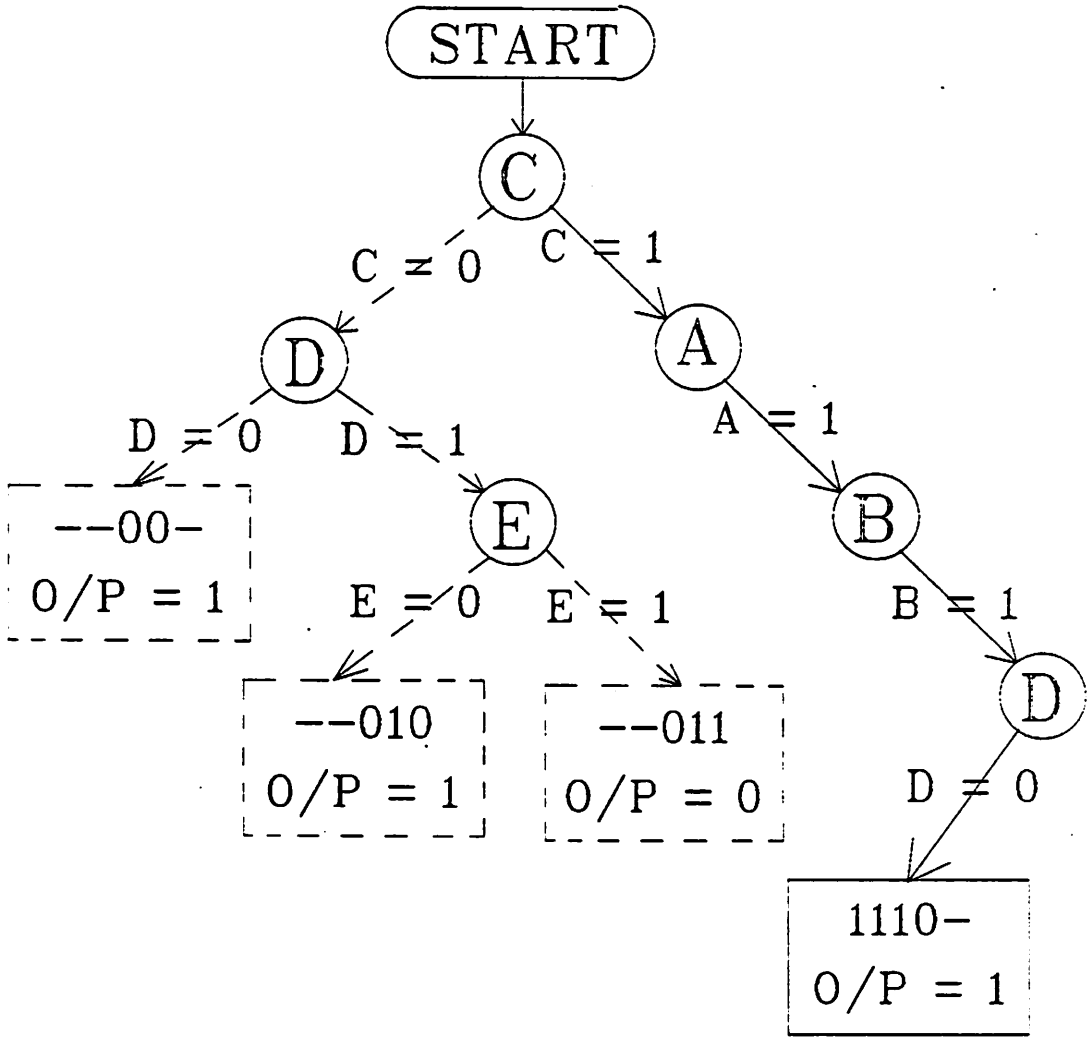


Fig. 6.5 Decision Tree of LOVER-PODEM

these *invalid states* is non-trivial since the internal encoding of the states in the logic-level finite automaton is not known. However, the output sequence information can be used to find some, if not all, of these invalid states.

The set of valid output vector cubes $VO = \{vo_1, vo_2, .., vo_z\}$ is constructed by inspecting *STT*1. Constructing this set takes $O(E)$ time, E being the number of edges in *STT*1. If the two machines are equivalent, all output vectors in *STT*2 $\in \overline{VO}$. Using a logic minimizer like ESPRESSO-II [23] $\overline{VO}$ is found. A logic minimizer is invoked so as to reduce following computations which are proportional to the number of literals (0 and 1) in $\overline{VO}$.

Lemma 1: If a state q produces an invalid output vector $vo \in \overline{VO}$, q is an invalid state.

Lemma 1 is the basis of finding invalid state encodings while generating *STT*2. The following section of code illustrates how invalid states are found given $\overline{VO}$ and the ON and OFF-sets of the OL block outputs.

$INVAL\_S = \phi$
foreach(cube $iv \in \overline{VO}$) {

/* *find all input cubes producing this invalid output cube* */

$$INVAL\_S = INVAL\_S \cup \{ \bigcap_{i \in \Gamma^+}^{N_s} C_i^{ON} \cap \bigcap_{i \in \Gamma^-}^{N_s} C_i^{OFF} \};$$

}

---

where iv is a cube of length N_s (inputs to the OL block), $\Gamma^+ = \{i : iv_i = 1\}$ and $\Gamma^- = \{i : iv_i = 0\}$. The - (don't care) literals in the cube iv are ignored in this computation. For example, given a cube iv = 10–1, $C_1^{ON} \cap C_2^{OFF} \cap C_4^{ON}$ is computed and added to *INVAL_S*. As one can see the number of cube set intersections performed in this step is proportional to the number of 0 and 1 literals in $\overline{VO}$ which is why a fast logic minimization while computing $\overline{VO}$ is employed. This technique cannot find an invalid state which produces a valid output cube. However, in large examples, typically a significant number of invalid states can be found using a small fraction of the total CPU time spent in the verification process as indicated in Section 6.4.7.

(3) *INVAL_S* is complemented to find the set of valid states *VAL_S*. The set of all valid input

cubes *VAL_I*, which contains all distinct input vector combinations (cubes or minterms) in *STT*1 is then constructed in $O(E)$ time, E being the number of edges in *STT*1. Typically, given a RTL description, the size of *VAL_I* is quite small in a cube representation. The edges in *STT*2 are generated using the NSL block enumerations. The section of code shown below illustrates this process.

---

foreach(state $QF \in VAL\_S$) {

/* *find all inputs to NSL producing this state as output* */

$INPUT\_PS = \{ \bigcap_{i \in \Gamma^+}^{N_i+N_s} C_i^{ON} \bigcap \bigcap_{i \in \Gamma^-}^{N_i+N_s} C_i^{OFF} \}$;

foreach(cube $ip \in INPUT\_PS$) {

$input = ip\langle 0{:}N_i - 1\rangle$;
$QP = ip\langle N_i{:}N_s+N_i - 1\rangle$;

if($QI \in VAL\_S$ AND $input \in VAL\_I$)
include edge $QI \rightarrow QF$ on *input* in *STT*2 ;
}
}

---

where *INPUT_PS* is a cube of length N_i+N_s (inputs to the NSL block), $\Gamma^+=\{i{:}QF_i=1\}$ and $\Gamma=\{i{:}QF_i=0\}$. For example, given $N_s = 2$ and $N_i = 2$, and $QF = 10$, first $INPUT\_PS = C_1^{ON} \bigcap C_2^{OFF}$ is computed. Then, if a cube, say 010-, exists in *INPUT_PS*, an edge between 01 (the first two bits of 010-) and *QF* is added to *STT*2 on the input combination 0- (the last two bits of 010-).

Checking to see if $input \in VAL\_I$ can significantly reduce the number of edges in *STT*2. The output corresponding to each state is found by simulating the state vector on the OL block.

(4) A state s_2 in *STT*2 which produces the same output as s_1 the starting state in *STT*1 is picked as the starting state of *STT*2. All the states which cannot be reached from s_2 in *STT*2

(if any) are deleted.

Cube set intersections require time complexity $O(n^2+m^2)$ given two sets of cubes with n and m cardinality. Two things are done to speed up cube intersections during the invalid state detection and edge generation process. Firstly, intersections within each cube (*iv* or *ip*) are performed in an order of increasing cube set cardinality so the number of intersected cubes at any point is minimum. Secondly, invalid output/state cubes are grouped in such a fashion that repetition of intersections between ON/OFF-sets of the same pair of outputs is minimized, without storing more than two intermediate results. This technique cannot find an invalid state which produces a valid output cube. However, in large examples, typically a significant number of invalid states can be found using a small fraction of the total CPU time spent in the verification process as indicated in Section 6.4.7.

6.4.3 Extraction from RTL Descriptions

6.4.3.1 Input RTL Description

The input description is at the register-transfer level, and has the following main constructs:

(1) Procedures and functions

(2) If and Select for control/branching

(3) Loops – While and For.

The description is ISP-like [17] except that clock boundaries are explicitly delineated using a *wait* statement on the rising/falling edge of the clock (or clock phase) ϕ_1. A sample input

description is shown in Figure 6.6.

6.4.3.2 Extraction from RTL Description

The extraction process is only concerned with the control flow in the RTL description, we wish to generate a DFA controller for the input specification. The DFA will have the control variables as the inputs (e.g. instruction bits, ALU status bits) and assert outputs (e.g. register load, ALU add) depending on the present state.

The following steps are carried out during the DFA extraction:

(1) In the first pass, a one-to-one correspondence between the controlling input variables and output signals of the RTL description and the logic level description is made. For example, in the description shown in Figure 6.6, the variables *run* and *pb* are two inputs. The output signals associated which each micro-instruction are specified along with the RTL description, e.g. the micro-instruction *MA = 0 @ pa* may require (a) the load signal of the *MA* register be high and (b) the load signal of the ALU be high with the ALU operation code 111.

(2) Given the inputs and outputs, the description is parsed starting from the routine MAIN and entering and exiting all procedures in the order they are called in. If a micro-instruction is encountered then the corresponding outputs of the micro-instruction are asserted in the output of the present state. If a wait(ϕ_1) statement is encountered a new state is generated.

(3) If a branch statement like IF or SELECT is encountered, two or more states are generated depending on the number of branching conditions and a transition edge between the previous state and each possible present state is created with the corresponding input pattern. The extraction process continues with each possible present state recursively enumerating all the possible combinations. The recursion may terminate at the end of the MAIN routine or terminate if any input condition is violated.

```
MAIN()
BEGIN
 run = 1;
 WHILE run DO
  BEGIN
   fetch_instruction();
   effective_address();
   execute();
   IF interrupt.enable EQL 1 THEN
     IF interrupt.request EQL 1 THEN
    BEGIN
     MBR = PC ;
     MP[0] = MBR ;
     PC = 1;
     wait(φ1);
    END
  END
END


! subroutine for effective address calculations

ROUTINE effective_address()
BEGIN

SELECT pb FROM
 [0]: BEGIN MA = 0 @ pa; END
 [1]: BEGIN MA = last.pc<0:4> @ pa; END
ENDSELECT ;

wait(φ1);

IF ib EQL 1 THEN
BEGIN
 MA = MP[MA] ;
 wait(φ1);
END

END  ! end of routine effective_address()
```

Fig. 6.6 Sample Input RTL Description

The State Transition Graph for the routine **effective_address()** is shown in Figure 6.7.

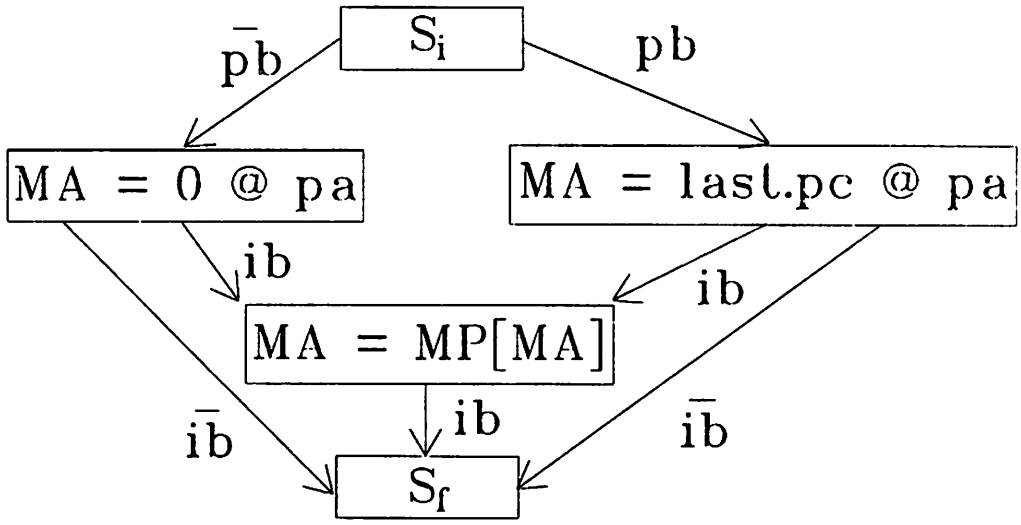


Fig. 6.7 State Transition Graph for *effective_address()*

only the local inputs and outputs are shown.

6.4.4 DFA Equivalence

6.4.4.1 Verifying the equivalence of two DFA's

Verifying that two incompletely specified finite automata are equivalent is done using a modified form of the method used to test completely specified finite automata [135]. Given the completely specified finite automata M_1 and M_2 accepting languages $L(M_1)$ and $L(M_2)$ respectively, $(L(M_1) \cap \overline{L(M_2)}) \cup (\overline{L(M_1)} \cap L(M_2))$ is accepted by some finite automaton M_3. M_3 accepts a non-empty language if and only if $M_1 \neq M_2$.

The languages accepted by finite automata are essentially the State Graphs of the finite automata. Given two State Graphs, a composite State Graph is constructed, which is effectively the *exclusive-or* of the two State Graphs. If a path exists from the starting state of the State Graph to any of the final states, the two original machines are not equivalent, and this path constitutes a *differentiating sequence*. If no such path exists, the machines are equivalent. A simple inductive proof [135] establishes this result. The construction of this composite State Graph is described in the sequel.

One way of handling incompletely specified DFAs [128] is to enter a third sequential machine, D, which accepts the inputs M_1 and M_2 don't accept and perform an extra intersection. Machines M_1 and M_2 are equivalent relative to the don't care condition, if

$$\overline{L(D)} \cap L(M_3)$$

is empty. However, a more efficient way is to add a single dummy state in each of *STT*1 and *STT*2 to which all the don't care transitions fanout to.

```
foreach(state q ∈ STT1 ) {
  fanout = φ ;

   foreach( fanout edge E from q )
     fanout = fanout ∪ E.input ;

   dcfanout = ¬fanout ;
   add dcfanout edges from q to dummy_s ;
}
```

The same is done for *STT*2. *STT*1′ and *STT*2′ are now completely specified DFA's and if they are equal it follows that *STT*1 and *STT*2 are equal.

A composite finite automaton *STT*3 given *STT*1′ and *STT*2′ is constructed as shown by the pseudo-code shown below. The states in *STT*3, are unordered pairs of states in *STT*1 and *STT*2. Each pair of edges in *STT*1′ and *STT*2′ are intersected. If the intersection is non-

empty, an edge is added between the state in *STT*3 given by the fanin and fanout states of the first edge and the state in *STT*3, given by the fanin and fanout states of the second edge.

```
foreach( edge e1 in STT1' ) {
  foreach( edge e2 in STT2' ) {
    if ( e1 ∩ e2 ≠ φ )
      include edge {e1.From , e1.To} → {e2.From , e2.To} in STT3 ;
  }
}
```

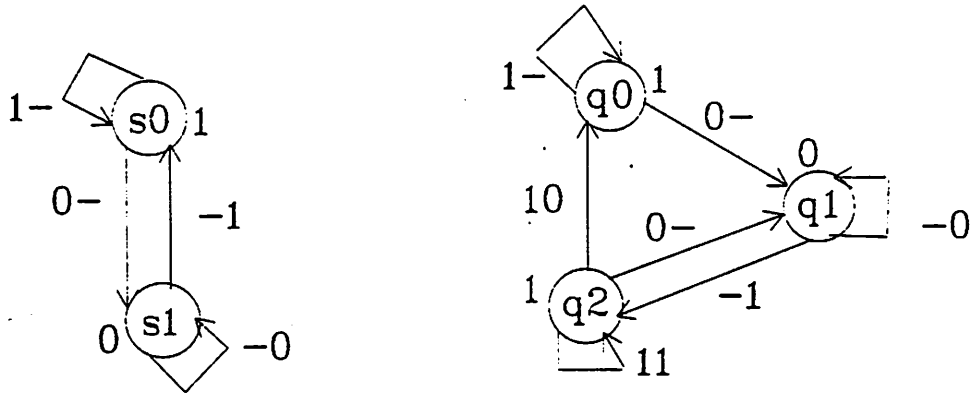
---

where *From* and *To* denote the fanout and fanin nodes of an edge. *STT*3 may have as many as $N_1 * N_2$ states given N_1 and N_2 states in *STT*1′ and *STT*2′ respectively. If a path exists from $\{s_1, s_2\}$ to any final node in *STT*3 the machines are not equivalent (if a path does not exist *STT*1 = *STT*2).

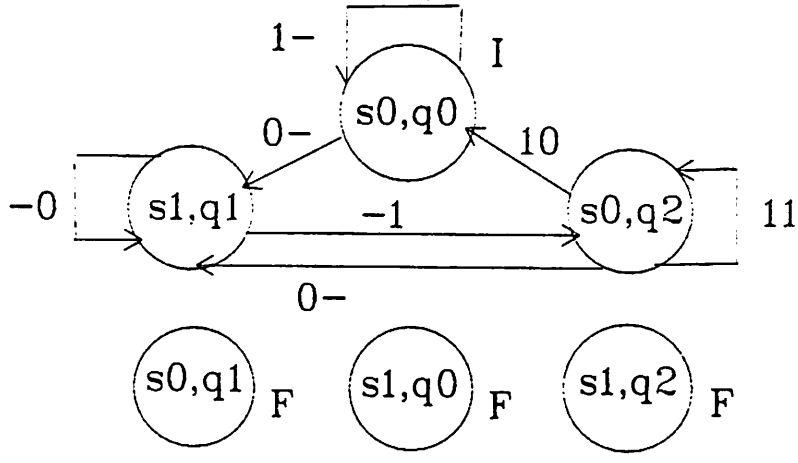
The set of final nodes in *STT*3 have to be found. A set of final states exists for each output of *STT*1 (and *STT*2). Assume, for simplicity, that *STT*1 has a single output. Then, if the starting state of *STT*1 asserts the output of 1 (0), the final states in *STT*1 for this output are all the states which assert an output of 0 (1). Given *STT*1 and *STT*2, each with a single output, the final nodes in *STT*3 are all the pairs of states in *STT*1 and *STT*2 such that one state of the pair (not both) is a final state of either *STT*1 or *STT*2. The final nodes in *STT*3 for multi-output finite automata are found as illustrated below.

```
for( i = 1 To No ) {
  foreach( state {q1 , q2} ∈ STT3 ) {
    if (( q1.output_i = 1  AND  q2.output_i = 0 )
                          OR
       ( q1.output_i = 0  AND  q2.output_i = 1 ))
      mark {q1 , q2} as a final node ;
  }
}
```

An example composite State Graph construction is illustrated in Figure 6.7. The two original State Graphs are equivalent (one is state minimized form of the other) and are shown in Figure 6.8(a). They have a single output. The output bit corresponding to each state in the two graphs is indicated near each state. The composite State Graph is shown in Figure 6.8(b). Since the original State Graphs are equivalent, there is no path from the initial state (marked



(a) Two state graphs



(b) The composite state graph

Fig. 6.8

I) to any of the final states (marked F) in the composite State Graph.

6.4.4.2 Starting or Reset States

In general, the starting state is not specified for logic-level finite automata, since the encoding of states is not known. The starting state for the register-transfer level finite automaton is easily found – it is merely the first state generated during extraction. So given s_1 in the RTL automaton and the output vector asserted by s_1, a set of all states S_2 is found which assert the same output in the logic-level automaton's STG. If for any $s_2 \in S_2$, no path exists from the $\{s_1,s_2\}$ to any final node then the machines are equivalent under $\{s_1,s_2\}$. The construction of the composite FSM is not affected by knowing or not knowing the starting state of the logic-level automaton – only the path finding process is repeated for each pair of possible starting states.

Given reset states/transfer sequences for logic-level finite automata, a two-phase enumeration-simulation approach can be used to verify the equivalence of two machines. This approach is described in Section 6.4.6.

6.4.4.3 Sub-Routine or Sub-Module Verification

Very often it is the case that one wishes to verify the operation of a sub-routine in a RTL description against a logic implementation. Since the logic-level description is not hierarchical it is difficult to make a correspondence across the two levels. Verification of the entire description in order to verify the sub-routine is obviously inefficient.

The extraction algorithms described so far are capable of extracting only that portion of the logic-level finite automaton's STG which corresponds to the given sub-routine in the RTL description without having to extract the entire STG of the automaton. Verification of a sub-routine can be done correspondingly faster than the entire RTL description.

Sub-routine verification proceeds as follows:

(1) The STG of the sub-routine is extracted treating the sub-routine as the entire input RTL description.

(2) Given this STG, the valid input and output sequence sets are found and invalid states in the logic-level finite automaton are detected using this information. These invalid states may contain states which are valid states in other sub-routines in the complete RTL description. Detecting these states produces a smaller STG and verification is correspondingly quicker.

6.4.5 Extension of Algorithms to Mealy Machines

A finite automaton whose output is associated both with the input and the state is called a Mealy machine. A Mealy machine is also a six-tuple (Q,I,O,NSL,OL,q_0), where all is as in the Moore machine, except that OL maps $Q \times I$ to O. For a Mealy machine we have:

$$o_i = f_i(ps_1,ps_2,..,ps_{N_s},i_1,i_2,..,i_{N_i}) \quad 1 \le i \le N_o$$

$$ns_i = g_i(ps_1,ps_2,..,ps_{N_s},i_1,i_2,..,i_{N_i}) \quad 1 \le i \le N_s$$

where ps_j and ns_j denote the present state and next state values respectively and i_k denotes the input.

Finding the complement of a NFA requires conversion to a DFA [135]. Hence, to verify the equivalence of two Mealy machine STG's by constructing $(L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2)$ we need to convert them into Moore machine STGs. This transformation is always possible, however the resulting Moore machine will have a larger number of states and edges than the original Mealy machine. The number of extra states required is $\sum_{i=1}^{N}(D_i - 1)$ where D_i is the number of different output vectors for state i, and N is the number of states in the Moore machine.

The invalid state detection process is different for a Mealy machine because f_i is now a function of both ps and i.

Lemma 2a: If $\{q,i\}$ always produces a invalid output vector for all $i \in VAL\_I$, then q is an

invalid state.

Lemma 2b: If $\{q,i\}$ always produces an invalid next state for all $i \in VAL\_I$, then q is an invalid state.

where $\{q,i\}$ implies that input i is being applied to the machine presently in state q. VAL_I is the set of valid inputs. For Lemma 2b to apply, at least one invalid state has to be detected using Lemma 2a. The two lemmas are alternately used until neither apply.

6.4.6 Verification by Enumeration and Simulation

6.4.6.1 Introduction

Two combinational logic circuits can be verified using an enumeration-simulation approach – The ON and OFF-sets of the outputs of the first circuit can be enumerated (as described in Section 6.3.2) and simulated on the other circuit to check if all input combinations produce the same values. This approach was first taken in [29]. The efficiency of this approach lies in the fact that cubes are enumerated and not minterms, i.e. implicit but exhaustive enumeration on the input space is performed.

The same approach can be generalized to sequential circuits where no correspondence exists between memory elements. The sequential circuits may represent Moore or Mealy machines – this approach deals uniformly with either kind of machine. Paths starting from the reset state of the first finite automaton (usually the one with the fewer number of latches) are enumerated. Each path consists of a sequence of inputs and asserted output values representing a sequence of edges in the STG of the automaton. (The input patterns are generally cubes and not minterms). These paths are acyclic in the sense that no fanout state pair, where each fanout state corresponds to one of the two machines being verified, for equivalence, appears more than once in the edges in the path. The sequence of input cube patterns in the path is simulated on the second automaton to see if the same output values are asserted; the same next states need not be asserted. If so, the path enumeration continues. If not, the machines

are declared non-equivalent. The attractive feature of this approach is that the entire STG of the finite automaton is not stored, merely a single path is stored. Only the valid states in the automaton's STG are visited, thus invalid state detection in this approach comes as a bonus.

6.4.6.2 Enumerating paths in the STG

The inputs to the path enumeration program is the combinational logic block of the finite state machine and information about latch inputs and outputs, i.e. present and next state lines. In the case of the Moore machine the two blocks in Figure 6.3 are merged into a single block. *This approach deals uniformly with Moore or Mealy machines.*

The STG enumeration proceeds in a depth-first fashion beginning with sequential cube-enumeration of all fanout edges from the given reset state, q_0. Whenever a new edge is found, it is added to the current path if a certain condition is satisfied. Let the next state produced in M_1, the machine being enumerated on, be n_1 for the current edge in the path. The corresponding state in M_2, the machine being simulated on, is n_2. The condition is that the state pair (n_1, n_2) should not appear more than once in the path. If the condition is not satisfied, the fanout of the current starting state (the state the current edge fans out of) continues to be enumerated, with the current path unchanged. Else, the edge is added to the path and n_1 is picked as a new starting state. The procedure is repeated until no more edges can be enumerated. All the edges in the complete STG will be implicitly and collectively enumerated in the paths. There is a hard limit, LIMIT, on the number of edges in each path to restrict memory requirements. Because of this limit un-enumerated states have to be "remembered" and placed on a stack for later enumeration. At any stage all the valid states in the STG and a single path is stored. The pseudo-code shown below illustrates the global strategy used. **EnumerateDfs()** is initially called with the reset state, q_0, of the machine and with an empty valid state set, VAL_STATES.

Like the general approach, this algorithm does not have to deal with the invalid states in the STG of the logic-level finite automaton. Only the states reachable from the reset state of

the machine are visited during the enumeration.

```
EnumerateDfs(State, VAL_STATES)
{

  UN_ENUM_STATES = ϕ;
  PathEnumerate(State, ϕ, UN_ENUM_STATES, VAL_STATES);
  foreach( state q ∈ UN_ENUM_STATES ) {
    EnumerateDfs(q, VAL_STATES) ;
  }
}
```

```
PathEnumerate(State, Path, UnEnumStates, ValStates )
{

  ValStates = ValStates ∪ State ;
  while (State.Enumerated is FALSE) {

    /* using PODEM, enumerate the next Edge fanning out from State */
    Edge = EnumerateStateFanout(State) ;
    simulate Edge on M2 ;

    if (Edge.Output(M1) ≠ Edge.Output(M2))
      M1 and M2 are not equivalent ;

    /* check if the state pair of the edge exists in the path */
    if ( (Edge.FaninState(M1), Edge.FaninState(M2)) ∈ Path.Spairs) {
      continue ;
    }
    else {
      ValStates = ValStates ∪ Edge.FaninState(M1) ;
      Path = Path + Edge ;
      update Path.Spairs ;
      if (Path.Length ≥ LIMIT ) {

        /* limit exceeded, push fanin state on un-enumerated stack */
        UnEnumStates = UnEnumStates ∪ Edge.FaninState(M1) ;
      }
      else {

        /* depth first enumeration continues */
        PathEnumerate(Edge.FaninState(M1), Path, UnEnumStates, ValStates) ;
      }
    }
```

```
    }
}
```

The algorithm used to enumerate the fanout edges from a state, **EnumerateStateFanout()**, is an extension to the implicit enumeration algorithm of PODEM [131]. Initially, the values of all primary inputs and next states of the logic-level finite state machine are set to unknown. The logic-level circuit is simulated with the present state lines fixed at their specified values. An unknown next state line is then picked and a path is backtraced from it to an unknown primary input with the objective to set the value of the chosen next state line to a known one. A 1 or 0 is assigned to that primary input. The circuit is then simulated again. The setting of primary inputs and simulation of the circuit is continued until all next state lines are set to known values – a fanout edge is enumerated. Whenever an edge is found, the algorithm backtracks to where a primary input is first set to a known value and assigns it an opposite value. The simulation and primary input setting are then repeated. When no more backtracking can be done, all the edges from a state are implicitly, but exhaustively enumerated.

An alternative to the backtracing/backtracking approach to STG enumeration described above is forward simulation on the input space given a starting present state. The forward simulation process begins with all the input lines set to unknown values. Inputs are set randomly to 0 or 1 in a pre-specified order till all the next state lines are all set to known values. Backtracking on primary input values is done after setting all next state lines. However, this approach is less efficient than the approach described earlier because a primary input value may be unnecessarily set in order to set the next state lines. This can lead to a great amount of redundant simulations. On the contrary, in the backtracing/bracktracking approach, the backtracing process makes sure that the next primary input to be set and the simulation fol-

lowing the value-setting always contribute to the setting of the next state lines.

6.4.6.3 Simulating paths

Every time an edge, corresponding to an enumerated input combination is produced, it is simulated on the second machine, M_2, to find the next state associated with the edge. The edge is simulated with the knowledge of the present state of M_2 (which is the next state of the previous edge in the path). Simultaneously, the output of the edge are checked to see if they are the same across M_1 and M_2. If they are, verification continues, else the machines are declared not equivalent.

Since the input patterns in the path are in general cubes and not minterms, *cube-splitting* [29] may be necessary on the input lines to produce known output and next state values. In practice, since parallel vector simulation is used, 16 or 32 edges may be stored and simulated simultaneously on the second automaton.

6.4.7 Examples and Results

For verifying RTL descriptions against logic-level descriptions, I give results for four different examples in Table 2. The statistics of these examples are described in Table 1. The first example is small and the total CPU time for extraction and verification is under 12 CPU seconds on a VAX 11/8650. Example 2 is large and is verified in about 1.5 minutes. The third example is a very large machine with 128 states and is also successfully verified within 10 CPU minutes. The first three examples are Moore machines, example 4 is a Mealy machine comparable in size to example 2, but takes almost twice as long due to conversion to a larger Moore machine for equivalence checking.

Note that for all the examples, the invalid state detection time is a small fraction of the total CPU time, but a very significant number of invalid states are found. A minimum bit encoding of states minimizes the number of invalid states in a logic level finite automaton, but typically state assignment programs like KISS [26] use encodings with a few more bits than

the minimum necessary to implement the machine since great savings in combinational logic can be made with extra state field bits.

Verifying equivalence between the two State Transition Diagrams has a time complexity of $O(E_1 * E_2)$ where E_1 and E_2 are the number of edges in the two machines. The number of edges in a machine grows approximately as the square of the number of states in the machine. Thus finding invalid states and invalid edges is a big gain – **Example 3**, when run without using don't care information, required 68 minutes to verify.

I also give results for verifying logic-level automata with known reset states for equivalence using the enumeration-simulation approach described in the previous section. Table 3 gives both the statistics and the CPU times required for six examples which have

EXAMPLE	RTL Description		Logic Level Description				
	#states in STT	#edges in STT	#inputs	#outputs	#latches	OLB #gates	NSLB #gates
1	5	10	2	2	4	9	15
2	33	300	10	10	7	220	388
3	128	529	27	56	8	368	667
4	29*	240*	8	16	6	0**	511

* After conversion to Moore, #states = 61, #edges = 417

** Only one block of logic for a Mealy machine.

Table 6.1 Description of examples

EXAMPLE	Logic Description			Cpu Times (seconds on VAX 8650)				
	#states initial	#invalid states detected	#edges	enum-eration	invalid state detection	edge gener-ation	equiv-alence check	total
1	16	11	2	1.0	1.9	2.4	6.0	11
2	128	94	1165	4.3	12.1	32.1	48.2	97
3	256	126	1372	21.1	71.2	126.2	368.1	587
4	64*	24*	912*	7.6	30.3	46.2	84.1	168

* After conversion to Moore, #states = 74, #edges = 1356.

Table 6.2 Run-Time Statistics Of Examples

been obtained from various industrial and university sources. The examples, **sbc.1** and **sbc.2** are single-cone output circuits from a 28 latch FSM in the Snooping Bus Controller of the Berkeley SPUR chip set [136].

Verification was performed between implementations of the same circuit with different encodings, implying no correspondence between the latches of the two circuits. The largest example, **sbc.2**, has 17 latches and 2764 valid states (total number of states is 131072) yet verification was possible in 5.36 hours on our VAX 11/8650. Note that over 2.6 million edges were enumerated, for this example. However, these edges did not have to be stored, since the paths were dynamically generated and simulated on the second finite automaton. Previous approaches to sequential verification are unable to deal with such large finite automata. Memory usage was restricted to less than 1Mbyte for all these examples. Cube enumeration drastically reduced the number of edges generated – minterm enumeration on each state would have resulted in over 3.7×10^{11} ($= 2764 * 2^{27}$) edges. The efficiency of this approach both in

EXAMPLE	#inp	#out	#gates	#latcheS	#valid states	#edges in STG	CPU time
cse	7	7	192	4	16	167	1.2s
sand	11	9	555	6	32	237	4.9s
planet	7	6	606	6	48	182	4.2s
scf	27	56	959	8	115	393	15.6s
sbc.1	31	13	465	13	2040	4846383	8.94h
sbc.2	27	17	492	17	2764	2662236	5.36h

s denotes CPU-seconds and h denotes CPU-hours on VAX 11/8650

Table 6.3 Verification using Enumeration and Simulation

terms of CPU time and memory usage is amply demonstrated by my results.

6.5 Conclusions

An effective method for the verification of two sequential machines at differing levels of abstraction has been presented. Previous work in this area involved verifying relatively small sequential circuits at the logic level. By exploiting the don't care information present at the register-transfer level (invalid input and output sequences) description of a sequential machine I have successfully compared descriptions of large machines at the RTL and logic levels (Tables 6.1 and 6.2). Don't care information can be exploited in the case of verifying two logic-level descriptions as well.

A memory and CPU time efficient enumeration-simulation strategy for verifying the equivalence of logic-level finite automata with known reset states has also been presented. This approach has been used successfully to verify the equivalence of finite automata with more than 2500 states (Table 6.3).

Future work in this area includes development of a more efficient algorithm for verifying the equivalence of two State Transition Graphs of Mealy machines and more efficient cube enumeration techniques at the logic level to speed up the verification process.

CHAPTER 7

Testing of Logic Circuits

7.1 Introduction

Testing of VLSI circuits is a process to ensure that a chip satisfies its functional specification. For testing a circuit, binary patterns, called test patterns or tests, are applied to the inputs of the circuit and the response of the circuit is compared with the expected one. Application of all possible input patterns, for combinational circuits, will guarantee that the chips passing the test are functionally correct. However, this exhaustive method becomes infeasible, in terms of testing time, when the number of inputs is large. In practice, a set of test patterns that are aimed to detect a high percentage of modeled faults is used. The most widely used fault model has been the *stuck–type* model [38]. Physical failures are assumed to correspond to a line in the gate-level description of the circuit stuck at a 0 or 1 value and an assumption is made that only one fault can occur at a time. It has been empirically shown that a high percentage of the chips passing the set of test patterns for stuck-type faults are correct working chips.

Test generation is a process which produces a set of tests that detect all or a large subset of potential faults in a logic circuit whose existence would cause the circuit to function incorrectly. Test generation for combinational circuits has traditionally been considered to be a search problem [130] [131]. A test pattern for a fault is generated by searching through the input space to find an input pattern that excites the fault and propagates its effect to one of the primary outputs. The cost of test generation can be very high and it has been proved that the problem of test generation is NP-complete [134]. It is especially expensive to generate tests for circuits that contain a large number of redundant faults. Redundant faults are faults for

which no test can be found after searching, implicitly or explicitly, across the entire input space. The cost for trying to generate tests for redundant faults can be more than 90% of the total test generation time. Redundant faults are due to logic redundancies in a circuit. Redundancies may be introduced intentionally, for reliability, performance, elimination of static hazards or other reasons, but often they are due to unoptimized designs. It can therefore be conjectured that designs which have gone through logic minimization, which in general aims to reduce the overall size of a circuit by removing logic redundancies, are more testable and easier to generate tests for.

Generating tests for sequential circuits is considerably harder than for combinational circuits. Even if the combinational part of a sequential circuit is made fully testable, it may still be impossible to obtain a high fault coverage for the sequential circuit. Some of the inputs and outputs of the combinational part are outputs and inputs respectively of the memory elements, i.e. flip-flops. Test patterns generated considering only the combinational part cannot be readily applied and fault effects cannot be observed directly at the inputs of these memory elements.

A popular approach to solving the problem of test generation for sequential circuits is to make all the memory elements controllable and observable, e.g. Complete Scan Design [36] [37]. Scan Design approaches have been successfully used to reduce the complexity of the problem of test generation for sequential circuits by transforming it into one of combinational test generation which is considerably less difficult. The design rules of Scan Design also constrain the sequential circuits to be synchronous so that the normal operation of the sequential circuit is free of critical races. However, there are situations where the cost in terms of area and performance of Complete Scan Design is unaffordable. In addition, even though the general sequential testing problem is very difficult, there may be cases where test generation can be effective. Simply making all the memory elements scannable in a sequential circuit without first investigating how difficult is the problem of generating tests for it could unduly

incur unnecessary area cost.

The difficulty in generating a test usually lies with: (1) setting the states of the memory elements into a certain combination so that the fault under test is excited; (2) propagating the fault effect to the primary outputs. An input sequence is usually required in both cases (if such a sequence exists). In general, the longer the length of the shortest input sequence needed to perform steps (1) and (2), the more difficult it is to find an input sequence to test the circuit. Both approaches mentioned above attempt to shorten the length of the input sequence. In the Scan Design approach, the length of the input sequence is reduced to one when all memory elements are made scannable.

Several approaches [93] [94] [39] [95] [96] [97] have been taken in the past to solve the problem of test generation for sequential circuits. They are either extensions to the classical D-Algorithm [93] [39] [95] [97] or based on random techniques [94] [96]. When the number of states of the circuit is large and the tests demand long input sequences, they can be quite ineffective for test generation. This is because no *a priori* knowledge of the length of the test sequence is available. In the extended D-Algorithm methods, a large amount of effort may be wasted in trying to find short sequence tests for faults that require long ones. Random testing techniques are based on continuous simulations and grading of test vectors according to simulation results. They can be very time consuming for difficult faults that have only a few long test sequences.

An approach to sequential testing has been developed, which represents a significant departure from previous methods. A new test generation algorithm [99], effective for small to medium-sized finite state machines, based on the concept of *state space enumeration* has been developed. The algorithm quickly generates tests for all the faults or a large subset of faults in the given circuit. Then, a *minimal subset of memory elements* is found, which if made observable and controllable, results in easy detection of all remaining irredundant but difficult-to-detect faults. This step represents an *Incomplete Scan Design* approach to the

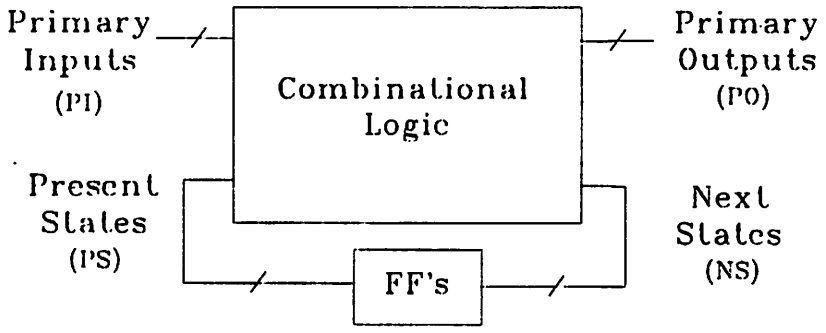
sequential testing problem. The test generation algorithm [99] is again used to generate tests for these faults in the modified circuit (the circuit with the identified memory elements made scannable). This algorithm can guarantee detection of all irredundant faults as in the Complete Scan Design case, but at much less area and performance cost.

In the next section, the problem of test generation for sequential circuits is described. In Section 7.3, representative approaches taken previously to solving this problem are reviewed. The new approach to sequential test generation is described in Section 7.4 and 7.5.

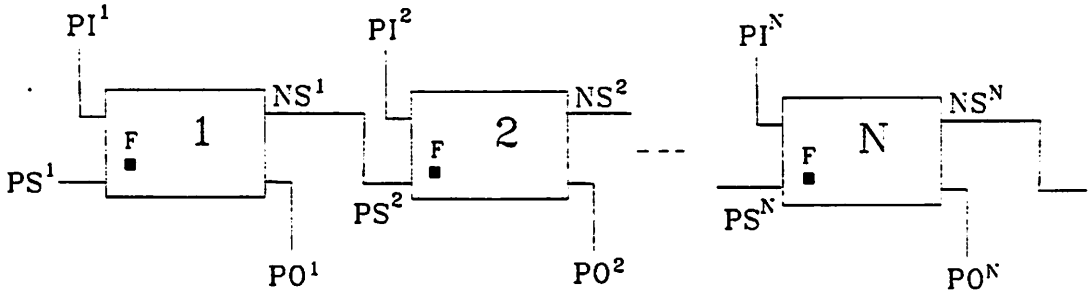
7.2 The Sequential Testing Problem

A general model of a synchronous sequential circuit, S, is shown in Figure 7.1(a). S is assumed to have a reset state R, from which all test sequences begin. S can be modeled as an iterative array C^p, shown in Figure 7.1(b), by cutting the feedback loops at the clocked flip-flops. The combinational circuits C_i, where $i = i,...p$, are all identical to the combinational portion C of the original sequential circuit and all flip-flops are modeled as combinational elements, referred to as pseudo flip-flops. The iterative array is logically equivalent to the sequential circuit – the temporal response of the sequential circuit is mapped into a spatial response of the iterative array. If an input sequence $x(1),x(2),... x(p)$ is applied to S in initial state $y(I)$ generating output sequence $z(1),z(2)... z(p)$, then the iterative array will generate the output z_i from cell i in response to the input x_i to cell i and $z_i = z(i)$ if $x_i = x(i)$ and $y_I = y(I)$. A single fault in S corresponds to the multiple fault f^p consisting of the same fault in every cell C_i of C^p. The goal is to generate input vector sequences for each possible fault in the combinational circuit C, which when applied to the faulty circuit detects the fault, by producing a different set of output vectors than in the fault-free circuit.

For a fault to be detected, first the states of the memory elements have to be set to a certain combination so the fault is *excited*. That is, S first has to be brought into a state from the reset state, R, which can excite the fault. This process is called *state justification.* After state justification, the effect of the fault has to be propagated to the primary outputs. This



(a) General form of a synchronous sequential circuit, *S*



(b) Equivalent combinational iterative array
Fig. 7.1

process is called *fault propagation.* In general, both these steps need a sequence of input vectors.

7.3 Previous Work in Sequential Testing

7.3.1 The Extended D-algorithm for Synchronous Circuits

As described in Section 7.2, a synchronous sequential circuit S can be modeled as an iterative array C^p, shown in Figure 7.1(b), by cutting the feedback loops at the clocked flip-flops. The test generation procedure for a self-initializing test sequence based on the iterative array model is as follows:

(1) Determine the maximum number of time frames, p, allowed for test generation.

(2) Choose an initial value for p and construct the iterative array model with y_I of unknown value. (If a reset state is given, y_I will be assigned the value of the reset state.)

(3) Choose the time frame q from which the D-drive must be organized. Apply the D-algorithm to find a test for the multiple fault f^q so that a D or $\overline{D}$ appears at one of the outputs $z_1, z_2, \ldots z_p$. If a test is found, exit; otherwise, continue.

(4) If possible, increment p by 1 and return to step 3); otherwise, exit with no test.

Since the length of test sequence cannot be determined *a priori*, a large amount of effort may be wasted in trying to generating tests with inappropriate choice of p.

7.3.2 Weighted random test-pattern generation

In a random test-pattern generator, a sequence of random patterns are applied to the circuit. Usually, the patterns differ in a single primary input. In general, all primary inputs (PIs) of the circuit have the same weights, i.e. each PI is exercised approximately the same number of times averaging over a long period of time. In a weighted random test-pattern generator (e.g. [94]), different weights are assigned to the PIs in proportion to their relative importance, i.e. some PIs are exercised more often than others. A single input changes between two consecutive patterns as before.

One way to determine the weight assigned to each PI is to measure the amount of gate switching activity produced inside the circuit as the result of exercising that PI. A set of random patterns is simulated on the circuit. The number of gates changing for the first time from a logic 1 to 0, and vice versa, due to the switching of any of the PIs, is counted. The switching activity count is then accumulated over the complete set of random patterns. By comparing the activity created by all the PIs, different weights can be determined for all the PIs. However, this method suffers from the drawback that the importance of the order of patterns applied to detect a fault is ignored. Furthermore, test sequences consisting of more than one

change between consecutive patterns cannot be generated.

A dynamic adaptive technique has been used to partially alleviate the problem of ignoring the order of test patterns mentioned above in weighted random test-pattern generation. This technique introduces the rates of changes of activity into the function of determining the weight for each PI. Results show that this technique achieves a significant improvement in fault coverage over the static weighted random test-pattern generators. A reduction technique is used to reduce the total number of random patterns generated as random approaches usually create a large number of test patterns. Random pattern techniques offer no guarantees of test coverage/redundancy identification unlike deterministic test pattern generators.

7.3.3 A new approach to sequential test generation

In this section, an approach to solving the problem of test generation for synchronous sequential circuits is described.

This approach to sequential test generation consists of three steps. First, using an efficient deterministic sequential test generation algorithm tests are generated for all or a large subset of the faults in the given circuit. If the fault coverage obtained is less than the target fault coverage, a minimal subset of memory elements are identified (Incomplete Scan Design) which if made observable and controllable result in easy detection of the remaining irredundant but difficult-to-detect faults. Then, the sequential test generation algorithm is applied to the modified circuit (the circuit with the identified memory elements made scannable) to detect the remaining faults. In Section 7.4, the sequential test generation algorithm is described. In Section 7.5, the algorithm used for the identification of the critical memory ele-

ments is described.

7.4 A Deterministic Sequential Test Generation Algorithm

7.4.1 Introduction

In this section, an algorithm for sequential testing [99] based on the PODEM justification algorithm is described.

It is assumed that the sequential circuit under test is synchronous and free of races under simple design rules. It is also assumed that there is a reset state for the synchronous sequential machine and memory elements such as D flip-flops are identified and represented as logical primitives to facilitate loop cutting in transforming the synchronous sequential circuit into an iterative array. First, a part of the State Transition Graph (STG) of the finite state machine is extracted, using purely structural information, i.e. the gate-level description of a sequential circuit. The construction of the partial STG is based on an efficient state-enumeration algorithm that finds paths from the reset state to different valid states (states reachable from the reset state) in the STG. For circuits with relatively few states, a partial STG including all valid states is built. For circuits with a large number of states, only a subset of valid states is included in the partial STG. The partial STG is then used in conjunction with efficient enumeration-based fault excitation-and-propagation and state justification algorithms for generating tests for line stuck-at faults. Tests have been successfully generated for finite state machines with a large number of states using reasonable amounts of CPU time and obtained close to maximum possible fault coverages.

The following section outlines the test generation process. Extraction of the fully or partially connected State Transition Graph from the logic-level finite state machine is described in Section 7.4.2. The enumeration-based fault excitation-and-propagation and state justification algorithms are described in Section 7.4.3 and 7.4.4 respectively. In Section 7.4.5, the detection of a special class of redundant faults is described. Results for a number of finite

state machines are presented in Section 7.4.6.

7.4.2 The Test Generation Process

Assuming the complete State Transition Graph (STG) of a sequential circuit is available, test generation for a fault under test can be done by first finding an input sequence $T1$ and an initial state $S0$ that excite and propagate the effect of the fault to the primary outputs within 4^n time frames, where n is the number of latches in the sequential circuit. A reset state is assumed to exist for the machine. Then, every path from the reset state to any state $S1$ that covers $S0$, a potential setup sequence, in the complete STG is fault simulated. If a path $T0$ (setup sequence) to a state $S1$ that covers $S0$ can be found under fault conditions, a test sequence $T2$ is generated by concatenating the path $T0$ with $T1$. Even though a setup sequence $T0$ may not be found, the fault may still be detected by one of the potential setup sequences through fault simulation. If this is the case, that particular potential setup sequence itself can serve as a test sequence $T2$. If no test sequence can be found, a new input sequence $T1$ and a new initial state $S0$ which is disjoint from all previously generated ones is searched and the procedure is repeated.

The algorithm is complete, i.e. if a fault is testable, a test will be found given sufficient time. The main drawbacks of this method are: (1) the memory storage for the complete STG may be unreasonably large and the generation of the complete STG may demand astronomical CPU time; (2) fault simulation of all potential setup sequences is extremely time consuming. A remedy to (1) is to generate the potential setup sequences on-the-fly using a backward justification algorithm that searches for paths from the reset state to the $S0$s under fault-free conditions. No information of the STG is required/used.

A test generation algorithm following the ideas presented above is as follows.

Algorithm Structure 1

(1) Find an (new) input sequence $T1$ and an (new) initial state $S0$ that will excite and propagate the effect of the fault under test to the primary outputs within 4^n time frames using the state-enumeration-based forward propagation algorithm (described in Section 7.4.3). If no solution exists, exit without a test.

(2) Find a (new) path $T0$ (potential setup sequence) from the reset state to the initial state $S0$ using a backward justification algorithm. If no solution exists, go to (1).

(3) Fault simulate the potential setup sequence $T0$. If it detects the fault, generate the test sequence $T2$ from $T0$ and go to (5). Else if it is a valid setup sequence, go to (4). Else if $T0$ neither detects the fault nor is a setup sequence go to (2).

(4) Concatenate the input sequence $T0$ that represents the path from the reset state to the initial state $S0$ with $T1$ to form $T2$ which is the test sequence for the fault under test.

(5) Exit with a test sequence.

Even though this algorithm is potentially effective, backward justification in general is difficult when the setup sequence is long. In addition, some states may need to be justified more than once. Therefore, an important enhancement is to generate *a partial STG containing as many valid states* (and paths from the reset states to them) as possible provided that the partial STG extraction process (through forward enumeration as described in Section 7.4.2) is carried out efficiently. Note that the partial STG may contain all the valid states in the complete STG but contains much fewer edges. States and edges may be *added* to the partial STG via backward justification during test generation.

The second drawback mentioned above, i.e. that fault simulation of all potential setup sequences is very time consuming, does not actually pose a problem. From experimental observations, if $T0$ is an invalid setup sequence, it is very likely to be a test sequence. Therefore, there is rarely the need for fault simulation of more than one potential setup sequence for a fault.

Finally, an efficient test generation algorithm combining the advantages of forward enumeration and backward justification by using the partial STG is as follows.

Algorithm Structure 2

(1) Find an (new) input sequence $T1$ and an (new) initial state $S0$ that will excite and propagate the effect of the fault under test to the primary outputs within a prescribed number of time frames using the state-enumeration-based forward propagation algorithm (described in Section 7.4.3). If no solution exists, exit without a test.

(2) Search for a path (potential setup sequence) $T0$ from the reset state to $S0$ in the partial STG. If it is found, go to (5).

(3) If the partial STG includes all valid states, go to (1).

(4) Find a path $T0$ from the reset state to the initial state $S0$ using the state justification algorithm (described in Section 7.4.4). If no solution exists, go to (1).

(5) Fault simulate the potential setup sequence $T0$. If it detects the fault, generate the test sequence $T2$ from $T0$ and go to (7). Else if it is a valid setup sequence, continue. Else go to (1).

(6) Concatenate the input sequence $T0$ that represents the path from the reset state to the initial state $S0$ with $T1$ to form $T2$ which is the test sequence for the fault under test.

(7) Exit with a test sequence.

The initial state $S0$ can be a cube containing don't care bits or a minterm with every state bit specified. In the case of a cube, a path from the reset state to a minterm covered by

*S*0 can serve the purpose of a setup sequence.

7.4.3 State Transition Graph Extraction

The algorithm used for State Transition Graph (STG) extraction is the same as the algorithm used in the verification subsystem for dynamic cube-enumeration of the STG of a finite automaton which was described in Chapter 6, Section 6.4.6.2. It is given the combinational logic block CLB of the finite state machine and information about latch inputs and outputs i.e. present and next state lines. If a partial STG is to be extracted, a limit is placed on the number of states at any given level from the reset state. A limit may be placed on the number of levels or the total number of distinct states to be enumerated.

The STG extraction first sequentially cube-enumerates all fanout edges from the given reset state. Whenever a new edge is found, it is added to the current STG if the next state it fans into does not exist in the STG. Each next state is then picked as a new starting state. The procedure is repeated until no more distinct valid states can be found. All the edges in the complete STG will be implicitly, but exhaustively enumerated. The partial STG constructed is a tree, i.e. there is only a single path from the reset state to any other state. This is to restrict the storage space for the partial STG so that synchronous sequential machines with very large number of states can be handled.

7.4.4 The Fault Excitation-and-Propagation Algorithm

The Fault Excitation-and-Propagation algorithm (FEP) is based on the decision tree concept of the test pattern generation algorithm PODEM. FEP uses the conventional iterative array model for generating an input sequence *T*1 and an initial state *S*0 to excite and propagate the effect of the fault under test to the primary outputs within a prescribed number of time frames. The iterative array is considered wholly as a combinational circuit with primary inputs of different time frames time-indexed and the present state lines of the first time frame treated as pseudo inputs. The initial state *S*0 is specified by the values of the pseudo inputs.

FEP first tries to propagate the fault effect to the primary outputs of the first time frame. If it fails, it will use the primary outputs of the second time frame for fault propagation and so on until the prescribed number of time frames is reached.

FEP uses *two decision trees*, one for the primary inputs of different time frames and the other for the initial state $S0$, as opposed to only one in PODEM. The two decision trees are built in a similar way through the backtracing and backtracking processes as used in PODEM. The present state lines of the first time frame are treated similarly as the primary inputs during the fault excitation-and-propagation process. Values of the present state lines and primary inputs of different time frames are continuously set one at a time through the backtracing process and the iterative array is simulated whenever a primary input or a pseudo input is set to a known value. The value-setting-and-simulation process continues until the effect of the fault under test is excited and propagated to the primary outputs of at least one of the time frames or when the backtracking limit is reached. Backtracking takes place whenever it has established that under the current set of primary input and pseudo input assignments, the effect of the fault under test cannot be excited and/or observed at the primary outputs of the specified time frame with further input assignments. Backtracking during the search for $T1$ and $S0$ is done on both decision trees.

FEP employs *the concept of disjoint state enumeration* to make sure that all the tests it generates for a specific fault will have disjoint initial states $S0$'s; this is necessary because of the loop in the test generation process described in Section 7.4.1. Whenever the search for a new test is begun, the primary input decision tree ($D1$) for the previous test is scratched completely, but the present state decision tree ($D2$) of the initial state $S0$ is retained. Immediately, backtracking is done on $D2$. Then the value-setting-and-simulation process is carried out as described above. The reason that tests generated for a specific fault by FEP should all have disjoint $S0$s is related to how FEP is used in the test generation process as described in Section 7.4.1. For a specific fault, a new test is requested only if one cannot find the path from

the reset state to the $S0$ in the previous test, neither in the extracted STG nor through the state justification algorithm described in Section 7.4.5. Therefore, all tests generated for a specific fault should have disjoint $S0$s.

A single decision tree could have been used instead of two separated ones as described above. And instead of completely resetting all primary input values to unknown, i.e. scratching the entire primary input decision tree, when a new search is started, one can simply backtrack on the single decision tree to where a pseudo input is first set to a known value and assign it an opposite value. But due to the inherent characteristics of the enumeration approach of PODEM, it is more efficient to begin a search with as small a number of preset inputs as possible. Therefore the double decision tree method is used.

7.4.5 The State Justification Algorithm

Given a goal state $S0$, the state justification algorithm (SJ) attempts to find a path (setup sequence) from the reset state to it. $S0$ can be a cube containing don't care state bits or a minterm with every state bit specified. In the case of a cube, SJ needs only to find a path to any minterm state that is covered by $S0$.

First, SJ sequentially enumerates all the fanin edges to $S0$. It then checks whether any state the edges fanout from covers the reset state. If such a state exists, a path is found. Otherwise, SJ picks each fanin state as a new goal state and carries out fanin edge enumeration again. The procedure is repeated until a path is found or no path can be found. SJ actually proceeds in a depth-first fashion and there is a limit on the maximum length of the justification sequence.

The edge enumeration algorithm is an extension to the enumeration algorithm PLOVER in [32]. The difference is that here we have multiple line (the next state lines) values to be justified simultaneously rather than a single output line as in PLOVER. The concept of state enumeration is also employed in SJ. There are two decision trees to be maintained as in Section 7.4.3, i.e. one ($D1$) for the primary inputs and the other ($D2$) for the present state lines.

All the present state lines and primary inputs are set to unknown values initially. Through backtracing and backtracking processes, the primary inputs and present state lines are continuously set to some known values, 1 or 0, until all the next state lines are found to be set to their specified values through simulation. Whenever the search for a new fanin edge is begun, $D1$ is completely scratched, but $D2$ is retained. Immediately, backtracking is done on $D2$. Then the enumeration procedure is repeated again. All edges (with disjoint fanin states) fanning out of a state are enumerated when no more backtracking is possible.

7.4.6 Detection of Redundant Faults

The difficulty in test generation for sequential circuits does not just lie with finding tests for the difficult testable faults. The determination of redundant faults is equally formidable. if not more difficult. Obtaining a low fault coverage does not necessary mean the test generator is inadequate if one can show that the fault coverage is close to the maximum achievable value. However, to determine whether faults, that no test has been generated for, are redundant or testable may demand astronomical CPU times. For the purpose of judging how close the fault coverage obtained by the test generator is to the maximum possible value, all the redundant faults based on Theorem 7.1 given below are found and the other undetected faults are treated as possibly testable faults. This gives a pessimistic estimate of the number of redundant faults in a given circuit.

Definition 7.1: An edge in the State Transition Graph is said to be *corrupted* by a stuck-at fault if the effect of the fault can be excited and propagated to the primary outputs and/or next state lines by the input vector corresponding to the edge with the present state lines values set to the fanin state of the edge.

Theorem 7.1: In order for a stuck-at fault to be detected, the fault should at least corrupt one fanout edge from a valid state that is reachable from the reset state in the State Transition Graph.

Proof: In order to detect a fault, we need a test sequence starting from the reset state and ending with a corrupted edge in the STG. If a fault does not corrupt any fanout edge from a valid state in the STG, no test sequence can detect the fault since no corrupted edge can be reached from the reset state. Q.E.D.

Determining this special class of redundant faults requires the extraction of a partial STG containing all valid states reachable from the reset states. The procedure to find these redundant faults is based on the FEP algorithm described in Section 7.4.3. A single time frame is used and all next state lines are treated as primary outputs. All tests are generated for a potential redundant fault, with disjoint initial states. If none of the initial states exists in the partial STG, the fault under test is redundant.

7.4.7 Results

Results for six finite state machines are given in Table 7.1. In Table 7.2, time profiles for each of the examples are given. In the tables *m* and *s* stand for minutes and seconds respectively. For each example in Table 7.1, the number of inputs (#inp), number of outputs (#out), number of gates (#gate), number of latches (#lat), number of equivalent faults (#eqv. faults), the number of test sequences (#test seq.), total number of test vectors (#vect), maximum test sequence length (max. seq. len.), fault coverage, percentage of provably redundant faults (using Theorem 7.1), total fault coverage including detected and provably redundant faults (tfc), and CPU time on the VAX 11/8800 are indicated. CPU times for extracting the partial State Transition Graph, test sequence generation, fault simulation, miscellaneous setup and for the entire test generation process are given in Table 7.2.

As can be seen the test generation technique obtains close to the maximum possible fault coverage in all the examples. The extraction of the STG consumes a relatively small amount of CPU time with respect to the total TPG time in all cases. Fault simulation constitutes a large percentage of total TPG time in most cases except in sse, as can be seen in Table 7.2. The fault simulator used incorporates the parallel-fault event-driven technique and a

CKT	#inp	#out	#gate	#lat	#eqv. faults	#test seq.	#vec	max. seq. len.	fault cov. (%)	red.* fault (%)	tfc§ (%)	CPU† time
cse	7	7	192	4	680	96	472	8	99.71	0.29	100.0	53.2s
sse	7	7	130	6	486	46	284	10	84.57	15.23	99.8	69.9s
planet	7	19	606	6	2028	80	1191	26	97.39	2.56	99.95	12.6m
sand	9	6	555	6	1932	165	1077	24	94.36	5.18	99.54	22.4m
scf	27	54	959	8	3338	136	2238	21	94.37	3.86	98.23	83.0m
sbc	40	56	1011	28	3008	168	1063	24	95.68	2.66	98.34	62.1m

* percentage of provably redundant faults
§ total fault coverage including detected and provably redundant faults
† All times are obtained on a VAX 11/8800

Table 7.1 Results for 6 example circuits

CKT	STG Extraction	Test Generation	Fault Simulation	Miscell.	Total
cse	0.9s	8.3s	43.8s	0.2s	53.2s
sse	0.4s	52.2s	17.1s	0.2s	69.9s
planet	3.2s	1.2m	11.4m	0.7s	12.6m
sand	4.6s	10.7m	11.6m	0.6s	22.4m
scf	13.9s	11.5m	71.2m	1.2s	83.0m
sbc	12.4m	28.3m	21.4m	1.3s	62.1m

Table 7.2 Time profiles for example circuits

more sophisticated one using concurrent techniques will significantly speed up the test generation process. The reason that test generation time is the dominant constituent in the total CPU time in sse is because a great amount of time is consumed in trying to find tests for the large number of redundant faults.

The first five examples are finite state machines obtained from various industrial sources. The largest example SBC is the snooping bus controller [136] in the SPUR chip set. It was synthesized using the multiple level logic optimization system MIS [6].

7.5 An Incomplete Scan Design Approach

7.5.1 Introduction

In the approach to sequential test generation as described in Section 7.4, two different steps of *forward* fault propagation and *backward* state justification are performed. Given the circuit and a fault to be detected from R (the reset state of the machine), First, a state, $S0$, and an input vector sequence, I, which can propagate the effect of the fault to the primary outputs are found. Then, a path from $S0$ to R is found using a backward justification algorithm. In this approach, the backward justification step is in general the bottleneck as regards CPU time for fault detection i.e. $S0$ and I are quickly found, but finding a path from R to $S0$ is much more difficult. The difficulties in backward justification are alleviated by extracting a partial State Transition Graph as described in Section 7.4.2. However, for large circuits (> 100 latches), a significant number of states cannot be justified, resulting in the corresponding fault remaining undetected. The Incomplete Scan Design algorithm identifies a set of memory elements, which when made controllable, modify the State Transition Graph of the sequential circuit so as to result in easy justification of previously unjustifiable states.

The overall structure of the Incomplete Scan Design algorithm is described in the next section. The heuristic selection process is described in Section 7.5.3. Results obtained using

this algorithm are presented in Section 7.5.4.

7.5.2 Overall Structure of the Algorithm

The algorithm incorporates the sequential test generation algorithm, STALLION, described in previous sections. It is given the sequential circuit S and a set of faults to be detected, F. It produces a set of test sequences, T, each beginning from the reset state of the machine, R, and identifies a set of memory elements, M, to be made scannable. The T are such that they detect the faults F in S^M, the sequential circuit, S, with the memory elements, M, made observable and controllable.

Incomplete Scan Design Algorithm

(1) Given the sequential circuit S, for each fault $f \in F$, attempt to find a fault-excitation-and-propagation sequence, P_f, which will propagate the effect of f to the primary outputs if possible else to the next state lines. The length of P_f is limited to *MAX_PROP_LEN*. If P_f does not propagate f to the primary outputs, a set of next state lines, NS_f, to which the effect of f can be propagated to, is found. The set of faults which can be propagated only to next state lines is called F^{NS}.

(2) All distinct state vectors in P_f^0, the first vector in the sequence P^f are found. Call this set of distinct state vectors, K.

(3) Generate *MAX_STATE* states, Q_i, at different levels, i, from R, in the State Transition Graph (STG) of S. i varies from 1 to *MAX_LEVEL*. The generation of these states uses the extraction algorithm described in Section 7.4.2.

(4) For each $k \in K$, find the memory lines to made scannable such that each state $q \in Q_i$ covers k. For each k generate *MAX_CHOICE* best (with the least number of lines) choices for line sets, which if made scannable will result in the covering of k by $q \in Q_i$.

(5) Given K and *MAX_CHOICE* choices of line sets for each $k \in K$, and a line set NS_f for each $f \in F^{NS}$, select a line set for each k and a single line from each NS_f line set so the number of distinct lines to be made scannable, M, is minimized.

In Step 1, propagation sequences are found for faults in the sequential circuit, S, using the sequential test generation algorithm, STALLION. While running STALLION, the present state lines of S are set last so the state vectors K, found in Step 2, have as many don't care bits in them as possible. STALLION produces a starting state, $S0$, and an input vector sequence, I, which propagate the effect of the fault to either the primary outputs or the next state lines of S. If the effect of the fault is propagated to the primary outputs, then it only remains to justify $S0$, else the complete set of next state lines to which the effect of the fault can be propagated, NS_f, has to be found as well.

In Step 3, a large set of states in the State Transition Graph of S, and paths from the reset state leading to each state are found. These states are extracted using the forward state enumeration algorithm, based on the PODEM enumeration algorithm (Section 7.4.2). The number of states and the lengths of the justification paths, are bounded by *MAX_STATES* and *MAX_LEVEL*, respectively.

The **distance** between two arbitrary bit vectors of length N, $A(i)$ and $B(i)$, where each bit can take the values of 0, 1 and 2 (don't care) is defined as the number of bits where $A(i)$ is 1 and $B(i)$ is 0 or *vice versa* over $i = 1, .. N$. Given a state $q \in Q_i$ and a state $k \in K$, the distance between q and k then gives the number of state line values that q and k differ in. It is easy to see that if the state lines which are different between q and k are made controllable, any justification sequence for q has to work for k. Don't care bits in k do not contribute to distance since they can take the values of 0 or 1. So to minimize distance between the states in K and Q_i, the number of don't care bits in K is maximized by setting the state lines last in Step 1.

In Step 4, for each pair of q and k, the set of lines which are to be made scannable for the justification sequence of q to be usable for k is found. Then, for each k, *MAX_CHOICE* such line sets with the least number of lines and secondarily the smallest justification sequence length are selected.

Given these *MAX_CHOICE* sets of lines for each k, a heuristic algorithm (Step 5) is used to select one particular line set, (corresponding to one particular $q \in Q_i$) for each k, so as to minimize the total number of distinct lines in all the line sets selected. Simultaneously, for each fault, f, which could not be propagated to the primary outputs, a line from NS_f (the set of next state lines to which f can be propagated) is selected. Two selection algorithms which have been employed are described in the next section.

After the heuristic selection process, a minimal number of state lines to be made scannable and justification sequences for K are identified. A set of test sequences, T, is formed by concatenating the justification sequences, J, with the propagation sequences, P, generated using STALLION. T will detect all undetected faults in S^M, namely, F.

7.5.3 The Heuristic Selection Process

The subproblem to be solved is as follows. We have N elements (each corresponding to a fault), each with a set of line groups. The number of line groups may vary. It is assumed that for any given element, none of the line groups is a superset or subset of any other line group of the same element. The goal is to identify a line group for each element such that the number of distinct lines in the selected line groups is minimum. Some of the elements may have a set of line groups each containing a single line. For example, during test generation, faults that can only be propagated to next state lines will result in an element with the property mentioned above.

Two heuristic algorithms which produce minimal solutions are described below.

Algorithm 1

```
for ( i = 1;  i ≤ N;  i++ ) {
   Element  =  e[i] ;
   for ( j = 1;  j ≤ Element.NumChoices;  j++ ) {
      Solution  =  greedy( Element.Choice[j], Element ) ;
   }
}
select best Solution ;


greedy( lineGroup, element ) {

  Lines  =  Lines ∪ lineGroup ;
  for ( i = 1;  i ≤ N;  i++ ) {
     if ( e[i]  ≠  element ) {
        for ( j = 1;  j ≤ e[i].NumChoices;  j++ ) {
           card  =  | Lines ∪ e[i].Choice[j] | ;
        }
        pick k so card is minimum ;
        Lines  =  Lines ∪ e[i].Choices[k] ;
     }
  }
  return( Lines ) ;
}
```

Algorithm 1 is a very fast greedy algorithm run with different starting points.

Algorithm 2

```
find all required lines, Lines ;

while ( choices to be made ) {

   pick N_e elements, e_1, .. e_Ne and choices
     for the elements, c_1, .. c_Ne minimizing
     | Lines ∪ e_i[c_i] |    i = 1, N_e ;

   mark choices made for e_i ;
}
```

Algorithm 2 first finds all lines which are definitely required, if any. For example, if for any element, a line exists in all its choices, that line is definitely required. Then N_e best elements and line groups in these elements are picked at a time. The complexity of this algorithm is $O(N^{N_e})$. The larger the N_e is, the better the solution can potentially be ($N_e = N$ is exhaustive search) but more CPU time is required. I have found that $N_e = 3$ gives near-optimal results within acceptable run times.

7.5.4 Results

Results obtained on several circuits are summarized in Table 7.3. In the table, the number of inputs (#inp), outputs (#out), gates (#gate), and latches (#lat) in each circuit is indicated. The percentage of combinationally redundant faults (which cannot be detected even using Complete Scan Design), initial fault coverage achieved by the sequential test generation algorithm, the number of latches made scannable, the final fault coverage in the modified circuit and the CPU times used for selection on a VAX 11/8650 are also given. The CPU times for sequential test generation varied between a minute for the smaller examples to an hour for the largest example **sbc** on a VAX 11/8650. It should be noted that the CPU time used for sequential test generation can be bounded by limiting the number of backtracks allowed. However, if this is done, quite possibly fewer faults will be detected and more scan latches may be required.

As can be seen, making a small subset of latches scannable increases the fault coverage obtained to the maximum possible value or very close to the maximum possible value for all the circuits. For instance in example **sckt4**, making 9 out of 21 latches scannable raised the fault coverage from 26.25% to the maximum possible value of 98.91% (1.09% of the faults are combinationally redundant). Depending on the fault coverage required, differing numbers of scan latches suffice. Trade-offs can be made for each example as indicated in Table 7.3.

The results demonstrate the advantages in using a combined approach of sequential test generation and Incomplete Scan Design. A large percentage of faults are detected using the

EXAMPLE	#inp	#out	#gate	#lat	red. fault %	initial fault cov.	scan latches	final fault cov.	CPU time (secs)
sse	7	7	130	6	0.0	84.57	3	100.0	4.9
sand	9	6	555	6	0.21	94.31	2	99.79	5.3
scf	27	54	959	8	0.51	94.67	2	99.49	1.8
donfile	2	1	232	12	0.0	63.60	5	96.34	11.1
							9	100.0	14.3
sckt4	3	6	160	21	1.09	26.55	6	92.93	2.1
							9	98.91	2.4
sbc2	35	51	1011	33	2.83	81.25	5	95.68	9.1
							10	97.17	17.1
lex1	27	52	395	97	0.0	75.86	39	97.94	6.3

Table 7.3 Incomplete Scan Design Results

efficient sequential test generation algorithm and the remaining irredundant faults are detected by the same algorithm after making a minimal subset of flip-flops observable and controllable.

7.6 Conclusions

In this chapter, a new approach to solving the test generation problem for sequential circuits was presented. An efficient deterministic test generation algorithm based on the concept of state space enumeration has been developed. This algorithm is effective for mid-sized sequential circuits (< 50 latches). For larger sequential circuits (> 50 latches), an effective approach to Incomplete Scan Design has been developed. After using the test generation algorithm on the given sequential circuit, a minimal number of memory elements are identified, which when made scannable, results in easy detection of the undetected faults. Excellent results have been obtained on large sequential circuits.

CHAPTER 8

Relationships between Logic Synthesis and Testing

8.1 Introduction

An intimate relationship exists between combinational logic optimization and synthesis algorithms and the testability of the synthesized circuit. Test generation algorithms can be, in general, modified for logic minimization to identify redundancies in a combinational circuit. However, removing all the redundancies of a circuit does not always produce an optimal solution. For example, a two-level network without any redundancies may have an equivalent multi-level representation that is smaller in some sense. Logic minimization can, in turn, guarantee that the optimized network is 100% testable. In [81], a synthesis procedure which guaranteed fully testable irredundant combinational logic circuits was proposed. Equally intimate relationships between the more complicated problems of sequential circuit synthesis and test generation have been envisioned.

Generating tests for sequential circuits is considerably harder than for combinational circuits. Even if the combinational part of a sequential circuit is made fully testable via logic minimization, it may still be impossible to obtain a high fault coverage for the sequential circuit. Some of the inputs and outputs of the combinational part are outputs and inputs respectively of the memory elements, i.e. flip-flops. Test patterns generated considering only the combinational part cannot be readily applied and fault effects cannot be observed directly at the inputs of these memory elements. Thus, an improperly designed sequential circuit may have large numbers of redundant and/or difficult-to-detect faults, in the sequential sense, even

though the faults are all detectable in the combinational sense.

One of the advantages of Scan Design is that all combinationally irredundant faults can be easily detected since direct access is provided to the flip-flop inputs and outputs. However, they may be cases where the area/performance penalty of Scan Design rules is unaffordable. Sequential test generation algorithms are very attractive. An efficient sequential test generation algorithm was described in the previous chapter. Now, we take a synthesis approach to solving the test generation problem.

In this chapter, we propose a synthesis and optimization procedure which beginning from a State Transition Graph description of a Moore or Mealy finite automaton produces a 100% testable logic-level implementation of the machine. This implementation is both fully as well as easily testable. The test sequences for all single stuck-at faults in the machine can be derived using test generation algorithms on the combinational logic blocks of the machine.

I show that a strong relationship exists between state assignment, logic optimization and testability of a sequential machine. A procedure of constrained state assignment and combinational logic optimization which ensures 100% testability for both Moore and Mealy finite state machines is outlined. Results obtained on benchmark examples show that the area penalties incurred due to the constraints imposed during state coding and logic optimization are small. The performance of the resulting circuit is usually *better* than a unconstrained design (This is because one of the constraints imposed requires combinational logic partitioning in the machine).

The relationship between combinational logic optimization and test generation is reviewed in Section 8.2. Basic definitions and terminologies used in sequential circuit synthesis and test generation are given in Section 8.3. In Section 8.4, we state the necessary conditions required for a fully and easily testable Moore machine. Extensions to Mealy machines are made in Section 8.5. In Section 8.6, we discuss how the state assignment algorithms described in Chapter 4 can be modified to produce a constrained encoding satisfying the testa-

bility criterion. Results obtained are presented in Section 8.7.

8.2 Relationship between combinational testing and logic minimization

A **Boolean network** is a multi-level structure for representing an incompletely specified logic function. A Boolean network, η, is a pair (F, PO), where $F = \{F_j, j=1,2,..m\}$ is a set of m given representations of the ON-sets f_j of incompletely specified functions $(X_j^{ON}, X_j^{DC}, X_j^{OFF})$. With each F_j is associated a "local output" logic variable y_j in the set $IV = \{y_1,..y_m\}$. The specified primary output set is $PO \subset IV$.

Given a Boolean network, η, a cube c of the two-level representation of F_j is **prime** if no literal of c can be removed without causing the resulting network η' to be *not* equivalent to η. Similarly, a cube c of F_j is **irredundant** if c cannot be removed from F_j without causing the resulting network η' to be *not* equivalent to η. A Boolean network η is said to be **prime** if all the cubes in each of the representations F_j of η are prime, and **irredundant** if all of these cubes are irredundant.

These two concepts are associated with local minima of a cost function which is nondecreasing in the total number of cubes and literals required to represent the incompletely specified logic functions, realized by the given Boolean network.

An **internal** stuck fault is a fault in which v_k (or $\overline{v_k}$) of cube c of representation F_j or a node n_j of a Boolean network is stuck at either its existing value v_k (or $\overline{v_k}$) or its opposite value $\overline{v_k}$ (or v_k). If each two-level function F_j is physically implemented with an AND-OR complex gate, each internal fault would correspond to an input stuck-at fault in the gate-level representation of the Boolean network.

Theorem 8.1: A Boolean Network is prime and irredundant if and only if it is 100% testable for internal stuck faults.

This theorem relates directly the testability of the Boolean network with concepts of logic minimization. The proof of the theorem for a Boolean network can be found in [81]. Similarly for a gate-level implementation of a Boolean network, the corresponding concept of **primality** and **irredundancy** can be defined for primitive gates AND, OR, NAND, NOR, NOT. An input stuck-at fault of a primitive gate is equivalent to an internal stuck fault of a node of a Boolean network corresponding to that primitive gate.

A primitive gate is **prime** if none of its inputs can be removed without causing the resulting circuit to be functionally different. A gate is **irredundant** if its removal causes the resulting circuit to be functionally different. A gate-level circuit is said to be **prime** if all the gates are prime, and **irredundant** if all the gates are irredundant.

Corollary 8.1: A gate-level circuit is prime and irredundant if and only if it is 100% testable for all single stuck-at faults.

8.3 Preliminaries

A **variable** is a symbol representing a single coordinate of the Boolean space (e.g. a). A **literal** is a variable or its negation (e.g. a or $\bar{a}$). A **cube** is a set C of literals such that $x \in C$ implies $\bar{x} \notin C$ (e.g., $\{a,b,\bar{c}\}$ is a cube, and $\{a,\bar{a}\}$ is not a cube). A cube represents the conjunction of its literals. The trivial cubes, written 0 and 1, represent the Boolean functions 0 and 1 respectively. An **expression** is a set f of cubes. For example, $\{\{a\},\{b,\bar{c}\}\}$ is an expression consisting of the two cubes $\{a\}$ and $\{b,\bar{c}\}$. An expression represents the disjunction of its cubes.

A cube may also be written as a bit vector on a set of variables with each bit position representing a distinct variable. The values taken by each bit can be 1, 0 or 2 (don't care), signifying the true form, negated form and non-existence respectively of the variable corresponding to that position. A **minterm** is a cube with only 0 and 1 entries.

The **distance** between two minterms is defined to be the number of bit positions they differ in.

A finite state machine is represented by its **State Transition Graph** (STG), $G(V,E,W(E))$ as defined in Chapter 4, Section 4.2.

Given n inputs to a machine, 2^n edges with minterm input labels fan out from each state. A STG where the next state and output labels for every possible transition from every state is defined to correspond to a **completely specified machine.** An **incompletely specified machine** is one where at least one transition edge from some state is not specified.

A starting or initial state is assumed to exist for a machine, also called the **reset state.** A **R-reachable** finite state machine has a STG such that for every possible state, q, in the STG an input sequence exists which when applied to the machine, initially at the reset state, places the machine in q. Thus every state is **reachable** from the reset state.

The fault model assumed is **single stuck-at.** A finite state machine is assumed to be implemented by combinational logic and feedback registers. Tests are generated for stuck-at faults in the combinational logic part.

A combinational logic network is said to be **irredundant** if all the faults in the network are testable.

To detect a fault in a sequential machine, the machine has to be placed in a state which can then excite and propagate the effect of the fault to the primary outputs. The first step of reaching the state in question is called **state justification.** The second step is called **fault excitation-and-propagation.**

An edge in a State Transition Graph of a machine is said to be **corrupted** by a fault if either the fanout state or output label of this edge is changed because of the existence of the fault. A path in a State Transition Graph is said to be corrupted if at least one edge in the

path has been corrupted.

8.4 Fully and Easily Testable Moore Machines

A general model for a Moore finite state machine was shown in Figure 6.3. It is realized by two logic blocks, the Output Logic (*OL*) block and the Next State Logic (*NSL*) block, and feedback registers. In a Moore machine, the outputs depend only on the present state of the machine.

Given n latches in the machine, the machine has 2^n possible states. However, the number of states in a State Transition Graph (STG) description of a machine need not necessarily be an integer power of 2.

I first prove the following result:

Theorem 8.2: Given a n-latch logic-level implementation of a Moore machine (shown in Figure 6.3), if (1) the combinational logic blocks *OL* and *NSL* are irredundant (2) the machine is R-reachable i.e. all 2^n states are reachable from the reset state and (3) all the 2^n states have distinct outputs, the machine is fully testable for all stuck-at faults in *OL* and *NSL*.

Proof: Consider a fault, F, in the *OL* block. Since the block is irredundant (Condition 1), a state, s, exists which detects F. This state, s, can be reached from the reset state, R, of the machine via an input sequence, I, because the machine is R-reachable (Condition 2). State s will be reached on applying I from R regardless of F since F is in the *OL* block. Therefore, a sequence exists, namely I, which can detect F.

Now consider a fault, F in the *NSL* block. Again, since *NSL* is irredundant, a state, s, and an input i exist which propagate the effect of this fault to the next state lines. Instead of obtaining the true next state, q, we obtain a faulty next state q^F. q and q^F have distinct outputs (Condition 3). Therefore, at the next clock cycle the effect of F is propagated to the primary outputs. We however, have to reach s from R. A path exists from s to R (Condition 2). However, this path may or may not have been corrupted by F. If the path has not been

corrupted, we can detect F after reaching s and applying input i. If the path has been corrupted, it means that for some edge in the path, the next state reached was different due to F. In this case, the fault is detected even *before* reaching s, since two different states were reached in the faulty and fault-free machine. Q.E.D.

The implications of each of the conditions of Theorem 8.2 are now analyzed. (1) is an essential condition. Obviously, a redundant fault in *NSL* or *OL* cannot be detected in the sequential machine. Redundancies are sometimes introduced for performance reasons, but mostly they are due to unoptimized logic [81]. An irredundant logic network would have minimum area. With recent advances in multi-level logic optimization, large networks can be made irredundant. If redundancies are required in the combinational logic for performance reasons, the proposed procedure will still guarantee testability and produce tests for all combinationally irredundant faults.

In general, State Transition Graph specifications of machines have reset states and are R-reachable. However, as mentioned previously, a STG specification of a machine need not necessarily have $N_s = 2^k$ states, $k = 1, 2...$ Given the number of encoding bits to be used, n ($n \geq \lceil \log(N_s) \rceil$), the number of states in a STG can be raised to 2^n. We have to ensure that these new states are reachable from the reset state to satisfy the R-reachability condition. Given a single unspecified transition edge (minterm or cube) from a single state in the original STG, edges can be added to the STG so as to ensure that all the added states are reachable (If the machine is completely specified, an extra input has to be added). Most STGs encountered in practical design are *very* incompletely specified.

Condition 3 is obviously unacceptable, since if the STG specification does not satisfy it, it cannot be made to do so without changing the functionality of the machine. This condition is now relaxed.

Consider the logic-level implementation of the Moore machine shown in Figure 8.1. The *NSL* block has been realized as n distinct single-output circuits or *partitions*. The following theorem shows that a *constrained state assignment* can ensure a fully testable circuit.

Theorem 8.3: Given a n-latch logic-level implementation of a Moore machine (shown in Figure 8.1), if (1) the combinational logic blocks *OL* and NSL_i, $i = 1,2.. n$, are irredundant (2) the machine is R-reachable and (3) if the state encoding of the machine is such that each pair of states asserting the same output has codes of distance-2 from each other, the machine is fully testable.

Proof: The faults in the *OL* block are detected as before in Theorem 8.2. Consider a fault F in the *NSL* block. Without loss of generality, assume that F is in the first partition. The effect of the fault when detected is to produce a 0 (1) instead of a 1 (0) at the NSL_1. In either case, the faulty next state produced, q^F, will differ from the true state, q, in at most one bit. Since state assignment has guaranteed that all states asserting the same outputs have been assigned distance-2 codes, q and q^F assert different outputs. This means that F is detected in the next clock cycle. Q.E.D.

A realization of a machine like the one shown in Figure 8.1 implies that logic cannot be shared between next state lines. Thus, a certain area penalty may be associated with such an implementation. The performance of the circuit does *not* suffer due to logic partitioning (and in fact may improve). However, the implementation shown is an extreme case and can be generalized. A partition may contain more than one NSL_i. This means that the logic between these lines *can* be shared.

The number of *NSL* partitions required relates to the number of states asserting the same output in the original STG. It is first shown that the state assignment constraint (Condition 3 of Theorem 8.3) can be satisfied quite easily.

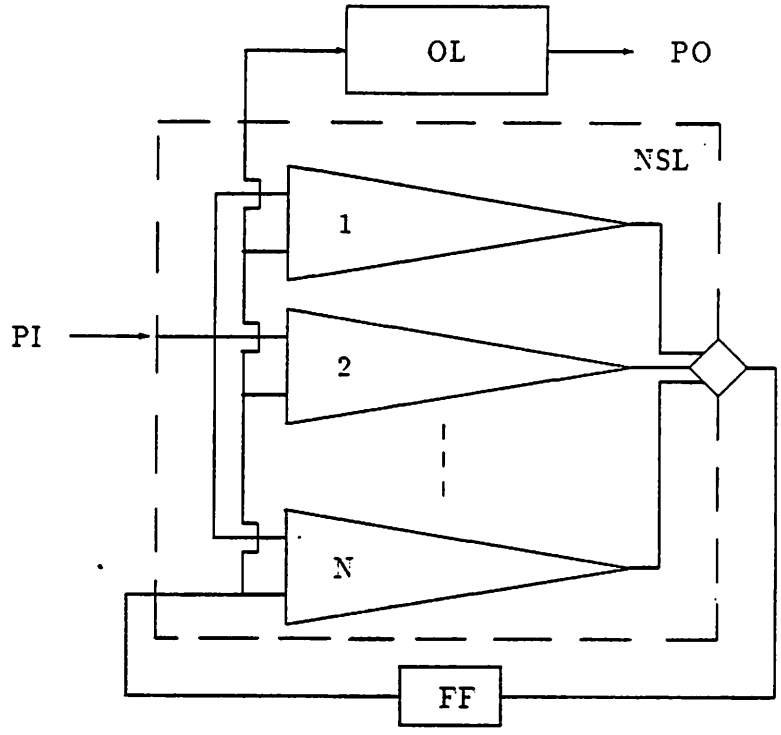


Fig. 8.1 Partitioned Moore Machine Model

Lemma 8.1: Given a State Transition Graph, if at most half the number of states assert the same output, a state assignment satisfying a distance-2 constraint between states with the same outputs can be found.

Proof: Given k bits, we have 2^k possible codes. These codes can be split into 2 sets each of cardinality 2^{k-1}, such that codes within each set are of distance-2. Given a STG with N_s states, we add states to raise the number of states to $2^{\lceil log(N_s) \rceil}$. The number of states in a distance-2 set is $2^{\lceil log(N_s) \rceil - 1} \geq \frac{N_s}{2}$.

The following result, which gives the required number of partitions of the *NSL* lines as a function of the number of states with the same output, is proved.

Theorem 8.4: If at most k states exist in a State Transition Graph which produce the same outputs, $\lceil log(k) \rceil + 1$ separate partitions suffice to obtain a fully testable machine.

Proof: In the worst possible case, if we have 2^n states in the machine, we have a situation where $\left\lfloor \frac{2^n}{k} \right\rfloor$ sets of states exist, the states within each set asserting the same output.

We need to ensure for each set that no two of these k states are ever produced as a fault-free faulty pair due to a fault in *NSL*. This means that the codes assigned to any two of these states must differ in at least two next state lines belonging to two distinct partitions. By Lemma 3.1, the number of bits required to generate 2 sets of 2^{p-1} distance-2 codes is p. To generate 2 sets of k distance-2 codes, we require $\lceil log(k) \rceil + 1$ partitions. We now have $n - (\lceil log(k) \rceil + 1)$ bits remaining. This means we can have

$$2^{n - \lceil log(k) \rceil - 1} \times 2 = \frac{2^{n-1}}{2^{\lfloor log(k) \rfloor}} \times 2 = \left\lfloor \frac{2^n}{k} \right\rfloor$$

sets each with k codes which differ in two next state lines belonging to two distinct partitions. Q.E.D.

There are thus three steps in producing combinational logic specifications for *OL* and *NSL* blocks from a State Transition Graph description. These steps are (1) raising the number of states in the State Transition Graph to 2^n, where n is the number of latches (2) obtaining constraints for the state assignment on the basis of state outputs and (3) state assignment obeying the constraint relations generated. A straightforward solution exists for Steps 1 and 2, however the optimality of the eventual implementation depend on the choices made during these steps. For example, in Step 1, transition edges connecting original states in the STG to the new states can be added in a variety of ways. The new states can be connected in a chain or separately connected from the original states. Similarly, if the number of required partitions is less than the number of next state lines, choices exist as to which next state lines to group together. Next state lines which can share logic maximally should be placed in the

same partition. In Step 3, an optimal state assignment which minimizes combinational logic while meeting the distance constraints has to be found. This step is further discussed in Section 8.6.

After obtaining the combinational logic specifications, logic optimization algorithms which can ensure an irredundant logic network (e.g. [81]) can be applied. If redundancies are required in the logic, this synthesis procedure ensures that all combinationally irredundant faults are sequentially irredundant as well.

To generate tests for the sequential machine, test vectors are generated for all faults in the *OL* and *NSL* blocks. Then, justification paths are obtained from the STG using simple breadth-first search. It is guaranteed (by the theorems proved in this section) that these paths concatenated with the test vectors applied to the primary inputs of the sequential machine will detect all possible faults in the machine so as to be observable at the primary outputs.

This procedure has ensured that a faulty state is always propagated to the primary outputs in a single clock cycle via state assignment. This can, in fact, be generalized to multiple-vector propagation. That is, state assignment constraints can be derived which ensure that a faulty state is propagated to the primary outputs in at most N clock cycles ($N \geq 1$). A state assignment algorithm can construct an optimal encoding which satisfies these constraints. For larger N, the constraints are less stringent but more difficult to state succinctly.

A re-statement of Condition 3 in Theorem 3.2 to ensure full testability via N-vector propagation sequences can be made. The re-statement for $N = 2$ is given below.

The state encoding of the machine should be such that each pair of states asserting the same output should have codes at least distance-2 apart or for each pair of states, q_1 and q_2, which assert the same outputs and have uni-distant codes, the following should hold. Assume that q_1 and q_2 differ in bit i. An input combination should exist which drives the fault-free machine from q_1 and q_2 to states s_1 and s_2 respectively, such that

(1) s_1 and s_2 assert different outputs and

(2) s_2', which is the state that differs from s_2 in bit i alone, asserts a different output from s_1.

8.5 Fully and Easily Testable Mealy Machines

A general model for a Mealy finite state machine is shown in Figure 8.2. It is realized by a single logic block and feedback registers. The output logic and the next state logic are both realized by one block. In a Mealy machine, the outputs depend on both the present state as well as the primary inputs. A model for a Mealy machine with each next state line realized as a separate circuit and with the output and next state logic separated is shown in Figure 8.3.

A theorem in direct correspondence to Theorem 8.3 for Mealy machines can be proven. First, I define the notion of O-equivalence between two states in a Mealy machine.

Definition 8.1: Two states in a Mealy machine are said to be **O-equivalent** if each pair of fanout edges on the same input from these states produces the same output.

Theorem 8.5: Given a n-latch logic-level implementation of a Mealy machine (shown in

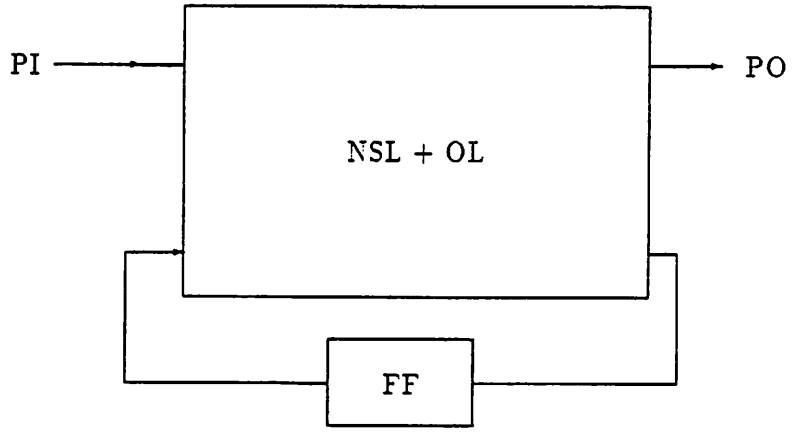


Fig. 8.2 General Mealy Machine Model

Figure 8.3), if (1) the combinational logic blocks *OL* and NSL_i, $i = 1,2.. n$, are irredundant (2) the machine is R-reachable i.e. all 2^n states are reachable from the reset state and (3) if the codes of states of the machine are such that each pair of O-equivalent states have codes of distance-2 from each other, the machine is fully testable.

Proof: Consider a fault in the *OL* block. There exists a state, s and input i which detects this fault by Condition 1. R-reachability and the fact that F is in the *OL* block imply that state s can be reached from R. F can thus be detected.

Consider a fault F in the *NSL* block. Without loss of generality, assume that F is in the first partition. Since this partition is irredundant, a state s and an input i_1 exist which can propagate the effect of the fault to the next state line. The effect of the fault when detected is to produce a 0 (1) instead of a 1 (0) at the NSL_1. In either case, the faulty next state produced, q^F, will differ from the true state, q, in at most one bit. Condition 3 guarantees that q and q^F are *not* O-equivalent since all O-equivalent states have distance-2 codes. This means that an input, i_2, exists which will produce a different output in the faulty machine (which is in q^F) from the fault-free machine (which is in q). We, however, have to reach s from R. A path exists from s to R (Condition 2). However, this path may or may not have been corrupted by F. If the path has not been corrupted, we can detect F after reaching s and applying input i_1 followed by i_2. If the path has been corrupted, it means that for some edge in the path, the next state reached was different due to F. We have a fault-free/faulty pair (q' , q'^F) By the argument above, an input i_3 which produces a different output for q' and q'^F exists, thus detecting F. Q.E.D.

The synthesis procedure for obtaining a fully testable Mealy machine is the same as the procedure outlined for the Moore machine in Section 8.4. To generate tests for the machine, as before, all the combinational logic tests for the *OL* and *NSL* blocks are generated. The justification path to the state detecting the fault concatenated with the primary input part of

the combinational test vector and the differentiating input vector (for the fault-free/faulty next state pair) constitutes the test sequence for a given fault.

8.6 Constrained State Encoding

State assignment is the process of assigning binary codes to the internal states of a finite automaton. The problem of optimal state assignment is to find an encoding of states which minimizes the combinational logic part of the sequential machine.

The combinational logic part of the sequential machine can be implemented using a Programmable Logic Array (PLA) or using multi-level logic. State assignment techniques targeting both these implementations have been proposed (e.g. [26] [74]). The program MUSTANG [74], described in Chapter 4, produces a state assignment that heuristically minimizes

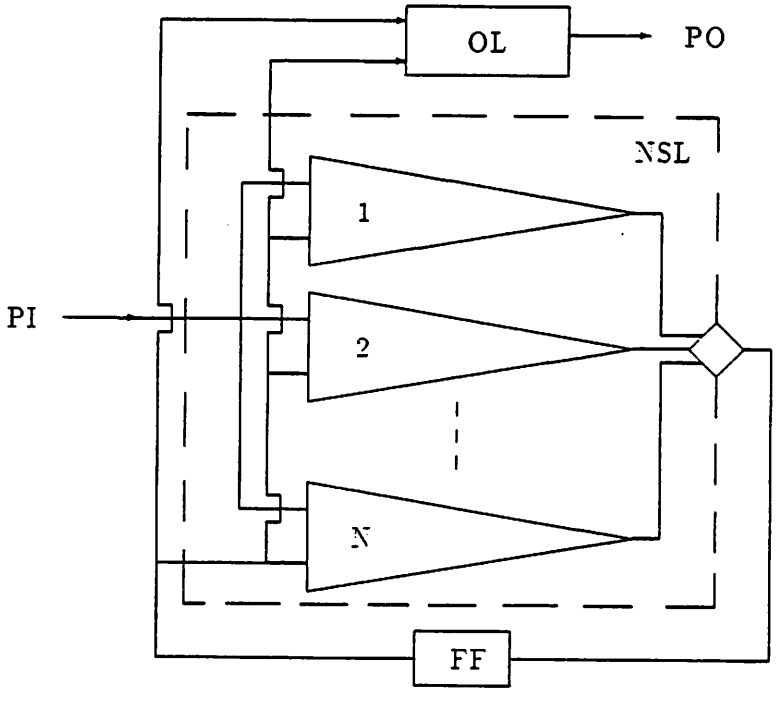


Fig. 8.3 Partitioned Mealy Machine Model

the number of literals in the combinational logic *after* multiple-level logic optimization.

The technique used by MUSTANG is based on maximizing common factors in the logic in an effort to reduce the area of the network. A weighted graph whose nodes represent each state of the machine is constructed. The weights between the edges in the graph reflect the "gains" in coding the corresponding states with uni-distant codes.

An embedding algorithm (Section 4.4.6) is used to actually assign binary codes to the states (nodes in the graph) so as to maximize the overall gain. The algorithm iteratively selects groups of states to be encoded. These states are given minimally-distant codes from the unassigned codes.

For the problem of satisfying distance constraints, the graph construction part remains the same. During embedding, when a group of states is selected, they are checked for distance-2 constraints. A minimally-distant set of codes satisfying these constraints is assigned to the states.

8.7 Results

Results obtained on five State Transition Graphs from the MCNC 1987 Logic Synthesis Workshop benchmark set are given in Table 8.1. First, the machines were encoded and optimized disregarding testability. The number of gates in the machine, the fault coverage obtained and the test generation time are given in Table 8.1 under the column labeled OPTIMIZE. Test generation was accomplished using an efficient test generation algorithm described in Chapter 7. Then, each of the machines were synthesized using the procedure described in Sections 8.4 and 8.5. Again, the number of gates, fault coverage obtained and the test generation time are given. Sequential test generation for these circuits was faster because combinational test generation and breadth-first search suffice to produce the test sequences. The example scf is a Moore machine, the others Mealy machines.

The area penalties incurred are due to three reasons : (1) the constraints imposed during state assignment (2) the addition of extra edges to the STG to obtain R-reachability and (3)

logic partitioning constraints. Empirical evidence has shown that (3) is easily the most significant factor – the next state lines may have to be realized as separate circuits. Additionally, for a Mealy machine, unlike in a unconstrained design, the next state and the output logic have to be separated.

Logic partitioning is extensively used to gain higher performance. A Mealy machine with separate next state and output logic blocks can be clocked faster than a machine with a single lumped block of logic. This is the case in the example designs of Table 8.1 as well. Thus, the fully and easily testable machines produced by logic partitioning may well represent a more desirable point in the area/performance tradeoff curve.

The number of gates in a circuit is, in general, indicative of the area required to implement the circuit. However, in some cases, this measure of area may not be very accurate. To obtain accurate estimates of circuit areas, the synthesized examples of Table 8.1 were placed and routed using the TimberWolf standard cell placement and routing package [61]. The areas of the resulting designs after place and route for the unconstrained and constrained cases are given in Table 8.2. For each example, the areas of the designs have been normalized to that of the unconstrained design.

EX	#inp	#out	#states	#lat	I - OPTIMIZE			II - TESTABLE		
					#gates	fault cov.	tpg time	#gates	fault cov.	tpg time
sse	7	7	13	4	91	84.57	69.9s	119	100.0	5.2s
tbk	6	3	16	4	181	98.57*	72.1s	231	98.57*	4.1s
scf	27	54	97	7	502	96.14	83.1m	541	100.0	71s
dfile	2	1	24	5	124	96.94	104s	144	100.0	2.0s
planet	7	19	48	6	417	98.82	373s	449	100.0	14s

s is CPU-seconds, m is CPU-minutes on a VAX 11/8650 running ULTRIX
* OL block was **not** combinationally irredundant

Table 8.1 Synthesis for Testability Results

EXAMPLE	I - OPTIMIZE		II - TESTABLE	
	#gates	area	#gates	area
sse	91	1.0	129	1.34
tbk	181	1.0	231	1.10
scf	502	1.0	541	1.01
dfile	124	1.0	144	0.98
planet	417	1.0	449	0.86

Table 8.2 Areas of Standard Cell Designs

In Table 8.2, some constrained designs are about the same size or smaller than the corresponding unconstrained ones. Logic partitioning, in these cases, has decreased routing complexity in the circuit to the extent of nullifying the increase in the number of logic gates. The cost function used in multi-level logic optimization is the number of literals (transistors) in the circuit [6], and is sometimes a poor estimate of the circuit area.

The number of test sequences required varied between 30-70 for these examples. The number of test sequences can be reduced by applying combinational test compaction strategies after generating all the test vectors for the combinational logic blocks. The average length of each sequence was about 5. Since the test vectors only access the primary inputs and only the primary outputs are observed, each vector can be applied in one clock cycle.

8.8 Conclusions

A synthesis procedure has been described that produces an optimized, fully and easily testable, logic implementation of a sequential machine from a State Transition Graph description of the machine. This logic-level implementation is guaranteed to be testable for all single stuck-at faults. No access to the memory elements is required. The test sequences for these faults can be obtained using combinational test generation techniques alone.

It has been shown that an intimate relationship exists between state assignment and the testability of a sequential machine. A procedure of constrained state assignment and logic optimization can guarantee a fully testable machine.

The testing time required in this method is smaller than that using a Scan Design methodology. Experimental results have shown that the area penalty incurred due to the constraints on the optimization are small. The performance of the synthesized design is usually better than a unconstrained design optimized for area alone. Future work includes generalizing these techniques to cascaded and interconnected finite state machine descriptions.

CHAPTER 9

Conclusions

In this dissertation, a framework for automated synthesis of integrated circuit chips from behavioral descriptions was presented. This prototype behavioral synthesis system incorporates a number of optimization tools for datapath and control synthesis as well as efficient verification and test strategies.

An introduction to existing synthesis systems was provided in Chapter 1. The different phases in synthesizing from behavioral descriptions were described and the optimization steps required during synthesis were discussed. The need for verification and test subsystems in a synthesis system was indicated. Finally, work done previously in this area was reviewed.

In Chapter 2, an overview of the synthesis system developed was presented. The first step in synthesis, hardware allocation, produces a structural specification of the circuit from a behavioral description. Datapath and control specifications are produced. Logic synthesis tools operate on these specifications to produce gate-level descriptions. Layout synthesis tools generate layout from this gate-level specification. A synthesis system is incomplete without verification and test strategies. Verification tools compare circuit specifications at different levels to ensure that the optimization tools have not introduced any logic errors in the circuit. Test generation algorithms produce a set of test vectors which are applied to the fabricated chip to check for correct functionality.

The new algorithms and tools developed for synthesis, verification and test were described in Chapters 3-8. Hardware allocation is the most creative step in synthesis. The structure of the circuit, i.e. the number of registers, arithmetic units and their interconnections, is decided during this step. The allocation problem was formulated as a two-dimensional

placement problem of operations in space and time. A simulated-annealing-based algorithm was used to solve the allocation problem. Excellent results were achieved because the optimization strategy was global. Tradeoffs between hardware resource cost and execution speed of the synthesized chip can be effectively explored using this algorithm. This algorithm was also extended to synthesize pipelined datapaths. Future work in this area includes optimal scheduling loops with data-dependent exits, where the number of iterations cannot be statically determined.

After hardware allocation, the finite state machine (FSM) controller associated with the resulting datapath must be synthesized. The tools used for this control synthesis step were described in Chapter 4. The specification of the FSM controller is derived from the two-dimensional placement of operations produced by the hardware allocation step. State assignment and logic optimization tools operate on this FSM specification to produce gate-level descriptions. All previous work in automatic state assignment of FSM's was aimed at two-level i.e. Programmable Logic Array (PLA) implementations of the combinational logic of the FSM. Multi-level logic implementations can be substantially faster and smaller than corresponding two-level implementations. New algorithms for state assignment of FSM's targeted toward multi-level logic implementations of the combinational part of the FSM were developed. These algorithms, implemented in the program MUSTANG, minimize an estimate of the area of the FSM after multi-level logic optimization. The estimate of area used is the number of literals in a factored form of logic. MUSTANG produces literal counts between 25 and 40% less than other state assignment techniques. However, these results could be improved by modeling more complicated multi-level logic optimizations like common kernel extraction.

A substantial fraction of datapath and controller area is occupied by combinational circuits. Combinational logic is used to realize arithmetic and Boolean operations as well as to implement finite state machines. The optimization of combinational circuits for area and

speed is a very important problem. Significant advances have been made in the use of algebraic techniques to factorize and decompose logic equations and realize minimal area multi-level logic networks. Boolean decomposition techniques are potentially more powerful. In Chapter 5, algorithms for Boolean decomposition which can be used for decomposing a PLA into a smaller set of interconnected PLAs, optimized for area and delay, were presented. These algorithms have produced excellent results on benchmark examples. Many steps in these algorithms have worst-case exponential time complexities, which may result in exorbitant decomposition times for large PLAs. Also, these algorithms are restricted to begin from a two-level representation of a function, which may not be available. Future work should address these limitations.

The verification subsystem in the synthesis system was described in Chapter 6. This subsystem incorporates tools for verifying sequential machines across different levels of abstraction. Thus, circuit specifications can be verified across optimization tools in the synthesis pipeline. Sequential circuit verification is a difficult problem and previous approaches to solving this problem have been restricted to verifying circuits with small amounts of memory (4-6 latches). New algorithms based on cube-enumeration (as opposed to minterm-enumeration) have significantly reduced the complexity of the verification problem and enabled successful verifications of sequential circuits with more than 15 latches.

In Chapter 7, the testing problem for VLSI circuits was addressed. Test generation is a process which produces a set of tests that detect all or a large subset of potential faults in a logic circuit whose existence would cause the circuit to function incorrectly. The logic circuit may be purely combinational or may be a sequential machine. Generating tests for sequential circuits is a considerably harder problem than combinational logic test generation, because no access is provided to the inputs and output of the memory elements. Approaches taken to solve the sequential testing problem have not proven to be effective thus far. The sequential testing problem is transformed into an easier combinational testing one by applying Scan

Design rules – making all memory elements controllable and observable. However, Scan Design may result in a significant area/performance penalty. The approach to sequential testing described in Chapter 7 represents a significant departure from previous methods. A new sequential test generation algorithm based on the concept of state space enumeration has been developed. This algorithm, used in conjunction with an Incomplete Scan Design approach to sequential test generation, produces excellent results for small-large sequential circuits. Developing deterministic algorithms, effective for circuits with over a 100 latches, with the capability of identifying redundant faults represents a major challenge in this research area.

In Chapter 8, the relationship between combinational logic optimization and test generation for combinational circuits was reviewed. A strong relationship was shown to exist between state assignment, logic optimization and the testability of a sequential circuit. A synthesis and optimization procedure for obtaining fully and easily testable Moore and Mealy finite state machines has been developed. A procedure of constrained state assignment and logic optimization was outlined in Chapter 8. This procedure not only results in a fully testable sequential machine, but also produces the test sequences required to test all single stuck-at faults in the machine via test generation on the combinational blocks of machine. Results obtained indicate negligible performance and small area penalties. Thus, this procedure can be used as a viable alternative to Scan Design approaches especially when testing time is a consideration. Optimal sequential synthesis procedures, without associated constraints, for fully testable non-scan sequential machines have not yet been proposed. Developing such procedures, represents a major theoretical challenge in this area.

The synthesis tools described in this dissertation produce gate-level descriptions of the circuit from its behavioral specification. Layout synthesis tools in the Berkeley Design Environment [63] [60] [59] [86] [62] generate layout for these gate-level specifications.

REFERENCES

1. S. W. Director, A. C. Parker, D. P. Siewiorek and D. E. Thomas, A Design Methodology and Computer Aids for Digital VLSI Systems, *IEEE Transactions on Circuits and Systems CAS-28*, (July 1981), 634-645.

2. A. R. Newton, D. O. Pederson, A. Sangiovanni-Vincentelli and C. H. Sequin, Design Aids for VLSI: The Berkeley Perspective, *IEEE Transactions on Circuits and Systems CAS-28*, (July 1981), 666-680.

3. J. Allen and P. Penfield, VLSI Design Automation Activities at M.I.T., *IEEE Transactions on Circuits and Systems CAS-28*, (July 1981), 645-665.

4. A. R. Newton and A. Sangiovanni-Vincentelli, CAD Tools for ASIC Design, *Proc. of the IEEE 75*, (June 1987), 765-776.

5. L. Trevillyan, An Overview of Logic Synthesis Systems, *Proc. of 24th Design Automation Conference*, Miami Beach, July 1987, 166-172.

6. R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli and A. Wang, MIS: A Multiple Level Logic Optimization System, *IEEE Transactions on CAD CAD-6*, (November 1987), 1062-1081.

7. J. Soukup, Circuit Layout, *Proc. of the IEEE 69*, (October 1981), 1281-1305.

8. A. Sangiovanni-Vincentelli, An Overview of Synthesis Systems, *Proc. of Custom Integrated Circuits Conference*, Rochester, May 1985, 221-225.

9. H. K. Hingarh and B. Marshall, Advanced gate arrays offering power/delay tradeoffs, *Proc. of Semi-Custom Integrated Circuit Technology, Symp.*, Institute for Defense Analysis, Science and Technology Division, May 1981, 39-53.

10. T. Kozawa, H. Horino, T. Ishiga, J. Sakemi and S. Sato, Block and Track Method for automated layout generation of MOS/LSI arrays, *Proc. of Int. Solid-State Circuits Conference*, Philadelphia, 1972, 62-63.

11. D. Johannsen, Bristle Blocks: A silicon compiler, *Proc. of 16th Design Automation Conference*, San Diego, 1979, 310-313.

12. H. E. Shrobe, The Datapath Generator, *Proc. Conf. Adv. Res. in VLSI*, MIT, Cambridge, MASS., January 1982.

13. J. R. Southard, MACPITTS: An Approach to Silicon Compilation, *IEEE Computer 16*, (December 1983), 74-82.

14. A. C. Parker, D. E. Thomas, D. Siewiorek, M. Barbacci, L. Hafer, G. Lieve and J. Kim, The CMU Design Automation System, *Proc. of 16th Design Automation Conference*, San Diego, June 1979, 73-80.

15. A. C. Parker, F. Kurdahi and M. Mlinar, A General Methodology for the Synthesis, Verification of Register-Transfer Designs, *Proc. of 21st Design Automation Conference*, Las Vegas, June 1984, 329-335.

16. J. Granacki, D. Knapp and A. C. Parker, The ADAM Advanced Design Automation System: Overview, Planner and Natural Language Interface, *Proc. of 22nd Design Automation Conference*, Las Vegas, July 1985, 727-730.

17. M. Barbacci, G. Barnes, R. Cattell and D. P. Siewiorek, *The Symbolic Manipulation of Computer Descriptions: ISPS Description Language*, Carnegie Mellon University, Research Report, 1979.

18. *VAX DECSIM Reference Manual*, Digital Equipment Corporation, Hudson, Mass., 1986.

19. N. Park and A. C. Parker, SEHWA: A Program for the synthesis of pipelines, *Proc. of 23rd Design Automation Conference*, Las Vegas, July 1986, 454-460.

20. D. P. Siewiorek, C. G. Bell and A. Newell, *Computer Structures: Principles and Examples*, Computer Science Series, McGraw Hill, 1982.

21. D. E. Thomas, C. Y. Hitchcock, T. J. Kowalski, J. V. Rajan and R. A. Walker, Automated Datapath Synthesis, *IEEE Computer 21*, (December 1983), 59-70.

22. J. A. Fisher, Trace Scheduling: A Technique for global microcode compaction, *IEEE Transactions on Computers C-30*, (July 1981), 478-490.

23. R. K. Brayton, C. T. McMullen, G. D. Hachtel and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, 1984.

24. D. B. Armstrong, A Programmed Algorithm for assigning internal codes to sequential machines, *IRE Trans. Electron Comput. EC-11*, (August 1962), 466-472.

25. T. A. Dolotta and A. J. McCluskey, The coding of internal states of Sequential machines, *IEEE Transactions on Electronic Computers EC-13*, (October 1964), 549-562.

26. G. D. Micheli, R. K. Brayton and A. Sangiovanni-Vincentelli, Optimal state assignment of finite state machines, *IEEE Transactions on CAD CAD-4*, (July 1985), 269-285.

27. T. Sasao, Multiple-Valued Logic and Optimization of Programmable Logic Arrays, *IEEE Computer 21*, (April 1988), 71-80.

28. G. D. Micheli, Symbolic Design of Combinational and Sequential Logic Circuits implemented by Two-level Macros, *IEEE Transactions on CAD CAD-5*, (October 1986), 597-616.

29. R. S. Wei, Logic Verification and Test Generation for VLSI Circuits, *Ph.D Dissertation*, Berkeley, September 1986.

30. J. P. Roth, VERIFY: an algorithm to verify a computer design, *IBM Technical Disclosure Bulletin 15*, (1973), 2646-2648.

31. S. Devadas, H. K. T. Ma and A. R. Newton, On the Verification of Sequential Machines At Differing Levels of Abstraction, *IEEE Transactions on CAD 7*, (June 1988), 713-722.

32. H. K. T. Ma, S. Devadas and A. L. Sangiovanni-Vincentelli, Logic Verification Algorithms and their Parallel Implementation, *Proc. of 24th Design Automation Conference*, Miami Beach, June 1987, 283-290.

33. S. Hwang and A. R. Newton, An Efficient Design Correctness Checker for Finite State Machines, *Proc. of Int'l Conference on Computer-Aided Design*, Santa Clara, November 1987, 410-413.

34. J. K. Ousterhout, Crystal: A Timing Verifier for Digital MOS VLSI, *IEEE Transactions on CAD CAD-4*, (July 1985), 336-349.

35. B. Konemann, J. Mucha and G. Zweihoff, Built-in Self Test for complex digital integrated circuits, *IEEE journal of Solid State Circuits SC-15*, (June 1980), 315-319.

36. E. B. Eichelberger and T. W. Williams, A logic design structure for LSI testability, *Proc. 14th Design Automation Conference*, New Orleans, June 1977, 462-468.

37. V. D. Agarwal, S. K. Jain and D. M. Singer, Automation in Design for Testability, *Proc. of Custom Integrated Circuits Conference*, Rochester, NY, May 1984.

38. M. A. Breuer and A. D. Friedman, *Diagnosis and Reliable Design of Digital Systems*, Computer Science Press, 1976.

39. R. Marlett, EBT: A Comprehensive Test Generation Technique for highly sequential circuits, *Proc. of 15th Design Automation Conference*, Las Vegas, June 1978, 332-338.

40. D. D. Gajski, Arsenic: Methodology and Implementation, *Proc. of 21st Design Automation Conference*, Las Vegas, June 1984, 676-678.

41. H. Trickey, *Compiling Pascal Programs into Silicon*, Stanford Computer Science Report STAN-CS-85-1059, July 1985.

42. H. Trickey, Flamel: A High-Level Hardware Compiler, *IEEE Transactions on CAD CAD-6*, (March 1987), 259-269.

43. M. R. Barbacci and D. P. Sieworek, The CMU RT-CAD System: An Innovative Approach to Computer-Aided Design, *AFIPS Conference Proceedings 45*, (1976), 643-655.

44. C. Y. Hitchcock and D. E. Thomas, A Method for Automatic Data Path Synthesis, *Proc. of 20th Design Automation Conference*, Miami Beach, June 1983, 484-489.

45. T. J. Kowalski and D. E. Thomas, The VLSI Design Automation Assistant: Prototype System, *Proc. of the 20th Design Automation Conference*, Miami Beach, June 1983, 479-483.

46. T. J. Kowalski and D. E. Thomas, The VLSI Design Automation Assistant: What's in a Knowledge Base, *Proc. of the 22nd Design Automation Conference*, Las Vegas, June 1985, 252-258.

47. J. V. Rajan and D. E. Thomas, Synthesis By Delayed Binding of Decisions, *Proc. of 22nd Design Automation Conference*, Las Vegas, June 1985, 367-373.

48. M. R. Barbacci, G. E. Barnes, R. G. Cattell and D. P. Siewiorek, The ISPS Computer Description Language, *Technical Report, Computer Science Department*, Pittsburgh, 1977.

49. L. J. Hafer, Automated Data Memory Synthesis: A Formal Method for the Specification, Analysis and Design of Register Transfer Level Design Logic, *Ph.D Thesis, Carnegie Mellon University*, Pittsburgh, June 1981.

50. C. Tseng and D. P. Siewiorek, Facet: A Procedure for the Automated Synthesis of Digital Systems, *Proc. of the 20th Design Automation Conference*, Miami Beach, June 1983, 490-496.

51. C. Tseng and D. P. Siewiorek, The Modeling and Synthesis of Bus Systems, *Proc. of 18th Design Automation Conference*, Nashville, June 1981, 471-478.

52. C. Tseng and D. P. Siewiorek, Emerald: A Bus Style Designer, *Proc. of 21st Design Automation Conference*, Las Vegas, June 1984, 315-321.

53. E. A. Snow, D. P. Siewiorek and D. E. Thomas, A Technology Relative Computer-Aided Design System: Abstract Representations, Transformations and Design Tradeoffs,

Proc. of 15th Design Automation Conference, San Diego, June 1978, 220-226.

54. M. C. McFarland, Global Transformations on Abstract Hardware Descriptions: A Formal Approach, *Carnegie-Mellon University Internal Report*, Pittsburgh, October 1979.

55. M. C. McFarland, On Proving the Correctness of Optimizing Transformations in a Digital Design Automation System, *Proc. of 18th Design Automation Conference*, Nashville, June 1981, 90-97.

56. B. M. Pangrle and D. D. Gajski, Design Tools for Intelligent Silicon Compilation, *IEEE Transactions on CAD CAD-6*, (November 1987), 1098-1112.

57. N. Park and A. C. Parker, Synthesis of Optimal Clocking Schemes, *Proc. of 22nd Design Automation Conference*, Las Vegas, June 1985, 489-495.

58. A. C. Parker, M. Mlinar and J. Pizarro, MAHA: A Program for Datapath Synthesis, *Proc. of 23rd Design Automation Conference*, Las Vegas, June 1986, 461-466.

59. C. Kring, GEM: A Generalized Array Synthesizer for VLSI, *U. C. Berkeley Internal Report*, Berkeley, 1987.

60. R. Rudell, WOLFE: Oct Interface to the Timberwolf Standard Cell Placement Program, *U. C. Berkeley Internal Report*, Berkeley, 1986.

61. C. Sechen and A. Sangiovanni-Vincentelli, The TimberWolf Placement and Routing Package, *IEEE Journal of Solid-State Circuits and Systems SC-20*, (April 1985), 510-522.

62. S. Chow, OCTOPUS: A Folded PLA Generator, *U. C. Berkeley Internal Report*, Berkeley, 1987.

63. J. Burns, A. Casotto, M. Igusa, F. Romeo, A. Sangiovanni-Vincentelli, C. Sechen, H. Shin, G. Srinath and H. Yaghutiel, MOSAICO: A Macrocell Placement and Routing System, *Proc. of VLSI '87*, Vancouver, August 1987, 133-147.

64. R. W. Hartenstein, *Hardware Description Languages*, Elsevier Science Pub. Co., Amsterdam, 1987.

65. J. R. Duley and D. L. Dietmeyer, A Digital System Design Language (DDL), *IEEE Transactions on Computers C-17*, (1968), 850-861.

66. M. Shahdad, An Overview of VHDL Language and Technology, *Proc. of 23rd Design Automation Conference*, Las Vegas, June 1986, 320-326.

67. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice Hall Software Series, 1978.

68. K. Jensen and N. Wirth, *PASCAL Users Manual and Report*, Springer-Verlag, NY, 1975.

69. S. Kirkpatrick, C. D. Gelatt and M. P. Vecchi, Optimization by Simulated Annealing, *Science 220, no. 4598*, (May 1983), 671-680.

70. S. Devadas and A. R. Newton, Algorithms for Hardware Allocation on Datapath Synthesis, *Proc. of Int'l Conference on Computer Design: VLSI In Computers*, New York, October 1987, 526-531.

71. S. Devadas and A. R. Newton, Datapath Synthesis from Behavioral Descriptions: An Algorithmic Approach, *Proc. of Int'l Conference on Circuits and Systems*, Philadelphia, PA, May 1987, 398-401.

72. P. G. Paulin and J. P. Knight, Force-directed Scheduling in Automatic Datapath Synthesis, *Proc. of 24th Design Automation Conference*, Miami Beach, July 1987, 195-202.

73. F. Romeo and A. Sangiovanni-Vincentelli, Probabilistic Hill Climbing Algorithms: Properties and Applications, *1985 Chapel Hill Conference on VLSI*, Chapel Hill, December 1985, 393-417.

74. S. Devadas, H. T. Ma, A. R. Newton and A. Sangiovanni-Vincentelli, MUSTANG: State Assignment of Finite State Machines for Optimal Multi-Level Logic Implementations, *Proc. of Int'l Conference on Computer-Aided Design*, Santa Clara, November 1987, 16-19.

75. S. Devadas, H. T. Ma, A. R. Newton and A. Sangiovanni-Vincentelli, MUSTANG: State Assignment of Finite State Machines Targeting Multi-Level Logic Implementations, *IEEE Transactions on CAD to appear*, (1989), .

76. R. K. Brayton and C. T. McMullen, Synthesis and Optimization of Multistage logic, *Proc. of Int'l Conference on Computer Design: VLSI in Computers*, NY, October 1984, 23-28.

77. K. Bartlett, G. D. Hachtel, A. J. D. Geus and W. Cohen, Synthesis and Optimization of Multi-level Logic under Timing Constraints, *Proc. of Int'l Conference on Computer-Aided Design*, Santa Clara, November 1985, 290-292.

78. R. K. Brayton, N. L. Brenner, C. L. Chen, G. D. Micheli, C. T. McMullen and R. H. J. M. Otten, The Yorktown Silicon Compiler, *Proc. of International Symposium on Circuits and Systems*, Kyoto, Japan, June 1985, 391-394.

79. J. Darringer, W. Joyner, L. Berman and L. Trevillyan, Logic Synthesis Through Local Transformations, *IBMJRD 25*, (July 1981), 272-280.

80. M. Hofmann, Automated synthesis of multi-level combinational logic in CMOS technology, *Ph.D Dissertation*, Berkeley, 1985.

81. K. A. Bartlett, R. K. Brayton, G. D. Hachtel, R. M. Jacoby, C. R. Morrison, R. L. Rudell, A. L. Sangiovanni-Vincentelli and A. R. Wang, Multi-level logic minimization using implicit don't cares, *IEEE Transactions on CAD 7*, (June 1988), 723-740.

82. R. A. Wood, A High Density Programmable Logic Array Chip, *IEEE Transactions on Computers C-28*, (September 1979), 602-608.

83. S. Devadas, A. Wang, A. R. Newton and A. Sangiovanni-Vincentelli, Boolean Decomposition of Programmable Logic Arrays, *Proc. of Custom Integrated Circuits Conference*, Rochester, May 1988, 2.5.1-2.5.5.

84. S. Devadas, A. R. Wang, A. R. Newton and A. Sangiovanni-Vincentelli, Boolean Decomposition in Multi-Level Logic Optimization, *Proc. of Int'l Conference on Computer-Aided Design*, Santa Clara, November 1988.

85. R. Rudell and A. Sangiovanni-Vincentelli, Multiple valued Minimization for PLA optimization, *IEEE Transactions on CAD CAD-6*, (September 1987), 727-751.

86. S. Devadas and A. R. Newton, GENIE: A Generalized Array Optimizer for VLSI Synthesis, *Proc. of 23rd Design Automation Conference*, Las Vegas, July 1986, 631-637.

87. J. Reed, A. Sangiovanni-Vincentelli and M. Santamauro, A New Symbolic Channel Router: YACR2, *IEEE Transactions on Computer-Aided Design CAD-4*, (July 1985), 208-220.

88. D. Braun, J. Burns, S. Devadas, H. T. Ma, K. Mayaram, F. Romeo and A. Sangiovanni-Vincentelli, CHAMELEON: A Multi-layer Channel Router, *Proc. of 23rd Design Automation Conference*, Las Vegas, July 1986, 495-502.

89. W. E. Donath and H. Ofek, Automatic identification of equivalence points for Boolean Logic Verification, *IBM Technical Disclosure Bulletin 18*, (January 1976), 2700-2703.

90. J. P. Roth, Hardware Verification, *IEEE Transactions on Computers C-26*, (1977), 1292-1294.

91. J. P. Roth, *Computer Hardware Testing and Verification*, Computer Science Press, Potomac, Maryland, 1980.

92. G. Odawara, M. Tomita, O. Okuzawa and T. Ohta, A Logic Verifier based on Boolean Comparison, *Proc. 23rd Design Automation Conference*, Las Vegas, June 1986, 208-214.

93. M. A. Breuer, A Random and an Algorithmic technique for fault detection and Test generation for sequential circuits, *IEEE Transactions on Computers C-20*, (November 1971), 1366-1370.

94. H. D. Schnurmann, E. Lindbloom and R. G. Carpenter, The Weighted Random Test-Pattern Generator, *IEEE Transactions on Computers C-24*, (July 1975), 695-700.

95. S. Mallela and S. Wu, A Sequential Test Generation System, *Proc. of International Test Conference*, Philadelphia, PA, October 1985, 57-61.

96. S. Nitta, M. Kawamura and K. Hirabayashi, Test Generation by Activation and Defect-Drive (TEGAD), *INTEGRATION, the VLSI Journal 3*, (1985), 2-12.

97. S. Shteingart, A. W. Nagle and J. Grason, RTG: Automatic Register Level Test Generator, *Proc. of 22nd Design Automation Conference*, Las Vegas, June 1985, 803-807.

98. H. K. T. Ma, S. Devadas, A. R. Newton and A. L. Sangiovanni-Vincentelli, Test Generation for Sequential Finite State Machines, *Proc. of Int'l Conference on Computer-Aided Design (ICCAD)*, Santa Clara, November 1987, 288-291.

99. H. T. Ma, S. Devadas, A. R. Newton and A. Sangiovanni-Vincentelli, Test Generation for Sequential Circuits, *IEEE Transactions on CAD 7*, (October 1988), .

100. H. T. Ma, S. Devadas, A. R. Newton and A. Sangiovanni-Vincentelli, An Incomplete Scan Design Approach to Test Generation for Sequential Machines, *Proc. of Int'l Test Conference*, Washington D. C., September 1988.

101. S. Devadas, H. T. Ma and A. Sangiovanni-Vincentelli, *Logic Verification, Testing and Their Relationship to Logic Synthesis in Design Systems for VLSI Circuits*, Nijhoff, July 1987.

102. S. Devadas, H. T. Ma, A. R. Newton and A. Sangiovanni-Vincentelli, Optimal Logic Synthesis and Testability: Two Faces of the Same Coin, *Proc. of Int'l Test Conference*,

Washington D.C., September 1988.

103. S. Devadas, H. T. Ma, A. R. Newton and A. Sangiovanni-Vincentelli, Synthesis and Optimization Procedures for Fully and Easily Testable Sequential Machines, *Proc. of Int'l Test Conference*, Washington D. C., September 1988.

104. W. Wolf, K. Keutzer and J. Akella, A Kernel Finding State Assignment Algorithm for Multi-Level Logic, *Proc. of 25th Design Automation Conference*, Anaheim, June 1988, 433-438.

105. M. C. McFarland and T. J. Kowalski, Assisting DAA: The Use of Global Analysis in an Expert System, *Proc. of Int'l Conference on Computer Design: VLSI in Computers*, NY, October 1986, 482-485.

106. G. D. Micheli and A. Sangiovanni-Vincentelli, Multiple Constrained Folding of Programmable Logic Arrays: Theory and Applications, *IEEE Transactions on CAD CAD-2*, (July 1983), 151-167.

107. P. Egan and C. L. Liu, Bipartite Folding and Partitioning of a PLA, *IEEE Transactions on CAD CAD-3*, (July 1984), 191-198.

108. A. Hashimoto and J. Stevens, Wire routing by Optimizing Channel Assignment Within Large Apertures, *Proc. of 8th D. A. Workshop*, Las Vegas, 1971, 155-169.

109. M. C. McFarland, Reevaluating the Design Space for Register-Transfer Hardware Synthesis, *Proc. of ICCAD-87*, Santa Clara, November 1987, 262-265.

110. *MSU Standard Cell Library*, Microelectronics Design Division, Institute for Technology Development, 1986.

111. D. A. Hodges and H. G. Jackson, *Analysis and Design of Digital Integrated Circuits*, McGraw Hill, 1983.

112. R. Gyurcsik, An Attached Processor for MOS-Transistor Model Evaluation, *Ph.D Dissertation*, Berkeley, 1986.

113. P. M. Kogge, *The Architecture of Pipelined Computers*, New York: McGraw-Hill, 1981.

114. E. Davidson, Effective Control for Pipelined Computers, *COMPCON Digest*, San Francisco, 1975, 181-184.

115. J. H. Patel and F. S. Davidson, Improving the througput of a Pipeline by the insertion of delays, *IEEE/ACM 3rd Annual Symposium on Computer Architecture*, Rochester, 1976, 159-163.

116. H. C. Torng, An Algorithm for finding secondary assignments of synchronous sequential circuits, *IEEE Transactions on Computers C-17*, (May 1968), 416-469.

117. G. D. Micheli, A. Sangiovanni-Vincentelli and T. Villa, Computer-Aided Synthesis of PLA-based finite state machines, *Proc. of Int'l conference on Computer-Aided Design*, Santa Clara, November 1983, 154-156.

118. A. J. Coppola, An Implementation of a State Assignment Heuristic, *Proc. of 23rd Design Automation Conference*, Las Vegas, July 1986, 643-649.

119. C. Tseng, A. M. Prabhu, C. Li, Z. Mehmood and M. M. Tong, A Versatile Finite state machine synthesizer, *Proc. of Int'l Conference on Computer-Aided Design*, Santa Clara, November 1986, 206-209.

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

121. J. Darringer, D. Brand, J. Gerbi, W. Joyner and L. Trevillyan, LSS: A System for production logic synthesis, *IBMJRD 28*, (September 1984), 537-545.

122. G. D. Hachtel and R. M. Jacoby, Verification Algorithms for VLSI Synthesis, *IEEE Transactions on CAD 7*, (1988), 616-640.

123. R. E. Bryant, Symbolic Verification of MOS Circuits, *1985 Chapel Hill Conference on VLSI*, Chapel Hill, December 1985, 419-438.

124. R. E. Bryant, Graph-Based Algorithms for Boolean Function Manipulation, *IEEE Transactions on Computers C-35*, (August 1986), 677-691.

125. M. Browne, E. Clarke, D. Dill and B. Mishra, Automatic Verification of Sequential Circuits using Temporal Logic, *Technical Report CMU-CS-85-100*, Pittsburgh, 1985.

126. F. Maruyama and M. Fujita, Hardware Verification, *IEEE Computer 23*, (Feb. 1985), 22-32.

127. D. Dill and E. Clarke, Automatic Verification of Asynchronous Circuits using Temporal Logic, *1985 Chapel Hill Conference on VLSI*, Chapel Hill, 1985, 127-143.

128. K. J. Supowit and S. J. Friedman, A New Method for verifying Sequential Circuits, *Proc. of 23rd Design Automation Conference*, Las Vegas, June 1986.

129. J. D. Ullman, *Computational Aspects of VLSI*, Computer Science Press, Rockville, Maryland, 1984.

130. J. P. Roth, Diagnosis of Automata Failures: a calculus and a method, *IBM journal of Research and Development 10*, (July 1966), 278-291.

131. P. Goel, An Implicit Enumeration Algorithm to generate tests for combinational logic circuits, *IEEE Transactions on Computers C-30*, (March 1981), 215-222.

132. C. Y. Lee, Representation of Switching Circuits by Binary Decision Diagrams, *Bell Syst. Tech. J 38*, (July 1959), 985-999.

133. S. B. Akers, Binary Decision Diagrams, *IEEE Transactions on Computers C-27*, (June 1978), 509-516.

134. O. H. Ibarra and S. K. Sahni, Polynomially complete fault detection problems, *IEEE Transactions on Computers C-24*, (March 1975), 242-249.

135. J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Reading, Mass., 1979.

136. M. Hill and al, Design decisions in SPUR, *IEEE Computer 19*, (November 1986), 8-22.

APPENDIX A

DEVICE MODEL ROUTINES FOR A CIRCUIT SIMULATOR

C Description

```
/*
 *  evalMosfet: this routine evaluates the MOS (Level1) model for current
 *      and conductance, and transconductance terms. It also loads
 *      the entries into the appropriate matrix locations provided
 *      in the MOSFET instance.
 *
 */

evalMosfet(mosfet)
MOSFET mosfet;
{
    BOOL forward,pmosfet;
    FLOAT vt, q, wolkp, wolkpla, vgst, vbd,vbs,vdst, vdiff, vds;
    FLOAT gmin = 1.0e-12;
    FLOAT exp() ,idbd,idbs,gdbd,gdbs;
    FLOAT phis;
    FLOAT gds, gmg, gmb, gsu, ids, ieq, sqrt(), fabs();
    FLOAT vd,vs,vb,vg;
    MODEL mosType;
    STRING pmos = "pmos";
/*
 *  get model mosType from mosfet
 */
    mosType = mosfet->mosfetType;
    wolkp   = mosfet->wolkp;
/*
*   check if pmos or nmos transistor.If pmos invert all the voltages
*/
    if (strcmp(mosType->modelName,pmos)==0)
        pmosfet = TRUE;
    else
        pmosfet = FALSE;

    if (pmosfet)
```

```
    {
     vd = - *(mosfet->pvd);
     vg = - *(mosfet->pvg);
     vs = - *(mosfet->pvs);
     vb = - *(mosfet->pvb);
     mosType->VBI = - mosType->VBI;
    }
   else
    {
     vd =  *(mosfet->pvd);
     vg =  *(mosfet->pvg);
     vs =  *(mosfet->pvs);
     vb =  *(mosfet->pvb);
    }

/*
*  To avoid singular matrices
*/
   *(mosfet->pgg) += gmin;

/*
 *  check for forward operation (vd > vs)
 */
   if( vd > vs ) {
      phis = mosType->PHIS;
      q = sqrt(MAX(phis, vs - vb+phis));
      vt = mosType->VBI + mosType->GAMMA*q;
/*
 *  forward but must be above threshold
 */
        if( (vgst = vg - vs - vt) <= 0.0 )  {
         *(mosfet->pdd) +=gmin;
         *(mosfet->pss) +=gmin;
         return;
      }
        vds = vd-vs;
        vdst = MIN( vds, vgst);
        wolkpla = wolkp*(1.0 + mosType->LAMBDA*vds);
        forward = TRUE;
   }
   else if( vd < vs ) {
      phis = mosType->PHIS;
      q = sqrt(MAX(phis, vd-vb+phis));
      vt = mosType->VBI + mosType->GAMMA*q;
/*
 *  inverse but must be above threshold
 */
        if( (vgst = vg - vd - vt) <= 0.0 )  {
          *(mosfet->pdd) +=gmin;
          *(mosfet->pss) +=gmin;
          return;
      }
        vds = vs-vd;
```

```
            vdst = MIN( vds, vgst);
            wolkpla = wolkp*(1.0 + mosType->LAMBDA*vds);
            forward = FALSE;
        }
        else {
/*
 *  vds == 0; no current flows
 */
            phis = mosType->PHIS;
            q = sqrt(MAX(phis, vs-vb+phis));
            vt = mosType->VBI + mosType->GAMMA*q;
              gds = wolkp*fabs(vg - vs - vt);
              *(mosfet->pdd) += gds;
              *(mosfet->pss) += gds;
              *(mosfet->psd) -= gds;
              *(mosfet->pds) -= gds;
              return;
        }
        vbs = MIN(vb - vs ,0.0);
        vbd = MIN(vb - vd ,0.0);
        idbs = mosType->IS *( exp(vbs/KTQ)*( 1 - vbs/KTQ ) -1 );
        idbd = mosType->IS *( exp(vbd/KTQ)*( 1 - vbd/KTQ ) -1 );
        gdbs = (mosType->IS)/KTQ * exp(vbs/KTQ);
        gdbd = (mosType->IS)/KTQ * exp(vbd/KTQ);
        vdiff = vgst-vdst;
        gmg = wolkpla*vdst;
        if( mosType->LAMBDA <= 0.0 ) {
            gds = wolkpla*vdiff;
        }
        else {
            gds = wolkp*(vdiff + mosType->LAMBDA*(vds*vdiff+vdst*(vgst-vdst/2.0)));
        }
        gmb = wolkpla*mosType->GAMMA*vdst/2.0/q;
        gsu = gds + gmg + gmb;
        if( forward ) {
              ids = wolkpla*(vgst-vdst/2.0)*vdst;
              ieq = ids - gmg*(vg-vs)
                  - gmb*(vb-vs) - gds*vds;
/*
 *  load matrix & rhs for forward operation
 */
            if (!pmosfet)
            {
             *(mosfet->pdr) -= (ieq-idbd );
             *(mosfet->psr) += (ieq+idbs);
             *(mosfet->pbr) -= (idbs+idbd);
            }
            else
            {
             *(mosfet->pdr) += (ieq - idbd);
             *(mosfet->psr) -= (ieq + idbs);
             *(mosfet->pbr) += (idbs+idbd);
            }
```

```
            *(mosfet->pdd) += (gds+gdbd);
            *(mosfet->pdg) += gmg;
            *(mosfet->pds) -= gsu;
            *(mosfet->pdb) += (gmb - gdbd);
            *(mosfet->psd) -= gds;
            *(mosfet->psg) -= gmg;
            *(mosfet->pss) += (gsu+gdbs);
            *(mosfet->psb) -= (gmb+gdbs);
          *(mosfet->pbb) += (gdbd + gdbs );
          *(mosfet->pbd) -= gdbd;
          *(mosfet->pbs) -= gdbs;

      }
      else {
            ids = wolkpla*(vgst-vdst/2.0)*vdst;
            ieq = ids - gmg*(vg-vd)
                - gmb*(vb-vd) - gds*vds;
/*
 *  load matrix & rhs for inverse operation
 */
          if (!pmosfet)
          {
           *(mosfet->pdr) += (ieq + idbd );
           *(mosfet->psr) -= (ieq - idbs  );
           *(mosfet->pbr) -= (idbs + idbd);
          }
          else
          {
           *(mosfet->pdr) -= (ieq + idbd);
           *(mosfet->psr) += (ieq - idbs);
           *(mosfet->pbr) += (idbs + idbd);
          }
            *(mosfet->pdd) += gsu;
            *(mosfet->pdg) -= gmg;
            *(mosfet->pds) -= gds;
            *(mosfet->pdb) -= (gmb+gdbd);
            *(mosfet->psd) -= gsu;
            *(mosfet->psg) += gmg;
            *(mosfet->pss) += (gds+gdbs);
            *(mosfet->psb) += (gmb - gdbs);
          *(mosfet->pbb) += (gdbd + gdbs);
          *(mosfet->pbd) -= gdbd;
          *(mosfet->pbs) -= gdbs;
      }

      /*
      *    invert the threshold voltage back again if it was a pmos Xstr
      */
        if (pmosfet)
          mosType->VBI   = - mosType->VBI;

      return;
}
```

Intermediate Description

```
#
#  evalMosfet: this routine evaluates the MOS (Level1) model for current
#      and conductance, and transconductance terms. It also loads
#      the entries into the appropriate matrix locations provided
#      in the MOSFET instance.
#

#
# REGISTERS
#
# forward, pmosfet
# vt, q, wolkp, vgst, vbd, vbs, vdst, vdiff, vds
# gmin, idbd, idbs, gdbd, gdbs
# phis, gdsm gmg, gmb, gsu
# ids, ieq, ktq, pmos
# vd, vs, vb, vg
# temp*
#
#
#



  (serial
    (parallel
      (equal MOSFET[WOLKP] wolkp)
      (equal 1.0e-12 gmin)
      (equal PMOS pmos)
      (equal KTQ ktq)
    )
#
#   check if pmos or nmos transistor. If pmos invert all the voltages
#
    (compare MOSFET[MODELNAME] pmos STATUS)
#
# branch on pmosfet or nmosfet
#
    (eior                          ! on STATUS
      (equal TRUE pmosfet)
      (equal FALSE pmosfet)
    )
#
# branch on pmosfet or nmosfet
#
    (eior
      (serial
```

```
        (parallel
          (equal MOSFET[PVD] vd )
          (equal MOSFET[PVG] vg )
          (equal MOSFET[PVB] vb )
          (equal MOSFET[PVS] vs )
        )
        (parallel
          (minus 0.0 vd vd)
          (minus 0.0 vg vg)
          (minus 0.0 vb vb)
          (minus 0.0 vs vs)
          (minus 0.0 MOSFET[VBI] MOSFET[VBI] )
        )
      )
      (parallel
        (equal MOSFET[PVD] vd )
        (equal MOSFET[PVG] vg )
        (equal MOSFET[PVB] vb )
        (equal MOSFET[PVS] vs )
      )
    )

#
#  To avoid singular matrices
#
    (parallel
      (add MATRIX[PGG] gmin MATRIX[PGG])
      (compare vd vs STATUS)
    )
#
#  check for forward operation (vd > vs)
#
    (eior                          ! STATUS
      (serial
        (implic
          (equal MOSFET[PHIS] phis)
          (minus vs vb temp1a)
          (add phis temp1a temp2a)
          (max phis temp2a temp3a)
          (sqrt temp3a q)
          (mult MOSFET[GAMMA] q temp4a)
          (add MOSFET[VBI] temp4a vt)
#
#  forward but must be above threshold
#
          (minus vg vs temp5a)
          (minus temp5a vt vgst)
          (compare vgst 0.0 STATUS)
        )
#
# check if above or below threshold
#
        (eior                      ! STATUS
```

```
            (parallel
              (add MATRIX[PDD] gmin MATRIX[PDD])
              (add MATRIX[PSS] gmin MATRIX[PSS])
            )
            (implic
              (minus vd vs vds)
              (min vds vgst vdst)
              (mult vds MOSFET[LAMBDA] temp6a)
              (add temp6a 1.0 temp7a)
              (mult wolkp temp7a wolkpla)
              (equal TRUE forward)
            )
          )
        )
        (serial
          (implic
            (equal MOSFET[PHIS] phis)
            (minus vd vb temp1b)
            (add phis temp1b temp2b)
            (max phis temp2b temp3b)
            (sqrt temp3b q)
            (mult MOSFET[GAMMA] q temp4b)
            (add MOSFET[VBI] temp4b vt)
#
# inverse but must be above threshold
#
            (minus vg vd temp5b)
            (minus temp5b vt vgst)
            (compare vgst 0.0 STATUS)
          )
#
# check if above or below threshold
#
          (eior                         ! STATUS
            (parallel
              (add MATRIX[PDD] gmin MATRIX[PDD])
              (add MATRIX[PSS] gmin MATRIX[PSS])
            )
            (implic
              (minus vs vd vds)
              (min vds vgst vdst)
              (mult vds MOSFET[LAMBDA] temp6b)
              (add temp6b 1.0 temp7b)
              (mult wolkp temp7b wolkpla)
              (equal FALSE forward)
            )
          )
        )
        (implic
#
# vds == 0; no current flows
#
          (equal MOSFET[PHIS] phis)
```

```
        (minus vs vb temp1c)
        (add phis temp1c temp2c)
        (max phis temp2c temp3c)
        (sqrt temp3c q)
        (mult MOSFET[GAMMA] q temp4c)
        (add MOSFET[VBI] temp4c vt)
        (minus vg vs temp5c)
        (minus temp5c vt temp6c)
        (fabs temp6c temp7c)
        (mult wolkp temp7c gds)
        (add MATRIX[PDD] gds MATRIX[PDD])
        (add MATRIX[PSS] gds MATRIX[PSS])
        (minus MATRIX[PSD] gds MATRIX[PSD])
        (minus MATRIX[PDS] gds MATRIX[PDS])
      )
    )
    (implic
      (minus vb vs temp8)
      (minus vb vd temp9)
      (min 0.0 temp8 vbs)
      (min 0.0 temp9 vbd)
      (divide vbs ktq temp10)
      (divide vbd ktq temp11)
      (exp temp10 temp12)
      (exp temp11 temp13)
      (minus 1.0 temp10 temp14)
      (minus 1.0 temp11 temp15)
      (mult temp12 temp14 temp16)
      (mult temp13 temp15 temp17)
      (minus temp16 1.0 temp18)
      (minus temp17 1.0 temp19)
      (mult MOSFET[IS] temp18 idbs)
      (mult MOSFET[IS] temp19 idbd)
      (mult MOSFET[IS] temp12 temp20)
      (mult MOSFET[IS] temp13 temp21)
      (divide temp20 ktq gdbs)
      (divide temp21 ktq gdbd)
      (minus vgst vdst vdiff)
      (mult wolkpla vdst gmg)
    )
    (compare MOSFET[LAMBDA] 0.0 STATUS)
#
# branch on lambda postive or negative
#
    (eior                              ! STATUS
      (mult wolkpla vdiff gds)
      (implic
        (divide vdst 2.0 temp22)
        (minus vgst temp22 temp23)
        (mult vds vdiff temp24)
        (mult temp23 vdst temp25)
        (add temp25 temp24 temp26)
        (mult MOSFET[LAMBDA] temp26 temp27)
```

```
        (add vdiff temp27 temp28)
        (mult wolkp temp28 gds)
      )
    )
    (implic
      (mult wolkpla MOSFET[GAMMA] temp29)
      (mult temp29 vdst temp30)
      (mult q 2.0 temp31)
      (divide temp30 temp31 gmb)
      (add gds gmg temp32)
      (add temp32 gmb gsu)
    )
#
# branch on forward operation or reverse
#
    (eior
      (serial
        (implic
          (divide vdst 2.0 temp33a)
          (minus vgst temp33a temp34a)
          (mult wolkpla vdst temp35a)
          (mult temp35a temp34a ids)
          (minus vg vs temp36a)
          (minus vb vs temp37a)
          (mult gds vds temp38a)
          (mult gmb temp37a temp39a)
          (mult gmg temp36a temp40a)
          (minus ids temp40a temp41a)
          (minus temp41a temp39a temp42a)
          (minus temp42a temp38a ieq)
        )
#
# load matrix & rhs for forward operation
# branch on pmosfet or nmosfet
#
        (eior
          (serial
            (parallel
              (minus ieq idbd temp43a)
              (add ieq idbs temp44a)
              (add idbs idbd temp45a)
            )
            (parallel
              (minus MATRIX[PDR] temp43a MATRIX[PDR])
              (add MATRIX[PSR] temp44a MATRIX[PSR])
              (minus MATRIX[PBR] temp45a MATRIX[PBR])
            )
          )
          (serial
            (parallel
              (minus ieq idbd temp43a)
              (add ieq idbs temp44a)
              (add idbs idbd temp45a)
```

```
            )
            (parallel
              (add MATRIX[PDR] temp43a MATRIX[PDR])
              (minus MATRIX[PSR] temp44a MATRIX[PSR])
              (add MATRIX[PBR] temp45a MATRIX[PBR])
            )
          )
        )
        (implic
          (add gds gdbd temp46a)
          (minus gmb gdbd temp47a)
          (add gsu gdbs temp48a)
          (add gmb gdbs temp49a)
          (add gdbd gdbs temp50a)
          (add MATRIX[PDD] temp46a MATRIX[PDD])
          (add MATRIX[PDG] gmg MATRIX[PDG])
          (minus MATRIX[PDS] gsu MATRIX[PDS])
          (add MATRIX[PDB] temp47a MATRIX[PDB])
          (minus MATRIX[PSD] gds MATRIX[PSD])
          (minus MATRIX[PSG] gmg MATRIX[PSG])
          (add MATRIX[PSS] temp48a MATRIX[PSS])
          (minus MATRIX[PSB] temp49a MATRIX[PSB])
          (add MATRIX[PBB] temp50a MATRIX[PBB])
          (minus MATRIX[PBD] gdbd MATRIX[PBD])
          (minus MATRIX[PBS] gdbs MATRIX[PBS])
        )
      )
      (serial
        (implic
          (divide vdst 2.0 temp33b)
          (minus vgst temp33b temp34b)
          (mult wolkpla vdst temp35b)
          (mult temp35b temp34b ids)
          (minus vg vd temp36b)
          (minus vb vd temp37b)
          (mult gds vds temp38b)
          (mult gmb temp37b temp39b)
          (mult gmg temp36b temp40b)
          (minus ids temp40b temp41b)
          (minus temp41b temp39b temp42b)
          (minus temp42b temp38b ieq)
        )
#
# load matrix & rhs for inverse operation
# branch on pmosfet or nmosfet
#
        (eior
          (serial
            (parallel
              (add ieq idbd temp43b)
              (minus ieq idbs temp44b)
              (add idbs idbd temp45b)
            )
```

```
            (parallel
              (add MATRIX[PDR] temp43b MATRIX[PDR])
              (minus MATRIX[PSR] temp44b MATRIX[PSR])
              (minus MATRIX[PBR] temp45b MATRIX[PBR])
            )
          )
          (serial
            (parallel
              (add ieq idbd temp43b)
              (minus ieq idbs temp44b)
              (add idbs idbd temp45b)
            )
            (parallel
              (minus MATRIX[PDR] temp43b MATRIX[PDR])
              (add MATRIX[PSR] temp44b MATRIX[PSR])
              (add MATRIX[PBR] temp45b MATRIX[PBR])
            )
          )
        )
        (implic
          (add gmb gdbd temp46b)
          (add gds gdbs temp47b)
          (minus gmb gdbs temp48b)
          (add gdbd gdbs temp49b)
          (add MATRIX[PDD] gsu MATRIX[PDD])
          (minus MATRIX[PDG] gmg MATRIX[PDG])
          (minus MATRIX[PDS] gds MATRIX[PDS])
          (minus MATRIX[PDB] temp46b MATRIX[PDB])
          (minus MATRIX[PSD] gsu MATRIX[PSD])
          (add MATRIX[PSG] gmg MATRIX[PSG])
          (add MATRIX[PSS] temp47b MATRIX[PSS])
          (add MATRIX[PSB] temp48b MATRIX[PSB])
          (add MATRIX[PBB] temp49b MATRIX[PBB])
          (minus MATRIX[PBD] gdbd MATRIX[PBD])
          (minus MATRIX[PBS] gdbs MATRIX[PBS])
        )
      )
    )
  #
  # only if pmosfet
  #
    (minus 0.0 MOSFET[VBI] MOSFET[VBI])
  )


  CONSTANT   0.0 1.0 2.0 TRUE FALSE 1.0e-12 PMOS KTQ
  SYMMETRIC  add mult

  INITIAL    MOSFET[WOLKP] MOSFET[PVD] MOSFET[PVS] MOSFET[PVB] MOSFET[PVG]
INITIAL    MOSFET[VBI] MOSFET[LAMBDA] MOSFET[PHIS] MOSFET[IS] MOSFET[GAMMA]
  INITIAL    MOSFET[MODELNAME]
```

```
   INITIAL   MATRIX[PDR] MATRIX[PSR] MATRIX[PBR]
   INITIAL   MATRIX[PDD] MATRIX[PDS] MATRIX[PDB] MATRIX[PDG]
   INITIAL   MATRIX[PSD] MATRIX[PSS] MATRIX[PSB] MATRIX[PSG]
   INITIAL   MATRIX[PBD] MATRIX[PBS] MATRIX[PBB]
   INITIAL   MATRIX[PGG]

   INITIAL   STATUS

   FINAL     MOSFET[WOLKP] MOSFET[PVD] MOSFET[PVS] MOSFET[PVB] MOSFET[PVG]
FINAL     MOSFET[VBI] MOSFET[LAMBDA] MOSFET[PHIS] MOSFET[IS] MOSFET[GAMMA]
   FINAL     MOSFET[MODELNAME]

   FINAL     MATRIX[PDR] MATRIX[PSR] MATRIX[PBR]
   FINAL     MATRIX[PDD] MATRIX[PDS] MATRIX[PDB] MATRIX[PDG]
   FINAL     MATRIX[PSD] MATRIX[PSS] MATRIX[PSB] MATRIX[PSG]
   FINAL     MATRIX[PBD] MATRIX[PBS] MATRIX[PBB]
   FINAL     MATRIX[PGG]

   FINAL     STATUS forward pmosfet
```