INCREMENTAL TOOLS FOR THE DESIGN AND VERIFICATION OF VLSI CIRCUITS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Arturo Salz
May 1993

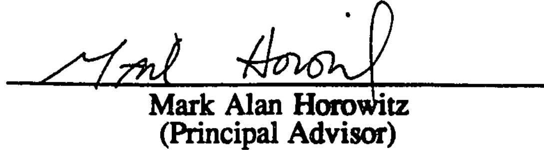
© Copyright 1993

by

Arturo Salz

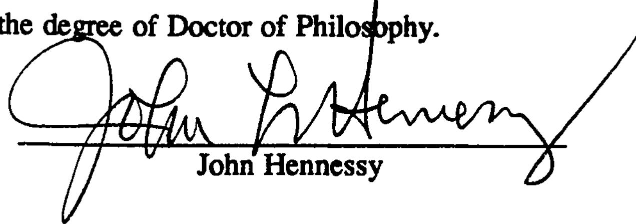
All Rights Reserved

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



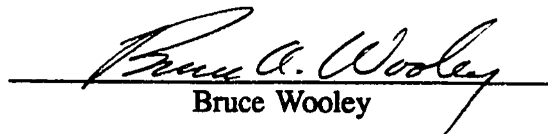
Mark Alan Horowitz
(Principal Advisor)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



John Hennessy

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



Bruce Wooley

Approved for the University Committee on Graduate Studies:

Abstract

Although shrinking geometries and larger VLSI chips have produced faster systems, they have also imposed greater demands on the speed of the design tools. These demands are exacerbated by the iterative nature of the design process whereby circuits are repeatedly modified and verified.

The VLSI design tools described in this thesis reduce design time by operating *incrementally*. An incremental tool takes advantage of the fact that most design changes affect only part of the circuit; thus, many of the computations performed while verifying a modified design are identical to those performed during previous verification runs. By reusing previous results, incremental tools can confine their work to the sections of the design affected by the changes, thereby allowing modifications to be verified in a time proportional to the size of the changes rather than the size of the design.

The incremental system described in this thesis consists of two components: simulator and circuit extractor. The simulator, *Irsim*, is a logic plus timing simulator that employs a *switched-resistor* MOSFET model that accurately simulates circuits at the transistor level. *Irsim* allows designers to modify the circuit being simulated and then incrementally update the waveforms of the sections of the circuit whose behavior is altered by the changes. To accomplish this, *Irsim* maintains a history of circuit activity during simulation and only resimulates the sections of the circuit that deviate from their history.

The circuit extractor is embedded within a hierarchical layout editor; it operates by identifying circuit changes from the layout modifications, and producing the circuit-level transformations needed to update the circuit's network. To accomplish this, the extractor keeps track of the modified layout areas and then extracts only the circuits contained within the modified areas from the layouts before and after the modifications; it then compares the graphs of the extracted subcircuits and reports the differences as network transformations.

Applicability of the system is demonstrated by stepping through the final phases of a large VLSI design. The results show that the incremental tools reduced design cycle runtime by an average of two orders of magnitude.

Acknowledgements

This work would not have been possible without the support, guidance, and encouragement of many people. I cannot possibly list everyone that contributed in making my stay at Stanford a fulfilling and instructive experience; I will mention just a few.

First and foremost, I would like to thank Mark Horowitz for his enduring support and unrelenting encouragement. Working with Mark and learning from him has been a genuine privilege. His guidance kept me focused, his uncanny ability to understand an idea and pinpoint its flaws after a brief conversation kept me alert, and his unfaltering encouragement kept me going.

I would also like to thank John Hennessy and Bruce Wooley for their support and guidance as members of my reading committee. I am also grateful to John Hennessy for helping me get started at Stanford. John Gill and Oyekunle Olukotun were kind enough to be on my orals examination committee; I gratefully acknowledge their efforts on my behalf.

For the test circuits, thanks are due to Mark Santoro for SPIM, Ted Williams for Divider, and the MIPSX team: John Acken, Anant Agarwal, Paul Chow, Scott McFarling, Steve Przybylski, Steve Richardson, Rich Simoni, Don Stark, and Steve Tjiang.

I am grateful to Darlene Hadding and Margaret Rowland for their invaluable assistance with administrative matters, and to Charlie Orgish and Laura Schrager for the long hours they invested in providing us with a dependable computing environment.

Many friends and colleagues contributed in making my passage through graduate school an enjoyable experience. I would like to thank John Vlissides, Craig Dunwoody, Doug Pan, Stuart Marks, and the other members of the Diners Club for many happy memorable meals. I am grateful for the friendship of Helen Davis, Rich Simoni, and

Steve Przybylski; their nocturnal popcorn and amicable conversation made those long nights seem short. I am also grateful to Russell Kao for his friendship and helpful discussions on simulation.

Finally, my most heartfelt gratitude goes to the four people that made this venture possible: my mother Lily, my grandparents Abraham and Sabina, and Andrea, my best friend and companion through life, as well as the most diligent (though unofficial) member of my reading committee. Their unending love and unwavering support were the inspiration that held me together during all the hours I spent away from them producing the document now before you. I thank them for believing in me; this work is dedicated to them.

This research was supported in part by grant MIP 8451822 from the National Science Foundation, and by the Defense Advanced Research Projects Agency under contract N00039-91-C-0138.

Contents

Abstract **iv**

Acknowledgements **v**

1 Introduction **1**

1.1 Incremental System Overview 4

1.2 Organization 5

2 Incremental Simulation **6**

2.1 Background And Related Work 8

2.2 Overview of Rsim 12

2.2.1 Event Processing 15

2.2.2 Event Management 16

2.2.3 Summary 17

2.3 Incremental Switch-Level Timing Simulation 18

2.3.1 An Incremental Simulation Example 18

2.3.2 Algorithm Description 19

2.3.3 Events During Incremental Simulation 25

2.3.4 Incremental States 28

2.3.5 Stage Activation 30

2.3.6 Stage Deactivation 34

2.3.7 History Management 34

2.3.8 Incremental Simulation Loop 37

2.3.9 Reducing Incremental Simulation Time 43

2.4 Performance 49
2.4.1 Worst Case Performance 50
2.4.2 Using Other Models 59
2.4.3 Non-Zero Time Resolution 61
2.4.4 Memory Requirements 62
2.5 Summary 64

3 Incremental Circuit Extraction 67
3.1 Introduction 68
3.2 Overview of Magic 72
3.3 Related Work 75
3.3.1 The Graph Isomorphism Problem 76
3.3.2 Graph Isomorphism Applied to Circuit Comparison 80
3.3.3 Incremental Circuit Update 84
3.4 Incremental Extraction of Circuit Changes 85
3.4.1 Tracking Layout Changes 86
3.4.2 Coalescence of Modified Regions 87
3.4.3 Extracting the Modified Regions 88
3.4.4 Comparing the Circuits 92
3.4.5 Circuit Comparison 99
3.5 Broken Connections 107
3.6 Name Resolution 110
3.7 Performance 111

4 Results 119

5 Conclusions 126

A Stage Analysis 130
A.1 Final Value Computation 130
A.2 Delay Estimation 132
A.3 Charge Sharing Computation 134

A.3.1 Pure Charge Sharing 135
A.3.2 Driven Charge Sharing 137

B Network Modification Commands **139**

C Fault Simulation **143**

Bibliography **146**

List of Tables

2.1 Transistor source-drain resistance as a function of gate level. 14
2.2 Number of events and running times for SPIM 52
2.3 Number of events and running times for DIVIDER 52
2.4 Dissection of execution times for SPIM circuit 55
2.5 Dissection of execution times for DIVIDER circuit 55
2.6 Number of events and running times for different resolutions 62
2.7 Memory usage for various tests circuits 63
3.1 Incremental extraction times for various modified areas 113
3.2 Comparison of incremental and hierarchical extraction plus flattening. . . 116
3.3 Circuit comparison running times for the same and different circuits. . . 118
4.1 Extraction times (in seconds) for each correction of MIPS-X 122
4.2 Simulation times for each correction of MIPS-X 123
4.3 Design cycle turnaround time for MIPS-X corrections. 125
C.1 Comparison of Incremental vs. Serial Fault Simulation. 145

List of Figures

1.1 A simplified view of the VLSI design process 2
1.2 Incremental system overview 4
2.1 Confining simulation to a portion of a modified circuit 7
2.2 Rsim model for an n-channel transistor 13
2.3 Decomposition of a static CMOS register into stages 14
2.4 Effects of aborted events on a CMOS NOR gate 17
2.5 Incremental simulation of a simple gate 19
2.6 Changing the structure of a node may not alter its behavior 21
2.7 Two ways in which a stage becomes active 22
2.8 The behavior of a stage can be affected by its outputs 23
2.9 Stimulus events at inactive inputs are used to stimulate active stages . . 26
2.10 Two ways in which a node can deviate from its history 27
2.11 Ignoring pending events may result in incorrect results 30
2.12 Incorrect resimulation due to aborted events 31
2.13 Possible timing of aborted and effective transitions during simulation . . 33
2.14 Several transitions may have to be rescheduled simultaneously 33
2.15 Possible cases due to an event occurring on an active node 38
2.16 Two ways in which stages are affected by an input 42
2.17 Removing internal nodes from a stage 44
2.18 Small timing differences result in delayed check-point and no-change events 46
2.19 Several situations in which a node transitions before a no-change event . 47
2.20 Number of event as a function of time for SPIM circuit 51
2.21 Number of events as a function of time for DIVIDER circuit 53

2.22 Simulation time as a function of the fraction of the circuit resimulated . 58
2.23 Relative worst-case overhead for the incremental algorithm using other models 61
3.1 Circuit changes due to cell overlap. 70
3.2 A corner-stitched plane of Magic tiles. 72
3.3 Abstract layers in Magic. 74
3.4 A CMOS XOR circuit and its corresponding graph representation. 76
3.5 Two renderings of the graph of the XOR circuit of Figure 3.4. 77
3.6 Partitioning a graph into singleton sets using vertex degree as the invariant. 79
3.7 The graphs of Figure 3.5 including circuit information. 80
3.8 Modified areas are flattened and coalesced into disjoint modified regions. 88
3.9 Determining the boundary of the modified region. 89
3.10 Incrementally extracting a simple layout change. 91
3.11 Transistors that intersect the modified region. 92
3.12 Comparing connectivity changes at the boundary of the circuit. 95
3.13 Algorithm to process boundary connectivity changes. 96
3.14 Result of comparing the boundary of Figure 3.12. 97
3.15 Boundary transistors that appear in several modified regions. 98
3.16 Circuit comparison algorithm. 99
3.17 Function to refine the partitions. 103
3.18 A circuit comparison example. 106
3.19 Incremental extraction running times as a function of area size. 113
3.20 Relative Incremental extraction running times. 114
A.1 Final value solution for static CMOS register circuit 131
A.2 Time constant computation for a static CMOS register circuit 133
A.3 Precharged circuit that uses pure charge sharing 135
A.4 A driven charge sharing circuit 137

Chapter 1

Introduction

The term *very large scale integration* (VLSI) reflects the remarkable changes that the field of electronics has undergone over the past few decades. In the 1940's electronic circuit design was synonymous with relays and vacuum tubes. Some 10 years after its invention in 1947, the transistor had largely supplanted those two devices. Today, as a result of continued progress in semiconductor technology, a complex electronic system consisting of millions of transistors can be fabricated in a single micro-electronic chip. While VLSI technology is still evolving, the use of integrated circuits is now firmly established; from watches to personal computers, they have, in some way, influenced every facet of our society. This revolutionary change in the conceptualization and fabrication of electronic systems has obliterated traditional design methodologies; the days when a circuit was tested on a board and errors were corrected using wires and solder are long gone.

The explosion in complexity made possible by the advent of VLSI has made the use of computer tools essential to the design process. A variety of tools supporting all phases of the design process have been developed and are relied upon by virtually every designer of integrated circuits. Computer tools can not only cope with the masses of data that would overwhelm a human designer, but they can also verify the functionality and determine the performance of a design before it goes through an expensive and time consuming fabrication process. Once a circuit is fabricated, finding errors becomes extremely difficult since the problems associated with attaching probes to microscopic wires severely limit the number of signals available for examination or manipulation. But

verification is more than just a convenience for detecting errors, it allows designers to explore different alternatives and improvements in ways that may be otherwise impractical or impossible.

While higher levels of integration have led to improvements in cost, size, power and performance of integrated circuits, they have also imposed greater demands on the tools used to design them. More complex circuits also need more tests, hence the time required to design and verify them grows more than linearly with their size. As circuits become larger, the computational requirements quickly outstrip the effectiveness of the tools. Consequently, attempts to build larger VLSI circuits are being increasingly limited by our ability to verify them.

Much work has been done to improve the basic performance of the tools. Most approaches, however, consider design and verification as separate processes; thus failing to recognize the iterative nature of the design process whereby the same design is repeatedly modified and verified. As a result, most existing tools are batch-oriented: they read a complete description of the circuit and operate on the entire design. If a section of the circuit is changed, the entire design must be verified again, regardless of how small the change or how large the design. For large designs, this approach has resulted in intolerably long turnaround times.

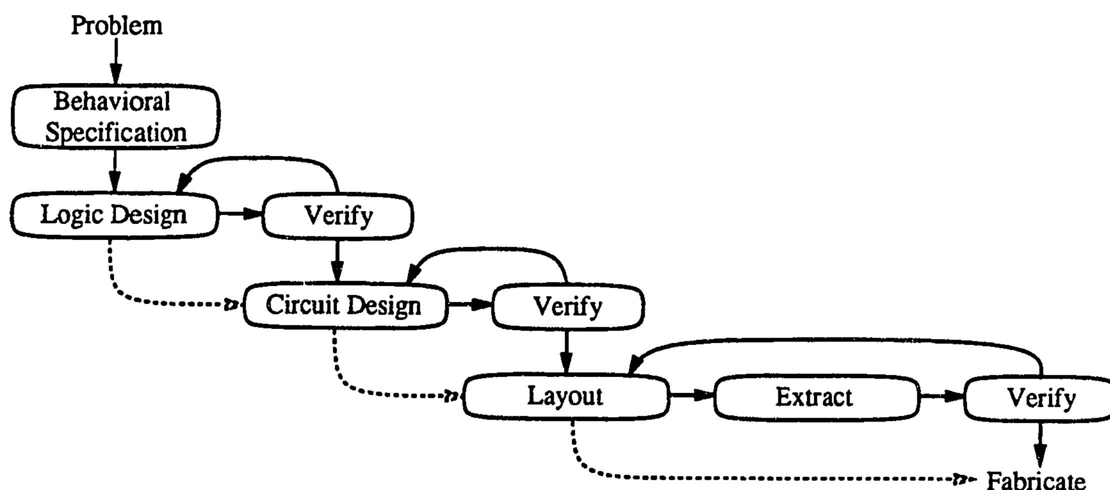


Figure 1.1: A simplified view of the VLSI design process

Ideally, a single forward pass through all the design steps would take a behavioral description into a layout suitable for fabrication (following the dashed arrows of Figure 1.1). But that is rarely the case; even with the use of modern design methodologies, a designer typically iterates many times between designing a section of the circuit and verifying that its implementation is correct.

Since at each step, the designer has incomplete knowledge of the design problem, she must estimate some important characteristics of the underlying sections. Once those sections are completed, the design can be tested to ensure that the estimates were correct. If the estimates were wrong, the process must be repeated, this time using the information gained from the previous design. Thus, design is intrinsically an iterative process consisting of what might be called *design cycles.* Since removing the cycles would require designers to have perfect foresight, there does not seem to be any way to completely remove the cycles. Nonetheless, the time spent in these cycles can be greatly reduced by building tools capable of working together in short design loops. A key observation is that designer productivity is determined by the time spent in these cycles; not the time spent in any one step.

The tools described in this thesis work together to operate *incrementally*. Incremental tools can reduce the time required to verify a modified design by confining their work to the sections of the design affected by the changes. This allows verification to be performed in a time proportional to the size of the changes rather than the size of the entire design. Since many changes are small, they have minor implications on the overall design. Therefore, by confining their work to the sections affected by these changes, incremental tools can compute the effect of the changes very quickly.

An incremental tool can confine its work to the affected sections of a modified design by tracking the designer's changes and reusing the information already gathered during previous cycles. Unaffected sections need not be recomputed since that information is already available. By making effective use of prior information an incremental tool can reduce the amount of redundant work, thereby allowing the designer to explore the design space yet receive quick feedback on the quality of the design.

1.1 Incremental System Overview

The incremental system presented in this thesis is shown in Figure 1.2. The system consists of three stages, corresponding to the three steps of the last design cycle shown in Figure 1.1: layout, circuit extraction, and verification.

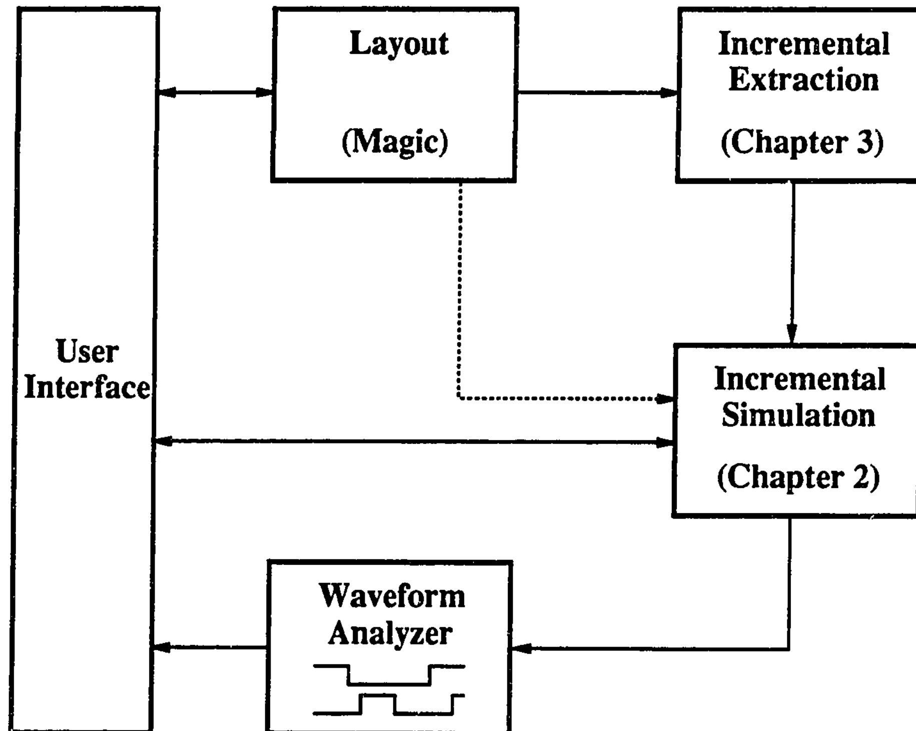


Figure 1.2: Incremental system overview

Layout is performed using the Magic layout editor[35]. Magic has been instrumented to perform incremental circuit extraction by producing a circuit level description of the modifications. Verification is performed using *Irsim*[40], an incremental version of the switch-level simulator *Rsim*[49]. Feedback to the user is in the form of a graphic waveform analyzer that can display the state of any node in the circuit for any time frame.

Much as incremental techniques are applicable to a variety of verification tools, the focus of this thesis is on simulation since it is an effective and, by and large, the most widely used verification method.

There are several reasons for working at the layout level. First, it is at this level that the circuit is almost complete, the design comes to a crunch, and the need for fast response is most keenly felt. Second, many of the more subtle errors arising from the connection of several blocks are discovered at this level. Third, actual circuit behavior can only be correctly estimated until transistor sizes and the unavoidable interconnect parasitics are extracted from the layout. Finally, Magic is the most commonly used tool for creating circuits in the Stanford design environment, thus providing a large pool of test circuits for which no other description is available.

1.2 Organization

The next chapter explores incremental simulation techniques. After surveying previous work on incremental simulation, we describe the implementation of an incremental logic plus timing simulator based on *Rsim's* switched-resistor MOSFET model. In addition, we analyze the efficiency of the simulator and evaluate its worst-case performance.

Chapter Three investigates techniques to incrementally extract the circuit modifications from the changes made to the circuit's mask-level description or layout. After discussing the approaches adopted by other researchers, we describe our implementation of a circuit extractor capable of updating the circuit description in time proportional to the size of the modifications.

Chapter Four demonstrates the applicability of incremental methods in a real design situation. We analyze the performance of the incremental tools when used to fix and simulate the various design flaws that occurred during the design of a VLSI microprocessor. Additionally, the efficiency of the incremental tools is compared with that of the conventional (batch) tools.

Finally, Chapter Five presents a synopsis of our experience with incremental techniques, summarizes the contributions of this thesis, and discusses areas for further research.

Chapter 2

Incremental Simulation

Simulators are tools used by designers to verify the functionality and performance of a design before it is fabricated. To use a simulator, the designer provides a *netlist* describing the design in terms of components and nodes; a node being essentially a wire used to connect components with one another. The designer then specifies input vectors, a series of voltages or logic levels to be applied at certain nodes, and calls on the simulator to evaluate the network and predict the voltages or logic levels of other nodes in the circuit. This process continues until the designer is satisfied that the circuit meets its required specifications. If it does not, the designer then modifies the circuit and simulates it again. Using a conventional simulator, with each set of changes, the designer must create a new netlist and repeat the whole process over the entire circuit. An incremental simulator, on the other hand, allows the designer to modify the circuit being simulated and then to incrementally update the simulation results of the parts of the circuit whose behavior is altered by the modifications. To accomplish this, the simulator saves a history of circuit behavior during the initial simulation, and then uses this history to confine subsequent simulations to the parts of the circuit affected by the modifications. The objective is then to minimize the number of evaluations needed to bring the modified circuit into a state consistent with the input vectors. Figure 2.1 illustrates two ways to confine the simulation of a modified circuit in order to reduce this number.

First, by tracing out all nodes whose behavior depends on a particular modification, either directly or through other components, the circuit can be topologically partitioned

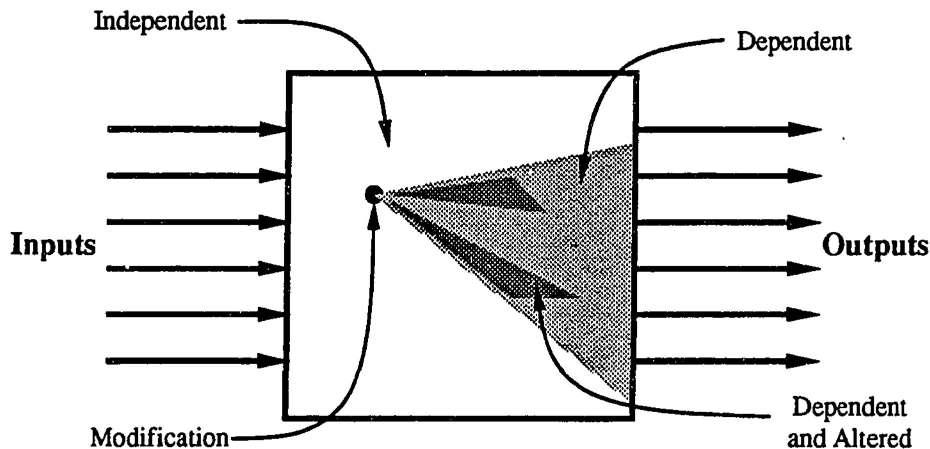


Figure 2.1: Confining simulation to a portion of a modified circuit

into regions whose behavior depends on the modification. Since nodes outside these regions are, by definition, independent of the modification, their behavior will not be altered and need not be simulated. By simply using the history of these outer nodes as input vectors for the dependent regions, a conventional simulation algorithm can confine its work to the dependent regions. As long as these regions are smaller than the overall circuit, the number of evaluations will also be smaller. Simulators that exploit this property are said to be incremental in space.

Second, at any particular time during simulation, the dependent regions can themselves be decomposed into regions with nodes whose behavior is actually altered by the modifications. Unlike a dependent region, however, the size of an altered region changes during simulation: it grows when nodes diverge from their previous behavior and it shrinks when they converge. By confining the simulation to the nodes that deviate from previous behavior, the number of evaluations can be further reduced. Simulators that exploit this property are said to be incremental in time.

This chapter describes Irsim, an incremental-in-time, logic-level, timing simulator for MOS circuits. The next section provides some background and discusses other approaches to reduce simulation time. Following this is an overview of Rsim, the simulator on which Irsim is based, including some of the improvements that have been incorporated into the original algorithm. Next is a description of Irsim's novel incremental algorithm, its

history management mechanism, and its implementation. The last section evaluates the incremental simulator's performance.

2.1 Background And Related Work

The predictions made by a simulator are based on models that describe the behavior of the individual components. Simulators can be classified according to the type of models they use. The three most common are:

- **Analytical** models based on the physics of the actual devices. The models typically consist of differential equations relating the current through the devices to their terminal voltages. To determine the new nodal voltages, numerical methods are used to solve a set of coupled nonlinear differential equations.

- **Functional** models based solely on the intended operation of the components. The models typically consist of boolean equations that relate a component's outputs to its inputs. The new values are found by simply evaluating these equations.

- **Hybrid** models that attempt to approximate the accuracy of analytical models while retaining the simplicity and efficiency of functional models.

Analytical models are used by circuit analysis programs such as *SPICE*[33]. This type of simulator is characterized by a high of degree accuracy. Obtaining this accuracy, however, requires the evaluation of every device model and solving a large set of equations at every time step. The computational requirements make it unrealistic to use such simulators on a large VLSI circuit.

Functional models are used to improve on the performance of circuit simulators by abstracting a circuit's logical function from its electrical behavior: components become logical blocks and voltages become logical levels. This type of model is found in gate-level simulators such as the *Yorktown Simulation Engine*[36], and in functional simulators such as *TEGAS*[46]. The use of logical values and blocks enables these simulators to exploit circuit latency: since the outputs of a block depend only on its inputs, a model need only be evaluated when its inputs change. This ability to *selectively trace* components can

greatly reduce simulation time. Although not all functional simulators are selective trace (for example *SSIM*[52] is a compiled-code simulator), most simulators exploit this cause and effect relationship through some sort of event scheduling mechanism. Simulators that use functional models can simulate large circuits in reasonable amounts of time, but for several reasons are unable to realistically model MOS circuits. First, MOS transistors are essentially bidirectional devices: signals may flow in either direction even if the designer intends a particular direction. This might lead to sneak paths or *charge sharing* effects, both of which need to be modeled. Also, functional models do not model charge storage at nodes; this precludes their use in circuits that rely on dynamic storage such as dynamic memories and precharged logic. Finally, functional models ignore the values of capacitances and transistor conductances, yet the state of MOS circuits depends strongly on these values.

Hybrid models attempt to bridge the gap between analytical and functional models by having as components the physical devices, but using simplified models that enable the simulator to incorporate many of the advantages exhibited by functional simulators. This type of model is typified by switch-level simulators such as *MOSSIM II*[6], *COSMOS*[37] and *Rsim*[49]. The basic idea behind the switch-level model is to simulate transistors using a simplified resistive-switch model. This model abstracts many of the nonessential details of the transistor while retaining most of its functionality. By adjusting the complexity of the resistive-switch model, a trade-off can be made by sacrificing accuracy for simulation speed.

Almost all previous work attempts to reduce simulation time by optimizing or accelerating the basic simulation function. However, there has been some prior work on incremental simulation. Hwang, Choi and Blank have implemented two incremental simulators based on the behavioral simulator *THOR*[2]: an incremental-in-space simulator[23, 24] and an incremental-in-time simulator[7]. Their incremental simulators operate on a flat netlist consisting of nodes and behavioral, functional, or gate models (the models are compiled C procedures). Users must manually enter the *net change tokens* describing the modifications to the flat netlist, supporting operations such as insertion and deletion of components, and connection and disconnection of nodes. For the incremental-in-space

algorithm, they report simulation speedups ranging from 2 to 30. This incremental algorithm exhibits little overhead: its worst-case performance, which occurs when most of the circuit is resimulated, is comparable to that of the non-incremental algorithm. For the incremental-in-time algorithm, they report simulation speedups ranging from 1 to 200. This algorithm exhibits much higher overhead than the first; some test cases show an increase in simulation time of three times or more. They attribute this increase to the overhead due to maintaining the history, and to the evaluation of zero-delay elements, which sometimes require more evaluations than in the conventional algorithm.

In his thesis, Choi[8] compares the two simulators described above, and introduces a third, hybrid algorithm that combines those two. The hybrid algorithm starts out as an incremental-in-time algorithm, resimulating only nodes that deviate from their previous behavior. However, once a node deviates from its previous behavior, it becomes part of an altered region that never shrinks. The algorithm thus resembles the incremental-in-space algorithm, the only difference is that the regions to be resimulated can grow during simulation. For the hybrid algorithm, Choi reports simulation time speedups that typically range between those of the other two: somewhat faster than the incremental-in-space algorithm, and slower than the incremental-in-time algorithm. In the worst case, however, the hybrid algorithm is never as slow as the incremental-in-time algorithm.

The incremental simulators described above require that the simulator be recompiled following the modifications. Unfortunately, this also requires that the history be saved on disk and then read in prior to any resimulation. The time required to compile the simulator, write, and read the history may negate any performance gains possible with the incremental simulator. Also, while behavioral models can simulate large circuits in reasonable amounts of time, as discussed earlier, they are often inadequate for modeling MOS circuits. Finally, the simulator does not compute transition times automatically; instead, the user must supply the delays, which is prone to error or wishful thinking on the part of the user.

In his thesis, Jones[28] proposed an incremental switch-level simulator embedded within a schematic capture system. The entire system is based on attribute grammar techniques[30] developed in the context of language-based programming environments[27]. Nodes are distributed throughout the hierarchy of the design, and the circuit

representation is updated as a byproduct of manipulating the design. All the circuit properties are represented by attributes of the design tree and are incrementally updated. Jones extended this principle to switch-level simulation by defining attributes for a node's logical state and strength. The simulator is based on the *MOSSIM II*[6] model, and uses a simple unit-delay timing mechanism. Because it operates on the design hierarchy, the system does not require a netlist compilation (or flattening) step. However, Jones focused only on updating the topology of the design and the state of the circuit for a fixed input stimulus; not for a series of inputs. He expects the amount of redundancy present in his representation to seriously degrade performance if the algorithm is extended to resimulate a series of inputs.

More recently, Jones[25] has implemented an incremental system involving a simulator that operates on a flattened netlist. The simulator is embedded within a schematic capture system that includes an incremental netlist compiler[26]. The netlist compiler works by propagating every change a user makes on the schematic to the flat representation. The simulator is also based on the *MOSSIM II* model, but allows designers to specify a delay for each node in the circuit. The simulator uses an incremental-in-time algorithm very similar to the one presented in this thesis, however, it can only handle identical rise/fall times and propagational delays, a special case of the more general algorithm presented in this thesis. Jones reports simulation time speedups that range from 0.33 to 864.

The approach we adopted is based on switch-level simulation. We did not consider an incremental-in-space algorithm since deriving such a model for a switch-level simulator was deemed impractical due to the bidirectionality of transistors, which, unlike a functional model, make it impossible to determine signal flow from the topology of the circuit alone; instead, all possible signal paths stemming from a modification must be included in the dependent regions. This would result in the resimulation of unnecessarily large portions of the circuit.

We use Rsim as the basis for our incremental-in-time simulator. Rsim is a logic-level timing simulator known for yielding reasonably accurate results without the long computation runs of circuit analyzers. There are several reasons for choosing Rsim. Since it is the most widely used simulator in the Stanford design environment, it provides us

with a large pool of real circuits and input vectors with which to test our simulator. Also, Rsim correctly models some the more subtle MOS timing effects such as charge sharing and input slope dependency. This timing accuracy may be a disadvantage with regard to incremental simulation, since even slight changes to the circuit may cause many signals to experience a time shift, hence requiring resimulation of large portions of the circuit although neither their logic levels nor their relative timing has changed. Nevertheless, since designers sometimes modify a circuit only to improve its performance, the simulation should reflect those changes. Moreover, small timing differences may sometimes cause a circuit to malfunction. In those cases, the more accurate results should more than compensate for the loss in performance.

Before presenting our incremental-in-time algorithm, the next section reviews Rsim's simulation algorithm.

2.2 Overview of Rsim

Rsim is a logic-level timing simulator based on a switch-level model that was developed by Terman with the goal of being able to simulate an entire VLSI circuit with acceptable accuracy. A detailed description of Rsim's underlying principles can be found in his thesis[49]. The version described here includes several improvements to his original algorithm: Horowitz[22] has modified it to account for the effects of input slope and resistor-capacitor trees, and Chu[11] has included improved charge-sharing and DC-analysis models.

Rsim represents a circuit as a set of nodes interconnected by transistors. Nodes are treated as wires with capacitance only to ground; the voltage across the capacitor represents the logical state of the node. To change this state, the capacitor must be charged or discharged. Since the charge across the capacitor cannot change instantaneously, a change in state involves a delay. The main task of the simulator is to detect transitions between logic states and their associated delays. The logic state of a node is determined by first estimating its voltage[1] and then quantizing it into one of three logic levels by comparing it to a pair of voltage thresholds appropriate to the technology, V_{low} and V_{high}:

[1]All voltages are normalized in the range [0,1], hence $V_{dd} = 1$ and $V_{gnd} = 0$.

0	logic low	voltages in the range $[0, V_{low}]$.
1	logic high	voltages in the range $[V_{high}, 1]$.
X	undetermined	unknown voltages or in the range $[V_{high}, V_{low}]$.

These logic levels correspond to the types of voltages one might expect from digital logic circuits; they not only provide an adequate level of abstraction but also simplify the computations by confining MOS transistors to operate in one of two states: fully conducting (ON) or not conducting (OFF). Accordingly, Rsim models a transistor as a bidirectional switch in series with a resistor; turning the transistor on "closes" the switch and connects the drain and source nodes via a resistor. The logic level at the transistor's gate controls the state of the switch (Figure 2.2). The value of the resistance, R_{eff}, is determined separately for each transistor, and is a function of the transistor type and its dimensions.

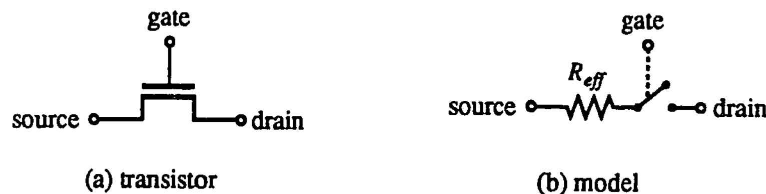


Figure 2.2: Rsim model for an n-channel transistor

The type of connection between the source and drain of a transistor as a function of its gate level can be easily tabulated for a variety of transistor types (Table 2.1). To account for transistor nonlinearity, the effective resistance, R_{eff}, of a transistor is actually modeled by two resistance values: R_{high} and R_{low}; the value used depends on whether the transistor is driving a particular node high or low. The uncertainty regarding the state of the switch due to an X level at its gate is modeled as an interval between its fully conducting and non-conducting states. All the network calculations use interval arithmetic; the bounds of the resulting intervals are used to convert voltages into logic levels. This mechanism is sufficient to deal effectively with X levels.

The electrical isolation provided by OFF transistors and the unidirectional coupling of the gate are used to partition the network into channel-connected *stages*, which are

Gate Level	State	Resistance
0	OFF	∞
1	ON	R_{eff}
X	ON/OFF	$[R_{eff}, \infty]$

(a) n-channel

Gate Level	State	Resistance
0	ON	R_{eff}
1	OFF	∞
X	ON/OFF	$[R_{eff}, \infty]$

(b) p-channel

Table 2.1: Transistor source-drain resistance as a function of gate level.

subcircuits that include all nodes connected together by conducting (and possibly conducting) transistors. A stage can be viewed as a functional model whose *inputs* are the nodes connected to the gates of its transistors, and whose *outputs* are the nodes contained within the stage. A stage is the basic unit of analysis, and is analogous to a functional model in that its outputs are isolated from its inputs, and a single stage determines the state of an output. Unlike a functional model, however, the composition of a stage is determined dynamically as transistors change state during simulation. Figure 2.3 illustrates this decomposition process.

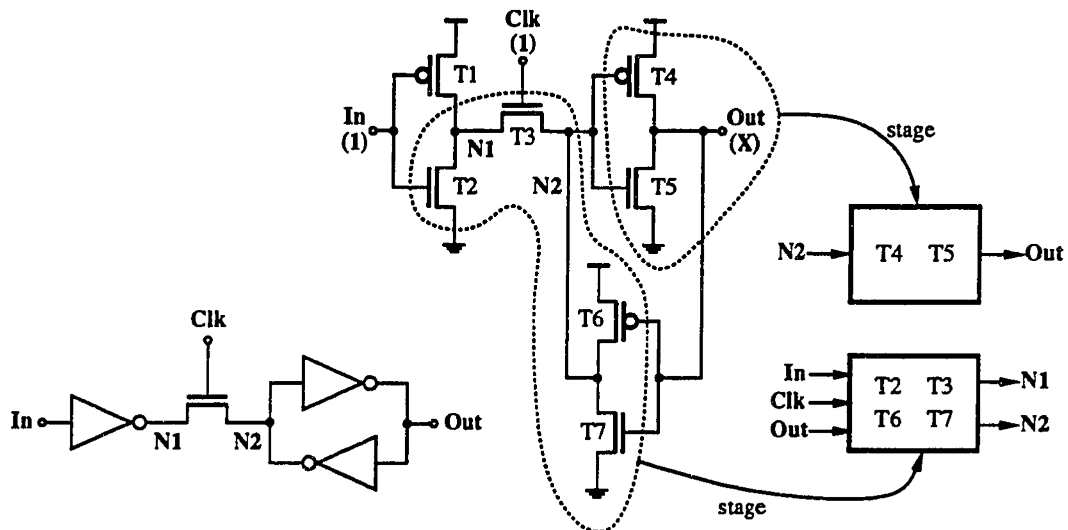


Figure 2.3: Decomposition of a static CMOS register into stages

Whenever a stage's inputs change state, causing nodes or transistors to be added or removed from it, the stage is evaluated by recomputing the state of all its output nodes.

Stages whose inputs are quiescent are not evaluated – the simulation algorithm is *event-driven.* Whenever a node is to change state, a new event is created and inserted into an event queue, which holds a list of events, sorted by time, indicating the amount of processing remaining to be done. Events specify a node, a new logic level, a transition time constant, and the time at which the node's state should change to the new level. Events are classified as *evaluation events*, which are generated when the evaluation of a stage causes an output node to change state, and *input events*, which are generated by the user providing input stimuli for the circuit.

2.2.1 Event Processing

Simulation begins by the user providing input stimuli for the circuit. For each of these stimuli, the simulator enters an input-event into the event queue; when the queue is empty, the network has "settled" and the simulator waits for further input. The simulator sequentially processes events from the event queue, advancing the current simulated time to the earliest time at which an event is scheduled, and stopping when either the queue is empty or a specified simulation time has elapsed. The basic event processing algorithm is:

1. Remove all events scheduled at the current time from the event queue.
2. For each event, change the state of the specified node to its new state.
3. Update the state of all transistors with gates connected to the nodes changing state. At the source and drain of each such transistor, trace the stage(s) formed and mark them for evaluation.
4. Evaluate each of the stages marked above, computing for each output node: its final value, its charge sharing value, and its delay should the node change value.
5. If the new value differs from the node's present state, or from the one it will assume due to a previously scheduled event, then schedule an event indicating that the node is to change state at the current time plus the delay computed.

Stages whose inputs are quiescent are not evaluated – the simulation algorithm is *event-driven.* Whenever a node is to change state, a new event is created and inserted into an event queue, which holds a list of events, sorted by time, indicating the amount of processing remaining to be done. Events specify a node, a new logic level, a transition time constant, and the time at which the node's state should change to the new level. Events are classified as *evaluation events*, which are generated when the evaluation of a stage causes an output node to change state, and *input events*, which are generated by the user providing input stimuli for the circuit.

2.2.1 Event Processing

Simulation begins by the user providing input stimuli for the circuit. For each of these stimuli, the simulator enters an input-event into the event queue; when the queue is empty, the network has "settled" and the simulator waits for further input. The simulator sequentially processes events from the event queue, advancing the current simulated time to the earliest time at which an event is scheduled, and stopping when either the queue is empty or a specified simulation time has elapsed. The basic event processing algorithm is:

1. Remove all events scheduled at the current time from the event queue.

2. For each event, change the state of the specified node to its new state.

3. Update the state of all transistors with gates connected to the nodes changing state. At the source and drain of each such transistor, trace the stage(s) formed and mark them for evaluation.

4. Evaluate each of the stages marked above, computing for each output node: its final value, its charge sharing value, and its delay should the node change value.

5. If the new value differs from the node's present state, or from the one it will assume due to a previously scheduled event, then schedule an event indicating that the node is to change state at the current time plus the delay computed.

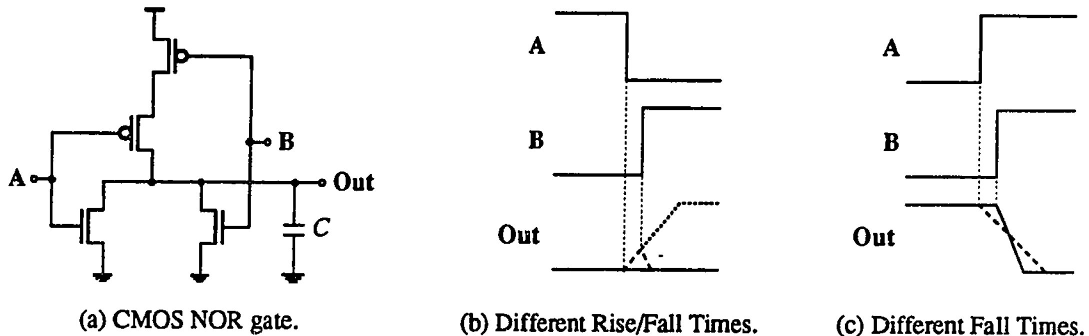
Since events are scheduled into the future, processing always proceeds monotonically forward in time, an important characteristic of this algorithm. The above discussion glossed over the many details involved in evaluating a stage. A more detailed description of these computations can be found in Appendix A.

2.2.2 Event Management

Scheduling an event entails creating a new event and inserting it, according to its scheduled time, into the event queue. To facilitate this operation, the queue is organized as a time wheel[5], a circular array that bucketizes events according to time. In addition, simulation time is quantized by rounding all times to the nearest $\frac{1}{10}ns$. Except for input stimuli, no signal can change instantaneously; if the quantization results in a delay of zero, Rsim uses a delay of one quanta instead. This not only produces more realistic results, but also allows detection of oscillating signals without any additional mechanisms.

If a stage's inputs change while its outputs have pending events, reevaluating the stage may lead to different output values. There are several ways to deal with this situation. In Rsim, the estimated transition delays are accurate enough to distinguish between two cases. If the new values are to take effect after the pending events, they are simply added to the queue. Conversely, if the new values are to take effect before the pending events, the simulator will abort the pending events – remove them from the queue – and, if needed, schedule the new values. The use of inertial delays with preemption is based on the principle that the most recently calculated event best reflects the current state of the circuit.

The NOR gate of Figure 2.4 shows two situations in which events are aborted. To simplify the discussion, all transistors are assumed to have the same resistance R. In Figure 2.4b, when input A falls, node **Out** begins to rise with time constant $2RC$. Before it reaches its final value, however, input **B** rises, causing **Out** to fall, this time with time constant RC. In Rsim, when **A** falls, an event is scheduled for **Out** to transition to high. Before this event takes effect, **B** rises, the stage is reevaluated, and the simulator determines that **Out** should become low before the previously scheduled transition is to take effect. The low→high transition is then aborted; since the new and current values



(a) CMOS NOR gate. (b) Different Rise/Fall Times. (c) Different Fall Times.

Figure 2.4: Effects of aborted events on a CMOS NOR gate

of **Out** are the same, no event is scheduled and the output remains low, without ever changing.

In Figure 2.4c, when input **A** rises, node **Out** begins to fall with time constant RC. Before it reaches its final value, however, input **B** rises, causing **Out** to fall with time constant $\frac{RC}{2}$. The situation in Rsim is similar to the one of Figure 2.4b above. When **B** rises, the simulator determines that **Out** should transition to low sooner than predicted before. Accordingly, the first event is aborted and a new event, which reflects the output's faster fall time, is scheduled in its place.

2.2.3 Summary

The Rsim simulation algorithm can be summarized as follows:

- Voltages are quantized into three logic levels: 0, 1, and X. These levels allow Rsim to model transistors as switches in series with a resistor. Two resistance values are used to predict node voltages and transition times.

- The circuit is partitioned into stages consisting of channel-connected subcircuits. The isolation between stages allows each stage to be analyzed separately.

- State transitions propagate through the network as a series of events. Each event leads to the analysis of the stages that may be affected by the transition. Events are sequentially processed, advancing monotonically forward in time.

2.3 Incremental Switch-Level Timing Simulation

The previous section reviewed the operation of the switch-level timing simulator Rsim, hereafter referred to as the conventional simulator. This section describes Irsim: an incremental-in-time simulator based on Rsim's circuit models.

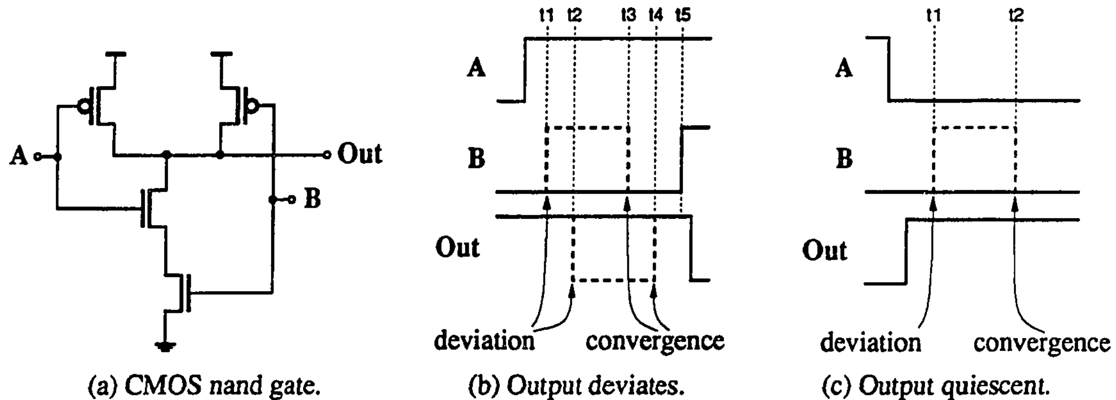
In an incremental simulator, in addition to providing input stimuli for the circuit, users can modify the circuit being simulated and incrementally update the simulation results of the portions of the circuit whose behavior is altered by the modifications. To use an incremental simulator, the circuit must have been simulated at least once. During this simulation, the simulator will produce a history of circuit activity, which is a record of every transition that took place. Subsequent incremental simulations use this history to resimulate only the portions of the circuit whose behavior deviates from their history. The basic concept is straightforward: given a history of past circuit behavior and a set of modifications to the circuit, simulate only those portions of the circuit whose behavior is altered by the modifications.

In an incremental-in-space simulator, all nodes that could possibly be affected by a modification are resimulated over the entire history. In contrast, an incremental-in-time simulator, such as Irsim, only resimulates those nodes whose behavior differs from that recorded in their history, and only for as long as their behavior remains different. This incremental simulation process is illustrated in the next section.

2.3.1 An Incremental Simulation Example

The operation of the incremental simulator is best illustrated through an example. Consider the nand gate of Figure 2.5. Shown in Figure 2.5b are the waveforms produced during a previous simulation (solid lines) and the ones produced by the current simulation (dashed lines) as a result of having modified some other part of the circuit.

Prior to time t1 all the signals are the same in both simulations so the gate need not be simulated. At time t1, input **B** deviates from its history by transitioning to high. This deviation causes the gate to be evaluated and, as a result, output **Out** also deviates from its history by transitioning to low at time t2. When **B** transitions to low at time t3, it converges with the value recorded in its history for that time. The gate is again



(a) CMOS nand gate. (b) Output deviates. (c) Output quiescent.

Figure 2.5: Incremental simulation of a simple gate

reevaluated, which results in **Out** transitioning to high at time t4, a time at which it converges with its history. At this time, all the signals have once again converged to their previous states and the gate need no longer be simulated: when **B** rises at time t5, **Out** will transition to low, but that transition does not have to be resimulated, since it is already recorded in the history and will not be affected now that the entire state of the circuit is the same as in the previous simulation. The key observation is that the nand gate only needs to be resimulated from t1 to t3, the time period during which its inputs differ from the history, not before and not after. Similarly, other gates with inputs connected to **Out** will be resimulated from t2 to t4.

Figure 2.5c shows a similar situation in which input **B** also deviates from its history from t1 to t2. In this case, however, node **Out** remains low, never deviating from its history. Although the nand gate is reevaluated twice, as before, no other part of the circuit needs to be reevaluated.

This example is considerably simplified in order to give an idea of the operation of the algorithm without becoming mired in detail. The following sections extend these ideas to the Rsim model and describe the algorithm in more detail.

2.3.2 Algorithm Description

The incremental algorithm consists of two steps: *Network Modification* and *Resimulation*. During the first step, the modifications are applied to the network, and all nodes directly

affected by a modification are identified. During the second step, the simulator updates the histories of nodes whose behavior is altered by the modifications.

Network Modification

The first step in an incremental simulation is to process the circuit modifications applied by the designer. Irsim incorporates facilities to add, delete, move, connect, and disconnect transistors or wires; as well as facilities to change the electrical parameters of the circuit, such as transistor sizes, node capacitances, and voltage thresholds. A complete list of the design modifications accepted by Irsim can be found in Appendix B.

During this step, the simulator updates the underlying netlist and produces a list of modified nodes. If the changes made at a particular node have the potential to change its timing or logical behavior, the node is marked *changed* and added to the list of modified nodes. Since these changed nodes have been subjected to changes that may alter their behavior, they and their corresponding stages will be resimulated over the entire history.

Not all nodes involved in a modification are marked changed. Many of the modifications are simply used to maintain the simulated network consistent with its corresponding layout, such as moving a transistor to another location, or changing the name of a node. Sometimes even structural changes will only modify the capacitance of a node; before marking these nodes as changed the simulator will check if the capacitance change is large enough to appreciably alter the timing of the node. Consider, for example, the circuit of Figure 2.6 in which a new connection is added to an existing node **N1**.

Although connecting the gate of a transistor to **N1** has effectively changed the structure of **N1**, its logic behavior will not be altered; only its timing, and then only if the capacitance of the node is appreciably changed. This simple static capacitance check is much more efficient than repeatedly evaluating node **N1** simply to determine that it does not deviate from its history. However, since the gate of the transistor is now connected to a different node, the simulator cannot predict what effect this change will have on nodes **N2** and **N3**, so these two nodes will be marked as changed.

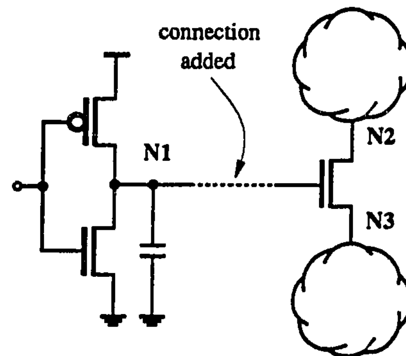


Figure 2.6: Changing the structure of a node may not alter its behavior

Resimulation

In a conventional event-driven simulator, simulation begins by providing input stimuli for the circuit; the effects of the stimuli are then propagated throughout the circuit by means of events. Although incremental simulation is initiated by modifying the circuit, a similar event-driven approach is used to propagate the effect of the modifications throughout the circuit. This enables the incremental algorithm to use the same circuit models and stage evaluation mechanism used by Rsim: an event queue is used to limit the number of stages that need to be examined at any time step, and processing always proceeds monotonically forward in time. Accordingly, the history is updated to reflect the new state of any node whose behavior is altered at time t before updating the history for any time after t. This allows the simulation to move consistently forward in time, without ever having to backtrack and revert the state of the network to any time prior to the current simulation time. Once the simulator reaches time t, the history of all nodes is up-to-date for all times prior to t, but continues to reflect the results of the previous simulation for times following t. When the simulator reaches the end of the history, it will have updated the entire history. At this point, the user may continue the simulation by providing additional input stimuli, or modify the circuit again.

Although the conventional and the incremental simulators are both event-driven, they are fundamentally different in how they handle events. In the conventional simulator an event requires the evaluation of all stages with inputs connected to the node changing

state, whereas in the incremental simulator an event requires the evaluation of only those stages whose state has deviated from its previous behavior. At the core of the incremental algorithm lies its ability to recognize which parts of the circuit need to be reevaluated. This is accomplished by extending Rsim's stage decomposition and selective trace mechanisms to account for the *incremental state* of a stage. During incremental simulation, a stage can be in one of two states: *active* or *inactive*. A stage becomes active if, at any specific time, either its composition or the state of any of its inputs differs from that of the previous simulation (Figure 2.7a). An active stage becomes inactive when its composition and the state of all its inputs and outputs converge to those of the previous simulation. Since the entire state of inactive stages is the same as in the previous simulation, their future behavior will also be the same; thus, only active stages need to be resimulated.

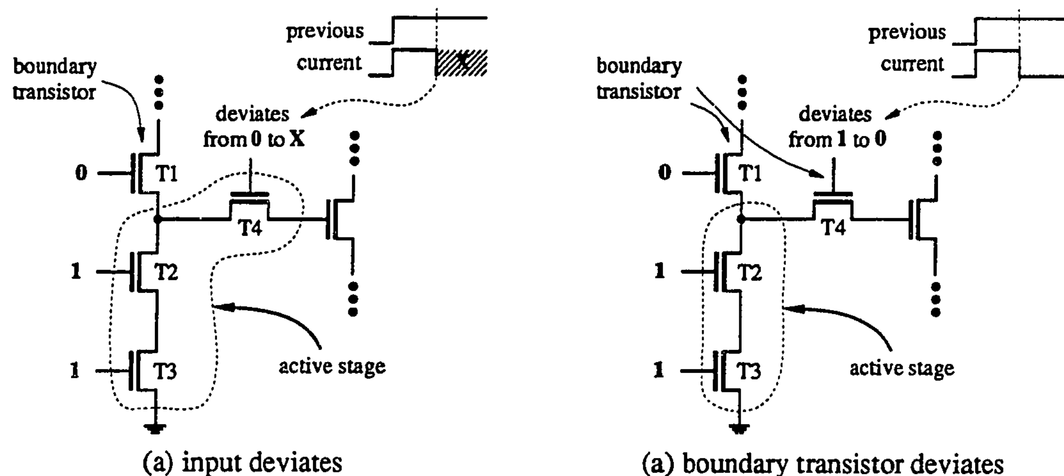


Figure 2.7: Two ways in which a stage becomes active

Although stages contain only conducting transistors, the incremental algorithm must also consider boundary transistors (off transistors at the boundary of a stage). For example, the boundary transistor, T1, shown in Figure 2.7a is not part of the stage and its state is the same as in the previous simulation, nonetheless, the simulator must track any changes at its gate node since the next transition could be the one that turns the transistor on, thus becoming part of the stage.

Even if the state of all the inputs and outputs of a stage is the same as in the previous

simulation, the stage can still become active by these boundary transistors altering its composition. This situation is shown in Figure 2.7b, where transistor T4 is turned off as a result of its gate deviating from its history. By becoming a boundary transistor, T4 has altered the composition of the stage, a stage in which all inputs and outputs remain the same as in the previous simulation.

Generalizing the above observations, a stage will be active for as long as any of the following *activation conditions* are met:

- The stage contains a changed node.
- An input has deviated from its history.
- The gate of a boundary transistor has deviated from its history.
- An output has deviated from its history.

Since the outputs of inactive stages never deviate from their history (they are not resimulated), the last condition above is different from the first three in that it can only occur once the stage is already active. This condition can not cause a stage to become active, it must, however, be checked before returning a stage to the inactive state. The reason for this condition is that a stage's outputs store charge; this charge might affect the behavior of the stage even after all its inputs have converged with their previous state. This situation is illustrated in Figure 2.8.

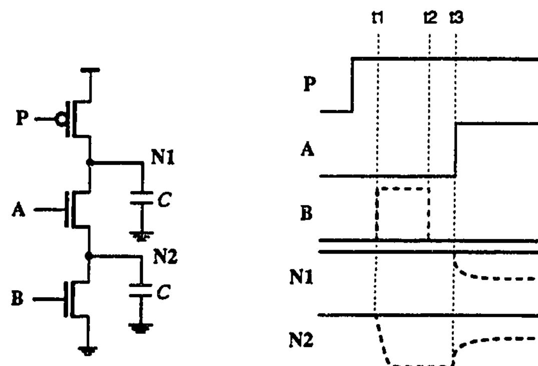


Figure 2.8: The behavior of a stage can be affected by its outputs

At time t1, input **B** deviates from its history; this causes **N2** to deviate from its history by transitioning to low. Then, at time t2, **B** transitions again and converges with its history. At this time, all the inputs to the circuit have converged to the values of the previous simulation, and only output **N2** differs from its history. When input **A** rises at time t3, nodes **N1** and **N2** will share their charge and deviate from their history. If the stage is prematurely deactivated at time t2, the simulator will fail to simulate this transition. This situation is avoided by requiring that a stage's outputs converge to their previous state before becoming inactive.

Resimulation begins at the changed nodes. As outlined in the previous section, stages that contain a changed node start out as active and remain so throughout the simulation. Each active stage is then simulated using the state transition history of its inputs to determine the new logical state of its outputs. Output nodes whose present state is different from the newly calculated one will have an event scheduled at the current time plus the delay computed for the transition. When these transitions become effective, the new states are compared against those recorded in the history. As long as the outputs of a stage do not deviate from their history, no new stages are activated. If an output does deviate from its history then every stage whose input is connected to that node becomes active. When a node converges with its history, the activation conditions of the stage containing the node are checked; if the conditions are not met, the stage becomes inactive and is no longer simulated.

Resimulation proceeds in this manner, sequentially processing events from the event queue, advancing the current simulated time to the earliest time at which an event is scheduled, and stopping when the end of the history is reached. A simplified view of the basic event processing algorithm is:

1. Remove all events scheduled at the current time from the event queue.

2. For each event, change the state of the specified node to its new value:

   (a) Compare the node's new value against the one recorded in its history.

   (b) **If** the node deviates from its history **then**
   Activate all stages connected to the node.

(c) **If** the stage containing the node converges to its previous state **then**
Deactivate the stage.

(d) **If** the event does not specify a transition **then**
skip steps 3, 4, and 5.

3. Update the state of all transistors with gates connected to the nodes changing state. At the source and drain of each such transistor, trace the stage(s) formed and if the stages are active mark them for evaluation.

4. Evaluate each of the stages marked above, computing for each output node: its final value, its charge sharing value, and its delay should the node change value.

5. If the new value differs from the node's present state, or from the one it will assume due to a previously scheduled event, then schedule an event indicating that the node is to change state at the current time plus the delay computed.

We can compare the incremental event processing algorithm to that of Rsim (Page 15). Steps 1, 2, 4, and 5 above are identical to the corresponding steps taken by Rsim. Step 2a performs the history comparison, and steps 2b and 2c deal with stage activation and deactivation, respectively. Step 2d is needed since not all events during incremental simulation imply a transition, this is explained below in Section 2.3.3. Step 3 is very similar to Rsim's, it differs in that the event is only propagated to active stages.

As the simulation progresses and nodes deviate from their history or converge with it, inactive stages may become active and vice versa. Also, as transistors turn on and off, the composition of active stages and boundary transistors changes during simulation. The main task of the simulator is to track all these changes – detect nodes converging or deviating from their history, activate and deactivate stages – while maintaining the history of all the nodes. The next section describes how an incremental simulator can track all these changes using new event types.

2.3.3 Events During Incremental Simulation

In the conventional algorithm, events are created in two ways: by the user providing input stimuli (input events) or by a transition at an output node due to the evaluation of its stage

(evaluation events). During incremental simulation there are two additional distinct types of events: *stimulus events* and *check-point events*. These events are different from the previous two in that they are not caused by circuit activity or user intervention; instead, they are extracted from the history and scheduled by the simulator to stimulate stages and detect transitions, respectively.

Stimulus Events

Once a stage becomes active, it must be evaluated for every transition occurring at its inputs or boundary transistors. In the conventional algorithm, events are always scheduled when a node changes state; these events signal the evaluation of stages whose inputs are connected to the node changing state. In incremental simulation, however, events are only scheduled on the outputs of active stages, and, in general, not all inputs to an active stage are themselves outputs of an active stage. Therefore, to signal a transition at these inactive inputs, a different type of event is used: a stimulus event. As long as a stage is active, all its inactive inputs (such as node **clk** in Figure 2.9) will have the next transition in their history scheduled as a stimulus event.

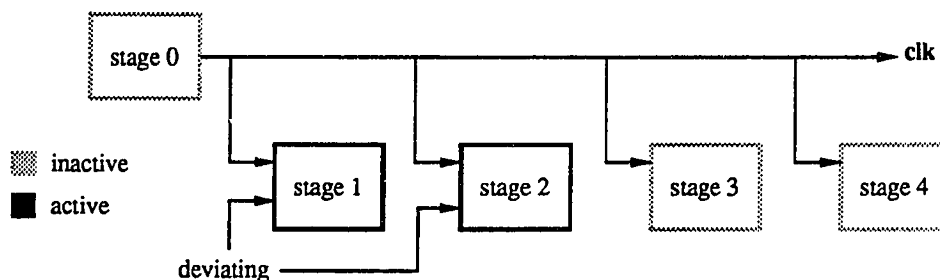


Figure 2.9: Stimulus events at inactive inputs are used to stimulate active stages

When a stimulus event takes effect, the value of the node it specifies is updated, all active stages for which the node is an input are evaluated, and the next transition in the node's history is scheduled as a stimulus event. Since there may be more than one active stage connected to the same input node (stages 1 and 2 in Figure 2.9, for example), stimulus events are not descheduled when a particular stage becomes inactive. Instead, the simulator waits until the event takes effect, and then checks if the node still needs to

be stimulated: if all stages with inputs connected to the stimulated node are inactive, the event is ignored, no stage is evaluated, and no more events are scheduled for the node. Although the simulator could keep track of the number of active stages connected to the stimulated node, and deschedule stimulus events when that number becomes zero, there are two reasons for not doing this. First, it is no more expensive to remove the event from the queue at the time it takes effect than to deschedule it. Second, another stage may become active and require the same stimulus event to be scheduled, in which case the event would have been descheduled only to be re-scheduled again; leaving the event in the queue avoids this extra work.

Check-Point Events

An important property of the incremental algorithm is that processing always proceeds monotonically forward in time. This monotonicity requires that a node deviating from its history be detected immediately as it occurs so stages that depend on the node can be activated and reevaluated for all further transitions. There are two ways in which a node can deviate from its history: by transitioning when the history shows a stable value (Figure 2.10a), or by remaining stable when the history shows a transition (Figure 2.10b).

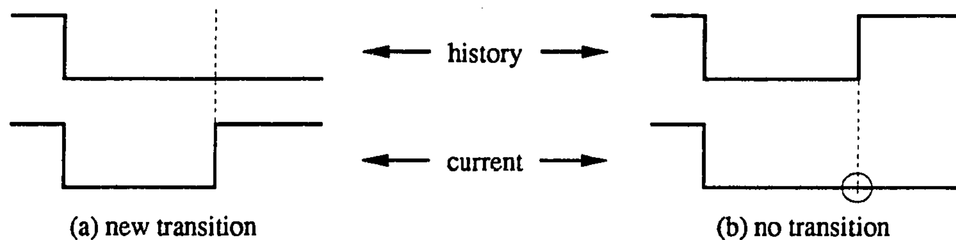


Figure 2.10: Two ways in which a node can deviate from its history

Detection of the first case is straightforward: since the transition must have been caused by the evaluation of an active stage, an event was scheduled to indicate the transition; the simulator can simply compare this event with the node's history. Detection of the second case is more difficult because there are no events scheduled for the node at that time. There are several ways to handle this situation.

The incremental-in-time behavioral simulator, THOR[7], repeatedly evaluates active

components and then compares the events that result from the evaluation with the corresponding transitions recorded in the history; detecting missing transitions is then trivial. This scheme is easy to implement in a simulator such as THOR[7], in which delays are fixed and events are never aborted. In Rsim, however, transition delays depend on the structure of the stage, the electrical parameters of its comprising elements, as well as on the input transitions. This makes it difficult to find a transition in the history that corresponds to the current evaluation. Furthermore, a transition could be aborted before it takes effect, thus invalidating any comparison made at the time the event was scheduled.

Our approach is to use a different type of event, a check-point event. This event indicates that an output was scheduled to change at a particular time during the previous simulation. When the check-point event is removed from the queue, the simulator can detect the missing transition by simply checking that no other event is scheduled for the node at the same time.

As long as a node is contained within an active stage, it will have the next transition in its history scheduled as a check-point event. When a check-point event is removed from the queue, the next transition in the node's history is scheduled as a check-point event. When a stage becomes inactive, the check-point events at its outputs are descheduled. Of course, check-point events are also used to detect when a node converges with its history, thus deactivating stages as soon as they converge to their previous state.

The main advantage of using check-point events is that they allow the use of an event-driven algorithm in which time increases monotonically. Their primary disadvantage is the overhead due to the scheduling of additional events; in the worst case, a node can have two events being continuously scheduled throughout the simulation: a check-point event and an evaluation event. Check-point events also complicate event processing since a node may have more than one event scheduled at the same time, a situation that never arises in the conventional algorithm.

2.3.4 Incremental States

Since the composition of stages varies during simulation, it is difficult to maintain their incremental state explicitly. Instead, as the network is decomposed into stages, their

incremental state is computed as a side effect of checking the activation conditions of their output nodes and boundary transistors. The simulator does maintain an incremental state for both nodes and transistors, each of which serves a different purpose.

A transistor's incremental state is active when transitions at its gate should not be ignored, that is, the transistor is either part of an active stage or a boundary transistor to one; otherwise the transistor's state is inactive. Whenever a stage is activated or deactivated, the incremental state of its transistors is updated. The incremental state of a transistor is used to quickly identify whether the transistor is part of an active stage, thus avoiding having to recompute the incremental state of all stages for which a particular node is an input. The incremental state of a transistor is examined in two situations: (1) to determine the stages to which a stimulus event should be applied (and resimulated), and (2) to determine whether stimulus events should be scheduled on the output nodes of a stage that becomes inactive; this occurs when the output node in question is an input to at least one active stage.

The incremental state of a node is examined to compute the incremental state of a stage, and also, as described later in Section 2.3.8, to decide which stages should be re-evaluated. A node can be in one of four incremental states:

- **Inactive:** The node is neither part of an active stage nor do any active stages depend on it. It is not being resimulated, and no events are scheduled for it.

- **Stimulated:** The node is inactive, but it is an input to one or more active stages. Although it is not being resimulated, the next transition in its history, if any, will be scheduled as a stimulus event.

- **Active:** The node is part of an active stage, but has not deviated from its history. It is being resimulated, and the next transition in its history will be scheduled as a check-point event.

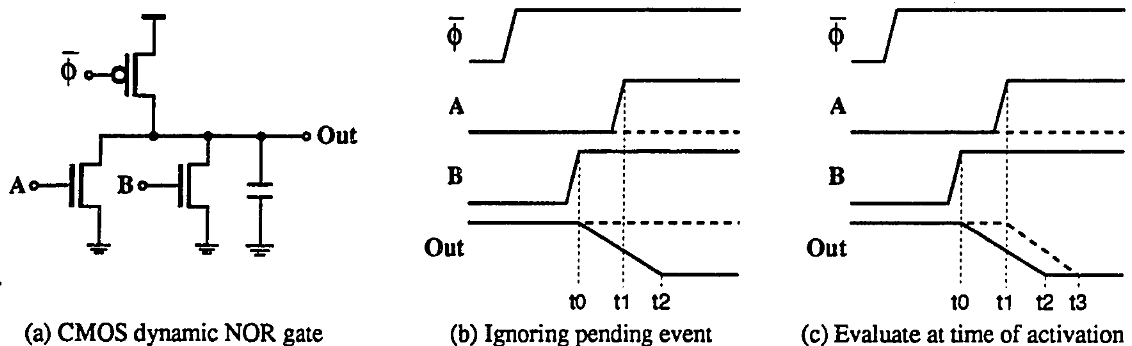
- **Deviated:** The node is active and it has deviated from its history. It is being resimulated and the next transition in its history will be scheduled as a check-point event.

2.3.5 Stage Activation

When a stage becomes active, its state must be initialized to that of the previous simulation at the time of activation. This entails extracting the state of its inputs and outputs from the history, updating the state of its transistors to reflect the current input values, and scheduling stimulus events on all its inactive inputs as well as check-point events on all its outputs. In addition, pending events must also be scheduled on its outputs; their need is explained below.

Pending Events

Transitions that were scheduled to occur in the future but were caused by an event prior to the time of activation must also be scheduled on the output nodes. Although these pending events are the result of an input transition while the stage was inactive, they are also part of the current state of the stage. Ignoring them can lead to incorrect results.



(a) CMOS dynamic NOR gate (b) Ignoring pending event (c) Evaluate at time of activation

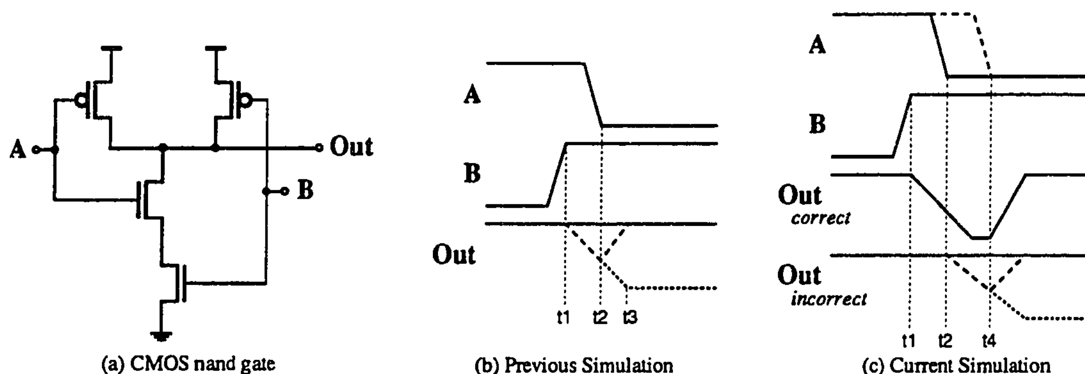
Figure 2.11: Ignoring pending-events may result in incorrect results

Consider what happens when node **A** in the example circuit of Figure 2.11 deviates from its history by remaining low at time t1 (dashed line in Figure 2.11b). At this time in the previous simulation, node **Out** had a pending transition to take effect at time t2, a transition that was caused by input **B** rising at time t0. Since in the current simulation none of the inputs has a transition at time t1, the stage will not be evaluated. If the pending transition is not rescheduled when the stage becomes active (at time t1), resimulation will incorrectly leave output **Out** with a stable high value. Note that the

correct output values could be obtained by always evaluating a stage at the time of activation, regardless of whether there are any input transitions; this, however, would not only make stage activation more expensive, but also lead to timing errors since in the absence of an input transition, any delay computed will be meaningless. In the example above, simply evaluating the stage at time t1 (Figure 2.11c) would result in a high→low transition for node **Out** to take effect at time t3 (dashed line), which is later than the actual transition.

Aborted Events

Aborted events are important because a previously aborted event may become an effective transition during incremental simulation. Omitting them can also lead to incorrect results. Consider what happens to the example circuit of Figure 2.12 when the transition of node A is delayed as shown in Figure 2.12c (dashed line).



(a) CMOS nand gate (b) Previous Simulation (c) Current Simulation

Figure 2.12: Incorrect resimulation due to aborted events

By remaining high at time t2, node A deviates from its history and causes the stage to be active from time t2 until time t4, when A converges to its previous simulation value. In the previous simulation, at time t2, node **Out** had a pending event; this event was aborted by the transition of A, which no longer occurs at time t2. With no input transitions at time t2, the stage will not be evaluated, thus leaving node **Out** with an incorrect high value, as in the previous example. In this case, however, even evaluating the stage at the time of activation produces an incorrect result: simulating the circuit

starting at time t2 will first predict a high→low transition on node **Out**, which will then be aborted when A falls at time t4. The resulting waveform would incorrectly show node **Out** with a constant high value. If we consider the event previously aborted at time t2, the situation is quite different, resulting in two transitions on node **Out**.

The key observation is that pending aborted events are also part of the current state of the stage. When a stage becomes active, all pending events at the time of activation, whether aborted or not, must be rescheduled on its outputs. Recreating possibly aborted events at the time of activation would require the ability to move back in time and repeatedly evaluate the stage. This would not only make stage activation quite expensive, but also violate the principle that time increases monotonically.

We have found that aborted events represent only 1 – 15% of all transitions; we therefore decided to record aborted events as part of a node's history. When the stage is activated, the simulator will scan the history of each output node looking for pending transitions; each such transition is then rescheduled as an evaluation event, just as if it had been scheduled by a previous stage evaluation. Any transition in the history that satisfies the following condition must be rescheduled:

$$\text{schedule time} < \text{activation time} < \begin{cases} \text{effective time} & \text{for an effective transition} \\ \text{aborted time} & \text{for an aborted transition} \end{cases}$$

Several transitions in the history can satisfy the above condition. This means that the simulator must scan the history to find all such transitions. To illustrate this, Figure 2.13 shows the 7 possible ways in which aborted (gray) and effective (black) transitions can occur during simulation.

In Figure 2.13, time increases from left to right. Each arrow points to the time at which the transition is to take effect, bullets represent the time at which the transitions are scheduled, and the dashed arrows indicate the time at which a transition is aborted. Note that Figures 2.13h–i represent situations that cannot occur, since the aborted transition would have been aborted when the effective transition was scheduled, as in Figure 2.13c. Depending on when a node becomes active, one or more of these transitions may be pending and will have to be rescheduled. For example, Figure 2.14 shows the four possible situations that may arise from Figure 2.13e.

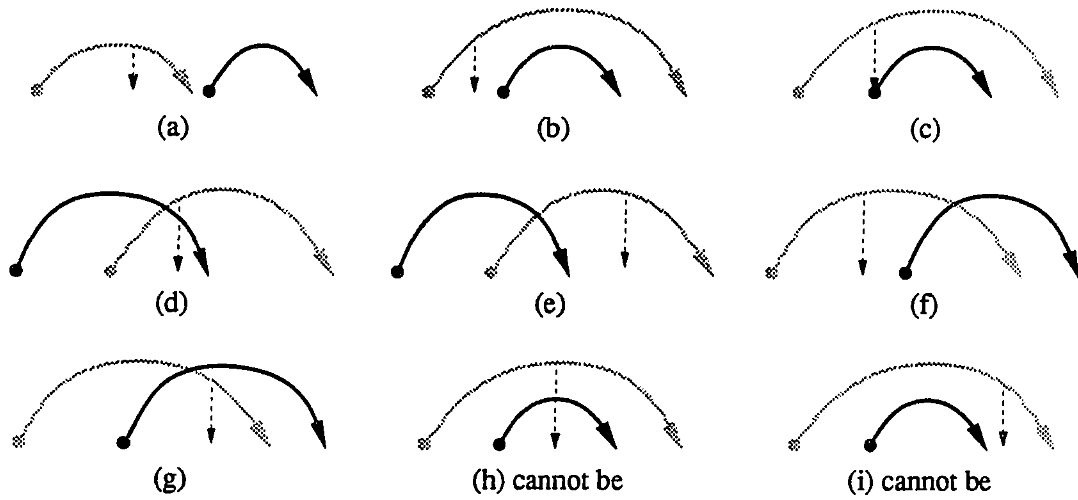
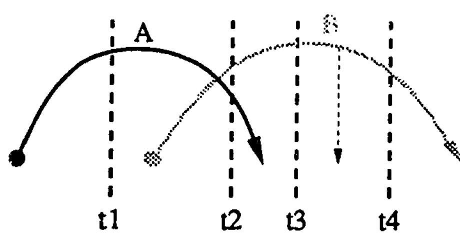


Figure 2.13: Possible timing of aborted and effective transitions during simulation



time of activation	transitions rescheduled
t1	A
t2	A and B
t3	B
t4	– none –

Figure 2.14: Several transitions may have to be rescheduled simultaneously

When an aborted transition is rescheduled, it is also removed from the history; if these transitions end up being aborted again, they will simply be re-recorded in the history. The implementation details are described later when discussing the history management mechanism in Section 2.3.7.

2.3.6 Stage Deactivation

In addition to the activation conditions outlined before, a stage must remain active until no pending events remain on any of its outputs. The reason for this is that some of those transitions may be aborted, and for that to happen the stage must remain active. Once all these conditions are met, the stage can be deactivated.

Deactivation of a stage is relatively simple; check-point events at its outputs are removed from the queue, and the incremental state of the nodes is updated. Often, an output node whose stage is deactivated drives a still-active stage. In this case the output node becomes stimulated and its next transition will be scheduled as a stimulus event. On the other hand, if the output node does not drive any active stages, then it becomes inactive (as described in Section 2.3.4).

2.3.7 History Management

The history of each node is kept as a linked list of transitions, which can be either effective or aborted. Each transition in the history carries enough information to be able to recreate the transition during resimulation. A history transition specifies: its target logic level, the time at which it was to take effect, its delay, its time constant, and whether it was an input stimuli. In addition, aborted transitions also specify the time at which they were aborted.

Effective transitions are recorded in the history at the time that the transitions take effect. Aborted transitions are recorded in the history at the time at which they are aborted. Since a transition can be aborted by a node remaining stable, an effective transition can be followed by an arbitrary number of aborted transitions. This simple history recording mechanism imposes a chronological ordering on the history; an effective transition is always followed by the next effective transition, or by a series of transitions that were

aborted after it took effect. Likewise, aborted transitions are always followed by a transition that was either aborted or took effect after they were aborted. When a stage becomes active and the history is scanned for pending events, the simulator need look no further than the first effective transition recorded after the time of activation.

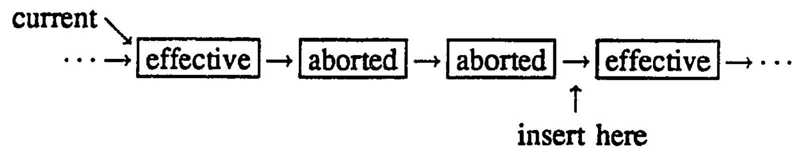
In addition to its history list, every node contains a *current pointer* that points to the history transition representing the current state of the node. Instead of maintaining the current pointer updated for all nodes in the circuit, the incremental algorithm only updates it for nodes whose state is needed during resimulation: for active and stimulated nodes it is kept updated as long as they remain in that state; for inactive nodes it is updated on demand. When the state of an inactive node needs to be updated, its pointer is advanced to the latest effective transition prior to the current simulation time; the logic level specified by the transition yields the current state of the node. As the simulation progresses and the history is updated, the current pointer of an active node will always point to the node's latest effective transition, never to an aborted transition. Since time advances monotonically forward, the pointer only needs to move in one direction, which is why a singly-linked list is an appropriate data structure.

When a node's behavior differs from that of the previous simulation, the simulator must modify its history. History modifications are performed by two basic operations: addition of a new transition and removal of a previous transition.

Adding a Transition

A new transition is inserted in the history when either a transition is aborted or a transition that did not occur in the previous simulation becomes an effective transition.

Transitions are always inserted in the history list before the next effective transition following the current pointer. Typically, they simply need to be inserted immediately following the current pointer. However, since there may be an arbitrary number of aborted transitions between any two effective transitions, inserting a transition may sometimes require a short scan of the history to skip past any aborted transitions.

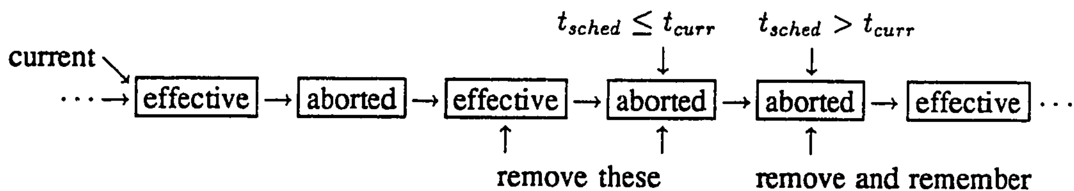


We have found that consecutive aborted transitions represent at most 15% of the aborted transitions, or about 0.1–2% of all transitions. Since this number is relatively small, a more sophisticated algorithm than the simple linear search we use is not warranted.

Since an aborted transition does not affect the state of the node (it only affects its history), the current pointer is never modified when inserting an aborted transition. Inserting an effective transition, on the other hand, will leave the current pointer pointing to the newly inserted transition that represents the current state of the node.

Removing a Transition

Whenever a transition that became effective during the previous simulation no longer occurs, that transition along with any aborted transitions that follow it are removed from the history list. Since missing transitions are detected as soon as they occur, they are always the next effective transition following the current pointer. The current state of the node remains the same when removing the next transition so the current pointer is not modified by this operation.



Aborted transitions removed from the history during this operation must be remembered by the simulator since the node may, at some later time, converge with the value of the deleted transition. If that happens, all remembered transitions that were scheduled after the time at which the node converges with its history are re-inserted in the history; remembered transitions that were scheduled before the time at which the node converged are freed and forgotten. This same process is used when a node becomes active: all

aborted transitions between the current pointer and the next effective transition that were scheduled after the time of activation are remembered. Remembered transitions whose scheduled time has expired are freed when either the node converges with its history or another transition is removed from the history.

2.3.8 Incremental Simulation Loop

Initially, all stages containing changed nodes are activated and stimulated using the transition history of their inputs. The simulator then enters its main loop and begins to sequentially process events. This processing may result in more stages becoming active or active stages becoming inactive. Resimulation ends when the simulator reaches the end of the history or there are no more events to process. At this point the user may again modify the circuit and resimulate it, or continue the simulation by providing additional input stimuli.

The particular actions taken by the simulator as it processes each event depend on the type of event and on the incremental state of the node it specifies. Processing evaluation and check-point events for active nodes can result in one of the eight situations shown in Figure 2.15.

As each evaluation and check-point event is processed, the state of the node they specify is updated and compared against its history. As a result of this comparison, the history is updated and the node is placed in one of eight *consideration lists*. Each list corresponds to one of the situations shown in Figure 2.15, as explained below. After all evaluation and check-point events have been thus processed, each consideration list is examined to determine which stages need to be evaluated, as follows:

(a) By remaining stable, the node deviates from its history; the node becomes active and deviated. All inactive stages with inputs connected to the node will be activated, but they will be marked for evaluation only if some other input undergoes a transition at the current time, a transition that can not be explicitly propagated through an event since the stages were inactive prior to the current time.

(b) By remaining stable, the node converges with its history; if the node controls any active transistors then it becomes stimulated, otherwise it becomes inactive. No

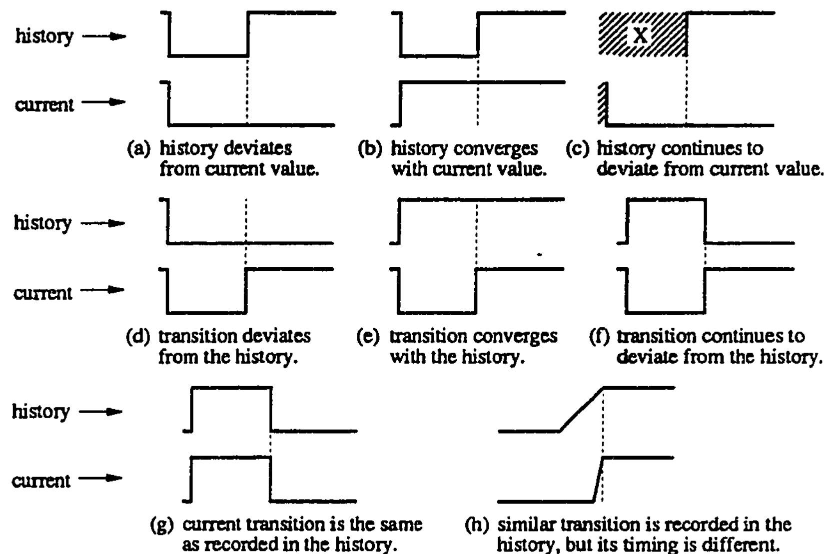


Figure 2.15: Possible cases due to an event occurring on an active node

stages are marked for evaluation.

(c) By remaining stable, the node continues to deviate from its history. No stages are marked for evaluation.

(d) A transition causes the node to deviate from its history; the node becomes active and deviating. All stages with inputs or boundary transistors controlled by the node are marked for evaluation.

(e) A transition causes the node to converge with its history; if the node controls any active transistors then it becomes stimulated, otherwise it becomes inactive. All stages with inputs or boundary transistors controlled by the node are marked for evaluation; these stages must be evaluated even if, as a result of this transition, their entire state has converged to that of the previous simulation, i.e., they just became inactive.

(f) The node transitions and continues to deviate from its history. All stages with inputs or boundary transistors controlled by the node are marked for evaluation.

(g) The node undergoes the same transition as recorded in its history. The node either

converges with or remains the same as its history; it becomes inactive or stimulated. Stages with inputs connected to the node that are still active at this point are marked for evaluation. This situation differs from (e) in that stages that became inactive as a result of this transition do not have to be evaluated.

(h) The node undergoes a similar transition as recorded in its history, only the timing is different. The node has either converged with or remains the same as its history. The situation is similar to (g) above, but in addition to active stages, inactive stages whose timing may be altered by the different time-constant will be activated and marked for evaluation. The timing of a stage may be altered by the time-constant if the transition turns a transistor on; this excludes boundary transistors.

At the same time that the consideration lists are examined and stages are marked for evaluation, the activation conditions of stages containing nodes converging with their history are examined; stages that do not meet these conditions are deactivated. After all events have been processed, the simulator proceeds to evaluate each stage marked for evaluation. As each stage is evaluated, its evaluation marks are cleared; this insures that each stage is evaluated only once during the current time step. The evaluation procedure, which is identical to the one performed by Rsim, will schedule an evaluation event for any output that is to change state. After all stages have been evaluated, the simulator scans the event queue and repeats the above process for the next set of events scheduled at the earliest time following the current time.

After processing evaluation and check-point events, but before evaluating any stages, other types of events are processed. The processing for all events is described below.

Stimulus Events

Stimulus events are never compared with the history of the node they specify, since by definition they are identical. To process a stimulus event, the state of all transistors controlled by the stimulated node that are part of an active stage (or at its boundary) is updated. If no such transistor is found then the event is ignored; otherwise the stages at the source and drain of each active transistor are marked for evaluation and the next transition at the stimulated node is scheduled as a stimulus event. By processing stimulus events

after evaluation and check-point events, the number of evaluations can be minimized, since some previously active stages may become inactive when the latter are processed, hence making their evaluation unnecessary.

Evaluation Events

When processing an evaluation event, the simulator will first check if the node it specifies has a check-point event scheduled at the current time. If this is the case then the event is processed alongside the check-point event, as explained in the next section; otherwise the node is assigned the new state, which is then compared with the history, and the node is inserted into the appropriate consideration list.

If the node was deviating from its history prior to the event then its new state is compared with its corresponding state during the previous simulation; if they are the same then the node converges with its history (Figure 2.15e), otherwise the node continues to deviate from its history (Figure 2.15f).

If the node is not currently deviating from its history then the absence of a check-point event at the current time implies that the node has just deviated from the history (Figure 2.15d).

The current transition is always added to the history. Since previous transitions that no longer occur are removed from a node's history, the simulator keeps a copy of the previous simulation value for as long as a node remains active. This value is updated whenever a check-point event is processed.

Check-Point Events

Processing a check-point event for a particular node must deal with the case when that node also has an evaluation event scheduled at the same time. In this case, after changing the state of the node to the new value, three possibilities arise:

1. The logic level and the timing specified by the two events are the same; the node has either converged with or remains the same as its history (Figure 2.15g). The history remains unchanged.

2. The logic level specified by the two events are the same, but the timing is different; the node has either converged with or remains the same as its history (Figure 2.15h). The old transition is removed from the history and the new one is inserted (since both transitions are the same, this case can be optimized by updating the timing information of the old transition).

3. The logic level specified by the two events are different; the node deviates from its history (Figure 2.15f). The old transition is removed from the history and the new one is inserted.

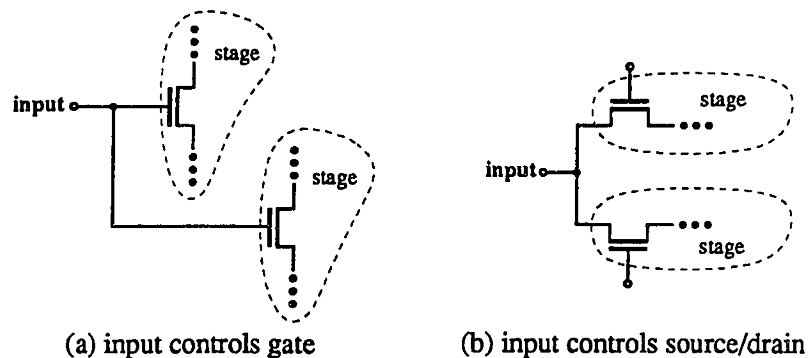
If the node specified by the check-point event does not have an evaluation event scheduled at the same time then the logic state specified by the event is compared with the node's current state: if the two logic states are the same, the node has just converged with its history (Figure 2.15b); otherwise the node deviates from its history (Figure 2.15a). In both these cases, the old transition is removed from the history.

Input Events

When a history transition to be scheduled as a check-point event corresponds to an input stimuli, instead of scheduling a check-point event, the simulator schedules an input event. There are two types of input events: *driven* and *undriven.* A driven input corresponds to the user specifying a particular value to be applied at a node, whereas an undriven input event corresponds to the user releasing the drive on a previously driven node. This type of situation is quite common with circuits containing bidirectional input/output pads: sometimes the node acts as an input and is driven by an input stimuli; at other times the node acts as an output, its value determined by the circuit.

Input events, both driven and undriven, must be treated specially since they can affect a stage in two ways (Figure 2.16).

An driven input event always causes the node to converge with its history, aborting any pending events the node may have. Stages whose inputs are controlled by a driven input event (Figure 2.16a) can be treated exactly the same as if it were a stimulus event, i.e., only active stages need to be evaluated. On the other hand, stages containing a node



(a) input controls gate (b) input controls source/drain

Figure 2.16: Two ways in which stages are affected by an input

driven by an input event (Figure 2.16b) will be split into two or more stages, perhaps causing some or all of them to become inactive.

To process a driven input event, each active stage containing the driven node must be considered separately and marked for evaluation. If all stages containing the driven node are inactive, the node itself becomes inactive. However, as long as a driven node is contained within at least one active stage, the node must remain active even though it has converged with its history and its state will not be changed by the evaluation process. This is necessary for two reasons; first, any transition on the driven node affects the behavior of the active stages and so they must be reevaluated. Second, the next event could be an undriven input event that causes the inactive stages to be merged with the active ones, thus becoming all active.

A driven node cannot change state until the drive is released or another stimuli is applied to the node; this means that a driven input history entry can only be followed by another driven input entry or by an undriven input entry. As part of the stage activation process, the simulator must identify all driven nodes and schedule their next transition either as a driven or an undriven input event, whichever follows their current pointer.

To process an undriven input event, the incremental state of the stage containing the previously driven node is examined; if the stage is still active, it will be marked for evaluation, otherwise the node also becomes inactive.

Leftover Events

These are events that remain in the event queue at the time the incremental simulation begins. This situation happens frequently in self-timed circuits that generate internal clocking signals. These events require special attention since they may still be valid at the end of the resimulation, i.e., the nodes they specify are inactive at the time they were scheduled, in which case they should remain in the event queue. Conversely, if the node the specify is being resimulated they should be descheduled.

Dealing with this condition is fairly simple; before resimulation starts, every evaluation event remaining in the queue is rescheduled as a *leftover event*, leaving the event time unchanged. When a leftover event is processed, the simulator checks the incremental state of the node it specifies. If the node is inactive at the time, the event is immediately rescheduled as an evaluation event; otherwise the event is discarded and, since the node is currently being simulated, any subsequent transitions will be scheduled for it as it is evaluated. When resimulation ends, any evaluation event still in the queue will remain in effect; if another resimulation is started, these events will themselves become leftover events.

2.3.9 Reducing Incremental Simulation Time

This section describes various techniques to improve the performance of the incremental algorithm; this can be achieved in three ways:

- Reducing the number of nodes in a stage.
- Reducing the number of active stages.
- Reducing the cost of stage evaluation.

The number of nodes in a stage can be reduced by eliminating all nodes that do not connect to any transistor gate and hence their values are not needed by other stages (nodes **X** and **Y** in Figure 2.17).

This scheme was first suggested by Terman for Rsim and it applies to both conventional and incremental simulation. The key idea is to merge transistor stacks of the same

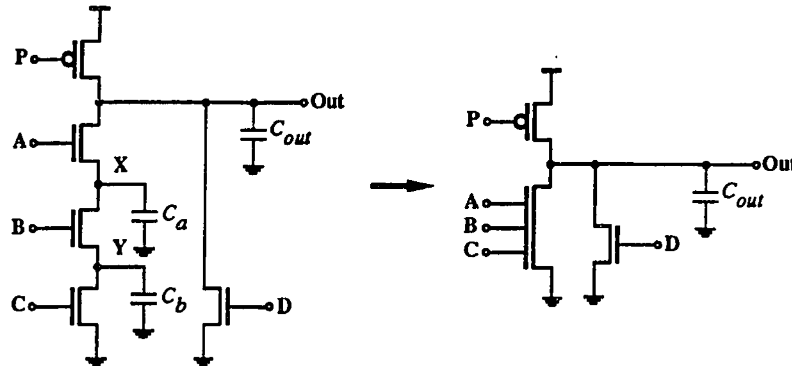


Figure 2.17: Removing internal nodes from a stage

type into a single compound transistor whose gate value is the logical conjunction of the values of the gates of the original transistor stack, and whose resistance is the sum of resistances of the original transistors.

The advantage of this scheme is speed: the reduction in the number of nodes makes stage evaluation faster, results in fewer events, and reduces the size of the circuit's history. The main disadvantage is the loss in accuracy due to the compound transistor's inability to account for the capacitance internal to the stack (C_a and C_b in Figure 2.17). This capacitance not only affects the timing of the surrounding circuit, but may also cause transitions by sharing its charge with the outer nodes, transitions that the simulator would fail to simulate. To avoid this charge sharing problem, the stacking option built into Irsim performs a static capacitance check before merging a stack; if the ratio of external to internal capacitance has the potential of changing the state of the outer nodes, the stack is not merged. For the example circuit of Figure 2.17 the following condition would have to be satisfied in order to merge the three transistors:

$$\frac{C_{out}}{C_a + C_b} > \frac{1 - V_{min}}{V_{min}},$$

where V_{min} is the minimum voltage fluctuation that can change the state of the node, and is equal to $\min(1 - V_{high}, V_{low})$.

Accounting for the effect of the internal capacitors on the timing of a driven transition can be accomplished by precomputing their contribution to the delay and adjusting the outer node's capacitance accordingly. For the example of Figure 2.17, this would result

in the following adjustment for C_{out}:

$$C_{adj} = \begin{cases} \frac{R_C+R_B}{R_A+R_B+R_C} C_b + \frac{R_C}{R_A+R_B+R_C} C_a & \text{when the compound transistor conducts} \\ 0 & \text{otherwise.} \end{cases}$$

In general, a different adjustment capacitor is needed for the source and drain terminals of the compound transistor. Unfortunately, there is no simple way to accurately model charge sharing transitions involving compound transistors; the timing errors that these transitions might produce should be considered before deciding to use the stack merging option.

Another way to improve the performance of the incremental algorithm is to reduce the number of active stages. To accomplish this, we have incorporated into Irsim a *resolution* parameter that eliminates stage activations caused by small timing differences. When the value of a transition is the same as recorded in the history and the difference in time is smaller than the resolution, the simulator will not activate the stages that depend on the transitioning node. This allows users to introduce limited timing errors that do not affect the functionality of the design.

Incorporating the timing resolution into the incremental algorithm is accomplished by introducing two new types of events: *delayed check-point events* and *no-change events*. Delayed check-point events are created when the new transition occurs slightly later than in the history (Figure 2.18a); the basic idea is to delay the history comparison and hence the corresponding stage activation until an expected evaluation event, which has already been scheduled within the resolution time, takes effect. No-change events are created when the new transition occurs slightly earlier that in the history (Figure 2.18b); they are used to ensure that no more transitions occur between the current transition and a similar transition recorded in the history.

At time t1, the check-point event of Figure 2.18a is processed before the evaluation event scheduled at time t2. Normally, this constitutes a deviation from the history that results in a transition being removed from the history. In this case, however, the node has an evaluation event that converges with its history within the resolution time, hence instead of changing the state of the node to deviating at time t1, this decision is postponed until time t2 by delaying the check-point event to ensure that the evaluation event takes effect. Six things may happen after t1:

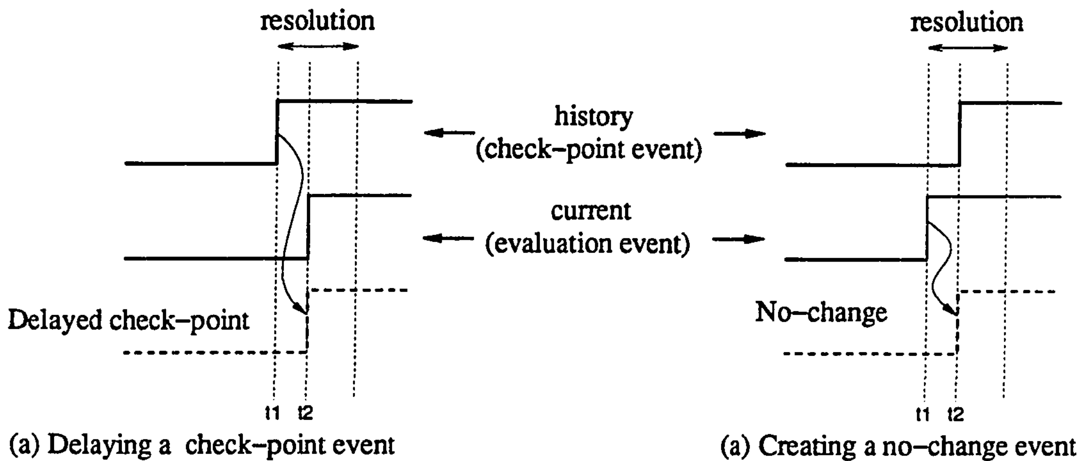


Figure 2.18: Small timing differences result in delayed check-point and no-change events

1. The evaluation event does indeed take effect at time t2; the node has therefore not deviated from its history and the stages that depend on it were never activated.

2. The evaluation event is aborted and another transition to high is scheduled sometime between t1 and t2. This situation is similar to the first case above, only better since the transition now occurs closer in time to the one recorded in the history. When the new evaluation event is processed, the delayed check-point event is descheduled, and, as before, the node has not deviated from its history and no stages were activated.

3. The evaluation event is aborted and then a transition to high is scheduled after t2 but before t1 plus the resolution. When the delayed check-point event is processed at time t2, the situation is identical to the original case of Figure 2.18a, and so the check-point event is again delayed until the time of the new transition.

4. The evaluation event is aborted and the node remains low. When the delayed check-point event is processed at time t2, the node is found to have deviated from the history. Accordingly, the transition is removed from the history and the stages that depend on the node must be activated. Since the node does not transition at t2, the activated stages are only reevaluated at this point if some other input undergoes a transition at t2.

5. The evaluation event is aborted and a transition to some other level, X for instance, is scheduled sometime between t1 and t2. When the new evaluation event is processed, the node deviates from its history; the delayed check-point event is descheduled and the stages that depend on the node are activated and reevaluated.

6. The evaluation event is aborted and then a transition is scheduled to take effect after t1 plus the resolution. This also represents a deviation from the history and it is processed as in 5 above.

In Figure 2.18b, the evaluation event is processed at time t1, before the check-point event scheduled at time t2. Normally, this also constitutes a history deviation that results in the insertion of a new transition in the history. However, since the history converges with the new value within the resolution time, the decision to change the state of the node to deviating is postponed until time t2 by turning the check-point event into a no-change event. If the node does not transition again until the no-change event is processed at time t2, then the node has not deviated from its history, and the stages that depend on the node were never activated. If, however, the node does transition again before time t2, the simulator will have failed to activate the stages that depend on the node for the duration of the change, as shown in Figure 2.19.

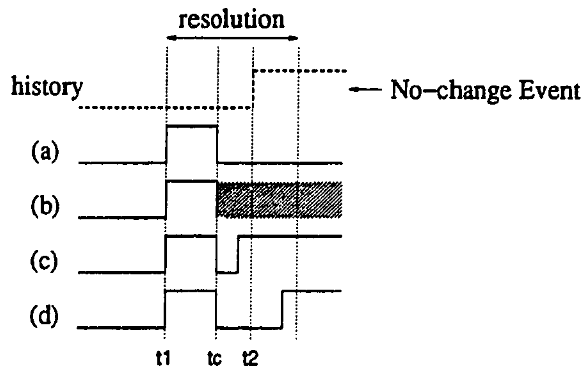


Figure 2.19: Several situations in which a node transitions before a no-change event

In Figure 2.19a, the node temporarily deviates from its history between t1 and the current time, tc, and then deviates again at time t2. The stages that depend on the node will have remained inactive from t1 to tc, neglecting to evaluate them for the duration

of the short-lived deviation. To rectify this mistake requires that the simulator revert the state of the circuit to that of time t1, activate the stages and continue simulating. This rollback in time, however, is beyond the capabilities of the algorithm in which time can only advance monotonically forward. The solution we adopted, albeit a crude one, is to issue an error message indicating the aforementioned condition; the designer can then rerun the simulation using a smaller resolution time. For an appropriate resolution value, this situation should be very infrequent; even if it does occur, the simulation may not be altogether wrong since the inertial delay model, given a small enough resolution, will most likely filter out the spike and eliminate its effect on the rest of the circuit, even if the stages had been evaluated.

The above discussion applies to all cases shown in Figure 2.19, the only difference is how the situation is handled after time tc. In Figure 2.19a, at time tc, the no-change event is turned back into a check-point event; when this event is processed, the node deviates from its history and the stages controlled by the node are activated. In Figure 2.19b the no-change event is also turned into a check-point event, but since the node deviates from the history at time tc, the stages it controls are activated and evaluated immediately. In Figure 2.19c the node rises again within the resolution, the simulation is back to the original case, and so the no-change event remains in effect. Finally, in Figure 2.19d the no-change event is turned into a check-point event, which when processed at time t2 will be rescheduled at time t4 as a delayed check-point event.

By using delayed check-point and no-change events, the incremental algorithm can minimize the number of stages that are activated, reduce the number of evaluations, and speedup resimulation. This mechanism, however, has some limitations. After several resimulations, the small timing errors introduced during each resimulation can accumulate to the point where the history becomes so severely corrupted that subsequent resimulations result in incorrect results with no indication as to what has gone wrong. Furthermore, the additional events required to implement the time resolution feature result in more overhead; this overhead may invalidate the speedup gained by not resimulating additional stages. These limitations suggest that the time resolution feature be used sparingly.

Finally, if a designer is only interested in the logical behavior of a circuit, not in its timing, it is possible to use a simplified stage evaluation model that reduces simulation

time even further. This can be accomplished by using any of a number of simpler evaluation algorithms, such as Bryant's MOSSIM II algorithm[6] or Terman's switch model[49][2]. The incremental algorithm is general enough that those models can be used without any changes to the algorithm, and in fact, this is true for any event-driven simulation algorithm, regardless of the models it uses. For example, using MOSSIM's model with assignable user delays would turn our algorithm into the one described by Jones in [25].

2.4 Performance

The incremental algorithm described in the previous sections has been implemented and incorporated into Rsim. The resulting simulator, Irsim, supports both conventional and incremental simulation; the results obtained using either method are identical[3].

As expected, the time that Irsim requires to complete a resimulation depends more on the number of re-evaluations than either the size of the circuit or the number of input vectors applied. When the number of re-evaluations is low, incremental simulation can be substantially faster than conventional simulation. Conversely, when the number of re-evaluations is high, incremental simulation may take as long, or even longer, than a conventional simulation. The two obvious questions are: how much faster? and how much slower? The answer to the first question is rather simple: if the modifications do not result in any changed nodes, incremental simulation will appear to be infinitely faster than conventional simulation. A more interesting question is how much faster (or slower) does it resimulate a typical modification. This question is addressed in Chapter 4, where we analyze the behavior of the incremental system during the final phases of a real design situation. The next section is devoted to answering the second question, which requires an analysis of the incremental algorithm's worst case performance.

[2]Actually, this model is already available in Irsim. Unfortunately, we have been unable to correctly simulate any of our benchmarks using this model, in either batch or incremental model.

[3]Of course, small timing differences are possible when using a time resolution greater than 0.

2.4.1 Worst Case Performance

To analyze the performance of the incremental algorithm, we modify an existing design and then resimulate the modified circuit using both the conventional and the incremental algorithms. The modifications we apply to the circuits do not represent real design errors, but rather attempts to artificially trigger the incremental algorithm's worst case performance, which occurs when the whole circuit is resimulated over the entire history. We then compare the time required by each of the two algorithms, identifying the various operations that contribute to the run-time of the simulation. The insight gained from this comparison is used to generalize our findings and show a definite upper bound on the degree to which incremental simulation can slow down execution, which, as we discuss later, depends not only on the algorithm itself, but on the complexity of the circuit as well.

Throughout our analysis, we use two real chips as test cases. Analyzing large designs helps insure that the algorithm is practical for use on real systems. The two circuits are:

- SPIM is a 64-bit by 64-bit iterating array multiplier designed by Mark Santoro of Stanford University[41]. To attain a high performance, the multiplier array generates its own internal clock whose frequency has been carefully tuned to match the internal delays. The circuit is designed in a 1.6μm CMOS technology, runs at 85MHz, and contains 41,804 transistors and 16,581 nodes.

- DIVIDER is a 54-bit, self-timed divider designed by Ted Williams of Stanford University[55]. The circuit is a direct implementation of an SRT division algorithm that uses a self-timed control chain to iteratively generate the result bits. It is designed in a 1.2μm CMOS technology, runs at 340MHz, and contains 15,353 transistors and 7,997 nodes.

To determine the overhead incurred by the incremental algorithm, we simulated the circuits using input vectors provided by the designers and then modified the circuits in such a way as to alter the timing of most of the nodes throughout the simulation. By altering the basic timing of the circuits so that all transitions occur at a slightly different time, we ensure that all work previously done by the initial simulation must be undone

(removing old transitions from the history) and then redone again (creating the new transitions). This form of resimulation results in worst case behavior not only because the whole circuit is resimulated over the entire history, but also because every old or new transition causes a node to either deviate or converge with its history, continuously changing its incremental state from active to inactive and vice versa.

In the case of SPIM, we added additional capacitance to the feedback line of the oscillator that generates the multiplier's internal clock. This change reduces the clock's frequency by about 6%, causing all multiplications to be resimulated. The circuit was simulated for 2μs, the time required to load and multiply 2 sets of numbers. The events generated during both the conventional and the incremental simulation are illustrated in Figure 2.20; these results are summarized in Table 2.2[4].

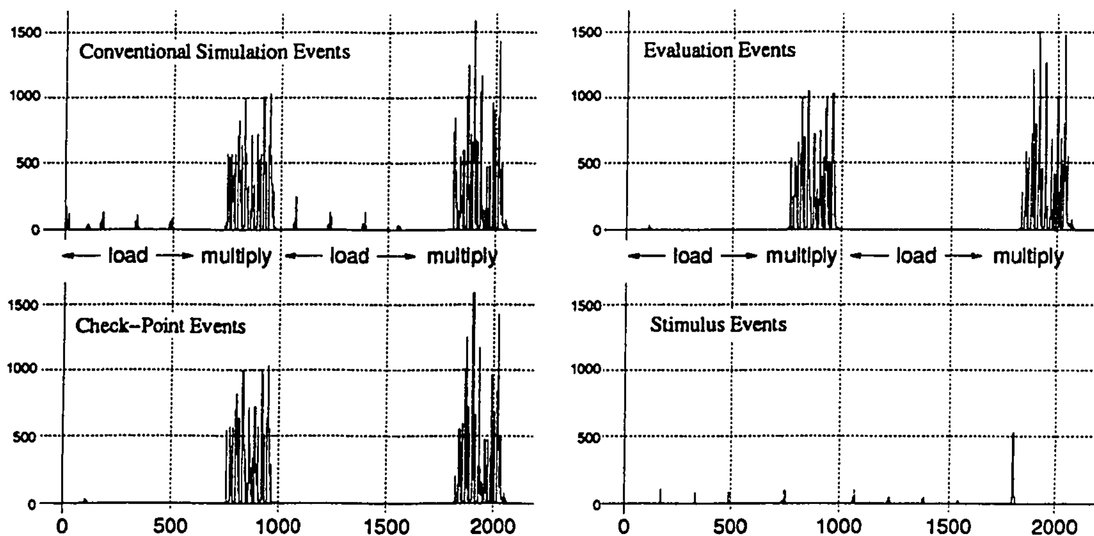


Figure 2.20: Number of event as a function of time for SPIM circuit

The number of events generated during incremental simulation is almost double those generated during conventional simulation. As expected, for every evaluation event, which corresponds to a new transition, there is also a check-point event, which corresponds to an old transition in the history occurring at a slightly earlier time. Also, the number of evaluations is almost the same in both simulations, achieving our goal of resimulating

[4]All tests were run on a Decstation 5000, which contains a 25MHz R3000 CPU, 64Kb instruction and data caches, and 24Mb of main memory. Irsim was compiled with the "-O" optimization level.

Simulation	Number of Events				Number of	Run Time
	Evaluation	Stimuli	Check-point	Total	Evaluations	(seconds)
Conventional	113,771	-	-	113,771	292,428	76.3
Incremental	99,762	5,368	99,769	204,889	260,741	85.0
Ratio (I/C)				1.8	0.89	1.11

Table 2.2: Number of events and running times for SPIM

the whole circuit over the entire history – the worst case behavior. The number of evaluations during incremental simulation is about 11% smaller because the operands are loaded using an external clock that was not modified, resulting in the operand loading not being resimulated, as indicated in Figure 2.20. Despite this small difference, the execution time for both simulations is approximately the same, with the incremental algorithm taking only 11% more time. We would not expect this overhead to increase by much had the remaining transitions been resimulated.

For the DIVIDER circuit, we added additional capacitance to the divider control chain; since this circuit is basically a ring oscillator, the net effect of this modification is to increase the delay of all transitions by about 5%. The circuit was simulated for 700ns, the time required to load and execute two divisions. The events generated during both the conventional and the incremental simulation are illustrated in Figure 2.21; these results are summarized in Table 2.3.

Simulation	Number of Events				Number of	Run Time
	Evaluation	Stimuli	Check-point	Total	Evaluations	(seconds)
Conventional	63,766	-	-	63,766	137,363	19.9
Incremental	60,074	1,880	60,076	122,030	133,242	26.2
Ratio (I/C)				1.9	0.97	1.31

Table 2.3: Number of events and running times for DIVIDER

Again, we find that the number of events generated during incremental simulation are about double the number for conventional simulation. Worst-case behavior is insured by the number of evaluations being nearly the same in both simulations. In this circuit,

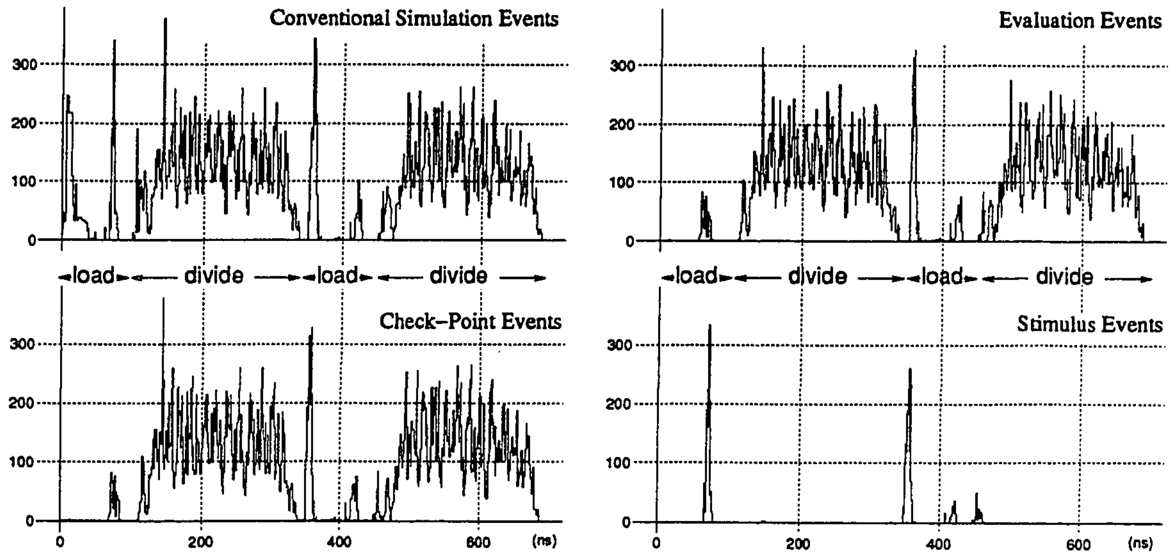


Figure 2.21: Number of events as a function of time for DIVIDER circuit

however, the relative increase in execution time during incremental simulation is almost three times greater than the corresponding time for SPIM, incurring an overhead of 31%. To understand why the overhead in this circuit is higher, we must examine the operation of the simulator for each circuit in more detail.

Tables 2.4 and 2.5 show the operations where the simulator spends time during the conventional and incremental simulations of SPIM and DIVIDER, respectively. The various rows and columns in Tables 2.4 and 2.5 represent the following:

- **time:** Actual amount of time spent in a particular operation.
- **%time:** Relative time of the corresponding simulation consumed by each operation, expressed as a percentage of the total time.
- **Ratio:** Ratio of incremental to conventional simulation time.
- **Overhead:** Percentage of the incremental overhead contributed by a particular operation.
- **Stage Evaluation:** Time spent performing charge-sharing analysis, and calculating final values and delays. It represents the same operations for both conventional

and incremental simulation.

- **Stage Decomposition:** Time spent decomposing the circuit into stages; during incremental simulation, it also includes the time spent updating the state of inactive nodes and checking the activation conditions.
- **Event Scheduler:** Time spent scheduling, descheduling, and aborting events, regardless of the type of event.
- **Event Processing:** Time spent scanning the event queue for the next event, removing the events from the queue, marking stages that need to be evaluated, and, during incremental simulation, comparing the history and building the consideration lists.
- **Stage Activation:** Time spent scanning the history for pending events, and determining for which nodes to schedule stimulus, check-point, or input events. It only occurs during incremental simulation.
- **Stage Deactivation:** Time spent resolving which output nodes become inactive or stimulated, and accordingly deschedule a check-point event or schedule a stimulus event (the actual scheduling time is not included). It only occurs during incremental simulation.
- **History Maintenance:** For conventional simulation it is the time spent recording the history; for incremental simulation it is the time spent updating the history.

We can make several important observations from the above tables. First, the amount of time required to maintain the history of either circuit is very small, consuming only close to 1% of the time, well within reasonable bounds. This result is in contrast to those cited by Hwang[7] and Jones[25], who report that history maintenance is the limiting factor for incremental-in-time simulation. Although this difference is partly due to Irsim's more computationally expensive model, which makes the relative overhead for maintaining the history very small, we suspect there are other factors involved; perhaps a sub-optimal implementation. Neither author presents any statistics to substantiate their claim.

Operation	Conventional		Incremental		Ratio	Overhead
	time (ms)	%time	time (ms)	%time	(I/C)	%
Stage Evaluation	67557.83	88.55	59591.43	70.09	0.88	-91.2
Stage Decomposition	3268.50	4.28	9779.11	11.50	2.99	74.5
Event Scheduler	3197.31	4.19	6006.90	7.06	1.88	32.2
Event Processing	1778.90	2.33	4557.96	5.36	2.56	31.8
Stage Activation	-	-	3643.45	4.29	-	41.7
Stage Deactivation	-	-	780.17	0.92	-	8.9
History Maintenance	488.21	0.65	666.29	0.78	1.36	2.0
Total	76290.75	100.00	85025.31	100.00	1.11	100.0

Table 2.4: Dissection of execution times for SPIM circuit

Operation	Conventional		Incremental		Ratio	Overhead
	time (ms)	%time	time (ms)	%time	(I/C)	%
Stage Evaluation	16424.16	82.23	16072.57	61.46	0.98	-5.7
Stage Decomposition	1009.44	5.05	2941.10	11.25	2.91	31.3
Event Scheduler	1309.44	6.56	2415.33	9.24	1.84	17.9
Event Processing	981.85	4.92	2407.90	9.21	2.45	23.1
Stage Activation	-	-	1532.11	5.86	-	24.8
Stage Deactivation	-	-	479.50	1.83	-	7.8
History Maintenance	248.26	1.24	301.70	1.15	1.22	0.9
Total	19973.15	100.00	26150.21	100.00	1.31	100.0

Table 2.5: Dissection of execution times for DIVIDER circuit

Second, both incremental simulations show a doubling in the amount of time spent in the event scheduler. This is not surprising since its time requirements are proportional to the number of events, which also doubles in the worst case. Similarly, event processing time increases by approximately 2.5 times during incremental simulation; this is because in addition to having to process twice as many events, it is a slightly more complex process than the one performed by the conventional algorithm. Both of these operations are relatively inexpensive: for SPIM, they account for 64% of the overhead, which corresponds to an increase in execution time of only 7%; for DIVIDER they account for 50% of the overhead, or 15.5% of the increase in execution time.

Third, the amount of time required by stage evaluation is dominant at all times, consuming between 82 – 89% of the time during conventional simulation. This result confirms our assumption that an incremental approach can substantially decrease simulation time by reducing the number of evaluations; our algorithm achieves that goal by attempting to replace this very expensive computation by other relatively inexpensive operations – maintaining the history, scheduling additional events, and keeping track of the incremental state of the circuit. The dominance of stage evaluation is such that not resimulating only 11% of the transitions in the SPIM circuit accounts for a 91% reduction of the overhead! Had these transitions been resimulated, the overhead, due to stage evaluation alone, would increase to 21%. While this number is closer to the overhead incurred by the DIVIDER, it still does not account for the discrepancy in the overhead of the two circuits.

We expected the incremental overhead to grow linearly with the number of re-evaluations, thus measuring the worst-case performance of any circuit would be sufficient to characterize it; our experiments, however, seem to indicate otherwise. At first we believed the difference in overhead was due to some non-linear behavior in the incremental algorithm, so we decided to trace the simulation of the test circuits. This analysis revealed that the complexity of the two circuits is significantly different: the average dynamic stage size for SPIM is 2.99 transistors, more than 60% of its stages contain more than 25 transistors, including some rather large stages of up to 920 transistors. DIVIDER, on the other hand, has an average dynamic stage size of 1.71 transistors, its largest stage containing only 37 transistors. After analyzing the two circuits it becomes obvious that the difference in overhead is due to the time required to evaluate the stages, a process that depends more than linearly on the size of the stage, roughly $\mathcal{O}(n^{1.13})$. Since SPIM is a more complex circuit, it spends relatively more time in stage evaluation than DIVIDER (about 7% more). In a more complex circuit, a few evaluations account for a much larger percentage of the time, reducing the impact that other operations may have on the execution time. Thus, even in the worst case, two circuits can experience different overheads due to incremental simulation; the more complex circuit having a lower overhead and vice versa. This difference in circuit complexity also explains why the relative overhead of the operations that depend on the complexity of the circuit –

stage decomposition, activation, and deactivation – is higher for SPIM; these operations require an amount of time proportional to both the number of events and the size of a stage.

Another important observation is that stage decomposition becomes a major contributor to the run-time of the incremental simulations, almost three times higher than in conventional simulation. This is not only because stage decomposition checks the activation conditions, which makes this operation somewhat slower than in conventional simulation, but also because in order to determine whether a stage has deviated or converged to its previous state, it must be performed for every event. Since the number of events doubles, we can expect this operation to take at least twice as long as in conventional simulation. Moreover, to determine what constitutes a stage, this process must look beyond the stage, checking boundary transistors as well as conducting transistors. The effect of this operation on the incremental overhead suggests that it may be worthwhile to maintain the stages throughout the simulation, and rather than re-building and computing their incremental state, simply update their composition and state incrementally. Unfortunately, the existence of current loops within a stage (parallel transistors, such as transmission gates, are quite common) make this a rather difficult optimization.

Since the degree to which incremental simulation may increase execution time depends on the complexity of the circuit, in order to evaluate the worst case performance of the incremental algorithm, we decided to simulate the worst-case behavior of the worst possible circuit. Luckily, the worst possible circuit is also the simplest, a circuit comprised of stages containing only one transistor, such as a CMOS inverter. To test this worst case scenario, we constructed a circuit consisting of a chain of 50 CMOS inverters; each inverter output is connected to the input of the next inverter in the chain, and we simulated it by applying a train of 2000 pulses at the input of the first inverter. We run two different sets of tests; in the first one, the capacitance at the output of each inverter was modified so as to delay the transitions of all subsequent inverters, resimulating the circuit after each such modification. In the second test, the circuit was incrementally built, starting with 49 inverters, simulating, adding the 50th inverter, and resimulating; then starting with 48 inverters, simulating, adding the last two inverters, and resimulating; continuing this way until only the first one remained and 49 inverters were added. In both

these tests the same fraction of the circuit is resimulated for the same transitions each time, however, when the inverters are incrementally added, their outputs have no history so they generate no check-point events. The results obtained are plotted in Figure 2.22.

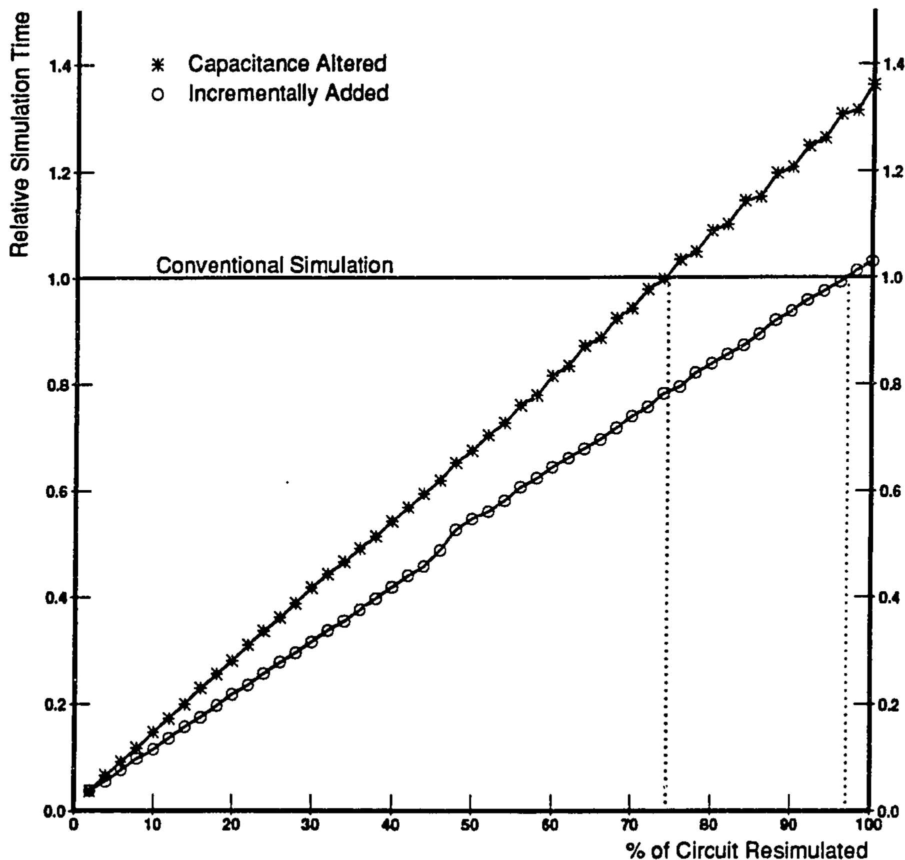


Figure 2.22: Simulation time as a function of the fraction of the circuit resimulated

The horizontal line in Figure 2.22 represents the time required to simulate the entire circuit once using the conventional algorithm; all other times are normalized to this time. Since each resimulated fraction of the circuit consists of identical subcircuits (inverters), execution time increases linearly for both tests. This was to be expected; what we did not expect was the slopes to be so strikingly different.

In the first test, incremental simulation consistently shows an overhead of 38%. Since

this represents the worst-case performance of the worst circuit configuration, it is an upper bound on the execution time degradation that any circuit might experience. Interestingly, incremental simulation will be faster until at least 74.5% of the circuit is resimulated for all time.

In the second test, the slope is nearly 1, resulting in extremely low incremental overhead; more than 97% of the circuit needs to be resimulated before incremental simulation shows any degradation. In the absence of check-point events, the added complexity of the incremental algorithm amounts to little overhead, reverting to the conventional algorithm. This indicates that our algorithm is particularly well suited for incremental refinement design techniques, in which the design is incrementally constructed by adding new components at each step. Although this means that designers must determine and apply all the input vectors before the circuit is complete, which may be difficult to do, the achievable savings in time make this an attractive option. Furthermore, it suggests a possible optimization to our algorithm: adaptive incremental simulation, whereby a node that continuously deviates from its history can be considered a changed node and resimulated for the remainder of the simulation, without scheduling its previous transitions as check-point events. This idea is similar to Choi's hybrid approach[8], except that the criteria for considering a node as changed is determined dynamically from its previous incremental behavior.

2.4.2 Using Other Models

We suggested earlier that our algorithm is not limited to Rsim's electrical models, and that it can be used with any event-driven simulator, regardless of the models it uses. We were curious as to how other models may affect the algorithm's worst-case performance. Since incorporating different models into Irsim is clearly beyond the scope of this thesis, we decided instead to extrapolate this information from the current implementation.

This extrapolation can be done by noting that except for "Stage Evaluation", all other parts of the program that contribute to the execution time of the simulator are independent of the models used. Since the same set of inputs applied to the same circuit should yield the same results (events), regardless of the models used, we can assume that all other

execution times listed in Tables 2.4 and 2.5 would remain approximately the same[5]. We can therefore factor out the effect of the model by subtracting the Stage-Evaluation time from the total execution time, replacing it with an expression that is proportional to the conventional algorithm's Stage-Evaluation time multiplied by the complexity of some other model relative to Rsim:

$$time_{inc} = Tirsim - Teval_{irsim} + K \cdot Teval_{rsim}$$

$$time_{conv} = Trsim + (K - 1) \cdot Teval_{rsim}$$

where $time_{inc}$ represents the model-independent worst-case incremental time, $time_{conv}$ represents the model-independent conventional simulation time, $Tirsim$ is the total execution time during incremental simulation, and $Teval_{rsim}$ and $Teval_{irsim}$ are the times required by stage evaluation during incremental and conventional simulation, respectively. The constant of proportionality, K, represents the relative complexity of some other model relative to Rsim: For a more complex model model, $K > 1$; for a less complex model $K < 1$.

The relative overhead due to the incremental algorithm can the be expressed as simply

$$\text{overhead} = \frac{time_{inc}}{time_{conv}} - 1.$$

Worst case behavior is ensured by using the conventional algorithm's evaluation time in both expressions, thereby assuming that the entire circuit is simulated for all time in both simulations. The results when K varies from 0 to 8 are plotted in Figure 2.23

As shown in the Figure 2.23, when $K = 1$, the overhead is that of Irsim. As the complexity of the model increases, the overhead tends to become more and more negligible. However, as the complexity of the model decreases, the overhead shows a sharp increase. Although an evaluation that takes zero time is unrealistic, it nonetheless represents the absolute worst possible degradation the incremental algorithm might experience; this upper bound is given by the "Inverters" curve, incurring a maximum overhead of 220%. This result is consistent with the worst-case performance reported by Choi[7], some of

[5]This is not strictly true; a more simplified model — a boolean functional model for example — might not simulate nodes internal to some blocks, thus yielding less events.

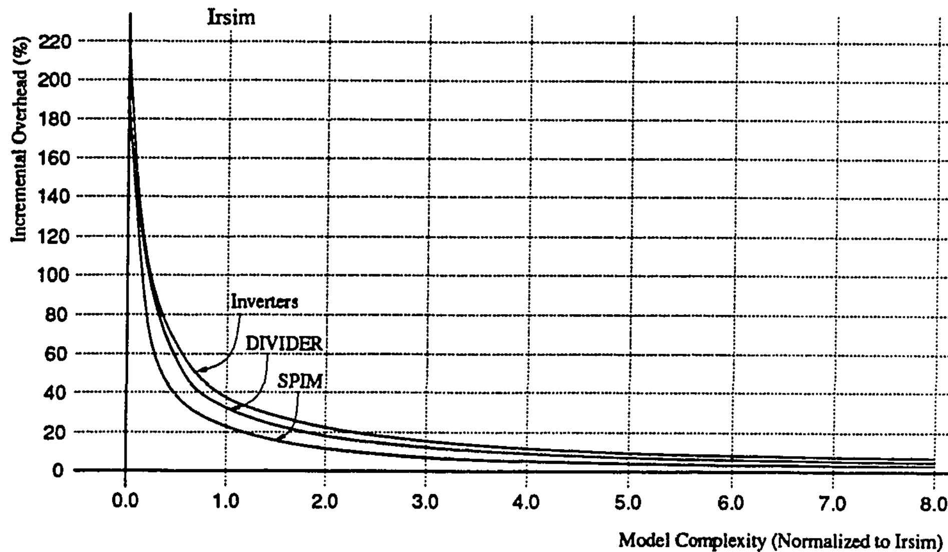


Figure 2.23: Relative worst-case overhead for the incremental algorithm using other models

whose tests result in an incremental execution time of three times that of the conventional algorithm.

Since other models may be faster (or slower) than Rsim by more than a constant factor, the preceding analysis is simply an approximation. Nonetheless, it is useful to estimate how other models may impact the performance of the algorithm. For example, we would not expect the worst-case overhead to increase by much more than 200%, regardless of the models used.

2.4.3 Non-Zero Time Resolution

To test the performance of the time resolution feature, we modified SPIM and DIVIDER as before, and then resimulated each circuit using resolutions of 0.1ns and 0.2ns. The results for this test are shown in Figure 2.6.

We can see from Table 2.6 that the higher resolution of 0.2ns results in a dramatic speedup for either circuit. This should not be at all surprising since it is this particular case that the time resolution addresses. As is obvious from the table, our capacitance change does not alter the timing of the circuit by more than 0.2ns; only the modified

		SPIM		DIVIDER	
Resolution		0.1ns	0.2ns	0.1ns	0.2ns
Number of Events	Evaluation	99,603	63	59,712	28
	Stimuli	4,643	38	1,870	28
	Check-Point	99,608	63	53,667	28
	Delayed Check-Point	5	47	7,011	28
	No-Change	1	0	3	0
	Total	203,860	211	122,236	112
Number of Evaluations		257,421	38	125,039	28
Time (seconds)		82.8	0.2	25.4	0.1
Speedup		0.93	199.0	0.73	382.5

Table 2.6: Number of events and running times for different resolutions

node is ever simulated. Since the only node to be resimulated in either circuit had its transitions delayed with respect to the history, there was never any need to schedule a no-change event.

At a resolution of 0.1ns, both circuits took about as long to run as the worst case analyzed before. In the case of SPIM, this resolution saved an additional 1% of the evaluations and ran almost 2% faster than with 0 resolution. For the DIVIDER circuit, it managed to save an additional 6% of the evaluations and ran about 4% faster than with 0 resolution. Even though the number of evaluations saved is greater than for SPIM, it does not represent much of an improvement. The differences are so small that the performance of the 0.1ns resolution is essentially that of the worst-case. It does indicate, however, that delayed events cause negligible performance degradation (no worse than the worst-case).

2.4.4 Memory Requirements

Finally, we address the memory requirements for maintaining the history. Table 2.7 shows the amount of memory devoted to the history of the various test circuits.

Efficient history maintenance mechanisms are necessary to keep the added cost of incremental simulation down, particularly during the initial simulation when the history

Test	Number of Transitions		History Size (Kbytes)	Time[a] (seconds)	Rate of Consumption[b] (Mbytes / MIPS · hour)
	Effective	Aborted			
SPIM	113,771	10,615	1457.6	76.3	3.7
DIVIDER	63,768	1,079	759.9	19.9	7.5
Inverters	101,819	0	1193.2	15.1	15.4

Table 2.7: Memory usage for various tests circuits

[a] Using the conventional simulation timing measurements
[b] Assuming an average of 18 MIPS for our machine.

is just being recorded. We did not attempt any sort of history compression scheme other than packing as much information as possible in the least amount of memory per history entry; this results in every entry using 3 words (12 bytes in our 32-bit machine). As Table 2.7 shows, our history maintenance mechanism uses an acceptable amount of memory by current standards. Note that the amount of memory consumed by "Inverters" represents an upper bound on the rate of memory consumption, corresponding to more than 24 million transitions an hour.

The locality exhibited by the simple linked-list implementation of the history works remarkably well in a paged virtual memory environment: as time increases monotonically, transitions that occur at the same time are allocated contiguous to one another, usually in the same physical page. Since only the pages containing the current history transitions are needed at any point in time, pages containing older transitions are not referenced any more and they become good candidates to be paged out. This results in negligible performance degradation due to second order memory effects, such as TLB or cache misses.

Currently, the only way to reduce the size of the history is by using the transistor stacking option described in Section 2.3.9, since the history of nodes internal to a stack is not maintained. If memory usage becomes a problem, the test can always be broken into several smaller tests. In all our experiments, some of which are quite large, memory requirements were not a problem.

2.5 Summary

VLSI designers iterate many times between capturing and simulating a design, typically making only minor alterations to the circuit. Resimulating the entire circuit can result in much of the time being wasted simulating sections of the circuit that are unaffected by the changes. An incremental simulator can eliminate this time by tracking the designer's changes and using the previous simulation results to confine the resimulation to the sections of the circuit affected by the changes. An incremental simulator is thus characterized by a runtime proportional to the size of the changes, not the size of the design.

There are essentially two algorithms that can confine resimulation to the sections of the circuit affected by a set of changes: incremental-in-space, and incremental-in-time. An incremental-in-space algorithm determines once (before simulation begins) which subcircuits may be affected by the changes, and then resimulates those subcircuits using a conventional simulation algorithm. This type of simulator is simple to implement and exhibits little overhead. However, because it pessimistically resimulates all possibly affected subcircuits, it does not fully exploit the previous results.

An incremental-in-time algorithm makes better use of the previous simulation history by only resimulating a subcircuit if its behavior is altered by the changes. The simulator begins simulating only the modified subcircuits, and then dynamically grows or shrinks the size of the subcircuits that need to be resimulated. This requires that the simulator continuously check which parts of the circuit need to become active or inactive. A subcircuit becomes active when any of its inputs deviates from its history; it becomes inactive when all its inputs and internal state reconverge with their history.

This chapter described the implementation of *Irsim*, an incremental-in-time simulator that uses a switched-resistor MOSFET model to accurately simulate circuits at the transistor level. The basic unit of simulation in *Irsim* is the stage: a subcircuit that includes all nodes electrically connected via conducting transistors. In a manner analogous to a conventional event-driven simulator, *Irsim* uses an event queue to limit the number of stages that need to be examined at each time step. However, a conventional event-driven simulator uses events exclusively to exploit circuit latency, whereas, *Irsim* uses events

to propagate the effect of the designer's changes through the circuit while advancing monotonically forward in time. This monotonicity complicates the algorithm by requiring that a deviation from the history (or convergence with it) be detected as soon as it occurs. Another source of complexity is *Irsim*'s inertial delay model, which requires that all conditions that affect the incremental state of a stage, such as pending transitions and aborted events, be taken into account. *Irsim*'s incremental algorithm accomplishes its task by augmenting *Rsim*'s event-driven mechanism to provide for history recording and comparison, selective event propagation to active stages, and by using different event types to signal the various conditions that arise during incremental simulation.

Irsim uses essentially three types of events: *evaluation events*, which are caused by a node changing state, and are similar to the events found in conventional simulators; *stimulus events*, which stimulate active stages whose inputs are inactive; and *checkpoint events*, which synchronize the resimulation with the history in order to detect missing transitions. These events are sufficient to track all changes in stage activation and deactivation that are needed by an incremental-in-time simulator.

Although the incremental-in-time algorithm makes optimal use of the history, the additional events and the continuous checking for active stages results in additional overhead. In the worst case, which occurs when the entire circuit is resimulated over the full history, incremental simulation can be slower than conventional simulation. To determine how this overhead may impact the performance of *Irsim*, we tested this worst-case scenario using several real, large designs. We found that the overhead due to incremental simulation may cause *Irsim* to take at most 38% longer than simulating the entire circuit using a conventional simulator; incremental simulation is faster until a design change causes a substantial part of the circuit (at least 75%) to be resimulated for all time.

Finally, we estimated how other evaluation models might affect the performance of the incremental algorithm. As the complexity of the model increases, the incremental overhead becomes more negligible. However, as the complexity of the model decreases and the time taken by the incremental algorithm becomes comparable to the model evaluation time, the overhead increases sharply. In *Irsim*, where the evaluation time is much higher than the incremental overhead, this tradeoff allows for substantial speedups, as discussed in Chapter 4. For a faster evaluation model, however, the cost of incremental

simulation might negate any possible speedups. In such cases, the incremental overhead can be reduced by increasing the unit of incremental simulation so that larger subcircuits are evaluated. For example, rather than a single stage, the simulator can maintain the incremental state of a set of interconnected stages; if any stage within the set deviates from its history, the entire set would be activated. Since the history of nodes internal to the set would no longer be needed, using larger units of incremental simulation provide a simple scheme to compress the history: maintain only the history of nodes at the boundary of the sets.

To be effective in reducing the time around the design cycle, an incremental simulator must be well integrated into a design system that allows incremental design modifications, and communicates these changes to the simulator. Techniques to incrementally update the circuit's network following a series of design modifications are examined in the next chapter.

Chapter 3

Incremental Circuit Extraction

The last chapter described a simulator that reduces the time required to validate a design by incrementally resimulating the parts of the circuit affected by design changes. To indicate these changes, the simulator requires a series of *network modification commands* which specify detailed changes to the netlist. Since circuits are rarely designed at the netlist level, these commands must be obtained by mapping changes to the description of the design being edited by the user into the corresponding netlist changes. This mapping, if done by hand, is a tedious and error prone task, and is naturally suited for automated design tools. Thus, the successful use of incremental simulation hinges on the problem of automatically identifying the modified portions of a design and conveying these changes to the incremental simulator. This chapter examines techniques to incrementally extract the modifications from a circuit's mask-level description.

The chapter is divided into five main sections. The first section briefly reviews what circuit extraction entails, and explores several alternatives to update a network incrementally. Following this is a brief overview of the Magic[35] layout editor, a review of previous work on circuit comparison, and a discussion of other approaches to incrementally update a circuit's description. Next is a description of our approach to incrementally extract the circuit modifications, its implementation within Magic, along with a description of the incremental circuit extraction and comparison algorithms. The final section discusses the performance of the incremental extractor.

3.1 Introduction

VLSI designs are normally created using interactive capture tools, such as layout or schematic editors, which support hierarchical composition and allow users to edit a pictorial representation of the design. The discrepancy between the topographical description used by these capture tools and the circuit representation used by verification tools prompts the need for circuit extractors and netlist compilers. These tools determine the electrical circuit implemented by the topographical representation being edited by the user, and are needed to provide a flat circuit representation required for efficient simulation. Thus, in order to simulate the design, the circuit implemented by a hierarchical layout or schematic must be extracted and compiled into a flat representation: a network in which every node points to the elements connected to it, and every element points to the nodes to which it is connected. This is necessary to efficiently evaluate elements and schedule events.

Inevitably, circuit extractors assume the role of translators between capture and simulation tools, and become a bottleneck in the design loop. Although the implementation details vary considerably — they may be flat[17] or hierarchical[43] — they all suffer from the same drawback: Each time the design is modified, they regenerate the entire (flat) network. One way to overcome this limitation is to use an extractor capable of identifying the circuit changes made to the layout (or schematic) and modifying the existing (flat) network accordingly. Updating the network incrementally eliminates the flattening step, and can also be used to quickly generate the network modification commands required by an incremental simulator.

Identifying network changes from the modifications made to a hierarchical layout or schematic, however, is not as obvious as it may seem. Small changes, such as deleting a component or a piece of material, may result in many changes to the underlying (flat) network. Conversely, large changes, such as moving an entire subcircuit from one location to another, may result in no electrical changes at all. The basic problem is the disparity between the two representations; before a network change can be identified as such, the circuit must be extracted, which is precisely what we are trying to avoid.

In addition to this problem, we determined that for the extractor to be useful, it would

have to meet the following requirements:

- Updating the network incrementally should yield the same circuit as regenerating the entire network. Special care must be taken to ensure that hierarchical names, interconnect parasitics, and device characteristics are preserved.
- It must be fast; preferably, the time it takes to detect and generate the network modifications should be proportional to the size of the modifications.
- Its operation should be transparent to the user; it should not impose additional restrictions on the operation of the editor or the layout style.
- It should generate the set of modifications that preserves as much of the unmodified network as possible. This reduces the number of modifications and the amount of work required by the incremental simulator to update the simulation results.

Since the problem is similar regardless of the topographical representation used, we chose to work at the layout level and adopted the Magic[35] layout editor as the front end to our incremental capture system. Many designers already use Magic and Rsim to create and simulate their designs, thus Magic provides a reasonable entry point to our incremental system. Modifying an existing tool rather than creating an entirely new editor saved us considerable development time. A secondary goal was to determine what changes are necessary to allow existing capture tools to operate in this new, incremental mode.

Depending on when the circuit changes are detected, there are essentially two ways to update the network. First, the changes could be detected after the user has finished modifying the layout. This can be done by simply extracting the entire circuit and comparing the resulting network to the one prior to the modification. The main advantage of this approach is that it can be used with existing capture tools. As the size of the design increases, however, the overhead associated with flattening and comparing two large, partially labeled networks can quickly become prohibitive. Besides being slow, this approach makes it difficult to find the set of changes that preserves most of the network.

Second, the circuit changes can be detected as they are being made by the user modifying the layout. Because it maintains both descriptions consistent at all times, this

approach does not have to maintain the unmodified version of the network nor does it require an expensive network comparison. The problem with this approach is detecting the changes quickly enough to still allow interactive use of the editor. This might be possible if only a small part of the layout needs to be analyzed. For example, one could devise a scheme in which the changes are identified at the hierarchical level being modified and then instantiated as many times as needed in their proper flat context. To make this scheme feasible, the hierarchy of the design would have to adhere to strict circuit boundaries, like a schematic. The hierarchy of a layout, however, does not obey any particular circuit boundaries and depends more on topological details like cell overlap.

Since any piece of material in a Magic layout is a potential connection to other parts of the hierarchy, a large part of the design might have to be examined to discover a change. Consider the example of Figure 3.1, in which a piece of material is deleted from **cellC**. Simply extracting **cellC** results in no electrical changes since the two nodes in the cell, **n1** and **n2**, remain unconnected. When **cellB** is taken into account, however, the extractor discovers that the modification disconnects node **n1** from nodes **n3** and **n2**. Finally, when **cellA** is considered, the extractor finds all four nodes to be connected to one another so no change is necessary.

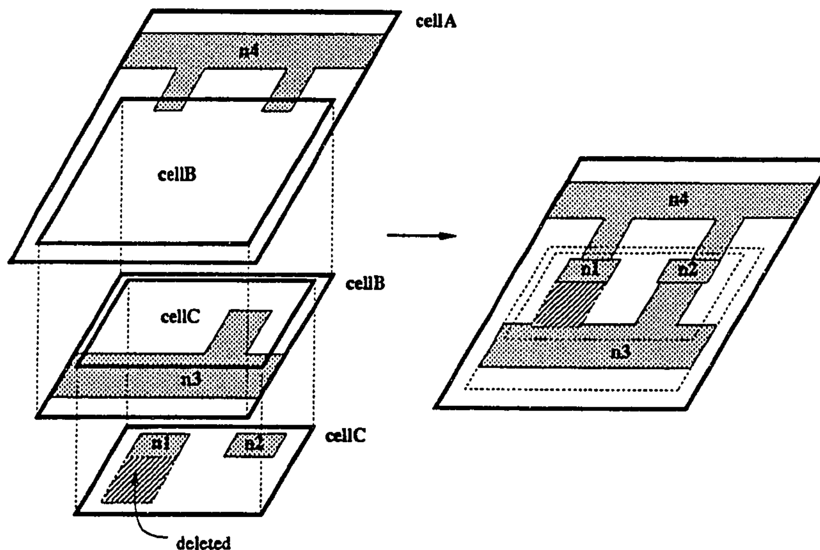


Figure 3.1: Circuit changes due to cell overlap.

In the preceding example, all three cells involved in the modification had to be examined, their nodes had to be traced and their interconnections computed; effectively extracting and flattening the three cells. To allow context free identification of the changes would require restrictions to cell overlap or explicit declaration of hierarchical connections, but doing so would make the editor harder to use, which violates one of our requirements.

An additional problem with trying to update the network while the layout is being modified is that changes can become undone or cancel out one another. It is not uncommon, for example, to modify a sizable part of the layout to accommodate a few additional wires or simply to resize a single transistor. In such cases, considering each change individually could result in sweeping changes to the network, yet considering them as a whole would uncover the much smaller, actual change. Detecting this condition after the network has been modified would force the simulator to perform the cumbersome task of maintaining and comparing the two networks.

The approach we adopted is a combination of the two described above. It is based on the observation that although detecting the changes while the layout is being modified can be expedient, there is no advantage in maintaining the network consistent with its layout at all times since the user will only resimulate it after having made all the changes. Therefore, instead of attempting to identify the changes as the layout is being modified, we simply keep track of the modified areas. When the user finishes editing the layout, only the modified areas are extracted in their proper (flat) context for both the modified and the original layout. The resulting subnetworks are then compared and the differences are reported as netlist changes.

While this approach still requires a network comparison, it can be done efficiently, for two reasons. First, the networks being compared are only as large as the modifications themselves, which are typically much smaller than the entire design. Second, since the layout outside a modified region is known to be unchanged, the nodes at the boundary of the region can be immediately recognized as equivalent in both networks. These boundary nodes provide the comparator with a good starting point, and enable it to quickly identify the changes within the region.

3.2 Overview of Magic

Magic is a layout editor for integrated circuits developed at the University of California at Berkeley. Magic introduced *corner stitching*, a novel data-structuring technique that accounts for much of its efficiency and many of its features, such as continuous design-rule checking, hierarchical circuit extraction, and routing. A more detailed description of Magic's features can be found in the 1984 Design Automation Conference Proceedings[35].

In Magic, a layout is represented as a hierarchical collection of cells. A cell, which can contain mask information as well as other sub-cells, is represented as a set of overlapping planes, each composed of rectangular *tiles* of varying types. Each plane (Figure 3.2) is completely covered by a mosaic of these tiles, which, depending on their type, represent either an area covered by some material or empty space. Because tiles do not overlap, each point in the plane is contained within a single tile. While there is no unique tile arrangement to describe a given layout, Magic provides a canonical form for the design by splitting the areas covered by the same material into maximal horizontal strips (see Figure 3.2). This arrangement also prevents a plane from becoming fragmented into many small rectangles, thereby reducing memory and run-time overhead.

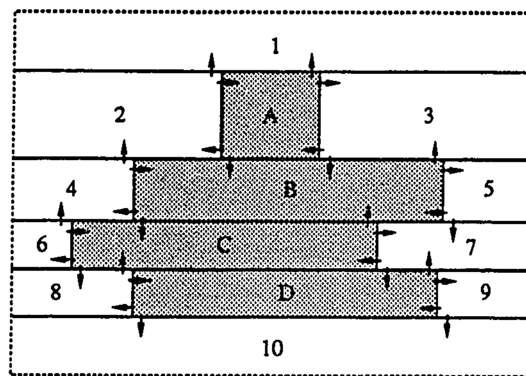


Figure 3.2: A corner-stitched plane of Magic tiles.

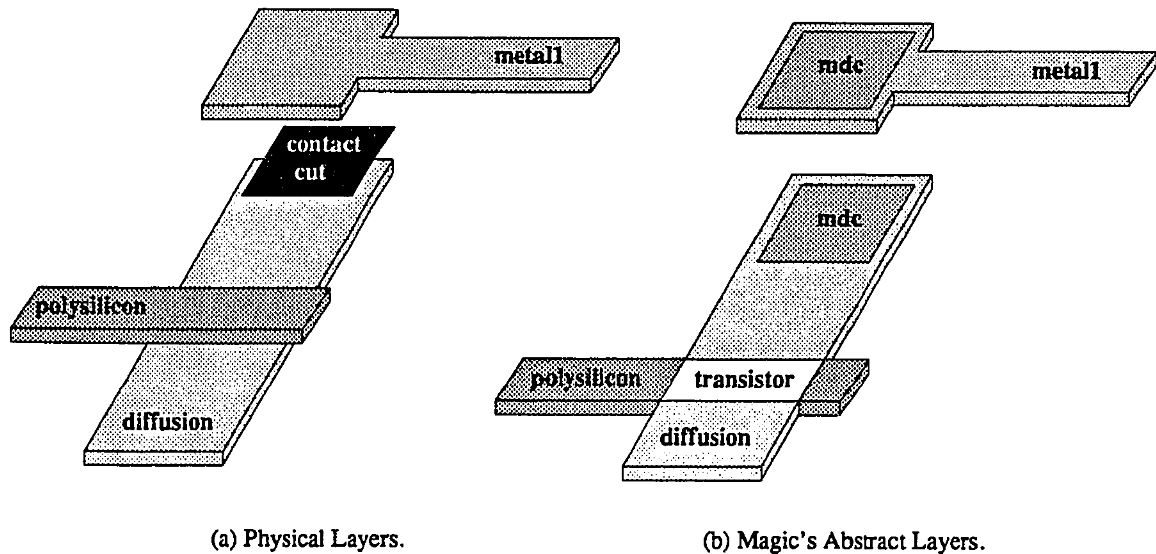
Tiles are assembled into planes by means of *corner stitches*, which are pointers to other tiles in the plane (the arrows in Figure 3.2). Each tile has four stitches that point to the tile's immediate neighbors: two at its lower-left corner and two at its upper-right

corner. Stitches allow local searching operations to run very fast; operations such as finding all tiles that touch a given tile — which is useful for design rule checking, circuit extraction, or determining connectivity — can be done by simply following stitches. For example, to find all tiles touching the top of tile B in Figure 3.2, first follow B's top-right pointer (to tile 3) and then follow bottom-left pointers (to tile A and then to tile 2) until reaching a tile whose left side is further left than B's. Similar algorithms exist for operations such as locating the tile containing a given point, finding all tiles in a given area, or traversing a connected region of tiles[34].

There are several ways in which corner-stitched planes might be used to represent the mask layers of the fabrication technology. One approach is to place all mask layers in one plane, using a different tile type for each possible layer overlap. The exponential number of tile types required, however, would fragment the plane into many small tiles. Another approach is to place each mask layer in a separate plane, using only as many tile types as mask layers. However, since many layout operations require information about layer interactions, this type of arrangement would result in much time being spent searching across planes.

Magic's approach lies between the two described above. Instead of working with the actual mask layers, Magic uses *abstract layers* that can represent not just a material, but composite structures, such as transistors and contacts. This is accomplished by using a technology dependent representation in which layers that commonly interact with one another are placed in the same plane. For example, a typical MOS technology with two metal layers requires only four planes: *active, well, metal1, and metal2.* The *active* plane contains the diffusion and polysilicon layers, plus their combinations, which form transistors (see Figure 3.3). Since interactions among the wells and the metal layers are rare, each is placed in a separate plane.

To represent connections between layers that lay in different planes, such as contacts, Magic uses a special composite layer that is replicated in both planes. For example, the contact formed by the diffusion, contact cut, and metal1 of Figure 3.3 are combined into a composite type **mdc**; a tile with type **mdc** is then placed in both planes: metal1 and active. An important observation is that abstract layers allow many circuit features to be represented explicitly; there is never a need to discover transistors by finding where



(a) Physical Layers.

(b) Magic's Abstract Layers.

Figure 3.3: Abstract layers in Magic.

polysilicon overlaps diffusion. Similarly, contacts need not be inferred; they are implied by the composite tiles.

In addition to the layer planes, cells contain a plane that holds information about its subcells. Magic does not support explicit cell connection ports; instead, cell interconnections are created by either abutting or overlapping cells. Magic allows nearly arbitrary overlap between cells; the only restrictions are that cells must be independently design-rule correct, and their overlap cannot create or destroy transistors. Because of this near-arbitrary overlap, any tile can become a point of connection to another cell. These connections pose a special problem since they are not explicit, and must be computed.

Magic's circuit extractor[43] is hierarchical and incremental. The extractor works by first extracting the circuit represented by the technology layers of a cell, independent of the context in which the cell is used. Next, the extractor identifies and flattens areas of interaction among subcells; these areas are then extracted and their capacitance and connectivity is adjusted accordingly. Because a cell can be extracted independent of its context, Magic can incrementally update extracted cells: when a cell is modified, only that cell along with all its ancestors need be extracted. Each cell is extracted into a separate file, and, prior to simulation, a separate program is used to compile the extracted

cells into a flat network representation. Each cell contains a timestamp indicating the time when the cell was last modified. When a cell is extracted, its timestamp is copied into the extracted cell's file. To determine which cells must be re-extracted, the cell's timestamp is compared with that of the extracted cell; if they differ, that cell along with all its ancestors need to be extracted.

Magic contains an incremental design-rule checker[48] which runs continuously in the background and uses designer latency to perform its work. The incremental checker works by recording the areas that have been modified by the user and only re-checks those areas. This is accomplished by adding an additional plane to each cell; this plane contains tiles describing the areas of the circuit that need to be verified. The checker continuously searches for these tiles; when it finds one, it verifies the area indicated by the tile, and then erases the tile. To handle cell overlap, the modified areas are recorded in the cell being edited as well as in each of the cell's ancestors, all the way up to the root of the design. The checker then proceeds in a manner similar to the extractor: First, it verifies the mask layers of a cell within the modified area. Next, it identifies areas of subcell interaction within the modified area, flattens the interactions, and verifies them.

3.3 Related Work

To update a network incrementally, the connectivity differences between the unmodified and modified versions of a circuit must be determined. Since these differences can modify the structure of the circuit, they can be identified by comparing the topology of the two circuits. The comparison has to detect isomorphism in order to establish which elements exhibit the same connectivity in both circuits and, if there are any differences, the transformations needed to convert one circuit into the other. Alternately, this can be regarded as comparing two circuits that are isomorphic but, due to some design changes, exhibit certain differences.

A convenient way to represent the topology of a circuit is a graph. A circuit's graph (Figure 3.4) contains a vertex for every electrical node or device in the circuit; its edges link devices with the nodes to which they are connected.

A graph representation completely captures the topology of the circuits, and reduces

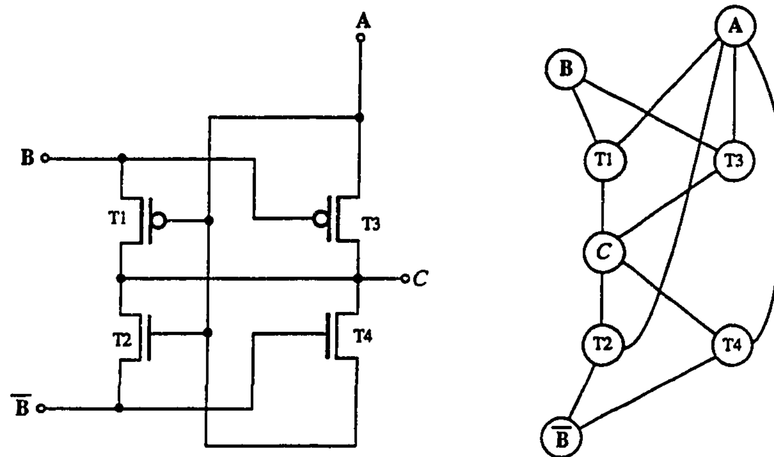


Figure 3.4: A CMOS XOR circuit and its corresponding graph representation.

the problem of comparing their topology to the well known problem of detecting isomorphism between two graphs.

3.3.1 The Graph Isomorphism Problem

Isomorphism of graphs is a well known concept: two graphs G_1 and G_2 are said to be isomorphic if there exists a one-to-one mapping σ of the vertices of G_1 onto the vertices of G_2 such that two vertices in G_1 are adjacent if and only if the corresponding vertices in G_2 are adjacent. Determining whether two graphs are isomorphic, however, is far from trivial. The basic problem is illustrated in Figure 3.5. Although the two graphs of Figure 3.5 appear to be different, they are merely different renderings of the same graph (that of the XOR circuit of Figure 3.4). This can be verified by redrawing the two graphs so they resemble one another. However, there is no unique or canonical way to draw a graph. To match the two graphs, therefore, one of them must be repeatedly redrawn by trying all possible topological permutations; a process that grows exponentially (or worse) with the size of the graph.

Because of its practical and theoretical importance, the graph isomorphism problem has been extensively studied in the past[1]. Although several attempts have been made to

[1] A comprehensive survey of papers on graph isomorphism algorithms, including a brief description of each paper, can be found in [38].

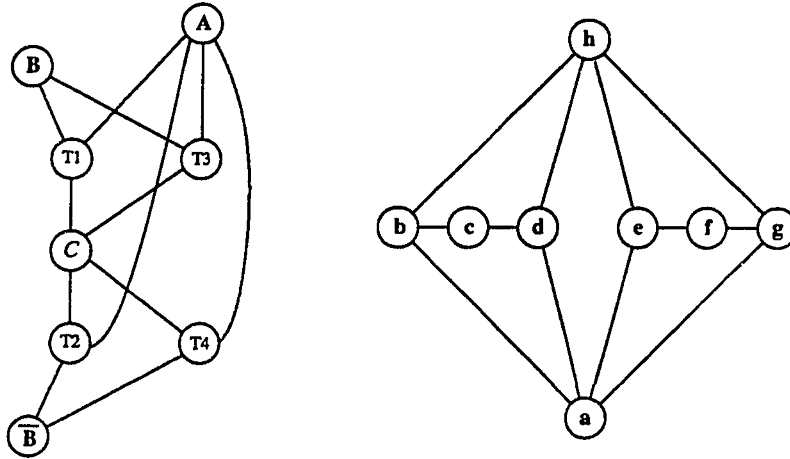


Figure 3.5: Two renderings of the graph of the XOR circuit of Figure 3.4.

find a mathematical function that can identify isomorphic graphs[18, 50], no such function that can be computed in polynomial time has been found. The problem is believed to be in the set of NP problems, but whether it belongs to P or is NP-complete remains unresolved[12]. Polynomial time algorithms for detecting isomorphism of planar graphs do exist[20, 19, 54]. Unfortunately, these algorithms rely on properties of planar graphs that circuit graphs, which are not planar, do not possess.

To analyze the complexity of the problem, consider two graphs, each with n vertices and p edges. If a different label is assigned to each vertex in the two graphs such that two vertices are assigned the same label only if they are equivalent, then isomorphism can be checked by simply comparing the edges of the two graphs; if they are identical, the graphs are isomorphic. Assuming the graphs can be labeled in linear time, the number of operations needed is then $\mathcal{O}(n+p)$. Now, if one of the graphs has been labeled in some arbitrary manner then an isomorphism can be detected by trying all $n!$ possible ways of labeling the second graph. This "brute force" approach will certainly find an isomorphism, if it exists, but it will require $\mathcal{O}((n+p)\cdot n!)$ operations; a prohibitively large number except for very small graphs.

There is no known algorithm that can solve the general problem efficiently (in polynomial time). There are, fortunately, several heuristic algorithms that, in practice, perform well on a large class of graphs. Although the details may vary considerably, most of

these algorithms adopt the same basic strategy which attempts to improve upon the "brute force" approach described above by reducing the number of permutations that need to be checked. The basic idea is to partition the graph's vertices into groups of vertices with the same topological features, thereby ruling out permutations that are clearly not isomorphic. If the number of vertices in these groups becomes sufficiently small (typically one), the number of comparisons decreases drastically and isomorphism can be tested quickly.

The information needed to partition a graph cannot be given by the vertices themselves but by their relation to other vertices. A common method is to use *vertex invariants* to label the vertices of the two graphs in such a way that corresponding vertices are assigned identical labels. A graph invariant is a property that is preserved by isomorphism, that is, a property that does not depend on how the graph is labeled. Examples of graph invariants are: the number of vertices, the number of edges, the degree of a vertex, the length of the smallest cycle enclosing a vertex, etc. Existing algorithms typically proceed in two steps: first, they generate an initial partition based on vertex invariants; next, they refine each partition into smaller groups by considering the neighborhood of each vertex. This process is then iterated, each time attempting to characterize a vertex by its relationship with ever increasing numbers of vertices more distantly connected, until isomorphism is detected or the partitions cannot be further refined. Figure 3.6 illustrates how a simple graph can be partitioned into singleton sets using vertex degree as the only invariant.

The graph of Figure 3.6 is initially partitioned by labeling each vertex with its degree (the number of incident edges). This partition is then repeatedly refined by relabeling vertices whose labels are not unique. In this example, a vertex is relabeled by the quick and simple method of adding the labels of its adjacent vertices. It is possible to use a more sophisticated relabeling method that propagates information about unique labels throughout the graph (as suggested in [13]). Although such methods make each iteration more expensive, they might partition the graph in less iterations.

If the partitioning results in singleton partitions, as in the previous example, then detecting isomorphism becomes an easy exercise. If, on the other hand, the refinement terminates with large partitions then all their permutations must be considered, an expensive process that is typically implemented as a backtracking algorithm[42]. The efficiency

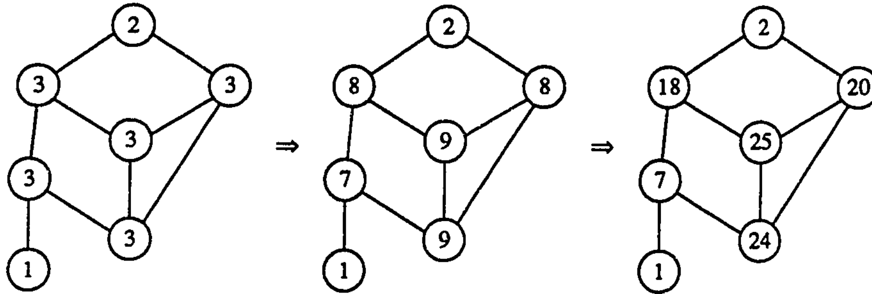


Figure 3.6: Partitioning a graph into singleton sets using vertex degree as the invariant.

of these algorithms, therefore, depends on using invariants powerful enough to expose the distinguishing features, allowing the graphs to be partitioned into sufficiently small groups. However, regardless of the invariants or the partitioning method, a graph can not be completely partitioned if it contains *automorphisms*, i.e., isomorphisms of a graph with itself. For example, it is not difficult to verify that there is not one but sixteen possible isomorphisms between the two graphs of Figure 3.5, as indicated by the table below:

A	a	a	a	a	a	a	a	a	h	h	h	h	h	h	h	h
B	c	c	c	c	f	f	f	f	c	c	c	c	f	f	f	f
$\overline{B}$	f	f	f	f	c	c	c	c	f	f	f	f	c	c	c	c
C	h	h	h	h	h	h	h	h	a	a	a	a	a	a	a	a
T1	b	d	b	d	e	g	e	g	b	d	b	d	e	g	e	g
T2	e	e	g	g	b	b	d	d	e	e	g	g	b	b	d	d
T3	d	b	d	b	g	e	g	e	d	b	d	b	g	e	g	e
T4	g	g	e	e	d	d	b	b	g	g	e	e	d	d	b	b

Thus, a graph may not be completely partitioned because either it contains some automorphisms or the partitioning method is not powerful enough to distinguish the different features. It is important to note that if a partition is known to be automorphic, then it

is possible to choose an arbitrary vertex from each of the two graphs and match them with one another (by assigning them a unique label, for instance). Determining whether a partition is in effect automorphic, however, is equivalent to detecting isomorphism, that is, a graph invariant that can be computed in polynomial time and is also capable of partitioning the graph into its automorphic partitions would provide a polynomial time solution to the graph isomorphism problem (see [38] for a proof). Unfortunately, no such invariant is known[14]. Nevertheless, this property can be useful if a partition is known (by some other means) to be automorphic.

3.3.2 Graph Isomorphism Applied to Circuit Comparison

The application of graph isomorphism algorithms to circuit comparison is simplified by using the additional information available in the circuit: vertices can be recognized as nodes or devices, device vertices can be further characterized by their type, and edges can be distinguished by the device terminals they link (gate, drain, etc.). By maximizing the number of distinguishing features, a finer initial partitioning can be expected. For example, when this information is added to the graphs of Figure 3.5, a unique isomorphism is easily found as shown in Figure 3.7.

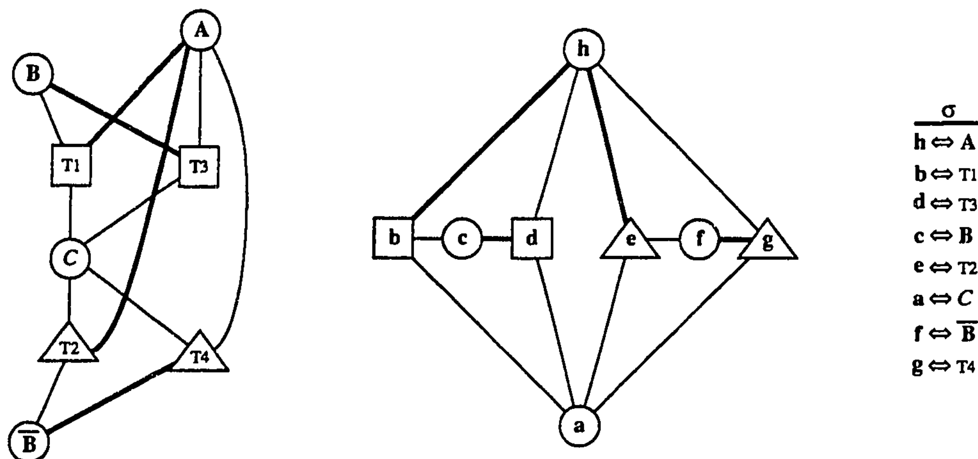


Figure 3.7: The graphs of Figure 3.5 including circuit information.

Traditionally, circuit comparators have been used as verification tools to ensure that the implementation of a circuit (typically the layout) is consistent with some other specification (schematic, stick diagram, netlist, etc). In this vein, the problem has been widely studied and numerous algorithms have been developed. While not all of these algorithms are explicitly based on graph isomorphism techniques, they can all be related to the graph partitioning algorithm described in the previous section; the differences, though many, depend more on implementation details such as the graph representation used, the relabeling method, the level at which the circuit is compared (logic, circuit, gate, etc), the type of reports produced, and the way that ambiguities and differences are handled. Typically, they proceed by identifying an item with a unique label (sometimes called signature or color) in one circuit, locating its counterpart in the other circuit, binding the two items, and relabeling the elements connected to the bound items.

One of the first to report using such a comparator is the *APPRAISE* system[1]. In this system, the initial partitioning is done by assigning a set of properties to each element based on the element's invariants (fanin, fanout, element type, etc.). Next, the partitions are enhanced by alternately binding devices and nodes in the two circuits. This process is iterated until no more elements can be bound. To decide which items to bind, the algorithm first generates a signature for each element by computing a weighted sum of the element's properties. Elements with similar signatures in the two circuits are then compared; if the signatures differ by less than some threshold (which is based on one of the signatures) then the two elements are bound to one another. Binding some elements within each iteration, rather than attempting to partition the entire circuit at once, allows the program to use the information gained from the elements bound during the previous iteration, thus bypassing ambiguous information until enough data has been built up to resolve the ambiguity. The major problem with binding elements by comparing signature differences against a threshold is its susceptibility to binding prematurely, which can result in apparent differences that might otherwise be resolved by further partitioning the circuit.

Another partitioning technique is employed by *WOMBAT*[45]. Rather than combining an element's properties into a scalar and using a threshold function to decide when two items should be bound, this algorithm maintains all the properties separate and only binds

two elements when their signatures are unique and identical. To minimize the number of comparisons, *WOMBAT* maintains a list of elements bound on any given iteration and restricts the comparison to elements directly connected to the ones on the list. Since not all circuits can be completely partitioned into singleton sets, however, this algorithm fails when the circuits contain automorphisms or ambiguous information.

A major problem with the signature partitioning technique is the algorithm's tendency to diverge in the presence of differences. Since the signatures are formed using neighbor information, even small differences can result in many elements having different signatures. These differences are then propagated throughout the circuit, preventing other isomorphic subcircuits from being matched. To avoid this problem, various heuristics that attempt to contain the propagation of differences have been proposed.

GEMINI[15] also uses a signature partitioning technique similar to the ones described above. The partitioning is done by assigning a label to each node based on circuit invariants. The partitioning is then refined by combining the label of each element with the labels of its neighbors, that is, a label is a function of both the element's current label and the labels of its neighbors. A hashing function is used to limit the size of the labels to a scalar. Instead of waiting until the circuits are completely partitioned, this algorithm also binds elements with identical, unique labels after each iteration. To avoid small differences from being propagated throughout the circuit, the partitions of the two circuits are compared after each iteration and the elements in partitions that do not have a corresponding partition in the other circuit are marked as suspect and their labels are not modified by subsequent iterations. After all the elements have been bound or marked as suspect, the program attempts to bind suspect elements using their neighbor information. If the partitioning does not result in singleton partitions, *GEMINI* arbitrarily binds two elements in corresponding partitions and attempts to continue partitioning the circuit. If this arbitrary match is later found to be incorrect, the program does not attempt to fix the error and fails.

A similar approach to contain differences from propagating through the circuit is adopted by *WLC*[31] and *CCOMP*[47]. These algorithms maintain the partitions explicitly and examine their contents at each iteration; if corresponding partitions of the two circuits are not composed of the same number of elements then those partitions are not partitioned

any further. Preventing the graph from being partitioned may allow other isomorphic subcircuits to be matched, however, it makes isolation of differences more difficult. Neither algorithm attempts to resolve ambiguities; partitions that contain more than one element of each graph are simply reported as mismatches.

If we put to one side the implementation details, we find that most other algorithms are fundamentally the same as the ones described above. With few exceptions, they can be characterized by the relabeling method and the heuristics they use, in addition to the local circuit invariants, to distinguish elements whose partitions cannot be refined into singleton sets. For example, the *LIVES* system[53] uses distance distribution and distance from terminal nodes for the initial partition. To resolve ambiguities, *LLC*[32] uses path information between ambiguous components. Wong[56] uses hierarchy and connections to known nodes to enhance the partitions; ambiguous partitions are resolved by arbitrarily matching all elements within partitions with the same number of components. *NECOM*[3] examines the vicinity of each matched node attempting to discover more unique features or differences. Tygar and Ellickson[51] use a randomized hashing technique to partition the circuits; the connections of matched components are then examined attempting to recursively match other elements connected to those components.

Many of these circuit comparators only indicate whether the two circuits are isomorphic, while others simply report which elements could not be matched, either by printing or annotating one of the circuits with the list of unbound components. This may be adequate when the comparison is intended to ensure consistency between two circuit descriptions that are expected to be the same, which perhaps exhibit minimal differences. However, using these algorithms to extract the differences between the modified and unmodified versions of a circuit is less natural. The main difference is that our problem does not necessarily have to prove an isomorphism; instead the goal is to find an isomorphism function, i.e., the transformations needed to convert one circuit into the other, a goal for which a list of unbound elements is of little use since it does not convey how the circuits differ. The fundamental problem with graph partitioning methods is that label refinement is the primary method used to identify matching circuit elements, but the labels themselves can only indicate equivalence; they offer no information regarding the nature of the differences. Furthermore, since in our case the circuit is known to be modified,

differences are very likely to occur; they cannot, therefore, be treated as exceptions by simply containing their effect on the rest of the circuit.

3.3.3 Incremental Circuit Update

Other researches have investigated a few approaches to update a circuit incrementally. Magic's circuit extractor[43], which is described in more detail in the next section, can incrementally extract hierarchical cells by examining their modification dates.

Beatty and Bryant[4] use an incremental method to reduce the amount of time required by the *COSMOS* simulator[37] to compile a circuit description into executable Boolean procedures. First, they extract a two-level hierarchy from a flat netlist by partitioning the network into channel-connected subnetworks. Next, they use a graph partitioning method similar to that used by *GEMINI*[15] to uniquely label the vertices of the graphs of each subnetwork. These labels are then sorted, yielding a quasi-canonical form for the subnetwork. A hash-code for the quasi-canonical form is computed and a hash table is examined; if the subnetwork in question is not in the table, it is compiled and the results are entered into the table. If the subnetwork does appear in the table, it need not be compiled again; it is simply replaced by the compiled procedure recorded in the table.

In [44], the authors present an incremental logic synthesis system. The input to the system is a logic description that is automatically converted into a gate representation from which a layout is synthesized. The layout is then optimized by hand. When the logic description is modified, however, the entire design is resynthesized, and the optimizations prior to the modification are lost. To avoid having to repeatedly optimize the entire design, the system attempts to preserve the parts of the layout that are unaffected by the changes and only resynthezises the modified parts. To accomplish this, the system compares the gate description of the modified and the unmodified designs in order to identify and preserve the gates whose connectivity remains unchanged. The comparison uses a partitioning algorithm to label each of the gates using such properties as the number of primary inputs, the number of inputs common to a pair of gates, the number of primary outputs, the gate fanout, etc. The labels are then refined using the

information of matched gates. After all gates have been labeled, gates with unique, identical labels are matched and preserved; unmatched gates in the unmodified description are deleted, unmatched gates in the modified description are added, and any inconsistent connections are adjusted. The authors mention nothing regarding automorphisms or ambiguous partitions; presumably they are considered unmatched. The program has two major drawbacks: first, it operates on the entire design; second, by restricting a match to gates that have identical labels, the comparison will fail to identify minor changes; for example, a gate whose output is connected to a different gate (which does not change the functionality of the gate) will not be matched by this process.

In [26] Jones presents a hierarchical schematic capture system that includes an incremental netlist compiler. The system works by mapping every modification to the schematic at any level of the hierarchy into manipulations of the netlist (component insertion or deletion, node connection or disconnection, etc). Because the schematic adheres to strict circuit boundaries, implementing this system is straightforward; any change to the schematic can be immediately recognized and need only be instantiated in its proper flat context as many times as the modified cell is used. As mentioned earlier, the problem with detecting the changes as the user modifies the design is that changes can be undone or cancel out one another. This system does not attempt to detect such changes, presumably because they are less common than when editing a layout.

To our knowledge, nobody has attempted to incrementally update a flat network by extracting only the modified areas of its corresponding hierarchical layout.

3.4 Incremental Extraction of Circuit Changes

This section describes the implementation of an incremental extractor designed to produce the network modification commands that will update the network following a layout modification. The extractor is integrated into the Magic layout editor and, unlike Magic's native extractor which operates hierarchically, it operates incrementally by confining its analysis to the modified portions of the layout. The result is a tool that can update the network in time proportional to the size of the changes, not the total size of the layout.

The key to fast extraction of circuit modifications is to examine only the modified

portions of the layout. To achieve this, the extractor keeps track of layout changes by recording the areas of the layout which are modified during an editing session. When the user finishes editing the layout, he calls on the extractor to update the network. The extractor then produces the network changes by examining the layout enclosed by the modified areas. This is done in three steps: First, the layout is topologically decomposed into its modified and unmodified regions by coalescing the previously recorded modified areas that interact with one another (either directly or hierarchically), thus yielding a set of disjoint modified regions, each of which is then analyzed independently. Next, the unmodified and the modified version of the subcircuit contained within each modified region is extracted from its corresponding layout. Finally, each pair of extracted subcircuits are compared, the differences are identified, and the required network modifications are produced. Although each modified region is analyzed independently, the extractor is able to maintain the correct relationship of every extracted subcircuit with the rest of the design by examining the boundary of the modified regions. When all the regions have been thus processed, the layout is cleared of all modified areas. At this point, the user may verify the design or continue editing the layout.

The next section describes how Magic's database is modified to keep track of layout changes. The following sections describe the extraction and comparison steps, including some of the more subtle problems that had to be solved.

3.4.1 Tracking Layout Changes

Tracking layout changes is accomplished by maintaining an additional *changed* plane with each Magic cell. This plane, which is invisible to the user, contains a series of *modified* tiles that indicate which areas of the cell have been modified during the editing session. Whenever the user changes a cell, a modified tile covering the area of the change is created and recorded in the cell's changed plane.

Implementing the modified-area recording mechanism is relatively easy since Magic uses a similar technique to perform incremental design-rule checking[48]. The only difference is that while DRC tiles are recorded in all of the modified cell's ancestors, modified tiles are only recorded in the changed cell itself. Keeping the changes local

to the modified cell minimizes the number of modified tiles needed during the editing session. It also better reflects the actual changes since modified tiles can easily be erased when changes are undone, an operation that would be extremely difficult if the modified tiles due to a single change were dispersed throughout the hierarchy.

With the exception of an additional command (to invoke the incremental extractor), the incremental extractor does not modify the user's view of the editor, nor does it impose any additional restrictions on the layout style. Furthermore, since recording the modified areas requires very little additional work, the overhead incurred by this mechanism is negligible. The operation of the incremental extractor is thus transparent to the user.

3.4.2 Coalescence of Modified Regions

Before extracting the modified portions of the layout, the extractor must determine the actual shape of the modifications. Once their shape is known, the layout can be decomposed into modified regions, that is, the layout can be regarded as a two-level hierarchy consisting of a series of modified cells contained within a single unmodified cell. The connections at the boundary of each modified region, which represent the interface between the unmodified cell and the circuit within the region, can then be used to analyze each region independently.

Since Magic allows almost arbitrary overlap, any cell may contain subcells that overlap other subcells or mask information; the modified areas of a cell can thus interact with the modified areas of other abutting or overlapping cells, thereby altering the shape of the modified areas. In addition, since a cell may appear any number of times in any other cells, the modified areas of a cell may interact in different ways with other modified areas, depending on the cell's context. Magic's near arbitrary overlap thus precludes context free identification of a modified region's shape. Instead, the extractor first flattens the hierarchical structure of the modified tiles into a dummy plane to produce a new corner-stitched plane that combines the modified information from all the cells in the design, as shown in Figure 3.8.

Once the modified areas have been flattened into a single plane, the shape of the modified regions is determined by coalescing all interacting tiles into a single region. Corner

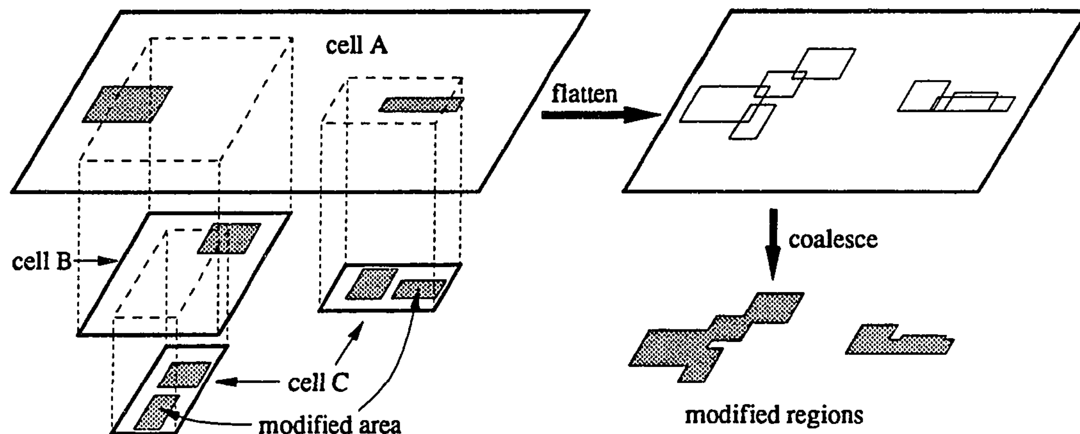


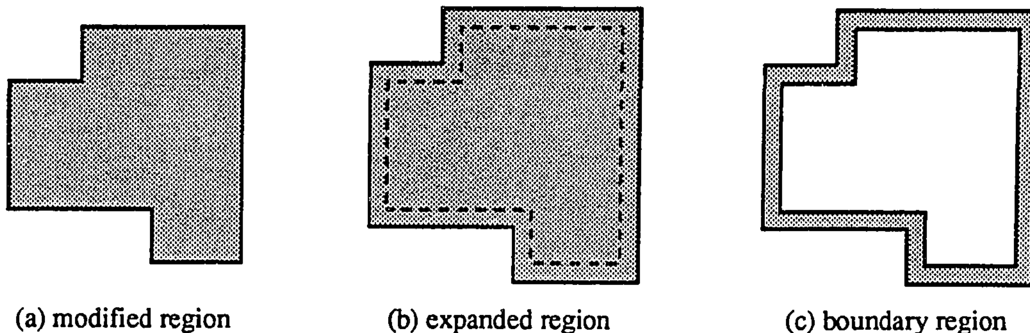
Figure 3.8: Modified areas are flattened and coalesced into disjoint modified regions.

stitching provides a simple mechanism to do this by following the stitches of adjacent modified tiles; this is similar to a node-finding algorithm except that it must consider tiles that touch only at the corners as being connected. When the region coalescing step is finished, all interacting modified-tiles will be grouped together into a single region, thus yielding a set of disjoint modified regions (Figure 3.8). These modified regions become the basic unit of analysis; subsequent steps process each of these regions one at a time.

3.4.3 Extracting the Modified Regions

For each modified region, the extractor produces the two circuits contained within the region: a *new* circuit, which is extracted from the modified layout being edited by the user, and an *old* circuit, which is extracted from the original, unmodified layout. To achieve this, the system maintains two design trees during incremental extraction: one that corresponds to the layout being edited by the user, and one that corresponds to the original, unmodified layout. The original design tree is built by reading, for each modified cell, the file holding the last version of the cell's layout. Maintaining two design trees allows quickly switching between the two by simply changing the pointer to the root cell of the design. Although this requires maintaining two copies of each cell, the memory overhead is minimized by allowing both design trees to share the mask planes of unmodified cells.

In addition to the old and new circuits, the extractor also produces the network at the boundary of the modified region. To determine what constitutes this network, the extractor computes a boundary region in the following way: First, the modified region is expanded by one lambda to produce an expanded region (Figure 3.9b). Next, the original modified region is erased from the expanded region to produce a one-lambda wide boundary region (Figure 3.9c).



(a) modified region (b) expanded region (c) boundary region

Figure 3.9: Determining the boundary of the modified region.

Once the boundary region has been determined, the layout encompassed by it is extracted to produce a set of boundary nodes. These nodes represent wires that connect the circuit within the modified region to the rest of the design; they can thus be regarded as *pins* or primary inputs to the circuit within the modified region. Since the boundary region lies completely outside the modifications, both old and new circuits share the same set of pins. These pins can, therefore, be used to establish a correspondence between the two circuits. To establish this correspondence, the extractor associates each pin with its corresponding nodes in each of the extracted circuits. Nodes that are associated with more than one pin represent connections between pins, and they are recorded by linking the interconnected pins to produce a pin-connectivity graph. Each pin p is processed once for each circuit in the following four steps:

1. Select an arbitrary tile which is part of pin p and locate its counterpart in the layout of the extracted circuit.

2. Find the node n that corresponds to the tile found above. When pins are processed, the circuit has already been extracted by marking each tile with the node to which

it belongs. Thus, finding the node that corresponds to a tile involves simply reading the node from the tile.

3. Check if n has already been marked as connected to another pin p_{other}. If so, add an edge from pin p to pin p_{other} in the current context (old or new); otherwise mark n as connected to pin p and mark pin p as connected to itself.

4. Bind p to node n in the appropriate (old or new) context.

After both circuits have been extracted and the pins have been processed by the above process, each pin will be bound to the node to which it is connected in the new and old circuits, each node connected to a pin will be marked with the corresponding pin, and the graph representing connections between pins in the new and old circuits will have been generated. Figure 3.10 illustrates a simple example in which a cmos nand gate is converted from a pseudo-nmos into a complementary implementation, including the layouts and diagrams corresponding to the design before and after the modifications (Figure 3.10a), the extracted parts of the layout (Figure 3.10b), and the resulting information (Figure 3.10c).

As shown in Figure 3.10, each extracted circuit corresponds to the layout enclosed by the modified region plus its boundary (i.e. the expanded region). By extracting the expanded rather than the modified region, the extractor does not have to detect abutting edges between the inside and outside of a modified region; this simplifies the node correlation and the computation of perimeter capacitance. Because the layout within the boundary region is identical in both circuits (old and new), the perimeter (and area) capacitance of boundary nodes is overestimated by exactly the same amount in both circuits. When the capacitance of the nodes is compared, the overestimated amount cancels out, thereby yielding the actual difference due to changes within the region.

Extracting the circuits contained within the modified regions hierarchically, in the same manner as Magic's extractor, would be extremely difficult. Because the modified region can be an arbitrary polygon, the layout of a cell and its interactions with other cells or subcells would have to be clipped to remain within this polygon. Also, since only parts of a cell may need to be extracted, and the parts may vary depending on the cell's context, extracting part of the cell hierarchically may be of little use since the extracted

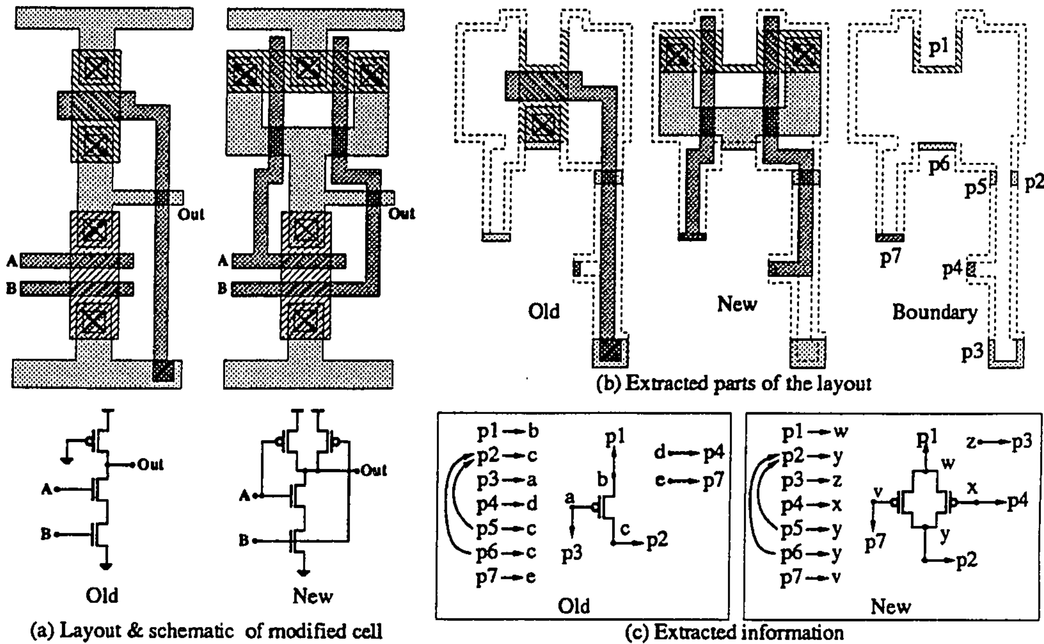


Figure 3.10: Incrementally extracting a simple layout change.

circuit cannot be reused. It is possible to devise a hierarchical system capable of storing the modified pieces of a cell and cross-registering the corresponding extracted circuits, but doing so would eliminate many of the benefits of a hierarchical database and would make the extractor much more complex. Instead, the incremental extractor flattens the hierarchical layout contained within the modified region and copies the clipped geometry into a dummy cell. Note that the boundary region does not have to be flattened; the extractor simply copies and clips the boundary portion of the layout from one of the flattened layouts into another dummy cell.

Transistors that intersect the modified region present a problem for the extractor. Since only part of the transistor is contained within the modified region, the extractor is unable to determine the transistor's size, location, or some of its terminals. A related problem is that changing a transistor's size results in only the enlarged (or shrunk) portion of the transistor being contained within the modified region (like **T1** in Figure 3.11a), which obscures the true nature of the change. A similar problem occurs when a transistor intersects the modified region more than once (like **T2** in Figure 3.11a); in this case each

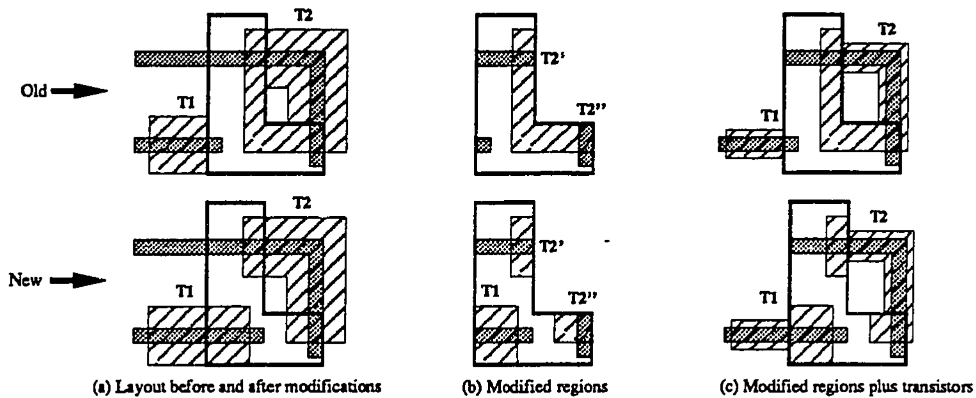


Figure 3.11: Transistors that intersect the modified region.

portion of the transistor might be incorrectly considered as a separate device.

For example, simply extracting the layout within the modified region of Figure 3.11b would incorrectly indicate that **T1** is a newly added transistor, and that a connection between **T2'** and **T2"** has been broken. To avoid all these problems, the extractor prevents partial transistors from being formed altogether. This is enforced during the flattening step by tracing and copying the entire structure of a transistor that intersects the modified region, as shown in Figure 3.11c. When tracing the outer part of an intersecting transistor, the amount of layout copied is confined to one lambda around the transistor; this limits the amount of extra layout and provides all the information needed to extract the transistor. Any additional nodes created by tracing intersecting transistors are prevented from being compared since they are outside the modified region and hence unmodified.

Once the hierarchy has been flattened into a single cell, the circuit network is extracted into a graph representation suitable for comparing its topology, which is described in the next section. The actual circuit extraction is done in a manner similar to Magic's basic circuit extractor; interested readers can refer to [43] for a detailed description of this process.

3.4.4 Comparing the Circuits

The previous sections described how the extractor keeps tracks of layout changes and how it uses that information to extract the circuits which correspond to the modified

portions of the layout. The remaining step is finding the transformations that convert the old circuit into the new one.

There are many combinations of transformations that will convert the old circuit into the new one, and several ways to find these transformations. The easiest approach is to assume that everything enclosed by the modified region has been altered, and simply replace the extracted subcircuit in its entirety. This approach requires no comparison and can quickly generate the transformations by simply removing the contents of the old circuit and, in its place, adding the contents of the new circuit. However, if only part of the circuit is different, which is not uncommon, this approach would result in needless changes to the network and hence unnecessary work to verify the changes. A more sophisticated approach is to generate the optimal set of transformations by finding the set of subcircuits that comprise the largest number of isomorphisms between the two extracted circuits. To find such a set of subcircuits, however, all possible isomorphisms between the two circuits would have to be examined; since this requires a solution to the very difficult (NP-complete) subgraph isomorphism problem, a practical implementation seems unlikely.

The approach we adopted represents a compromise between the two extremes outlined above. Rather than attempting to find all possible isomorphisms, the circuits are compared in order to find a single, trivial isomorphism. This is accomplished by comparing the extracted circuits, starting at the modified region's boundary and moving progressively towards its center, attempting, at each step, to match elements whose connectivity is the same in both circuits. If at any point during the comparison, an element cannot be matched (because it has no counterpart in the other circuit), an isomorphism is forced by adding or removing the unmatched element, the transformation is recorded, and the comparison continues. Thus, at the end of the comparison, both circuits will always be the same; if any elements were added or deleted, the transformations will have been recorded. If the circuits were in fact the same, no transformations will be recorded and none are produced.

The comparison is done in two steps: *Boundary Comparison* and *Circuit Comparison.* During the first step, the differences at the boundary of the two circuits are identified. Although these differences are the result of changes within the modified region, they

require special attention because they affect not only the extracted circuit but also the network outside the modified region. During the second step, the differences that arise from topological changes within the modified region are identified.

Boundary Comparison

To determine the changes enclosed by a modified region, the extracted circuits are compared independent of the context in which they appear: the circuits are treated as isolated cells that share a common set of pins, but whose contents may be different. However, the extracted circuits are not isolated from the rest of the layout and, therefore, some of the changes made within the modified region — such as shorting two boundary nodes — can alter the circuit outside the region. These changes can be detected by examining the connectivity at the boundary of the modified region, and their effects can be isolated by adjusting the connectivity of the circuits both inside and outside the region. Isolating connectivity changes at the boundary of the region allows the circuits to be compared independent of their context, thus making incremental extraction possible.

Connectivity changes at the boundary of the region also modify the extracted circuits' interface to the rest of the layout. This interface, which consists of nodes that interact across the boundary, is essentially the pins extracted from boundary region, and these pins are the same for both circuits. However, the pins are merely connection points to the outside of the modified region. The actual inputs to the circuits, and hence their interface, are the nodes that connect to those pins and, since the pins can be interconnected differently in each circuit, they need not be the same for both circuits. To allow context free comparison of the circuits, a uniform interface (i.e. the set of boundary nodes common to the two circuits) must be found.

The purpose of the boundary comparison is thus twofold: to provide both circuits with a uniform interface, and to isolate the changes that alter the circuit outside the modified region. Since boundary nodes exist both inside and outside the modified region, a change within the modified region does not create or destroy these nodes; it can only establish or break a connection between them. These changes can, therefore, be detected by examining the connectivity of the boundary nodes, and is easily implemented by comparing the previously obtained pin-connectivity graphs.

Since the pin-connectivity graphs are generated by processing the pins in the same order for both circuits, the edges representing connections between pins (see Figure 3.12) always point to the first pin to be processed of any interconnected set of pins (i.e. the arrows in Figure 3.12 all point to the left). This ordering guarantees that if the pins are compared by examining them in the same order as previously processed, then a pin will either point to itself or to a previously examined pin. This property allows the boundary comparison algorithm (shown in Figure 3.13) to detect all broken or new connections by examining the connections of each pin only once. A broken connection is detected when two pins are connected only in the old circuit, i.e., only the pin in the new circuit points to itself. Conversely, a new connection is detected when two pins are connected only in the new circuit.

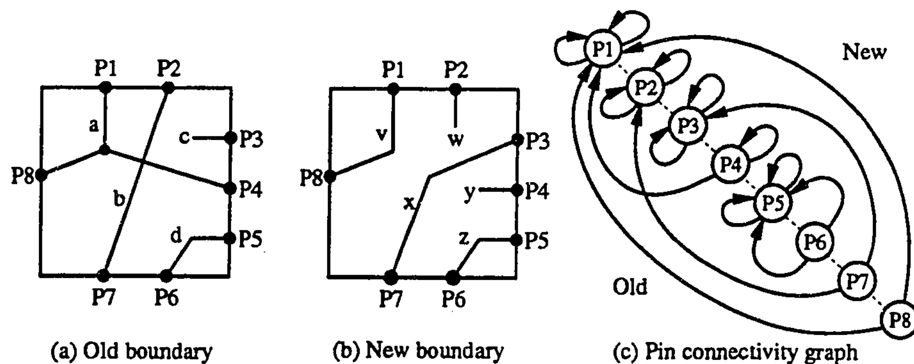


Figure 3.12: Comparing connectivity changes at the boundary of the circuit.

The algorithm of Figure 3.13 assigns a to every pin a label that is initially equal to the pin's position in the graph. Connectivity differences can then be found by comparing the labels of the pins to which a pin is connected in either circuit; if the labels are different (line 5 in Figure 3.13) then, after processing the difference, the two pins are assigned the same, lowest label (line 12 of Figure 3.13). For the purpose of the comparison, this has the effect of merging the two pins and prevents detecting the same difference more than once. For example, in Figure 3.12, the connections P1↔P4 and P4↔P8 are different, but since P1 and P8 are connected in both circuits, one of the differences is redundant and need not be processed again. The procedure "Merge" of Figure 3.13 takes the (old or new circuit) nodes that correspond to the two pins whose connectivity differ and combines

1. **for** $j = 1$ **to** N **do**
2. label(pin_j) $\leftarrow j$
3. $p_{old} \leftarrow$ pin to which pin_j is connected in the old circuit
4. $p_{new} \leftarrow$ pin to which pin_j is connected in the new circuit
5. **if** label(p_{old}) $\neq$ label(p_{new}) **then**
6. **if** label(p_{new}) $\neq$ label(pin_j) **then**
7. Issue(`connect` $pin_j \leftrightarrow p_{new}$)
8. Merge(old_node(pin_j), old_node(p_{new}))
9. **if** label(p_{old}) $\neq$ label(pin_j) **then**
10. Record(`break` $pin_j \leftrightarrow p_{old}$)
11. Merge(new_node(pin_j), new_node(p_{old}))
12. label(p_{old}) $\leftarrow$ label(p_{new}) $\leftarrow$ min(label(p_{old}), label(p_{new}))

Figure 3.13: Algorithm to process boundary connectivity changes.

them into a single node. When a new connection is detected (line 6 in Figure 3.13) the command to connect the corresponding nodes in the flat network is issued immediately, and the two nodes are merged in the old circuit (in which they are unconnected). When a broken connection is detected (line 9 in Figure 3.13), the two nodes are merged in the new circuit (as though they were still connected), however, the command to break the connection is not issued at this point but simply recorded for later processing.

After comparing the pin-connectivity graph using the algorithm of Figure 3.13, both circuits will have the same interface: the same number of inputs, each connected to the same unique pin. Since newly connected as well as disconnected nodes appear connected in the two circuits, they are both in the state in which they would be had no boundary nodes been disconnected. In fact, since the connect commands have been issued at this point, when applied to the flat network, it will also be in this same state. For example, when the above algorithm is used to compare the boundary of Figure 3.12, it will discover the following differences: {`break` P1 $\leftrightarrow$ P4}, {`break` P2 $\leftrightarrow$ P7}, {`connect` P3 $\leftrightarrow$ P7}, and modify the circuits as shown in Figure 3.14.

There are two reasons for postponing broken connections. First, in order to generate the same interface for both circuits, it is much simpler to merge the disconnected nodes

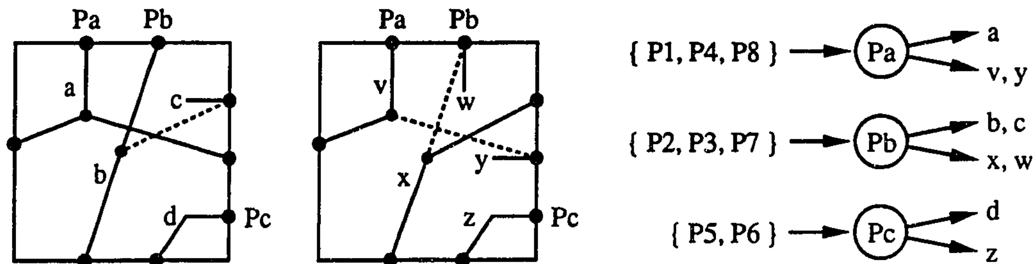
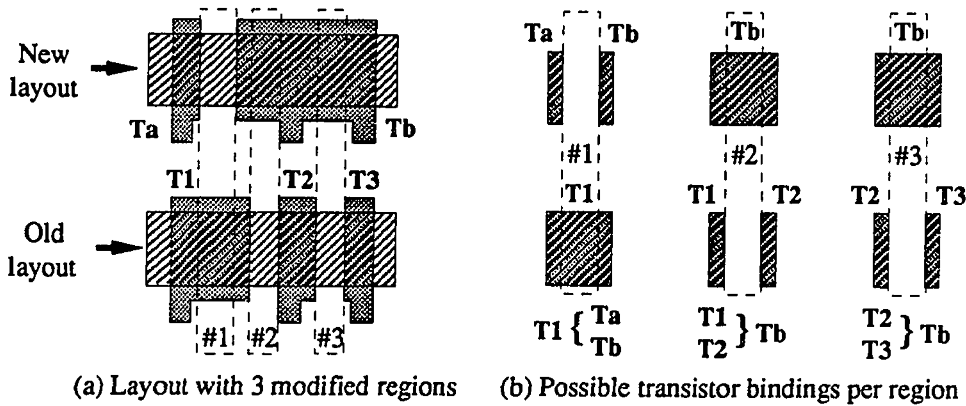


Figure 3.14: Result of comparing the boundary of Figure 3.12.

in the new circuit than it would be to split a node in the old circuit. To split a node, the exact point at which the connection was broken would have to be identified, which can be done by keeping track of every connecting wire within a node, but doing so would make the extractor much more complex (and slow). Second, processing a broken connection entails tracing each broken node (even outside the modified region) in order to determine how the transistors are redistributed along each broken branch of the original node. If a node is broken at multiple places (in different modified regions, for example), each break would cause the node to be re-traced. This redundant work is avoided by recording the broken connections while comparing the boundary and then processing all the recorded breaks in one fell swoop. Since connecting two nodes more than once is idempotent, new connections do not suffer from these problems and can thus be issued right away.

Transistors that intersect the boundary region of the two circuits can be correlated in much the same way as pins are used to correlate the boundary nodes of the two circuits. This can be done by binding the two transistors (one from each circuit) that cross the boundary at the same place. Matching boundary transistors thus maximizes the distinguishing features of the two circuits and simplifies the circuit comparison step. However, situations in which a boundary transistor appears in more than one region, like those of Figure 3.15, must be handled carefully.

The situation of Figure 3.15, in which a transistor itself is merged or split, presents a problem to geometrically matching boundary transistors. Because each modified region is processed independently and each region presents multiple choices to which a transistor can be matched (Figure 3.15b), the same transistor may be matched inconsistently



(a) Layout with 3 modified regions (b) Possible transistor bindings per region

Figure 3.15: Boundary transistors that appear in several modified regions.

throughout the various regions in which it appears. This would lead to incorrect or impossible transformations; for example, if while processing region #1, **T1** is arbitrarily bound to **Ta** then **Tb**, having no counterpart in that region, will be deleted. Then, when region #2 is processed, **T2** might be bound to **Tb**, which no longer exists, and proceed to add **T1**, which already exists. To avoid this type of error, a unique binding that is consistent among all the regions in which a boundary transistor appears must be obtained.

There are many ways to produce a unique, consistent binding among such ambiguous boundary-transistors. For example, the binding that retains the largest number of boundary transistors could be obtained by considering all their possible bindings before comparing the circuits. Doing this, however, would eliminate much of the advantage yielded by comparing each region independently and, since split or merged transistors are very infrequent, it is hardly worthwhile. Instead, the comparator tries to bind boundary transistors geometrically, but avoids inconsistent bindings by maintaining a table of previously encountered boundary transistors. Thus, when attempting to bind a boundary transistor, the table is examined; if the transistor is not in the table then the transistor is bound to its newly found counterpart and the binding is entered in the table. If the transistor is already in the table then the previously recorded binding is used. In addition to their bindings, the table is also used to record any transformations already applied to a boundary transistor — such as changing its size, for example — and prevents them from being reported more than once. This simple mechanism provides a quick method

for handling the most frequent case and, if the boundary information is ambiguous, it produces a consistent binding regardless of the order in which the regions are processed.

It is important to note that if the modified region contains no internal nodes (only boundary nodes and perhaps transistors), all the circuit changes will be detected by the boundary comparison, without having to compare the circuits any further. Since the complexity of the boundary comparison is $\mathcal{O}(p)$, p being the number of pins in the boundary, changes that alter only the connectivity of the design — such as re-routing part of the layout — can be identified very quickly.

3.4.5 Circuit Comparison

Once the boundary has been compared and adjusted to provide a uniform interface, the extracted circuits can be compared and the differences that arise from topological changes within the modified region can be identified. The circuit comparison algorithm (shown in Figure 3.16) is based on a graph isomorphism approach similar to those described in Section 3.3.2. It uses a graph partitioning heuristic to determine whether the two circuits are isomorphic; if they are not, it isolates the differences and produces the transformations needed to make them isomorphic.

1. $(N,T) \leftarrow$ PartitionCircuits(G_{old}, G_{new})
2. $N_{prev} \leftarrow \{\}$ $T_{prev} \leftarrow \{\}$
3. **repeat until** $N = \{\}$ **and** $T = \{\}$
4.   **if** $N \neq N_{prev}$ **then**
5. $T_{prev} \leftarrow T$
6. $T \leftarrow$ RefinePartitions(T)
7.   **if** $T \neq T_{prev}$ **then**
8. $N_{prev} \leftarrow N$
9. $N \leftarrow$ RefinePartitions(N)
10.   **if** $T = T_{prev}$ **and** $N = N_{prev}$ **then**
11. $T \leftarrow$ HeuristicMatch(T, N)

Figure 3.16: Circuit comparison algorithm.

The objective of the algorithm is to uniquely label each of the circuit elements (nodes

and transistors) by repeatedly subdividing the circuits into partitions of elements having the same connectivity until each partition contains no more than one element from the same circuit. To achieve this goal, each partition is assigned a unique label used to further refine the partitions. Since elements that have the same connectivity are found in the same partition, all elements within a particular partition are assigned the same label. If the circuits are the same, they can typically be divided into singleton partitions containing one element from each circuit; these elements represent exact matches (i.e. the same element in the two circuits) and can be bound to one another. Elements that cannot be thus matched represent circuit differences, which can be isolated by adding or removing the unmatched elements so as to force an isomorphism. These adjustments represent the desired transformations that convert one circuit into the other.

The circuit comparison algorithm can be broken into three steps: initial partitioning, refinement, and transformation generation. During the initial partitioning (line 1 in Figure 3.16) the boundary information is used to decompose the circuits into matched and unmatched elements. Nodes are partitioned by assigning the same unique label to each pair of boundary nodes that correspond to the same pin; all remaining nodes are placed in N, the set of unmatched node partitions. Likewise, transistors are partitioned by first assigning the same unique label to each pair of boundary transistors that were geometrically bound. Next, any remaining transistors are divided by forming a separate partition for each device type — one for n-devices, one for p-devices, etc — and placing these partitions in T, the set of unmatched transistor partitions.

Besides the boundary information, other graph invariants, such as the number of vertices or their degree, could be used to partition the circuit graphs into a finer initial partition. When the circuits are different, however, these invariants would cause the circuits to diverge much too early in the process. These differences would then propagate through the graph, thereby preventing the algorithm from discovering partially isomorphic subcircuits. This is avoided by using the connections to the boundary as the major means by which equivalent elements are detected.

After the initial partitioning, the algorithm enters its main loop and begins the refinement phase. In each pass, the refinement process is applied alternately to the transistor (line 6 in Figure 3.16) and the node partitions (line 9 in Figure 3.16) until both partitions

are empty. In the first iteration, the labels of boundary nodes are used to repartition T into smaller sets of equivalent transistors. The new transistor partitions are assigned unique labels which are then used to repartition N into smaller sets of equivalent nodes. The process is then iterated, this time using the new node labels in addition to the labels of boundary nodes. Thus, each iteration propagates the original boundary information, which naturally advances on all sides simultaneously, towards the topological center of the circuits. As the refinement progresses and elements become unique, they are removed from their corresponding unmatched set and either matched, added, or deleted so as to keep the two circuits the same. The process thus resembles a zipper moving spirally from the boundary towards the center of the circuits, closing, at each step, any differences between the two circuits.

If the circuits contain one or more automorphisms, there will be partitions that cannot be subdivided beyond the set of equivalent elements. When the algorithm detects that the partitions remain constant from one pass to the next (line 10 in Figure 3.16), it uses a different heuristic to refine the transistor partitions. Since automorphic transistors can be permuted without altering the graph, the "HeuristicMatch" function selects the smallest transistor-partition and arbitrarily matches two of its transistors based on other information such as device size and layout location. The heuristic breaks only the smallest set of ambiguous transistors so as to use the least non-topological information to continue partitioning the graphs. Because the partitioning uses only the boundary information, it is unlikely for the heuristic to match non-equivalent transistors; even if it were to do so, however, this is not a fatal condition since an isomorphism will be enforced, producing, at worst, some unnecessary transformations. Note, however, that if "HeuristicMatch" is called with an empty node partition ($N = \{\}$), then all nodes have already been matched and a transistor automorphism is assured[2]. In this case, rather than matching only two transistors, the function uses the size, connectivity, and location information to match all transistors in T in one fell swoop.

The refinement process is shown in Figure 3.17, and consists of three basic steps: re-partitioning, matching, and adjustment. In the first step, the elements of a partition are

[2]This situation is not uncommon; to adhere to spacing constraints, designers sometimes implement very wide devices by laying out several smaller transistors connected as a single transistor, hence creating a series of indistinguishable, automorphic transistors.

subdivided into partitions having the same connectivity. This is efficiently accomplished by computing a connectivity signature for each element in the partition, and then grouping the elements according to their signature. The signature of an element is a function of the labels of the elements to which it is connected and the type of terminal (gate, source, or drain) associated with the connection:

$$S_e = \mathcal{F}(\ \mathcal{G}(label_1,\ conn_type_1),\ \mathcal{G}(label_2, conn_type_2)),\ \cdots\ )$$

Before assigning an element to a partition, its connectivity must be compared with that of the partition. This is because the signatures are essentially hashing codes, hence, elements with different connectivity may sometimes (although unlikely) be assigned the same signature. Since a transistor's source and drain are symmetrical, neither the signature nor the connectivity comparison distinguish between these two terminals.

After the partition has been subdivided into new partitions, the matching step examines the size of each new partition. Partitions containing one element from each of the two circuits (line 11 in Figure 3.17) represent an exact match; the two elements are bound to one another and assigned the same unique label. Partitions that contain only elements from the new circuit (line 7 in Figure 3.17) represent possibly newly added elements. Conversely, partitions that contain only elements from the old circuit (line 9 in Figure 3.17) represent possibly deleted elements. Both these elements are placed in their respective differing element set, which are examined at the end of the matching step. Any other partition containing elements from both circuits remain in the set of unmatched elements (N or T) and will be considered in the next iteration.

During the adjustment step of the refinement process, the two sets of differing elements are considered. Since these elements cannot be directly matched to any counterparts in the other circuit, the algorithm could simply delete all elements in $diff_{old}$ and add all elements in $diff_{new}$. However, the connectivity of some of these elements may differ only because they were slightly modified. Consider, for example, a node that contains several identically connected transistors yet its connectivity was modified by disconnecting a single transistor. If such a node were to be considered unmatchable, then that node along with all its connected transistors would have to be deleted and added as required. Furthermore, the small difference would then propagate through the transistors, resulting

```
1.  function RefinePartitions( S )
2.      S_next ← {}
3.      for j = 1 to |S| do
4.          V ← new partitions of S_j having the same connectivity
5.          diff_old ← {}    diff_new ← {}
6.          foreach new partition p in V do
7.              if |p.old| = 0 then
8.                  diff_new ← diff_new ∪ {p.new}
9.              else if |p.new| = 0 then
10.                 diff_old ← diff_old ∪ {p.old}
11.             else if |p.new| = 1 and |p.old| = 1 then
12.                 match p.new ↔ p.old
13.                 assign same unique label to p.new and p.old
14.             else
15.                 assign same unique label to all elements of p
16.                 S_next ← S_next ∪ {p}
17.         foreach element a in diff_new do
18.             d ← element in diff_old for which δ(a, d) is minimum
19.             if δ(a, d) < δ_max then
20.                 diff_old ← diff_old − {d}
21.                 match a ↔ d
22.                 assign same unique label to a and d
23.             else
24.                 assign unique label to a
25.                 added ← added ∪ {a}
26.         foreach remaining element d in diff_old do
27.             assign unique label to d
28.             deleted ← deleted ∪ {d}
29.     return( S_next )
```

Figure 3.17: Function to refine the partitions.

in a large portion of the circuit being unmatched. The algorithm, therefore, avoids this type of situation by trying to match the elements of the two differing sets ($diff_{old}$ and $diff_{new}$) before deciding that an element cannot be matched. It does this by finding the pair of elements for which $\delta(a,\ d)$ is a minimum; this function is a measure of the difference between possibly added element a and possibly deleted element d, and is implemented as the ratio of differing connections (adjacent elements with unequal labels) to the total number of connections to a and d (the larger of the two). If a and d are only slightly different (line 19 in Figure 3.17) then they are matched to one another and assigned the same label so as to eliminate the difference. The constant δ_{max}, which represents the maximum tolerable difference, is the maximum $\delta(a,\ d)$ for which elements a and d will be considered matchable (in the current implementation $\delta_{max} = \frac{1}{2}$). This final matching increases the execution time of the comparison, but allows the binding of elements with similar but not identical connectivity. If this type of binding were disallowed, any small difference could propagate through the circuit and prevent identification of other unmodified subcircuits.

At each iteration in the refinement process, any element may become uniquely labeled but, since the partitions are refined using only the labels of adjacent elements, those most likely to be labeled uniquely are the ones connected to elements already labeled uniquely, which are typically closer to the boundary. Because of this, the adjustment step only considers the differing elements that result from a single partition, not all differing elements. This not only limits the number of elements that must be matched at a particular step, but is a necessary condition for the convergence of the algorithm, since the labels assigned to added, deleted, or matched elements during previous iterations have already been used to further partition the circuits. The argument in favor of this approach is that if a differing element could not be matched with any other differing elements from its own partition (which are typically at the same topological distance from the boundary), then it is unlikely to be matched with a differing element from a partition whose elements are at a different topological distance from the boundary. This can be best understood by considering the initial transistor partitions: if, for example, a p-transistor could not be matched with any other p-transistor (in its own partition) then it would make no sense to try to match it with a transistor from the n-transistor partition. Although a more

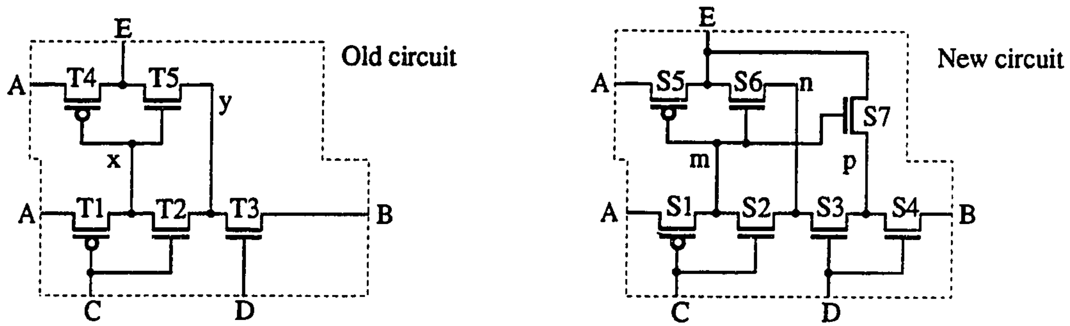
sophisticated matching algorithm that considers all possible matchings, such as those described in [29], could have been used, the advantages yielded by such methods might be negated by the additional overhead. Furthermore, since the circuits are expected to be different, it is not clear that a more sophisticated algorithm would produce a better match. The current implementation is simple, fast, and has been adequate for all our test cases.

When the refinement process finishes, it will have produced three sets of nodes and transistors: *matched, added,* and *deleted.* The final step of the comparison, transformation generation, examines these sets and generates the corresponding network modification commands. The sets are examined in the following order: *added-nodes, deleted-transistors, matched-transistors, added-transistors, deleted-nodes,* and *matched-nodes.* This order guarantees that an added node has already been created before connecting any transistors to it, and that a node is deleted once there are no more transistors connected to it. For added or deleted elements, no additional processing is needed; the corresponding command is simply issued. Matched elements, on the other hand, need to be compared with their counterparts to determine if any of their features have changed, as described below.

For matched nodes, their parasitic capacitance and hierarchical name are compared; if either is different, the appropriate command is issued. For matched transistors, their size, layout location, and terminal connections are compared; if either is different, the appropriate command is issued. Note that some of these comparisons do not represent electrical changes, such as the name of a node or the swapping of a transistor's source-drain terminals; they are nonetheless needed to maintain the flat network consistent with its layout.

A simple example of the circuit comparison process is illustrated in Figure 3.18. In the illustration, the greek letters represent the unique labels assigned by the comparison algorithm; the tables on the left side show the elements being compared along with their signatures, which are formed by concatenating the labels of their neighboring elements, and use the following syntax: "$label_{gate} \ldots / label_{source,drain} \ldots$".

In the initial partition, every pin is assigned a unique label, and the remaining nodes (x, y, m, n, and p) are assigned label ϕ and placed in the same node partition. Transistors



Initial Partition:

pins = { A B C D E }
↑ ↑ ↑ ↑ ↑
α β χ δ ε

N = { x y m n p }
m ↑ ϕ

T = { {T1 T4 S1 S5} {T2 T3 T5 S2 S3 S4 S6 S7} }

1⁰ Transistor Refinement:

T1	T4	S1	S5
χ/αϕ	ϕ/αε	χ/αϕ	ϕ/αε

T2	T3	T5	S2	S3	S4	S6	S7
χ/ϕϕ	δ/βϕ	ϕ/εϕ	χ/ϕϕ	δ/ϕϕ	δ/βϕ	ϕ/εϕ	ϕ/εϕ

matched
T1↔S1 ←π
T4↔S5 ←κ
T2↔S2 ←λ
T3↔S4 ←μ

added
S3 ←θ

T = { T5 S6 S7 } ←ρ

1⁰ Node Refinement:

x	y	m	n	p
κρ/πλ	/λμρ	κρρ/πλ	/λθρ	/θμρ

matched
x↔m ←τ
y↔n ←υ

added
p ←ω

N = { }

2⁰ Transistor Refinement:

T5	S6	S7
τ/ευ	τ/ευ	τ/εω

matched
T5↔S6 ←ξ

added
S7 ←ψ

T = { }

Transformation Generation:

added–nodes → **create new node** p

added–transistors → **add transistor** S3 (g=D s=y d=p)

add transistor S7 (g=x s=A d=p)

matched–transistors:

T1	S1
χ/ατ	χ/ατ

T2	S2
χ/τυ	χ/τυ

T5	S6
τ/ευ	τ/ευ

T4	S5
τ/αε	τ/αε

T3	S4
δ/βυ	δ/βω

→ **move source(** T3 **) from** x **to** p

Figure 3.18: A circuit comparison example.

are divided according to type, and placed in the corresponding partition.

In the first iteration, four pairs of transistors have identical unique signatures (T1-S1, T2-S2, T3-S4, and T4-S5) and can thus be matched directly. Of the remaining transistors, T5, S6, and S7 have the same signature, which results in a new partition. This leaves S3 as the sole differing transistor, so it is placed in the set of added transistors. The new labels assigned to the transistors are then used to refine the node partition N. This partitioning, however, does not yield any exact matches (identical pairs of signatures) so the algorithm tries to match $diff_{old}$ ={x y} with $diff_{new}$ ={m n p}. Since $\delta(\text{x, m}) = \frac{1}{5}$, nodes x and m are very similar and can be matched. Likewise, y and n are matched since $\delta(\text{y, n}) = \frac{1}{3}$. Note, however, that since $\delta(y,\ p) = \frac{1}{3}$, the algorithm could just as well have matched these two nodes. It is not difficult to verify that had it done so, the next transistor refinement would have matched T5 with S7 instead of S6, yielding an equivalent set of transformations. After matching node x with m, and node y with n, a single unmatched node (p) remains in the partition so it is placed in the set of added nodes. At this point the node partition (N) becomes empty.

In the second iteration, transistors T5 and S6 have identical signatures and can be directly matched. This results in S7 to be added to the set of added transistors, and leaves the transistor partition (T) empty. During the transformation generation step, commands are generated to create new nodes for every node in the added-node set (node p), and new transistors for every transistor in the added-transistor set (S3 and S7). Finally, the signatures of every pair of matched transistors are compared; since transistors T3 and S4 differ, a command must be issued to move one of its terminals so as to leave both transistors with identical signatures.

3.5 Broken Connections

The final step of the incremental extraction is to process the broken connections at the boundary of the modified regions. As mentioned in Section 3.4.4, these modifications are simply recorded when comparing the circuits, their actual processing and the issuing of the corresponding network modification commands is postponed until all the modified regions have been compared.

Each broken connection record specifies the layout location of each of the two nodes to be split and the difference in capacitance between the original node and the two split nodes (taken as a single node). This capacitance is needed because a node may appear to have been broken in some modified region, yet still be connected in some other section of the layout. If this is the case then the node is not split, but its capacitance may still need to be adjusted.

Processing a broken connection entails tracing the two nodes in their entirety, extracting, for each node, the transistors connected to it, its parasitic capacitance, and its name. Since a node may be broken into more than two nodes, some of its new branches may have been recorded several times. To avoid tracing these branches more than once, the broken connections are processes in two steps. In the first step, all nodes recorded as broken are traced, marking every tile that belongs to a node with the corresponding node. Before tracing a node, these marks are examined to determine whether the node has already been traced. In the second step, the resulting nodes are examined, the differences are determined, and the corresponding network modification commands are issued. Since the layout is hierarchical, a node consists of one or more hierarchical nodes interconnected by tiles that abut or overlap connecting tiles of other cells. To trace a node, all its hierarchical nodes need to be found. This is done by finding an initial tile that corresponds to the broken node's recorded location; the hierarchical node that corresponds to the tile is extracted and placed into a pending hierarchical node list. Hierarchical nodes are then processed one by one until none remain in the list. Each hierarchical node is processed in five steps:

1. Check to see if this hierarchical node has already been extracted. If it has then use the extracted information in the following steps; otherwise trace the node at this level of the hierarchy to obtain its parasitic capacitance, the transistors to which it is connected, and its hierarchical name. While tracing the node, every tile connected to the node is marked with the corresponding node.

2. Add this hierarchical node to the list of nodes composing the overall node, add the capacitance of this hierarchical node to the overall node's capacitance, and increment the transistor count of the node by the number of transistors in this

hierarchical node.

3. If this hierarchical name is a "better" name than the previously obtained name for the node then set the name of the overall node to this hierarchical name. The following rules determine what constitutes a better name[3]:

   Choose: a user-defined name over a machine-generated one.

   Otherwise the name with the shortest hierarchical path.

   Otherwise the shorter name.

   Otherwise the smaller lexicographical name.

4. Search other cells that abut or overlap the current cell looking for tiles electrically connected to any of the tiles in this hierarchical node.

5. For each tile found above, determine the tile's hierarchical node (as in 1 above). If the hierarchical node is not yet part of this node then add that node to the list of pending hierarchical nodes. In addition, adjust the overall node's capacitance by subtracting the capacitance overestimated by the hierarchically overlapping or abutting tiles.

After tracing every broken node, all nodes that were broken from a single original node are grouped together. Each such group is then examined to determine the modifications, as follows: if the node consists of a single node then the node was not broken and only its capacitance (and perhaps its name) may have to be changed. Otherwise, find from the N broken nodes, the node with the largest number of connections N_{max}. For N_{max}, issue only the commands to set the node's new name and capacitance, if needed. For the remaining $N-1$ nodes, issue first the command that creates a new node N_i; then, for each transistor terminal connected to the new node issue a command to reconnect the transistor's terminal from node N_{max} to node N_i.

[3]In a hierarchical layout, a node will have as many different names as the cells in which it appears. In order to limit the amount of memory needed to store all such names, when Magic flattens the network, it produces a single, unique name for every node. The rules described above will generate the same name as Magic's circuit flattener.

3.6 Name Resolution

To apply the transformations, the network modification commands must address existing nodes and transistors in the network. This requires finding for every element in the network a unique name that is understood by the tool that will process the commands. For transistors this poses no particular problem since their layout location is unique and easily established. For nodes, however, obtaining their names can be quite involved as it requires tracing all hierarchical nodes that make up the node, a process similar to the one used to process broken connections. If this process were to be applied for every command that addresses an existing node, the extractor would spend much of the time tracing nodes throughout the layout. This is particularly critical for very long nodes that traverse much of the layout, such as clocks or power supplies.

This limitation can be easily overcome by addressing a node using any of the transistors to which the node is connected. A node can thus be addressed using the location of any transistor to which the node is connected, and the type of transistor terminal associated with the connection. This still requires tracing the hierarchical nodes, but the process can be stopped as soon as a transistor is found, and typically requires tracing only one cell. Furthermore, since the extracted circuits already contain some transistors, these transistors can themselves be used to address the nodes to which a transformation is to be applied, without having to trace any part of the layout.

Since the modified regions are processed one at a time, some of the transistor locations or terminal connections may have already been modified by the transformations due to previously processed regions. This creates a problem for addressing nodes, since the transistor properties used to address a node (as found in the old layout) may have been invalidated by some previous transformation. Note that this same problem exists even if the actual node names were to be used, since they can also be altered. To establish which properties are no longer valid, the extractor would have to determine how previously generated transformations affect the elements of other regions, which, unfortunately, requires knowledge about the entire network. This problem is solved by recognizing that the flat network is already being maintained by the verification tool that applies the transformations. Thus, before issuing the transformations, the extractor issues a series of

commands to associate a simple label (a number) with every node that will be addressed by some transformation. These labels provide a handle to existing nodes (in the old network), regardless of how the node is later modified by the transformations. After all regions have been processed, but before processing the broken connections, the extractor issues the network modification commands using the previously generated labels instead of the node names. A detailed description of the network modification commands is given in Appendix B.

3.7 Performance

A meaningful analysis of the complexity of the incremental extractor is difficult, for two reasons. First, each step of the process depends on very different types of information, so there is no single metric upon which to base the analysis. For example, layout flattening and circuit extraction depend primarily on the number of tiles found in the modified region, pin extraction and comparison depend on the shape and size of the boundary region, and circuit comparison depends on the size and topology of the extracted circuits as well as their differences. Second, although worst case time bounds for some of these operations can be derived, they are not particularly meaningful since worst-case behavior never occurs in practice. For example, a worst-case time bound of $\mathcal{O}(n^2 + t^2)$ for comparing two circuits with n nodes and t transistors is not difficult to derive, but it never occurs in real circuits; typically, unique labels propagate quickly through the circuits, allowing the comparison to be performed in linear time.

To analyze the performance of the incremental extractor, we carried out several experiments on the layout of the cache controller from the MIPS-X microprocessor[21]. We chose this particular layout because of its nearly-constant high density — a sizable part of the layout consists of various types of very compact memory cells such as associative memory, valid bits, etc. Since the number of both tiles and transistors per unit of area is almost constant throughout the layout, the performance of the incremental extractor can be characterized by the area of the modified regions.

To analyze the impact that layout and circuit sizes have on the speed of the extractor, we forced the extractor to incrementally extract differently sized portions of the layout by

marking several areas of the root cell as modified but without actually altering the layout. Since the layout was unchanged, none of the tests produced any modifications. Table 3.1 lists the sizes of the extracted circuits and the running times of the various operations. The rightmost column shows the time required to extract and flatten the entire circuit using Magic's hierarchical extractor. The bottom row gives the ratio between the time for incremental extraction and the time for hierarchical extraction plus flattening[4]. The times shown in Table 3.1 are plotted in Figure 3.19.

As shown in Table 3.1, the correlation between layout area and circuit size is reasonably good; it breaks down only for the largest areas (> 60%). This is because these larger areas contain parts of the less densely populated control circuitry that is located on the periphery of the layout. The data shows that until the changes comprise 40% of the layout area, incremental extraction has a clear and consistent advantage over hierarchical extraction plus flattening. For larger layouts, we expect the advantage to increase since the changes remain relatively small while the unmodified section grows larger. Note that even for the largest area, which contains 93% of the circuit, incremental extraction took only twice as long as the conventional extractor. This is hardly surprising since the incremental extractor has to flatten and extract the circuit twice and, in addition, compare the two rather large circuits.

An important observation is that the runtime of each of the major steps of the incremental extractor depend linearly on the size of the modified area (see Figure 3.19). Figure 3.20 shows the fraction of time required to perform each operation of the incremental extraction, as a function of the size of the modified area. For small areas, the time is dominated by reading the modified cells and building the old version of the design tree. This operation takes the same small amount of time (about 40ms) in every test since all of them built the same design tree in which only the root cell had to be read in. For changes smaller than 0.5% of the total area (those with less than 50 transistors), the circuit comparison requires very little time because the extracted circuits consist almost exclusively of boundary elements. This is no longer true for the test that extracts 0.5% of the area so the circuit comparison time shows a marked increase, taking roughly half

[4]To make a fair comparison, Magic's flattening time does not include the time to write the flat netlist, only the time to read and flatten the extracted circuit hierarchy.

		Incremental Extractor									Magic
% of Area Modified		0.02	0.1	0.5	5.0	15.0	30.0	45.0	60.0	75.0	100.0
Transistors		1	2	19	586	2230	4711	7420	9628	12008	13014
Nodes		4	6	30	342	1098	2178	3388	4236	5145	5425
% of Circuit		0.02	0.04	0.26	5.0	18.0	37.4	58.6	75.2	93.0	100.0
Boundary Nodes		4	6	27	132	290	362	520	539	387	-
Time (ms)	Tree Build	39	43	43	47	43	39	43	36	43	-
	Region Coalesce	4	2	2	2	3	2	1	4	4	-
	Flatten	4	20	113	2821	10468	22624	36013	46546	58730	21100
	Extract	11	11	71	1805	6839	15186	24084	32928	43259	40600
	Pin Compare	4	4	7	73	247	398	676	731	711	-
	Circuit Compare	8	4	230	1250	3351	5898	9370	11573	19764	-
	Total	70	84	466	5998	20951	44147	70187	91818	122511	61700
Speedup		881	735	132	10	3	1.4	0.88	0.67	0.5	1.0

Table 3.1: Incremental extraction times for various modified areas

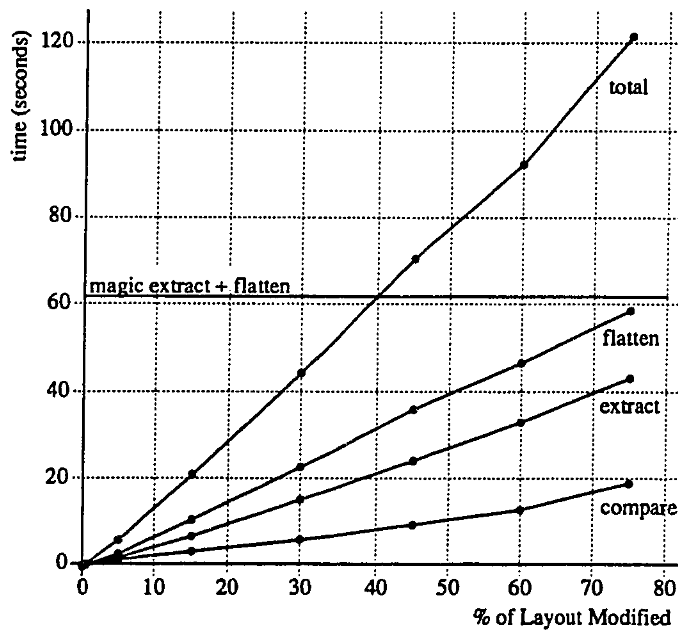


Figure 3.19: Incremental extraction running times as a function of area size.

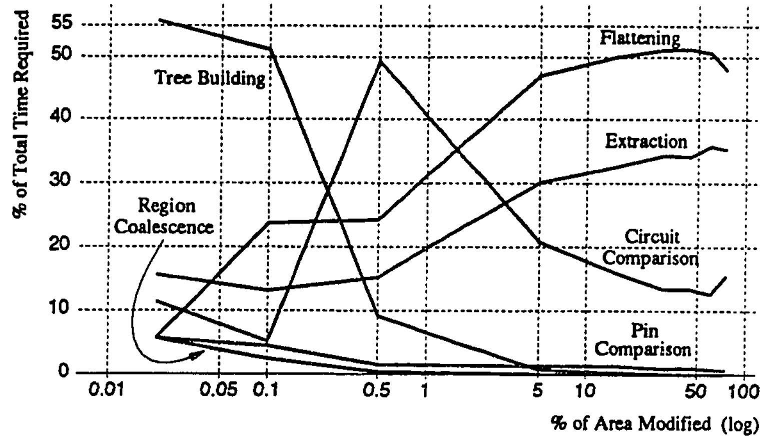


Figure 3.20: Relative Incremental extraction running times.

of the total time. However, as the size of the modified area increases beyond 0.5%, the time required by the circuit comparison decreases and remains fairly constant at about 12%–15%. For large areas, the extraction time is dominated by flattening the modified regions; all the remaining steps take about half of the time. This suggests that substantial speed improvements are possible by extracting the modified regions hierarchically. Doing this is not an easy problem, however. As described in Section 3.4.3, when arbitrary cell overlaps are allowed, the modifications within a cell interact with the modifications of other overlapping cells. The circuit that corresponds to the modified region and its boundary cannot be easily determined by examining only the modified areas within a cell.

Except for circuit comparison and processing of broken connections, which depend on the circuits' topology, all other operations performed by the incremental extractor depend primarily on the size of the modified region and the density of the layout within it. Given the near-constant high density of the layout from which Figure 3.19 is obtained, we can expect the time requirements of these operations to depend linearly on the size of the modifications.

To compare the speed of the incremental extractor with Magic's incremental, hierarchical extractor, we performed a second set of tests in which the layout was subjected to

various random changes. Since Magic is a hierarchical editor, design changes were introduced at different levels of the hierarchy; thus, Magic's extractor only had to re-extract the modified cells and their ancestors. Also, because some of the modified cells were instantiated multiple times, a single change sometimes resulted in multiple modifications to the flat network. A change was generated by selecting an arbitrary cell and randomly applying one of the following changes:

1. Delete a single transistor and short the nodes connected to its source-drain terminals.
2. Add a new transistor between three existing nodes.
3. Add a new transistor, delete another transistor, and short two nodes.
4. Break a connection between two nodes and reconnect them to some other nodes.
5. Add two transistors, delete another two transistors, and reconnect their source, drain, and gate terminals to some other nodes.
6. Merge two transistors and short their gate nodes, remove another transistor and reconnect the nodes connected to its source-drain to another pair of nodes.
7. Split a node and add a CMOS inverter between the two broken wires.
8. Delete a CMOS inverter and short the nodes that were connected to its input and output.

After applying each of the above changes, the network was updated using the two extractors: the incremental extractor to generate the corresponding modifications to the flat netlist, and Magic's hierarchical extractor to extract the modified cells (and their ancestors) and flatten the hierarchy into a flat netlist. Table 3.2 gives the circuit sizes and running times for the two extractors[5] . In the table, "Hierarchical Speedup" is the ratio between the time to extract the modified cells plus the time to flatten the hierarchy, and the time to extract all the cells plus the time to flatten the hierarchy; "Incremental Speedup" is the ratio between the time to incrementally extract the modifications and the

[5]The circuit comparison times seem relatively higher here because they include the time to determine the actual node names.

		Test	1	2	3	4	5	6	7	8
MAGIC		Cells Extracted	7	4	6	8	4	5	5	5
		Transistors Extracted	5	50	45	20	64	8	50	50
	Time (ms)	Extract	32100	17800	21000	21800	19400	25400	17800	17800
		Flatten	18900	19400	19200	19400	19500	19400	19400	19400
		Output Netlist	5610	4830	5820	6850	6110	4230	6800	6800
		Total	56610	42030	46020	48050	45010	50030	44000	44000
		Hierarchical Speedup	1.21	1.63	1.49	1.42	1.52	1.37	1.55	1.55
INCREMENTAL		% of Area Examined	0.24	0.01	0.02	0.01	0.01	0.03	0.02	0.02
		Number of Transistors	512	1	16	0	3	48	6	6
		Number of Nodes	5632	11	96	448	7	336	11	11
		Number of Changes	1536	3	40	32	8	96	4	6
	Time (ms)	Tree Build	50	180	109	86	98	43	46	47
		Region Coalesce	75	2	4	11	4	8	4	3
		Flatten	2477	7	103	255	19	309	12	11
		Extract	690	3	12	23	10	76	12	12
		Pin Compare	1117	4	40	81	4	77	4	8
		Circuit Compare	7931	133	806	1038	801	287	12	35
		Broken Nodes	0	0	0	1895	0	0	31	0
		Output Changes	43	1	4	15	1	12	4	1
		Total	12383	330	1078	3404	937	812	125	117
		Incremental Speedup	4.6	127.4	42.7	14.1	48.0	61.6	352.0	376.1

Table 3.2: Comparison of incremental and hierarchical extraction plus flattening.

time to extract only the modified cells plus the time to flatten the hierarchy (the "Total" row in the Magic section).

Magic's hierarchical extractor is effective in reducing the number of extracted cells to only a few (out of a total of 64), each containing a relatively small number of transistors. However, since it always needs to extract the root cell, the extractor spends most of the time computing interactions between cells and subcells, even though most of them were not touched by the modifications. Furthermore, the time to flatten the hierarchy almost negates the speedup gained by hierarchical extraction; the time to generate the flat netlist by extracting only the modified cells is only marginally better than extracting all the cells

in the design. The incremental extractor, on the other hand, is about 60% slower than Magic's extractor when analyzing the same area; nevertheless, because it only examines a very small portion of the layout, it can reduce the time to update the flat network an average of 128 times. With the exception of test 1, all other extractions took less than four seconds, providing almost immediate feedback. Even test 1, which modifies the basic memory cell used many times throughout the circuit, is still 4.6 times faster than hierarchical extraction.

Although the changes applied in the experiment are not real modifications in the sense that they correct no errors, nevertheless, they are representative of the types of small changes and adjustments that are likely to occur during the design. Note that since incremental extraction only examines an area proportional to the size of the change, a similar change will take the same amount of time regardless of the design size. For example, the time to update the network following a change to add an inverter (as in test 7) will be roughly 0.12 seconds on any design. The only operation that may, in the worst case, depend on the size of the overall design is processing broken connections, since it may have to traverse the entire layout. Even in such a situation, however, incremental extraction may be no worse than hierarchical extraction plus flattening. Chapter 4, which presents a more realistic test of the capabilities of the incremental extractor, analyzes one such situation.

To determine the impact that circuit differences may have on the speed of the circuit comparison, we incrementally extracted the same non-trivial portion of the layout with and without modifications. The modified layout was subjected to the same changes as in the previous test but, in addition, the entire area of the modified cell was marked as modified. For the unmodified version, the area of the corresponding cells were marked as modified but no changes were applied. Marking the entire area of a cell sometimes resulted in a modified region much larger than the cell itself; this occurs when different instances of a cell overlap or abut one another, as in the memory arrays. Table 3.3 shows the circuit sizes and the time taken by the circuit comparison of the unmodified (same) and modified (different) layouts. In the table, "Pins" is the number of boundary nodes, and "Changes Found" is the number of network modification commands generated when comparing the differing circuits.

Test	Old Circuit		New Circuit		Pins	Changes Found	Time (ms)		Variance (slowdown)
	Fets	Nodes	Fets	Nodes			Same	Different	
1	2560	2160	2048	1648	1136	4224	3523	3531	0.23%
2	51	60	52	61	39	3	512	512	0.00%
3	72	160	72	152	128	48	1481	1329	-10.26%
4	640	387	640	387	131	256	372	375	0.81%
5	208	137	207	137	57	2	2699	2859	5.93%
6	97	162	49	114	101	157	901	610	-32.30%
7	8	7	10	8	4	11	3	3	0.00%
8	10	10	8	9	5	7	4	4	0.00%

Table 3.3: Circuit comparison running times for the same and different circuits.

The degree to which circuit differences slow down the execution of the circuit comparison is minimal. Moreover, when the circuits differ significantly, the comparison can be even faster than when the circuits are identical. In fact, the tests for which the runtime variance was greatest were those which ran faster when the circuits were different. The reason for this is that in those two cases (tests 3 and 6), the new circuit was smaller than the old circuit so the comparison converged faster, i.e., less iterations were needed to partition the circuits. The first experiment showed that the time for comparing two identical circuits depends linearly on the size of the circuit; the last experiment confirmed that this is also true for circuits that differ. Thus, the incremental extractor is capable of updating the network in time proportional to the size of the modifications.

Chapter 4

Results

The previous chapters described how to incrementally update the information produced by circuit extraction and logic, timing simulation. By working in tandem, these two incremental tools provide a design system whose time requirements throughout the design cycle are proportional to the size of the modifications. This chapter evaluates the extent to which this incremental system can reduce the time around the design cycle in a real design situation.

To realistically test the capabilities of the incremental system, we recreated the last stages of the design of the MIPS-X processor[9]. MIPS-X is a 32-bit RISC microprocessor designed at Stanford University by Mark Horowitz and a team of students. It is designed in a 2μm, two-level-metal, n-well CMOS technology; it runs at a clock speed of 20MHz, and contains 18,641 nodes and 47,204 transistors[1].

To recreate the final stages in the design of MIPS-X, we used the error log in which the designers recorded the design flaws and corrections that took place near the end of the design. The system was tested by first reintroducing all the errors listed in the log and then applying each of the corrections, one at a time. For each correction, the circuit was re-extracted and re-simulated using both the incremental and the conventional tools. The MIPS-X error log contained the following entries:

branch-squash : The branch squash state machine starts out in RUN mode. After reset

[1]Although MIPS-X contains an internal instruction cache, the on-chip cache memory was not included in our experiments. The transistor count listed above also excludes the cache memory.

there are 2 undetermined instructions in the pipeline. Since there is no control over what these instructions are, they can result in unknown side effects.
solution : Change the reset state of the 2-bit state machine to start out in squash mode. This inhibits the 2 instructions in question from altering any state.

register-read : There is a race between the bit-line precharge and the latch that stores the bit-line value.
solution : Reduce the size of the register file precharge transistors so the latch always wins the race.

cache-reset : The reset line for the address tags in the cache controller is connected to the master reset. To test the cache controller, the values stored in the tags must not be reset when the chip is in *cache test* mode.
solution : Qualify the reset line to the cache controller with the $CacheTest$ signal, i.e., make $CacheReset = MasterReset \bullet \overline{CacheTest}$.

psw-bit : The system/user bit in the Processor-Status-Word was accidentally shorted to ground during global routing. This results in all instructions executing in the system address space.
solution : Break the connection to ground.

cache-miss : The $CacheTest$ signal, which allows sequentially reading or writing all locations of the cache, disables $CacheWrite$ while in cache test mode. This must be corrected to allow writing into the cache during testing.
solution : Change signal's logic from $CacheWrite = CacheMiss \bullet \overline{CacheTest}$ to $CacheWrite = CacheMiss$.

address-clock : The multiplexor that selects what gets placed on the address bus (instruction or data addresses) is implemented with pass-transistors. The timing of the select signal is stable on one of the non-overlapping clocks; this causes the address bus to change during the wrong clock cycle.
solution : Qualify the select signal with the appropriate clock so that the multiplexor stores the address value for an entire clock cycle, i.e., change the select signal from $AddressSelect = pc/data$ to $AddressSelect = pc/data \bullet clock$.

address-mux : The address multiplexor (which was changed above to become a dynamic latch) is not restored; this causes the address bus to lose its value during long cycles.
solution : Make the latch static by adding feedback transistors to every bit of the address bus.

write-glitch : A glitch was found in the memory write signal. This was caused by a slow signal into a 2-input nand that generates memory writes.
solution : Speed up the signal by appropriately resizing the inverter that drives the signal.

Extraction times for each correction using the incremental and hierarchical (Magic) extractors are shown in Table 4.1. As shown in the table, even small changes that require extraction of only a few cells still take a considerable amount of time using Magic's hierarchical extractor. Magic's incremental-hierarchical extractor does reduce extraction time but only by roughly a factor of 3 — extracting all the cells in the design took 633.5s, flattening took 94.6s, and generating the output required 40.0s; this yields a total of 768.1s. In contrast, the incremental extractor only examined the fraction of the layout corresponding to the actual changes, and reduced extraction time an average of almost 2 orders of magnitude, from 243.6 to 4.2 seconds. Almost all corrections required less than 5 seconds to incrementally update the network. The only exception is the "psw-bit" correction, which required almost as much time as hierarchical extraction plus flattening. Since the "psw-bit" correction had to break a connection involving one of the power supplies (the Ground node), most of the time (over 98%) was spent processing the broken connection. This is not surprising since the power supplies weave around most of the layout, touching nearly every cell in the design; thus, processing the broken connection required flattening almost the entire layout but in a much less efficient manner than Magic's extractor. In spite of that, incrementally updating the network was still 18% faster than extraction plus flattening.

The times required to re-simulate the circuit following each correction are shown in Table 4.2. The simulation was carried out by applying the test scripts supplied by the designers; the scripts initialized the circuit's state and then exercised the processor by running 537 cycles of the Ackerman benchmark. As expected, the time required to

	Operation	branch squash	register read	cache reset	psw-bit	cache miss	address clock	address mux	write glitch
M	Cells Extracted	6	4	2	4	2	4	8	3
A	Extract	112.50	114.80	109.50	115.60	110.80	104.60	109.20	103.40
G	Flatten	93.70	94.30	94.10	94.20	93.90	94.00	94.40	93.70
I	Output	40.00	40.50	42.30	39.90	39.80	40.10	39.70	39.70
C	Total	246.20	249.60	245.90	249.70	244.50	238.70	243.30	236.80
I	Transistors	8	4	6	6	10	12	84	6
N	Changes	12	5	11	6	35	22	263	7
C	Tree Build	0.41	0.36	0.65	0.35	0.68	0.34	0.39	0.39
R	Reg. Coalesce	0.01	0.00	0.00	0.00	0.01	0.01	0.04	0.01
E	Flatten	0.08	0.01	0.07	0.02	0.17	0.07	1.77	0.50
M	Extract	0.03	0.01	0.01	0.01	0.02	0.02	0.25	0.01
E	Pin Compare	0.02	0.01	0.01	0.01	0.03	0.02	0.42	0.01
N	Ckt. Compare	1.74	2.06	0.57	0.01	1.83	0.62	19.04	2.16
T	Broken Nodes	0.00	0.00	0.00	208.44	0.58	0.00	0.00	0.00
A	Output	0.02	0.01	0.01	2.07	0.12	0.01	0.02	0.01
L	Total	2.31	2.46	1.32	210.91	3.44	1.09	21.93	3.09
	Speedup	106.58	101.46	186.29	1.18	71.08	218.99	11.09	76.63

Table 4.1: Extraction times (in seconds) for each correction of MIPS-X

	Batch Simulator			Incremental Simulator						
FIX	Events	Evalua-tions	Time (seconds)	Events: evaluate	Events: chk-pnt	Events: stimuli	Events: Total	Evalua-tions	Time (seconds)	Speedup
psw bit	694174	1804347	1089.8	304780	259583	83737	648219	614019	509.2	2.1
register read	694174	1804350	1084.6	23846	24806	21392	70044	40788	25.7	42.2
address clock	693094	1811997	1078.6	13782	14833	9176	37791	34147	16.5	65.4
address mux	694111	1813578	1086.5	8479	7475	4794	20748	18221	9.9	109.7
branch squash	647157	1744758	1065.6	470	600	889	1959	1090	1.3	819.7
cache reset	694051	1804249	1083.2	144	299	1297	1740	231	1.1	984.7
cache miss	694108	1804388	1080.1	284	323	587	555	994	0.6	1800.2
write glitch	693937	1813465	1088.4	325	499	226	1050	475	0.4	2721.0

Table 4.2: Simulation times for each correction of MIPS-X

incrementally resimulate the circuit depends more on the locality of the changes than either the size of the circuit or the length of the test; thus, the corrections that had little effect on the overall circuit run extremely fast under the incremental simulator.

Some of the corrections dealt exclusively with exceptional or testing conditions. For example, "cache-reset" and "cache-miss" were related to a test feature that was not used during the test. Nonetheless, the circuit had to be resimulated to ensure that these changes did not introduce other errors. Simulating these corrections incrementally resulted in dramatic improvements in runtime (3 orders of magnitude).

Other errors were localized to a particular portion of MIPS-X. For example, "register-read" is constrained to the register file, and "address-clock" and "address-mux" are confined to the address generation module. These corrections modified one or more of the system-wide clocks, thereby affecting portions of the circuit that are exercised on every cycle. Nevertheless, incremental simulation still reduced simulation time substantially.

The "branch-squash" correction affects a considerable portion of the processor since it must undo the work performed by the two unknown instructions. However, it is limited to a few cycles following the reset sequence; this locality in time also results in significant speedup. Similarly, the "write-glitch" correction is very close to a primary output so the changed nodes affect a small number of stages; these stages were only active during memory-write cycles, which occurred infrequently during the test. This correction thus exhibits locality in both space and time, resulting in a very large speedup.

The "psw-bit" correction affects a significant part of the processor throughout the simulation: the program counter, address generator, instruction cache, and instruction decoder. This requires a large portion of the circuit to change state. Furthermore, since the correction affects all instructions, it does not exhibit locality in time. Even in this situation, incremental simulation was twice as fast as conventional simulation. The key observation is that when the effects of a modification are localized in either space or time (or both), incremental simulation can update the circuit very fast, often in a fraction of the time required by a conventional simulator.

Table 4.3 compares the design cycle turnaround time for the MIPS-X corrections. The table shows that for relatively small changes, incremental techniques show a clear and consistent advantage over batch techniques. For this particular set of corrections, the

Correction	BATCH			INCREMENTAL		
	Extraction	Simulation	Total	Extraction	Simulation	Total
psw-bit	249.70	1089.8	1339.50	210.91	509.2	720.11
address-mux	243.30	1086.5	1329.80	21.93	9.9	31.83
register-read	249.60	1084.6	1334.20	2.46	25.7	28.16
address-clock	238.70	1078.6	1317.30	1.09	16.5	17.59
cache-miss	244.50	1080.1	1324.60	3.44	0.6	4.04
branch-squash	246.20	1065.6	1311.80	2.31	1.3	3.61
write-glitch	236.80	1088.4	1325.20	3.09	0.4	3.49
cache-reset	245.90	1083.2	1329.10	1.32	1.1	2.42
Total (seconds)			10611.50 (3 hours)			811.25 (13 min)

Table 4.3: Design cycle turnaround time for MIPS-X corrections.

incremental tools reduced the entire design cycle time from 3 hours to just a few minutes. Moreover, while the conventional tools required about 22 minutes to extract and simulate each correction, the incremental tools extracted and simulated every correction (except one) in well under one minute, providing almost instantaneous feedback to the designer.

An important consideration is that the work on MIPS-X was done on machines that were 15 times slower than the machine used to collect the data, so the large differences in time would have been that much more significant (on those machines, each correction would take an average of 5 hours 30 minutes using the batch tools versus 3 minutes using the incremental tools).

Chapter 5

Conclusions

VLSI design is an iterative process whereby the same design is repeatedly modified and verified. Yet, regardless of how small the modifications or how large the design, most existing design tools operate on the entire design. For large designs, the time to verify an entire chip becomes a bottleneck in the design cycle, which leads to lower designer productivity and higher design costs. Unfortunately, attempts to reduce this time often result in a trade of accuracy for speed. This thesis presents another alternative: incremental techniques. Incremental tools can reduce verification time, but rather than sacrificing accuracy, they do so by tracking the designer's changes and making effective use of the information gathered during previous cycles.

This dissertation describes incremental algorithms to perform two of the most time consuming tasks in the design cycle: circuit extraction and logic timing simulation. Both algorithms take advantage of the fact that most design changes are relatively small and often affect only a fraction of the design. The ability to track design changes and analyze only the modified sections of the design provide our incremental algorithms with their fast response.

The incremental simulator maintains a history of past circuit activity in order to resimulate only the parts of the circuit that, as a result of the design changes, deviate from the history. Augmenting a conventional event-driven mechanism to include check-point and stimulus events allows the incremental simulator to use the same evaluation models employed by *Rsim*; it is thus able to reduce simulation time without sacrificing accuracy.

The ability to track and resimulate only those stages whose behavior deviates from the history gives the incremental simulator its fast response. Keeping track of which stages deviate from their history, however, results in additional overhead. Our experiments indicate that in the worst case, which occurs when the entire circuit is resimulated, our incremental simulator can take up to 38% longer than *Rsim*; as long as less than 75% of the circuit is resimulated, the incremental simulator is faster than *Rsim*. Since few changes affect such a large portion of the circuit, incremental simulation should be the method of choice for most changes.

The incremental extractor minimizes extraction time by keeping track of the layout areas that are modified by the designer during an editing session. It extracts the circuits that correspond to the layout before and after the modifications, compares the two circuits, and reports the differences as network modifications. To compare the circuits, the extractor makes use of the information available at the boundary of the modified layout. This enables the comparison to quickly determine the necessary network transformations by using a fast graph partitioning algorithm in order to find a near-isomorphism. Our experiments show that the comparison algorithm is fairly insensitive to circuit differences and also that the time required to incrementally extract the differences depends linearly on the size of the modified area; the result is a tool that can update the network in a time proportional to the size of the changes. However, extracting two versions of the circuit results in additional overhead which makes the extractor faster than hierarchical extraction plus flattening only as long as less than 40% of the layout is modified. Since such a sizable part of the layout is rarely modified, the extractor should be faster for most changes.

Implementing the incremental simulator, *Irsim*, required major changes to *Rsim*. The data structures that describe the circuit were modified to enable network changes, and the event manager was modified to perform history recording. In testing out the incremental simulator we found a number of errors in the switch-level models used by *Rsim*, including a problem in which the outputs depended on the order of evaluation. Since during incremental simulation one needs to be able to regenerate each output, these problems caused the incremental simulation to take longer than necessary (it also made it extremely difficult to verify the results). We therefore decided to incorporate into *Irsim* the improved

simulation models developed by Horowitz and Chu[11, 10, 22]. These changes greatly enhanced the accuracy and usefulness of *Irsim*, which is currently in use at several hundred academic and industrial sites.

Once the incremental system was assembled, we tested it on several corrections to errors that actually occurred in the design of a large VLSI project. The incremental system was able to reduce the entire verification process from 3 hours to a few minutes. For most of these corrections, the incremental system took only a few seconds, providing near-instantaneous feedback on the quality of the design, and decreasing turnaround time by an average of two orders of magnitude.

Although the system is operative now, there are many changes and extensions that would enhance its utility. The biggest impediment to incrementally simulate a large design is the amount of memory needed to maintain the history. Although this limitation can be overcome by splitting a large test into several smaller tests, this solution is both clumsy and error prone. Another solution, selectively storing part of the history (as suggested by Choi[8]) is not an easy problem, however. Because there is no way to determine a-priori which parts of the history will be needed during a subsequent resimulation, there is no realistic information on which to base the decision regarding what parts of the history to maintain. Furthermore, maintaining the complete history provides *Irsim* with the ability to move backwards in time as well as to incorporate a graphic analyzer that allows interactive display of any node in the circuit. This analyzer is extremely helpful, and greatly improves a designer's ability to understand the behavior of the circuit. Storing only part of the history would limit the usefulness of both of these features.

Another enhancement to the incremental algorithm that merits further research would be the ability to maintain a stage's composition and incremental state throughout the simulation, and rather than rebuilding the stage for each event, simply update it incrementally. However, this is not a trivial optimization due to the presence of current loops within a stage.

There are several areas in incremental extraction that also deserve more attention. The two biggest problems in incremental extraction are the inability to hierarchically extract the modified parts of the layout, and the unbounded time needed to process broken connections. Both of these problems are hard to overcome. Extraction without flattening

is quite difficult unless cell overlap is severely restricted or hierarchical connections are explicitly declared; such restrictions make the editor harder to use and are thus annoying to designers. There is also no obvious approach to determine the composition (devices, parasitics, and name) of broken nodes without traversing the nodes in their entirety.

Additional work to the interface of the two tools would also make them easier to use. The current prototype implementation of the incremental extractor is somewhat clumsy; the user must incrementally extract the changes before saving the modified cells, and must always save them after having extracted the changes. If this order is not preserved, the boundaries of the modified regions may not be the same and will surely lead to catastrophic results. It would also be desirable to provide for a smoother data transfer between the two tools; currently this is done through intermediate files that create the potential for human error.

Incremental techniques can be applied to many other processes that are iterated several times, each time making minor changes. One such process is fault simulation, which can be regarded as a process that repeatedly inserts minor changes to the circuit and simulates their effect. Appendix C describes a prototype implementation of a fault simulator based on this concept. Although incremental techniques can be effective in reducing the runtime of these processes, there is some additional overhead involved in using these techniques. The cost of analyzing the entire design must be carefully weighted against this overhead in order to amortize this cost and maximize the benefit. In our system, this additional overhead is handsomely paid off by reducing the cost of extracting and simulating the entire design, making it possible to speedup the process by two to three orders of magnitude. In the future, as designs become larger and more complex, every step of the design cycle will have to be analyzed in order to speed the process. The work presented in this dissertation provides an understanding of the types of tradeoffs that will have to be considered in that analysis.

Appendix A

Stage Analysis

Section 2.2 describes how *Rsim* partitions a circuit into stages. This section describes the computations involved in the evaluation of a stage: logic level calculation, delay estimation, and charge-sharing analysis.

A.1 Final Value Computation

To compute the final (or steady-state) value of a node, the simulator must solve for the voltage in the resistor network formed by the stage containing the node. This voltage can be determined by using Kirchoff's current-voltage laws. If, however, the stage contains transistors with unknown (X) gate levels, the existence of a single solution is not guaranteed; since these transistors can be either on or off, they induce different circuit configurations, each one resulting in a different voltage for the same node. To uncover all possible solutions using Kirchoff's laws would require an exhaustive evaluation of the stage by trying out all possible combinations of transistor states. This is not only inefficient but unnecessary since the logical state can be determined from the voltage range comprising all possible solutions by simply checking that this range lies within a single logical state. Several algorithms have been developed to approximate this voltage range. One of them is Terman's algorithm, which characterizes the voltage range of a node by its Thevenin equivalent. This scheme, however, suffers from various problems, including an evaluation order dependency that makes it impossible to obtain the same

results during incremental simulation.

The current version of Rsim uses a resistor-divider model developed by Chu[11]. This model characterizes the effect of the network on a particular node by its equivalent resistance to the two power supplies: R_{up}, its resistance to 1, and R_{down}, its resistance to 0. These two resistances are, in general, intervals (R_-, R_+) since the equivalent transistor resistances from which they are derived might themselves be intervals. The two resulting intervals form a resistor divider; the voltage between them represents the output node's voltage range. The final logic state is obtained by comparing the bounds of this voltage to the thresholds:

$$\text{Final value} = \begin{cases} 0 & \text{if } R_{down+}/(R_{down+} + R_{up-}) \leq V_{low} \\ 1 & \text{if } R_{down-}/(R_{down-} + R_{up+}) \geq V_{high} \\ \text{X} & \text{otherwise} \end{cases}$$

For example, in the circuit for a static CMOS register of Figure 2.3, one stage containing nodes **N1** and **N2** includes two transistors with an X gate level: T6 and T7. The simulator must determine the state of nodes **N1** and **N2**, which are undetermined at this point. The corresponding resistor divider model and the equivalent resistances that Rsim calculates are:

$$R_{up}1 = [R_3 + R_6, \infty] \qquad R_{up}2 = [R_6, \infty]$$

$$R_{down}1 = [\tfrac{R_2(R_3+R_7)}{R_2+R_3+R_7}, R_2] \qquad R_{down}2 = [\tfrac{(R_2+R_3)R_7}{R_2+R_3+R_7}, R_2 + R_3]$$

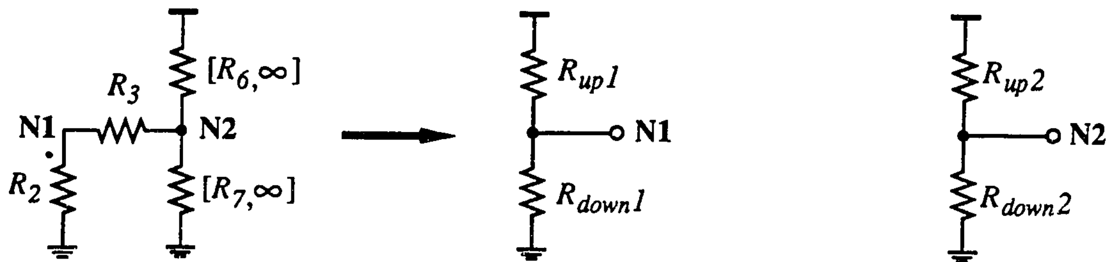


Figure A.1: Final value solution for static CMOS register circuit

To allow incoming values to be stored at node **N2**, this circuit is designed such that the input transistors (T1, T2 and T3) have a stronger drive than the feedback transistors (T6 and T7). To guarantee proper operation of the example shown above, in which a 0 is being stored, the following conditions are derived from the resistor-divider model:

$$\frac{R_2}{R_2+R_3+R_6} \leq V_{low} \quad \text{for node N1}$$
$$\frac{R_2+R_3}{R_2+R_3+R_6} \leq V_{low} \quad \text{for node N2}$$

A.2 Delay Estimation

If a node's final value differs from its present state, the simulator must determine how long it will take for the node to reach the new level. The approach adopted by Rsim is to estimate the node's waveform by modeling the output of a stage by an equivalent, single-capacitor, single-resistor circuit; the waveform of the equivalent circuit is a single exponential and its time constant can be used to determine the delay. The basic idea is straightforward; since Rsim uses resistors to model conducting transistors, a stage represents a resistor-capacitor (RC) tree: a network of floating resistors and grounded capacitors driven by a voltage source. An RC tree is a linear system, and as such, its behavior, as a function of time, is a sum of exponentials. Since the output waveform of a digital circuit is often dominated by the slowest of these exponentials, i.e. the lowest frequency pole of its transfer function, a single exponential is a good approximation for the waveform.

To determine the equivalent circuit, Rsim uses the single-time-constant model developed by Rubenstein-Penfield-Horowitz[39]. Their model is derived by matching geometric waveform characteristics – the boundary conditions and the area under the voltage waveform – between the RC tree and the equivalent circuit. The area under an exponential waveform is simply its time constant; the corresponding area for a node in an RC tree can be easily determined from the topology of the circuit, as follows. Assume that one of the nodes in the tree is grounded, and that all the capacitors are initially charged high; the voltage, V_e, at any node e is equal to the voltage drop across the resistors between e and ground. This voltage can be found by replacing each capacitor by its equivalent current source, $i_k = -C_k \frac{dV_k}{dt}$, and then using superposition to add the contribution of each current source on node e:

$$V_e = \sum_k^n R_{ke} i_k = -\sum_k^n R_{ke} C_k \frac{dV_k}{dt},$$

where n is the number of nodes in the stage, i_k represents the current due to the capacitor

C_k at node k, and R_{ke} is the resistance to ground shared by nodes e and k. The area under V_e is then given by

$$\tau_{D_e} = \int_0^{\infty} V_e dt = \sum_k^n R_{ke} C_k$$

The time constant, τ_{D_e}, is equal to the centroid of the circuit's impulse response; a quantity used by Elmore[16] to define the delay through a linear amplifier, hence known as Elmore's delay.

Computing τ_{D_e} can be done by a depth-first traversal of the tree, rooted at the output node e, and accumulating the resistance-capacitance products on the path to the driving node. To exemplify this, assume that in the circuit of Figure A.2, node **Clk** has just transitioned from 0 to 1, causing node **N2** to transition from 1 to 0. To estimate its delay, Rsim calculates the following time constant for node **N2**:

$$\tau_{D_{N2}} = R_2 C_1 + (R_2 + R_3) C_2$$

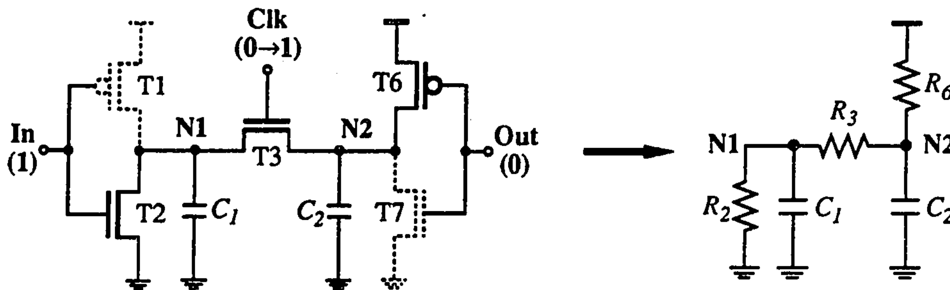


Figure A.2: Time constant computation for a static CMOS register circuit

The delay is also affected by the rate of change of the input signal. Horowitz[22] has modified Rsim to account for the effects of input slope on the output. He applies an input ramp with rise-time T_{in}, and models the transition of a stage's output in two parts. Initially, the output current is a linear function of the input voltage; this produces a quadratic output voltage waveform, $V = f(t^2)$. Once the output voltage reaches the voltage drop across the output resistance, the current becomes independent of the input voltage and depends only on the intrinsic time-constant, τ_D, of the output; this produces

an exponential output voltage waveform, $V = f(e^t)$. Depending on which of these two waveforms dominates the output transition, the delay can be approximated by:

$$t_d \simeq \begin{cases} \sqrt{2T_{in}(1 - V_s)C_L/g_m} & \text{if dominated by input transition} \\ \tau_D \ln 1/V_s & \text{if dominated by output delay,} \end{cases}$$

where g_m is the input transistor's transconductance, C_L is the output capacitance, and V_s is the switching voltage of the stage. Horowitz proposes approximating the delay by the root-mean-square of these two values, and using the slope of the input transition at the switching point: $T_{in} = \frac{\tau_{in}}{1-V_s}$. The delay can then be written as

$$t_d = \sqrt{(\tau_D \ln V_s)^2 + 2\frac{\tau_{in}}{g_m}C_L}.$$

For the example of Figure A.2, Rsim computes the following delay for node **N2**:

$$t_{d_{N2}} = \sqrt{(\tau_{D_{N2}} \ln 2)^2 + 2\frac{\tau_{Clk}}{g_{m_{T3}}}(C_2 + C_1\frac{R_2}{R_2 + R_3})}.$$

After computing this delay, the simulator will then schedule an event $t_{d_{N2}}$ units from the current time, indicating that node **N2** is to change state to 0 with time constant $\tau_{D_{N2}}$. If no other activity occurs in the circuit, at time $t_{current} + t_{d_{N2}}$, the value of node **N2** will be changed to 0. The whole process will then be repeated to compute the state of node **Out**, this time using $\tau_{in} = \tau_{D_{N2}}$ to compute its delay.

A.3 Charge Sharing Computation

When a transistor turns on, its source and drain nodes redistribute their charge so that they are at the same potential. If the two nodes are initially charged to different voltages, this charge sharing can change their state. In general, charge sharing refers to the redistribution of charge when two or more stages charged to different voltages are connected together. Chu[11, 10] has incorporated into Rsim two types of charge sharing models: *pure charge sharing* and *driven charge sharing*.

A.3.1 Pure Charge Sharing

Pure charge sharing occurs when two undriven stages are connected together. In this case, all connected nodes will share their charge and reach the same final voltage, which depends on the ratio of total charge to total capacitance. This type of charge sharing is frequently used intentionally in VLSI circuits such as precharged logic. An example of such a circuit is shown in Figure A.3, where the output of precharged signal **N1** is connected to an inverter through a pass transistor. When input **Load** goes high, the value of node **N1** will be transferred to node **N2** exclusively by charge sharing.

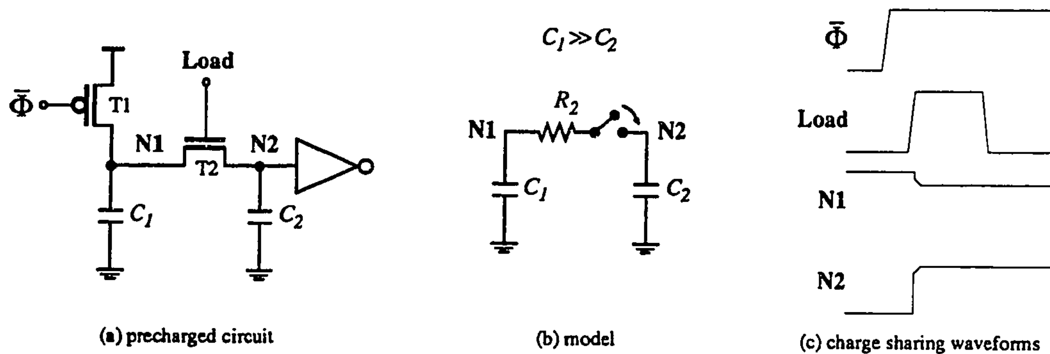


Figure A.3: Precharged circuit that uses pure charge sharing

The minimum and maximum voltages due to charge sharing can be found by collecting all the charge in the stage. These two voltages can then be compared with the thresholds to determine the new logic state. Since X capacitors contribute an indeterminate amount of charge, Terman suggests a scheme in which they only contribute towards raising the maximum voltage or lowering the minimum voltage; the idea is to make a conservative estimate whereby X capacitors can only bring the final voltage closer to X, as follows:

$$V_f = \begin{cases} 0 & \text{if } \frac{C_{high}+C_x}{C_{low}+C_{high}+C_x} \leq V_{low} \\ 1 & \text{if } \frac{C_{high}}{C_{low}+C_x+C_{high}} \geq V_{high} \\ X & \text{otherwise} \end{cases}$$

The delay due to a pure charge-sharing transition cannot be estimated using the single-time-constant approximation of Section A.2. That model expresses the voltage waveform

in terms of capacitor currents towards a voltage source; in pure charge sharing, however, all the nodes are floating and there is no voltage source ($R_{ke} = \infty$). Chu's approach is to express all voltages with respect to an arbitrary reference node and then use charge conservation to decouple the relationship. In a manner similar to that of the single-time-constant approximation, he models the stage by an equivalent, single-capacitor, single-resistor circuit that characterizes the delay by a single time constant. The time constant is determined by calculating the area between the voltage waveform and a time-independent line representing the final voltage, and then choosing an exponential function with the same area and boundary conditions. This results in the following time-constant:

$$\tau_e = \frac{C_T \tau_{A_e^r} - \sum_k^n C_k \tau_{A_k^r}}{C_T(V_e - V_f)},$$

where

$$\tau_{A_e^r} = \sum_k^n R_{ke}^r C_k (V_k - V_f)$$

and C_T is the total capacitance in the stage, C_k and V_k are the capacitance and voltage at node k, and R_{ke}^r is the resistance to the reference node r shared by nodes k and e.

Despite the apparent complexity of the above expressions, they can all be easily computed from the topology of the circuit by traversing the stage twice: the first time to compute $\tau_{A_e^r}$, the second time to compute $\sum_k^n C_k \tau_{A_k^r}$. For example, consider the circuit of Figure A.3 in which **N2** undergoes a charge sharing transition from 0 to 1. Using **N1** as the reference node ($r \equiv$ **N1**), Rsim calculates:

$$\begin{array}{rcccl}
\text{since } C_1 \gg C_2 & \rightarrow & \frac{C_1}{C_2+C_1} > V_{high} \quad \rightarrow & V_f & = \; 1 \\
\tau_{A_{N1}^{N1}} & = & 0 \cdot C_1(1 - V_f) + 0 \cdot C_2(-V_f) & = & 0 \\
\tau_{A_{N2}^{N1}} & = & 0 \cdot C_1(1 - V_f) + R_2 C_2(-V_f) & = & -R_2 C_2 \\
\tau_{N2} & = & \frac{(C_1+C_2)(-R_2C_2) - C_2(-R_2C_2)}{(C_1+C_2)(-1)} & = & R_2 \frac{C_1 C_2}{C_1 + C_2}
\end{array}$$

The result is the same if node **N2** were chosen as the reference, which can be verified in a similar manner as above. The details of this model can be found in Chu's thesis[11].

A.3.2 Driven Charge Sharing

Driven charge sharing is similar to pure charge sharing except that one or more of the nodes is driven by a voltage source. In this case (Figure A.4), the stage is composed of two components: a *driving tree* and a *charging tree*. When the two trees are connected together, nodes in the charging tree will be driven to their final value by the voltage source; nodes in the driving tree, however, will experience a momentary spike as they share their charge with nodes in the charging tree. Depending on the amplitude of the spike, nodes in the driving tree may undergo a temporary change of state.

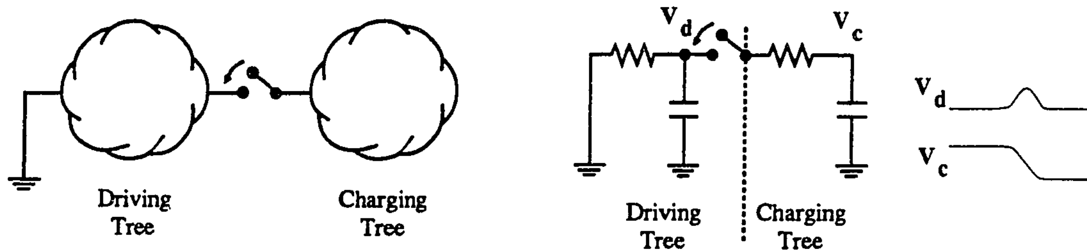


Figure A.4: A driven charge sharing circuit

This type of charge sharing occurs in many circuits. For example, node **N1** in the static register circuit of Figures 2.3 and A.1 is subjected to a spike whenever the **Clk** input turns transistor T3 on. Analyzing this type of charge sharing is more complicated; since the node starts and ends at the same voltage, it can not be modeled by a single time constant. Instead, two exponentials are needed to represent the spike. The first one is set by the time required for the redistribution of charge between the two trees, while the second one is set by the time required to return the node to its initial, driven value.

To analyze a spike, Chu maps a driving tree node into a reduced, two-capacitor, two-resistor circuit, which is the simplest circuit that exhibits a two-time-constant behavior. The mapping is done by matching three geometric waveform characteristics of the driving tree node with those of the reduced circuit: the area under the voltage waveform, the sum of the poles of the transfer function, and Elmore's delay. These are derived in a manner similar to the pure charge sharing equations, and are given by the following:

$$\tau_{A_e} = \sum_k^n R_{ke} C_k V_k \qquad \tau_{P_e} = \frac{\sum_k^n R_{ke} C_k \tau_{A_k}}{\tau_{A_e}} \qquad \tau_{D_e} = \sum_k^n R_{ke} C_k$$

The reduced circuit is solved explicitly and the results are stored in a table that is indexed through a normalizing function of these three quantities. The table contains two values for every entry: a spike amplitude and a charge sharing delay. When a spike is detected during simulation, Rsim looks up these values in the table. If the amplitude of the spike is too small to change the state of the node, the spike is ignored; otherwise two events are scheduled. The first one, a charge sharing event, changes the state of the node to that of the spike; its delay is determined from the table. The second one, a driving event, returns the node to its initial state; its timing is computed using the single-time-constant approximation.

Appendix B

Network Modification Commands

The incremental extractor generates a network modification file that contains the various network modification commands that specify how a circuit is to be modified; this file is interpreted by Irsim which applies the transformations to the network. The file consists of a series of lines, each of which begins with a keyword; the keyword beginning a line determines how the remainder of the line is interpreted. Commands are classified into four groups: Auxiliary, Topological, Parametric, and Correspondence.

Auxiliary commands do not specify any changes, instead they define node numbers that serve as indirect node references; other commands use these node numbers to refer to a node that may be altered. Topological commands are used to modify the topology of the underlying network by adding, removing, or reconnecting transistors and nodes. Parametric commands are used to modify the electrical parameters of the network. Correspondence commands are used to maintain the simulated network consistent with its corresponding layout.

In all commands, capacitance values are specified in femtoFarads, and linear dimensions, such as transistor sizes, are specified in centimicrons / lambda.

1. **Auxiliary Commands:**

   - == ⟨*node-number*⟩ ⟨*node-name*⟩
     Defines ⟨*node-number*⟩ as a reference for the node whose hierarchical name is ⟨*node-name*⟩.

- = ⟨*node-number*⟩ ⟨*node-ref*⟩
  Defines ⟨*node-number*⟩ as a reference for the node referred to by ⟨*node-ref*⟩. A ⟨*node-ref*⟩ has the following format:
  @=⟨*terminal*⟩⟨*x*⟩,⟨*y*⟩
  where ⟨*terminal*⟩ is one of the three letters **g**, **s** or **d**, and ⟨*x*⟩ and ⟨*y*⟩ are the location of a transistor in the layout. For example, "@=s20,55" means the node connected to the source of the transistor located at coordinates 20,55.

2. **Topological Commands:**

- **new** ⟨*cap*⟩ ⟨*node-name*⟩
  Creates a new node with hierarchical name ⟨*node-name*⟩, whose capacitance to ground is ⟨*cap*⟩.
- **Eliminate** ⟨*node-number*⟩
  Removes the node whose hierarchical name is ⟨*node-name*⟩.
- **eliminate** ⟨*node-number*⟩
  Same as the command above, but instead of a hierarchical name, the node is specified by a node number (defined earlier through an auxiliary command).
- **connect** ⟨*node-number*⟩ ⟨*node-number*⟩
  Connects the two specified nodes forming a single compound node. The *best* hierarchical name of the two nodes is chosen as the name for the compound node.
- **break** ⟨*node-number*⟩ ⟨*number*⟩ ⟨*cap*⟩ ⟨*node-name*⟩
  Break off a new node from the node referred to by ⟨*node-number*⟩. The new node is assigned node number ⟨*number*⟩, hierarchical name ⟨*node-name*⟩, and ⟨*cap*⟩ capacitance to ground.
- **add** ⟨*type*⟩ ⟨*x*⟩ ⟨*y*⟩ ⟨*length*⟩ ⟨*width*⟩ ⟨*gate*⟩ ⟨*source*⟩ ⟨*drain*⟩
  Add a new transistor of type ⟨*type*⟩. Currently, ⟨*type*⟩ may be one of **n**, **p** or **d**, for n-type, p-type, and depletion, respectively. ⟨*x*⟩ and ⟨*y*⟩ specify the location of the transistor, ⟨*length*⟩ and ⟨*width*⟩ specify its size; and ⟨*gate*⟩,

⟨*source,*⟩ and ⟨*drain*⟩ are node numbers specifying the nodes to which the gate, source, and drain of the transistor are to be connected.

- **delete** ⟨*x*⟩ ⟨*y*⟩
  Deletes the transistor at location ⟨*x*⟩,⟨*y*⟩.
- **move** ⟨*x*⟩ ⟨*y*⟩ ⟨*gate*⟩ ⟨*source*⟩ ⟨*drain*⟩
  Moves the terminals of the transistor at location ⟨*x*⟩, ⟨*y*⟩ from their current nodes to the nodes specified by the node numbers ⟨*gate*⟩, ⟨*source*⟩, and ⟨*drain*⟩. The node specifiers can be a single dot, ".", indicating that the corresponding transistor terminal is not to be moved.
- **Move** ⟨*x*⟩ ⟨*y*⟩ ⟨*node-number*⟩ ⟨*terminal*⟩
  Moves a single terminal of the transistor at location ⟨*x*⟩, ⟨*y*⟩ from its current node to the node specified by ⟨*node-number*⟩. The transistor terminal to be moved is specified by ⟨*terminal*⟩ which can be one of **gate, source**, or **drain.**

3. **Parametric Commands:**

- **Cap** ⟨*node-number*⟩ ⟨*cap*⟩
  Cap ⟨*node-number*⟩ =⟨*cap*⟩
  In the first form, the capacitance of the node specified by ⟨*node-number*⟩ is changed by ⟨*cap*⟩ units. To reduce the capacitance of a node, a negative value may be used. In the second form, the command specifies the capacitance to be assigned to the node.
- **size** ⟨*x*⟩ ⟨*y*⟩ ⟨*length*⟩ ⟨*width*⟩
  Changes the size of the transistor located at position ⟨*x*⟩, ⟨*y*⟩ to the size specified by ⟨*length*⟩ and ⟨*width*⟩.
- **threshold** ⟨*node-name*⟩ ⟨*vlow*⟩ ⟨*vhigh*⟩
  Changes the voltage thresholds of the node with name ⟨*node-name*⟩ to the normalized voltage values specified.
- **delay** ⟨*node-name*⟩ ⟨*tplh*⟩ ⟨*tphl*⟩
  Changes the user-assignable delay of the specified node. ⟨*tplh*⟩ and ⟨*tphl*⟩ refer to the low→high and high→low transition times respectively.

4. **Correspondence Commands:**

- **xchange** ⟨*x*⟩ ⟨*y*⟩
  Exchanges the source and drain nodes of the transistor at location ⟨*x*⟩, ⟨*y*⟩, i.e., source becomes the drain and vice versa.

- **position** ⟨*x*⟩ ⟨*y*⟩ ⟨*new-x*⟩ ⟨*new-y*⟩
  Changes the location of the transistor from coordinates ⟨*x*⟩,⟨*y*⟩ to coordinates ⟨*new-x*⟩,⟨*new-y*⟩.

- **rename** ⟨*node-number*⟩ ⟨*node-name*⟩
  Assigns the name ⟨*node-name*⟩ to the node specified by ⟨*node-number*⟩.

- **hier-rename** ⟨*node-number*⟩ ⟨*node-name*⟩
  hier-rename ⟨*node-number*⟩ ⟨*node-name*⟩ ⟨*curr-name*⟩
  In its first form, the name of the node specified by ⟨*node-number*⟩ is changed to ⟨*node-name*⟩ only if ⟨*node-name*⟩ is a better hierarchical name than its current name. In its second form, the node's name is changed only if its current name is ⟨*curr-name*⟩.

Appendix C

Fault Simulation

One possible use of the incremental simulator is as a fast fault simulator. The basic idea is to simulate the "good circuit" once, and then repeatedly introduce a fault and incrementally simulate the faulty circuit. Each fault can be introduced as a circuit modification and then the circuit resimulated until the effect of that change is observed on an output. Since the incremental simulator only simulates the effect of the change, the per-fault time should be very fast. This scheme can easily accommodate any fault that can be modeled as circuit modifications. Furthermore, since Irsim is a timing simulator, it can identify timing faults as well as functional faults.

To test the effectiveness of an incremental fault simulator, we implemented a simple "stuck-at" fault simulator using Irsim as a base. To start a fault simulation, the user first simulates the circuit once; the history recorded during this initial simulation corresponds to the behavior of the good circuit. Next, the user must specify the circuit's primary outputs and the timing of the sampling for each of the primary outputs; signals can be sampled using either a fixed time interval or the rising/falling edge of some other signal in the circuit. For example, the following entry indicates that the primary outputs A_0, A_1, A_2, and A_4 are to be sampled on the falling edge of signal phi1:

```
trigger phi1 0
        A0 A1 A2 A4
***
```

Implementing the fault simulator required some minor changes to the basic incremental algorithm. First, a *trigger* event type was introduced to indicate the sampling times. When this event is processed, the simulator checks all primary outputs that are sampled by the event; if any of them is currently deviating from the good machine's history then the fault is reported as detected at the current simulation time. Second, the algorithm was modified so it terminates as soon as the first fault at a primary output is detected; this requires cleaning up of any other events that may be queued at that time. Finally, instead of updating the simulation history of the good machine (as the conventional incremental simulator does), the fault simulator retains the initial history and only maintains the current history entry of the faulty circuit. If this were not done, the good machine's history would have to be rebuilt by either reading it from disk or by applying the anti-fault (the modifications that eliminate the fault) and resimulating once more. This would be rather inefficient since the history can be quite large and must be rebuilt for every fault.

To model stuck-at faults, the simulator simply connects an always-conducting transistor with very low resistance between the node to be tested and the appropriate power supply. This requires minimum changes to the network and can be done very fast. Fault seeding is very simple-minded; it simply tests all nodes that are not inputs to the circuit.

The output produced by the fault simulator includes the following information:

- For each detected fault: the output at which the fault was detected, the type of fault (stuck at 1 or stuck at 0), the node at which the fault was injected, and the simulation time at which the fault was detected.
- For each undetected fault: the type of fault (stuck at 1 or stuck at 0), and the node at which the fault was injected.
- The total number of faults seeded.
- The number of detected faults.
- The number of undetected faults.
- The fault coverage as the percentage of faults that are detected.
- The number of probably detected faults.

The last item corresponds to outputs whose history deviate from the good circuit at the time of sampling, but they do so by deviating to or from an **X** logic state. Since **X** represents an undetermined or unknown logic value, these faults may not be observable so they are listed separately. Note that when this type of fault is detected the simulation does not terminate, but continues until either a deterministically observable fault is found or the end of the history is reached.

To characterize the performance of the stuck-at fault simulator, several circuits were tested using the test scripts provided by the designers. Table C.1 compares the time required to fault-simulate the circuits using the incremental fault simulator and using a serial fault simulator version of the same simulator.

Circuit	Number of Transistors	Faults Tested	Fault Coverage	Time (hours:min) Serial	Time (hours:min) Incremental	Speedup
Spim	41,804	33,062	52.2%	639:24	24:21	34.8
Divider	15,335	15,950	49.2%	157:50	15:6	10.45
Mipsx	47,202	3,262	42.7%	1903:51	117:35	16.2

Table C.1: Comparison of Incremental vs. Serial Fault Simulation.

Bibliography

[1] R. Allgair and D. Evans. "A Comprehensive Approach to a Connectivity Audit, or A Fruitful Comparison of Apples and Oranges". In *Proceedings of the Design Automation Conference*, pages 312–321. ACM/IEEE, 1977.

[2] Robert Alverson, Tom Blank, Kiyoung Choi, S.Y. Hwang, Arturo Salz, Larry Soule, and Thomas Rokicki. "THOR User's Manual: Tutorial and Commands". Technical report csl-tr-88-348, Stanford University, January 1988.

[3] E. Barke. "A Network Comparison Algorithm for Layout Verification of Integrated Circuits". *IEEE Transactions on Computer-Aided Design*, 3(2):136–141, April 1984.

[4] D. Beatty and R. Bryant. "Fast Incremental Circuit Analysis Using Extracted Hierarchy". In *Proceedings of the 25th Design Automation Conference*, pages 495–500. ACM/IEEE, 1988.

[5] M.A. Breuer and A.D. Friedman. *"Diagnosis & Reliable Design of Digital Systems"*. Computer Science Press Inc., Potomac, Maryland, 1976.

[6] R.E. Bryant. "A Switch-Level Model and Simulator for MOS Digital Systems". *IEEE Transactions on Computers*, C-33(2):160–177, February 1984.

[7] K. Choi, S.Y. Hwang, and T. Blank. "Incremental-in-time Algorithm for Digital Simulation". In *Proceedings of the 25th Design Automation Conference*, pages 501–505. ACM/IEEE, June 1988.

[8] Kiyoung Choi. *"Incremental Approach to Digital Simulation"*. PhD thesis, Stanford University, June 1989.

[9] Paul Chow, editor. *"The MIPS-X RISC Microprocessor"*. Kluwer Academic Publishers, Norwell, Massachusetts, 1989.

[10] C.-Y. Chu and M. A. Horowitz. "Charge-Sharing Models for Switch-Level Simulation". *IEEE Transactions on Computer-Aided Design*, CAD-6(6):1053–1061, November 1987.

[11] Chorng-Yeong Chu. *"Improved Models for Switch Level Simulation"*. PhD thesis, Stanford University, November 1988. Technical Report CSL-TR-88-368.

[12] D. Corneil. "Recent Results on the Graph Isomorphism Problem". In *Proceedings of the 8th Manitoba Conference on Numerical Mathematics and Computing*, pages 13–31, 1978.

[13] D. Corneil and C. C. Gotlieb. "An Efficient Algorithm for Graph Isomorphism". *Journal of the Association of Computing Machinery*, 17:51–64, 1970.

[14] D. Corneil and D. Kirkpatrick. "A Theoretical Analysis of Various Heuristics for the Graph Isomorphism Problem". *SIAM Journal of Computing*, 9(2):281–297, 1980.

[15] C. Ebeling and O. Zajieck. "Validating VLSI Circuit Layout by Wirelist Comparison". In *Proceedings of the International Conference on Computer Aided Design*, pages 172–173. IEEE, September 1983.

[16] W. Elmore. "The Transient Response of Damped Linear Networks with Particular Regard to Wideband Amplifiers". *Journal of Applied Physics*, 19:55–63, January 1948.

[17] Anoop Gupta. "ACE: A Circuit Extractor". In *Proceedings of the 20th Design Automation Conference*, pages 721–725. ACM/IEEE, June 1983.

[18] F. Harary. "The Determinant of the Adjacency Matrix of a Graph". *SIAM Journal*, 4(3):202–210, July 1962.

[19] J.E. Hopcroft and R.E. Tarjan. "Isomorphism of Planar Graphs". In *Complexity of Computer Computations*, pages 143–150. PlenumPress, New York, N.Y., 1972.

[20] J.E. Hopcroft and J.K. Wong. "Linear Time Algorithm for Isomorphism of Planar Graphs'. In *Proceedings of the 6th Annual ACM Symposium on the Theory of Computing*, pages 172–184, Seattle, Wash., April 1974.

[21] Mark Horowitz, Paul Chow, Don Stark, Richard T. Simoni, Arturo Salz, Steven Przybylski, John Hennessy, Glenn Gulak, Anant Agarwal, and John Acken. "MIPS-X: A 20 MIPS Peak, 32-bit Microprocessor with On-Chip Cache". *IEEE Journal of Solid State Circuits*, SC-22(5):790–799, October 1987.

[22] Mark A. Horowitz. *"Timing Models for MOS Circuits"*. PhD thesis, Stanford University, 1983.

[23] S.Y. Hwang, T. Blank, and K. Choi. "Incremental Functional Simulation of Digital Circuits". In *International Conference on Computer Aided Design Digest of Technical Papers*, pages 392–395. IEEE, November 1987.

[24] S.Y. Hwang, T. Blank, and K. Choi. "Fast Functional Simulation: An Incremental Approach". *IEEE Transactions on Computer-Aided Design*, 7(7):765–774, July 1988.

[25] Larry G. Jones. "An Incremental Zero/Integer-Delay Switch-Level Simulation Environment". To appear in *IEEE Transactions on Computer Aided Design.*

[26] Larry G. Jones. "Fast Incremental Netlist Compilation of Hierarchical Schematics". In *International Conference on Computer Aided Design Digest of Technical Papers*, pages 326–329. IEEE, December 1986.

[27] Larry G. Jones. *"Incremental VLSI Design Systems Based on Circular Attributes Grammars"*. PhD thesis, Pennsylvania State University, December 1986.

[28] Larry G. Jones and Janos Simon. "Hierarchical VLSI Design Systems Based on Attribute Grammars". In *International Conference on Computer Aided Design Digest of Technical Papers*, pages 58–69. IEEE, January 1986.

[29] D.E. Knuth. *"Marriages Stables"*. Les Presses de L'Universite de Montreal, Montreal, Canada, 1976.

[30] Donald E. Knuth. "Semantics of Context-Free Languages". *Mathematical Systems Theory*, 2(2):127–145, June 1968.

[31] K. Kodandapani and E. McGrath. "A Wirelist Compare Program for Verifying VLSI Circuits". *IEEE Design and Test of Computers*, pages 46–51, June 1986.

[32] Peter M. Maurer and Alexander D. Schapira. "A Logic-to-Logic Comparator for VLSI Layout Verification". *IEEE Transactions on Computer Aided Design*, 7(8):897–907, August 1988.

[33] L.W. Nagel. "SPICE2: A Computer Program to Simulate Semiconductor Circuits". In *Memorandum No. ERL-M520, Electronics Research Laboratory*. University of California, Berkeley, 1975.

[34] John. K. Ousterhout. "Corner Stitching: A Data-Structuring Technique for VLSI Layout Tools". *IEEE Transactions on Computer-Aided Design*, CAD-3(1):87–100, January 1984.

[35] John K. Ousterhout, Gordon T. Hamachi, Robert N. Mayo, Walter S. Scott, and George Taylor. "Magic: A VLSI Layout System". In *Proceedings of the 21st Design Automation Conference*, pages 152–159. ACM/IEEE, June 1984.

[36] Gregory Pfister. "The IBM Yorktown Simulation Engine". In *Proceedings of the IEEE*, pages 850–860, June 1986.

[37] R.E., Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler. "COSMOS: A Compiled Simulator For MOS Circuits". In *Proceedings of the 24th Design Automation Conference*, pages 9–16. ACM/IEEE, June 1987.

[38] R. Read and D. Corneil. "The Graph Isomorphism Disease". *Journal of Graph Theory*, 1:339–363, 1977.

[39] Jorge Rubenstein, Paul Penfield Jr., and Mark A. Horowitz. "Signal Delay in RC Tree Networks". *IEEE Transactions on Computer Aided Design*, CAD-2(3):202–210, July 1983.

[40] Arturo Salz and Mark Horowitz. "IRSIM: An Incremental MOS Switch-Level Simulator". In *Proceedings of the 26th Design Automation Conference*, pages 173–178. ACM/IEEE, June 1989.

[41] Mark Santoro and Mark Horowitz. "A Pipelined 64 x 64b Iterative Array Multiplier". In *1988 International Solid State Circuits Conference Digest of Technical Papers*, pages 36–37. IEEE, 1988.

[42] D. Schmidt and L. Druffel. "A Fast Backtracking Algorithm to Test Directed Graphs for Isomorphism Using Distance Matrices". *ACM*, 23(3):433–445, 1976.

[43] Walter S. Scott and John K. Ousterhout. "Magic's Circuit Extractor". In *Proceedings of the 22st Design Automation Conference*, pages 286–292. ACM/IEEE, June 1985.

[44] T. Shinsha, T. Kubo, Y. Sakataya, and K. Ishihara. "Incremental Logic Synthesis Through Gate Logic Structure Identification". In *Proceedings of the 23th Design Automation Conference*, pages 391–397. ACM/IEEE, 1986.

[45] R. Spickelmier and R. Newton. "WOMBAT: A New Netlist Comparison Program". In *Proceedings of the International Conference on Computer Aided Design*, volume CAD-2, pages 70–82, April 1983.

[46] S. A. Szygenda. "TEGAS2 - Anatomy of a General Purpose Test Generation and Simulation System for Digital Logic". In *Proceedings of the 9th Design Automation Workshop*, pages 116–127. ACM/IEE, June 1972.

[47] M. Takashima, Takashi Mitsuhashi, Toshiaki Shiba, and Kenji Yoshida. Programs for Verifying Circuit Connectivity of MOS/LSI Mask Artwork. In *Proceedings of the 19th Design Automation Conference*, pages 545–550. ACM/IEEE, 1982.

[48] George S. Taylor and John K. Ousterhout. "Magic's Incremental Design-Rule Checker". In *Proceedings of the 21st Design Automation Conference*, pages 160–165. ACM/IEEE, June 1984.

[49] Christopher J. Terman. *"Simulation Tools for Digital LSI Design"*. PhD thesis, Massachusetts Institute of Technology, October 1983.

[50] J. Turner. "Generalized Matrix Functions and the Graph Isomorphism Problem". *SIAM Journal of Applied Mathematics*, 16(3):520–526, May 1968.

[51] T. Tygar and R. Ellickson. "Efficient Netlist Comparison Using Hierarchy and Randomization". In *Proceedings of the 22th Design Automation Conference*, pages 702–708. ACM/IEEE, 1985.

[52] L.-T. Wang, N.E. Hoover, E.H. Porter, and J.J. Zasio. "SSIM: A Software Levelized Compiled-Code Simulator". In *Proceedings of the 24th Design Automation Conference*, pages 2–8. ACM/IEEE, June 1987.

[53] T. Watanabe, M. Endo, and N. Miyahara. "A New Automatic Logic Interconnection Verification System for VLSI Design". *IEEE Transactions on Computer-Aided Design*, pages 46–51, June 1986.

[54] L. Weinberg. "A Simple and Efficient Algorithm for Determining Isomorphism of Planar Triply Connected Graphs". *IEEE Transactions on Circuit Theory*, CT-13:142–148, 1966.

[55] T.E. Williams, M. Horowitz, R. L. Alverson, and T.S. Yang. "A Self-Timed Chip for Division". In *Advanced Research in VLSI: Proceedings of the 1987 Stanford Conference*, pages 75–96. Stanford University, MIT Press, Cambridge, 1987.

[56] Y. Wong. "Hierarchical Circuit Verification". In *Proceedings of the 22th Design Automation Conference*, pages 695–701. ACM/IEEE, 1985.