THE UNIVERSITY OF CALGARY

TLSIM: A TIMING AND LOGIC DIGITAL CIRCUIT SIMULATOR

by

Idan Shoham

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

CALGARY, ALBERTA

JUNE, 1993

ISBN 0-315-88628-5

Canada

Name Idan Shoham

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

Electronics & Electrical - Engineering
SUBJECT TERM

0544
SUBJECT CODE

U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS
- Architecture 0729
- Art History 0377
- Cinema 0900
- Dance 0378
- Fine Arts 0357
- Information Science 0723
- Journalism 0391
- Library Science 0399
- Mass Communications 0708
- Music 0413
- Speech Communication 0459
- Theater 0465

EDUCATION
- General 0515
- Administration 0514
- Adult and Continuing 0516
- Agricultural 0517
- Art 0273
- Bilingual and Multicultural 0282
- Business 0688
- Community College 0275
- Curriculum and Instruction 0727
- Early Childhood 0518
- Elementary 0524
- Finance 0277
- Guidance and Counseling 0519
- Health 0680
- Higher 0745
- History of 0520
- Home Economics 0278
- Industrial 0521
- Language and Literature 0279
- Mathematics 0280
- Music 0522
- Philosophy of 0998
- Physical 0523
- Psychology 0525
- Reading 0535
- Religious 0527
- Sciences 0714
- Secondary 0533
- Social Sciences 0534
- Sociology of 0340
- Special 0529
- Teacher Training 0530
- Technology 0710
- Tests and Measurements 0288
- Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS
- Language
  - General 0679
  - Ancient 0289
  - Linguistics 0290
  - Modern 0291
- Literature
  - General 0401
  - Classical 0294
  - Comparative 0295
  - Medieval 0297
  - Modern 0298
  - African 0316
  - American 0591
  - Asian 0305
  - Canadian (English) 0352
  - Canadian (French) 0355
  - English 0593
  - Germanic 0311
  - Latin American 0312
  - Middle Eastern 0315
  - Romance 0313
  - Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY
- Philosophy 0422
- Religion
  - General 0318
  - Biblical Studies 0321
  - Clergy 0319
  - History of 0320
  - Philosophy of 0322
- Theology 0469

SOCIAL SCIENCES
- American Studies 0323
- Anthropology
  - Archaeology 0324
  - Cultural 0326
  - Physical 0327
- Business Administration
  - General 0310
  - Accounting 0272
  - Banking 0770
  - Management 0454
  - Marketing 0338
- Canadian Studies 0385
- Economics
  - General 0501
  - Agricultural 0503
  - Commerce-Business 0505
  - Finance 0508
  - History 0509
  - Labor 0510
  - Theory 0511
- Folklore 0358
- Geography 0366
- Gerontology 0351
- History
  - General 0578
  - Ancient 0579
  - Medieval 0581
  - Modern 0582
  - Black 0328
  - African 0331
  - Asia, Australia and Oceania 0332
  - Canadian 0334
  - European 0335
  - Latin American 0336
  - Middle Eastern 0333
  - United States 0337
- History of Science 0585
- Law 0398
- Political Science
  - General 0615
  - International Law and Relations 0616
  - Public Administration 0617
- Recreation 0814
- Social Work 0452
- Sociology
  - General 0626
  - Criminology and Penology 0627
  - Demography 0938
  - Ethnic and Racial Studies 0631
  - Individual and Family Studies 0628
  - Industrial and Labor Relations 0629
  - Public and Social Welfare 0630
  - Social Structure and Development 0700
  - Theory and Methods 0344
- Transportation 0709
- Urban and Regional Planning 0999
- Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES
- Agriculture
  - General 0473
  - Agronomy 0285
  - Animal Culture and Nutrition 0475
  - Animal Pathology 0476
  - Food Science and Technology 0359
  - Forestry and Wildlife 0478
  - Plant Culture 0479
  - Plant Pathology 0480
  - Plant Physiology 0817
  - Range Management 0777
  - Wood Technology 0746
- Biology
  - General 0306
  - Anatomy 0287
  - Biostatistics 0308
  - Botany 0309
  - Cell 0379
  - Ecology 0329
  - Entomology 0353
  - Genetics 0369
  - Limnology 0793
  - Microbiology 0410
  - Molecular 0307
  - Neuroscience 0317
  - Oceanography 0416
  - Physiology 0433
  - Radiation 0821
  - Veterinary Science 0778
  - Zoology 0472
- Biophysics
  - General 0786
  - Medical 0760

EARTH SCIENCES
- Biogeochemistry 0425
- Geochemistry 0996
- Geodesy 0370
- Geology 0372
- Geophysics 0373
- Hydrology 0388
- Mineralogy 0411
- Paleobotany 0345
- Paleoecology 0426
- Paleontology 0418
- Paleozoology 0985
- Palynology 0427
- Physical Geography 0368
- Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES
- Environmental Sciences 0768
- Health Sciences
  - General 0566
  - Audiology 0300
  - Chemotherapy 0992
  - Dentistry 0567
  - Education 0350
  - Hospital Management 0769
  - Human Development 0758
  - Immunology 0982
  - Medicine and Surgery 0564
  - Mental Health 0347
  - Nursing 0569
  - Nutrition 0570
  - Obstetrics and Gynecology 0380
  - Occupational Health and Therapy 0354
  - Ophthalmology 0381
  - Pathology 0571
  - Pharmacology 0419
  - Pharmacy 0572
  - Physical Therapy 0382
  - Public Health 0573
  - Radiology 0574
  - Recreation 0575
  - Speech Pathology 0460
  - Toxicology 0383
- Home Economics 0386

PHYSICAL SCIENCES
Pure Sciences
- Chemistry
  - General 0485
  - Agricultural 0749
  - Analytical 0486
  - Biochemistry 0487
  - Inorganic 0488
  - Nuclear 0738
  - Organic 0490
  - Pharmaceutical 0491
  - Physical 0494
  - Polymer 0495
  - Radiation 0754
- Mathematics 0405
- Physics
  - General 0605
  - Acoustics 0986
  - Astronomy and Astrophysics 0606
  - Atmospheric Science 0608
  - Atomic 0748
  - Electronics and Electricity 0607
  - Elementary Particles and High Energy 0798
  - Fluid and Plasma 0759
  - Molecular 0609
  - Nuclear 0610
  - Optics 0752
  - Radiation 0756
  - Solid State 0611
- Statistics 0463

Applied Sciences
- Applied Mechanics 0346
- Computer Science 0984
- Engineering
  - General 0537
  - Aerospace 0538
  - Agricultural 0539
  - Automotive 0540
  - Biomedical 0541
  - Chemical 0542
  - Civil 0543
  - Electronics and Electrical 0544
  - Heat and Thermodynamics 0348
  - Hydraulic 0545
  - Industrial 0546
  - Marine 0547
  - Materials Science 0794
  - Mechanical 0548
  - Metallurgy 0743
  - Mining 0551
  - Nuclear 0552
  - Packaging 0549
  - Petroleum 0765
  - Sanitary and Municipal 0554
  - System Science 0790
- Geotechnology 0428
- Operations Research 0796
- Plastics Technology 0795
- Textile Technology 0994

PSYCHOLOGY
- General 0621
- Behavioral 0384
- Clinical 0622
- Developmental 0620
- Experimental 0623
- Industrial 0624
- Personality 0625
- Physiological 0989
- Psychobiology 0349
- Psychometrics 0632
- Social 0451

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled, "TLSIM: A TIMING AND LOGIC DIGITAL CIRCUIT SIMULATOR," submitted by Idan Shoham in partial fulfillment of the requirements for the degree of Master of Science.

Dr. Jun Gu, Supervisor
Dept. of Electrical & Computer Engineering

Dr. L. T. Bruton
Dept. of Electrical & Computer Engineering

Dr. B. Nowrouzian
Dept. of Electrical & Computer Engineering

Dr. X. Mao
Dept. of Mechanical Engineering

Date: September 1/93

Abstract

This thesis discusses the design of the *Timing and Logic SIMulator* (TLSIM): a fast, accurate digital circuit simulator. The development of TLSIM is broken down into a number of issues, each of which is described in detail, resulting in a design choice. Among these issues are the selection of signal and device models, and a scheduling algorithm. Efficient methods for event cancellation, high-impedance handling and function evaluation are also presented.

TLSIM incorporates a computationally efficient device model, based on the following assumptions: circuits can be decomposed into interconnected, unidirectional devices; node voltages always saturate, and delays may be lumped at device outputs. In addition, TLSIM uses a four-valued signal representation (0,1,X,Z).

TLSIM's execution speed is compared to commercial software, and favourable results are measured. This speedup is attributed to a number of design features, including the level of abstraction of TLSIM's device model, its event scheduling and cancellation mechanisms, and optimizations related to memory allocation.

Preface

I dedicate this thesis to my dear grandfather, Adolf Haimov Gershon, who recently passed away after a long battle with cancer.

I would like to thank my family and friends, who have been very supportive during the development of TLSIM, and the writing of this thesis. I would also like to thank my supervisor, Dr. Jun Gu, for our long discussions and his helpful comments.

Finally, I would like to thank the Natural Sciences and Engineering Research Council, who awarded me a PGS-A award during my studies, and who provided Strategic Grant number MEF0045793 to Dr. Gu. This work would not have been possible without NSERC's support.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

As Very Large Scale Integration (VLSI) technology advances, designers are building ever-larger integrated circuits. Currently, state-of-the-art microprocessor designs incorporate up to two million transistors. This figure is the result of a trend in fabrication technology, where the minimum feature size in MOS technologies has been dropping by approximately $0.2\mu m$ every two years, for at least the last decade [23, 14]. This trend is expected to continue for at least another four years.

Two problems arise because of the rising level of integration. The first is that the cost for lithography and fabrication of high device density prototype wafers is rising [23]. The second is that design verification for digital circuits becomes more difficult as circuit complexity increases. These problems together mean that while design errors are becoming even less tolerable, the ability of current tools to verify design correctness is diminishing.

Clearly, there is a constant need for better design verification tools. In the initial stages of circuit design, one of the primary tools used to check designs is a circuit simulator. A circuit simulator reads in a description of the circuit and the signals applied to its inputs, and predicts its behaviour. Using this tool, a designer can check whether a circuit will perform correctly for a given set of inputs.

1.1 Circuit Simulators

There are several types of circuit simulators currently in use [19, 11, 33, 1, 37, 40, 22, 3, 4, 9, 39]. They differ primarily in the trade-off they make between execution speed and the accuracy of their predictions. Models using a higher level of abstraction take less time to execute, but give less detailed predictions about circuit behaviour.

Circuit simulators may be characterized by the models they use to represent signals and devices in the circuit. Signal models may be continuous [21] or discrete, and discrete models may use two or more signal states. There are many possible device models. Among these are: continuous models [11], using resistors, capacitors, etc., switch-level models [38, 9, 12, 33, 31, 1, 5, 2], using ideal switches, resistors, etc., gate models [39, 4] and functional models [18, 29]. In general, the device and signal models are related, but a number of combinations are possible. For instance, a switch-level simulator may use either a continuous or discrete signal representation.

The most precise, but slowest signal model is the continuous model. Continuous domain simulators, such as *Simulation Program with Integrated Circuit Emphasis* (SPICE), solve a set of simultaneous equations in order to calculate the transient response of every circuit element. By using continuous variables to represent signals in the circuit, these simulators obtain good accuracy. Unfortunately, this is accomplished at the cost of very slow execution. Continuous-domain simulators are normally used to design basic components, such as standard cells and gates. The performance characteristics of these components are then used as parameters for simulators with a higher level of abstraction.

Switch-level simulators are faster but less accurate. These simulators model a

circuit as a collection of switches, resistors and capacitors. Switch-level simulators achieve better performance than continuous-domain simulators by using only a small set of discrete values to represent node voltages. Some of them are able to approximate propagation delays by estimating each driving transistor's channel resistance and dynamically calculating the charging / discharging capacitance to each node [1]. Although they do provide an excellent degree of behavioural verification, the ability of these simulators to *accurately* predict the timing characteristics of a circuit is unclear [10].

Gate-level simulators achieve better performance by disallowing *bidirectional* switches [3, 6]. Requiring circuit elements to be strictly unidirectional has a number of significant computational advantages. In general these simulators are faster than switch-level simulators. Gate-level simulators, however, are incapable of directly modeling certain circuit structures, especially MOS circuits that make use of charge storage and sharing effects [9].

Higher-level simulators generally use a procedural language to model complex elements [18, 29]. These are the fastest, because the internal operation of large circuit blocks is not calculated. However, these simulators are the least accurate, because the timing behaviour of these blocks is impossible to predict without simulating their internal operation.

1.2 Compiled vs. Interpretive Simulation

One of the major differentiating factors between simulators is whether or not they generate executable code. Some simulators represent the circuit to be simulated as

a data structure, and apply various algorithms to change its state over simulation time. Other simulators convert the circuit specification into source code (typically in assembler or C), which is then compiled into an executable program. These two approaches are illustrated in Figure 1.1.

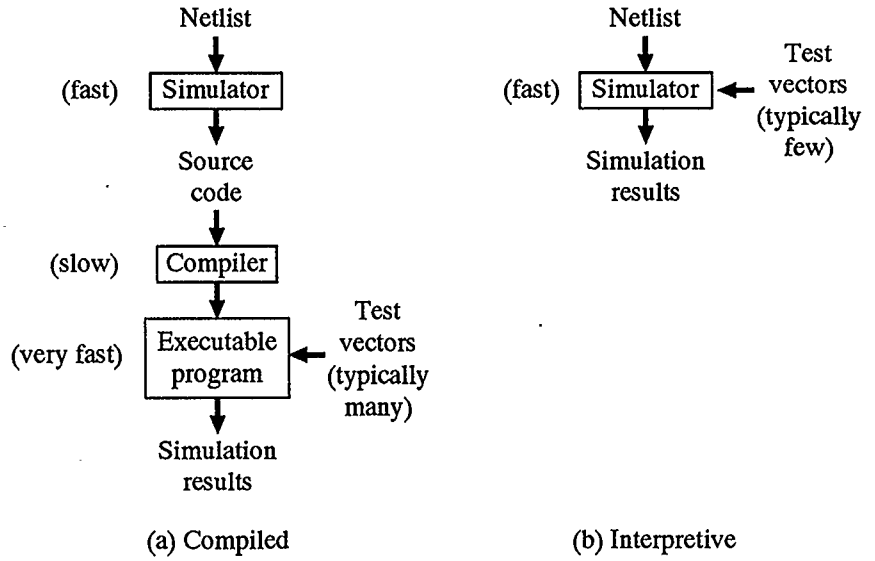


Figure 1.1: The compiled and interpretive simulation approaches

The first approach, called *interpretive simulation*, has the advantage that simulation commences almost immediately after the simulator is invoked, since the effort required to construct a graph of the circuit is minimal. However, due to the complex data structures involved, interpretive simulation is relatively slow. *Compiled simulation*, on the other hand, incurs a large amount of overhead for compilation. However, once the circuit has been compiled, very few machine instructions are required to model devices such as gates and flip-flops. Accordingly, compiled simulation proceeds much faster.

The choice of either compiled or interpretive simulation depends on two factors:

the required flexibility in the device model and the number of test vectors to be processed in a typical simulation run. In order to gain a performance advantage, compiled simulators must use only simple devices, such as gates, which map well to machine instructions. Accordingly, compiled simulators are less flexible in their choice of a device model. For small test vectors, interpretive simulators are faster, since they do not incur compilation overhead. For exhaustive testing, it is usually preferable to use a lower level of abstraction, so compiled simulation is acceptable. If the number of test vectors is sufficiently large, the performance gain attained by replacing complex device evaluation functions with short sequences of machine instructions makes up for the compilation overhead.

The relationship between simulator performance and test vector size is illustrated in a general way in Figure 1.2.

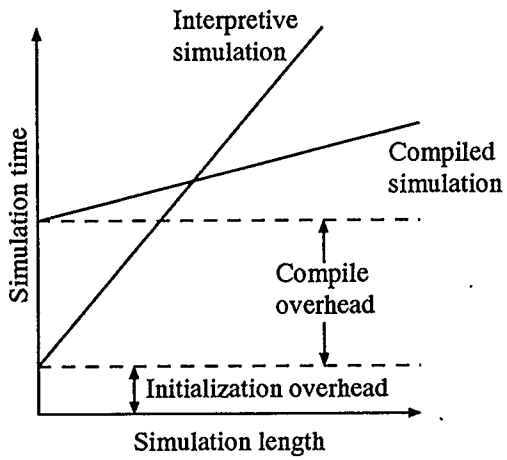


Figure 1.2: Performance of compiled vs. interpretive simulation

1.3 The TLSIM Digital Circuit Simulator

This thesis describes the design of the *Timing and Logic SIMulator* (TLSIM): an interpretive digital circuit simulator which uses an efficient hybrid device model. TLSIM is meant for use in circuit *design*. It uses a functional device model, combining features of both the gate-level and algorithmic models. This model produces results at least as accurate as traditional gate-level simulators, while at the same time achieving much of the performance and flexibility previously associated only with algorithmic models.

TLSIM is meant for circuit design, *not* for exhaustive circuit testing. This design goal is crucial to many of the design decisions made in its development. In the circuit design process, a user enters a circuit description, and tests it with some input vectors. This process is normally repeated very often, so it is important for a simulator to give fast turn-around. Accordingly, start-up overhead in simulation should be minimized. In particular, a compiled circuit simulator would be inappropriate for circuit design, as *circuit compilation* could easily take more time than the simulation runs. With this in mind, TLSIM was implemented as an interpretive simulator.

1.4 Project Background

TLSIM was developed as a component in a larger project, whose goal was to develop a set tools for the computer-aided design of Path Programmable Logic integrated circuits [16]. This tool set consists of a PPL circuit editor, a netlist extractor, the TLSIM simulator, an interactive waveform editor, and a host of smaller utilities.

Although it is used as a component in this larger system, TLSIM was built with

flexibility in mind, and is capable of simulating any digital circuit.

1.5 Thesis Outline

The rest of this thesis is organized as follows:

Chapter 2 gives background relevant to digital circuit simulator design in general. It describes different types of simulators, and discusses the major design issues which must be addressed before implementing a simulator.

Chapter 3 describes how the issues identified in Chapter 2 are addressed in the TLSIM simulator. It describes the basic design decisions made in the development of TLSIM, and justifies why they were made.

Chapter 4 gives detailed descriptions of some of the key algorithms in the TLSIM simulator. The need for each algorithm and the manner in which it contributes to simulator performance is discussed here.

Chapter 5 gives an analysis of the computational complexity of the algorithms from Chapter 4.

Chapter 6 presents some experiments which were undertaken to quantify the performance of TLSIM. It gives execution times for various benchmark circuits, and compares them to execution times incurred by Verilog, a commercially available circuit simulator.

Chapter 7 contains a more in-depth discussion of some of the design decisions made in the TLSIM simulator. This discussion emphasizes how these choices are relevant within the broader framework of circuit design, rather than only simulation.

Chapter 8 concludes this thesis, with a summary of the significant results.

Chapter 2

The Components of a Simulator

This chapter gives an overview of the major issues in the design of a circuit simulator. The first of these are the selection of signal and device models. Sections 2.2 and 2.3 discuss how signals and devices may be represented in a simulator. In the case of an event-driven simulator, an ordered queue of pending events must be maintained. Section 2.4 gives an overview of this scheduling problem. Section 2.5 discusses different approaches to setting the initial state of the circuit being simulated. Finally, section 2.6 discusses the merits of various degrees of expressiveness in a simulator's input language.

2.1 Definitions

Before proceeding with this chapter, we need some definitions:

Device: A device is a circuit component. Devices are atomic – within the simulation framework, they may not be decomposed into smaller circuit elements. For instance, in a gate-level simulator devices are gates, while in a switch-level simulator devices are transistors, resistors and capacitors.

Node: A node is an equipotential conducting region in a circuit. It acts as a single electrical point. In practice, nodes represent metal or polysilicon wires and traces.

Nodes are said to have a *fan-in.* A node's fan-in is the set of all devices whose outputs are connected to it. Nodes also have a *fan-out.* A node's fan-out is the set of all devices whose inputs are connected to that node.

Netlist: A netlist is a description of device interconnections used to model a circuit's topology. A netlist consists of *devices* and *nodes*, where devices are connected to one another using nodes.

Netlists may be *flat*, containing only devices and nodes, or they may be *hierarchical*, in which case they may also contain other netlists.

Circuit Inputs: The inputs to a circuit are the set of nodes whose fan-in is empty. During simulation, the user is responsible for setting the state of these nodes.

Circuit Outputs: The outputs of a netlist are the set of nodes whose fan-out is empty. During simulation, the simulator reports the state of these nodes to the user.

Device Output: Uni-directional devices, such as gates, are said to have outputs. A device's output is the value which it tries to apply to its node. Note that more than one device may be connected to a single node, so a device's output is not necessarily equal to the node's value. Note also that one device may have multiple outputs.

The relationship between a device output and a device's output *node* is illustrated in Figure 2.1.

Event: To model device propagation delays, some simulators *schedule* transitions at nodes, and later retrieve them in a time-increasing order. An event is a data structure which identifies a node, a time and a new signal value.

Queue: When a simulator makes use of events, it needs a mechanism for scheduling and retrieving events. In the context of circuit simulation, a queue is a mechanism for event scheduling which guarantees that event retrieval will occur in a time-increasing order.

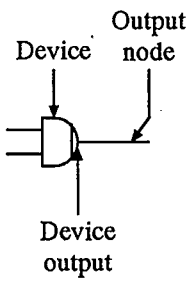


Figure 2.1: The relationship between a device output and a device's output node

2.2 Signal Representation

A circuit simulator functions by modeling how devices in a circuit modify signals over time. Before considering how these devices work, a signal representation must be chosen:

There are a number of ways to model electrical signals in a digital circuit. Since normally signals in the circuit refer to node voltages, it is natural to represent signals with real numbers, which stand for voltage levels. Because of computational complexity, we might restrict the voltage levels to a discrete set of approximate values.

At the extreme, we could use just two values – representing the binary digits 0 and 1. We might also generalize to a four-valued system, $\{0, 1, X, Z\}$, where X represents an unknown value, and Z indicates the absence of any signal.

Each of these signal representation schemes bears more detailed consideration, as it leads to different simulation strategies.

2.2.1 Continuous Signals

The use of continuous signals requires that devices operate on continuous inputs and produce continuous outputs. Because of the smooth variation of outputs with input values, this representation necessitates that device equations be solved simultaneously. This is only feasible for small circuits. Due to computational cost, it is impossible to simulate even moderate-sized digital circuits, with 100 or more gates, with this model.

2.2.2 Discrete Signal Levels

Since digital circuits are meant to operate only on binary numbers, rather than represent node voltages using real numbers, it is possible to use a set of discrete values. Typically, discrete schemes use a combination of voltage level and charge strength to characterize the signal at a node. "Charge strength" may indicate the dynamic node capacitance or the node's capacitive charge with respect to the power or ground levels. For instance, Table 2.1 shows one signal representation which uses two logic levels and four charge strengths. Alternately, a mixed model might use a continuous variable to represent node capacitance or accumulated charge.

Using this representation, charge decay may be modeled by a sequence of signal

Table 2.1: A 2-valued, 4-strength signal representation

Charge Strength	Node voltage	
	level=0	level=1
Decaying	1	5
Weak	2	6
Normal	3	7
Strong	4	8

states at a node. If a buffer was initially at logic 1 and was then turned off, a simulator could model the node voltage as a sequence of signal values: first 7, then 6, and finally 5. These transitions would be scheduled with appropriate time delays.

2.2.3 Four-Valued Signal Representation

Digital circuits operate on continuous voltages, but these voltages are meant to represent only two binary values, 0 and 1. In order to reduce the computational complexity of simulation, it seems natural to use a two-valued signal representation.

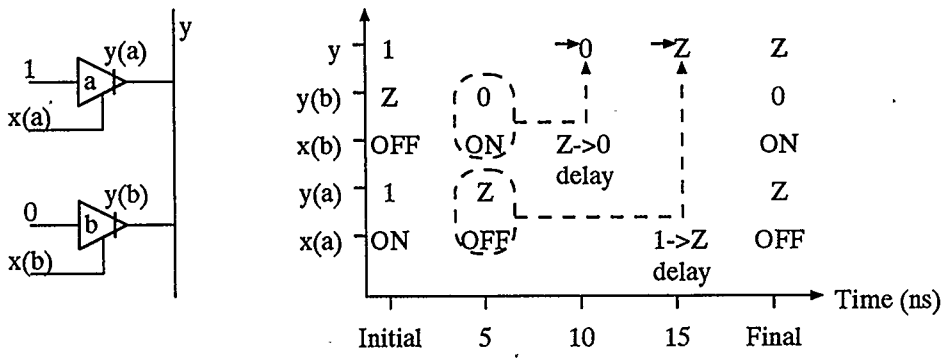
Unfortunately, the two-valued model has severe limitations. Real digital circuits actually make use of three signal values: 0, 1, and *high-impedance*, i.e., no signal. Without a high-impedance state (Z), it would be impossible to simulate tri-statable buses, fundamental to modern circuits. Furthermore, when a circuit is turned on, all of its nodes quickly settle to one of the three values mentioned above. Before beginning the simulation, however, a simulator has no way of calculating which node starts at which value, since some nodes may power up to random values determined by thermal noise. To account for this, a fourth signal value must be added.

Rather than assigning an arbitrary initial value to every node, we may designate

X to represent signals whose value has not been specified or cannot be calculated by the simulator. The X state models the initial value of all nodes in the circuit. In addition, the X state may be used by the circuit designer as a deliberate input value, in order to study the effect of floating terminals on the circuit.

2.2.4 The High Impedance State

Special care must be taken with the high impedance state in a four-valued simulator. Due to different propagation delays, two devices connected to a single node may cause a dangerous sequence of events to occur, as shown in Figure 2.2.



(a) A sample tristate node (b) Event sequence with an error in the final state

Figure 2.2: A problem with event sequencing and high impedance

In the figure, two tri-statable buffers, a and b, are connected to a single node. Initially, a is on, b is off, and y is set to 1, reflecting a's input. At $t = 5ns$, a turns off while b turns on. Let the turn-on delay for these buffers be $5ns$, and the turn-off delay be $10ns$. The two events at $t = 5ns$ will cause both buffers' outputs to change states, scheduling two events at $t = 10ns$ and $t = 15ns$ at node y. At $t = 10ns$, the value of y will change from 1 to 0. At $t = 15ns$, another event will set the value of

y to Z. The final value of y will therefore be Z.

Since the final state of buffer b is ON, the *correct* final value of y is 0. The error results from a transient conflict between $y(a)$ and $y(b)$, which are both asserted on node y from $t = 10ns$ to $t = 15ns$. Blindly processing events from the queue can cause simulation errors when the Z signal value is involved. A four-valued simulator must take some precaution to ensure that transient signal conflicts do not introduce errors into the simulation.

2.3 Device Representation

By selecting a particular signal representation, we limit the number of device models to just those that are compatible with the signal model. With a continuous signal representation, devices must be modeled according to their time-domain voltage/current characteristics. Using the logic level/charge strength or four-valued signal models, there are many more possibilities. For instance, we may model devices as ideal switches, gates, truth tables, procedures, etc.

Due to its computational limitations, we will not give further consideration to the continuous signal representation. There remain, however, four major device models applicable to discrete signals: switches, gates, functional blocks and procedures. These models are discussed in this section.

This section also contains a brief discussion of event-driven simulation, which is normally used with the gate, functional and algorithmic device models, since these models all exhibit the property of uni-directional signal propagation.

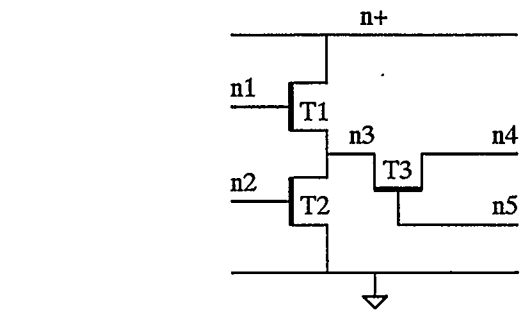
2.3.1 Switch Primitives

Metal-Oxide Semiconductor (MOS) digital circuits may be decomposed into transistors, resistors and capacitors. The transistors, in turn, may be approximated using ideal switches, which conduct when turned on. Using the logic level/charge strength signal model, it is possible to estimate the behaviour of these switches.

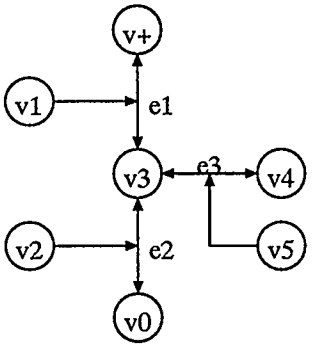
With the switch model, a simulator represents the circuit using an undirected graph $G(E, V)$, whose vertices V represent electrical nodes in the circuit, and edges E represent the subset of transistors which are conducting at any given time. This model is illustrated in Figure 2.3. In Fig. 2.3(a), a sample circuit consisting of three transistors and five electrical nodes is shown. If every transistor is "on," then this circuit would be represented by a graph as shown in Fig. 2.3(b). If only transistor T1 is turned on, the graph would look like Fig. 2.3(c).

In an idealized model, we may consider conducting transistors to have no channel resistance. This implies that every set of nodes in the circuit which are connected to one another by conducting transistors may be thought of as one effective node, with the same voltage everywhere. If transistor T1 is the only transistor turned on, then nodes n+ and n3 form a single "effective node," as shown in Fig. 2.3(c). In this case, since the voltage of n+ is fixed (n+ is the power supply), the voltage at n3 is also known.

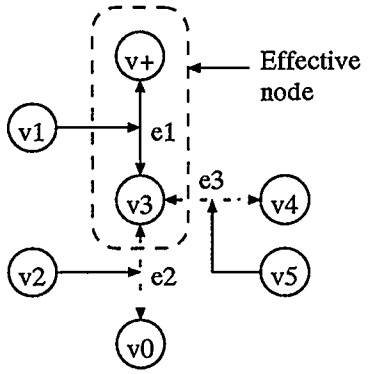
The task of a switch-level simulator is to iteratively identify which transistors in the circuit are switched on. This calculation is equivalent to finding which edges $\{E_i\}$, out of all possible edges in the circuit, are part of the circuit graph at every time step. Given the set of turned-on transistors (edges $\{E_i\}$), the simulator identifies "effective



(a) Example circuit



(b) Edge / vertice representation



(c) An effective node

Figure 2.3: Graph representation of a circuit

nodes" which contain either the power or ground node, and propagates these voltages across the "effective nodes." By iterating this procedure, the simulator is eventually able to calculate the voltage at every node in the circuit.

Due to the complex bidirectional signal flow in the switch-level circuit model, simulators must solve large sets of simultaneous equations to identify which transistors are turned on at every time step. Fortunately, it is possible to reduce the computation by making use of the fact that the set of "effective nodes" only changes incrementally at each time step. Furthermore, there are efficient algorithms for solving the circuit's state equations with this model, which make use of constraints on signal and transistor behaviour. These optimizations together mean that typically switch-level simulations are only about an order of magnitude slower than the next higher level of abstraction – gate level simulation [9, 6].

2.3.2 Discrete Event Incremental Simulation

At all levels of abstraction higher than the switch model, devices are considered to propagate signals in a single direction only. The network graph thus becomes a *directed* graph $G(E, V)$, where vertices V represent electrical nodes in the circuit, and edges $\{E\}$ represent *unidirectional* devices.[1]

Consider the circuit illustrated in Figure 2.4. $v1 - v10$ are vertices representing electrical nodes in the circuit. Nodes have a definite state, which usually denotes a voltage level. Nodes are connected to one another by means of devices. Edges representing devices in the circuit are labeled $e1 - e4$. In a unidirectional model, edges are directed. That is, they have definite inputs and outputs. For instance, the

[1] In practice, a single "edge" may start from multiple vertices, and end at multiple vertices.

state of vertice $v7$ affects edge $e4$, and may thus affect vertice $v10$. However, there is no way for $v10$ to affect the state of $v7$.

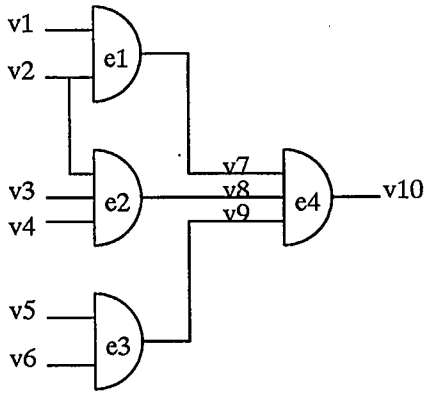


Figure 2.4: Sample unidirectional circuit

The directed nature of a circuit graph leads to the concepts of *fan-in* and *fan-out* (see Section 2.1). The fan-in of a vertice is the set of edges which affect its state (i.e., devices whose outputs are connected to that node). The fan-out of a vertice is the set of edges that begin in that vertice (the devices whose inputs are connected to that node). In the example shown, the fan-out of $v1$ is $\{e1\}$, and the fan-out of $v2$ is $\{e1, e2\}$. The fan-in of $v7$ is $\{e1\}$, and that of $v8$ is $\{e2\}$.

Using a directed graph for circuit simulation is advantageous since the circuit may be simulated incrementally. Given a known circuit state at some time t_0, inputs applied to the circuit at time t_0 will cause some device outputs to change state after various delays, at $t_0 + \Delta T_1$, $t_0 + \Delta T_2$, $\cdots$, where $\Delta T_1, \Delta T_2, \cdots$ are device propagation delays. The simulator need only reevaluate the state of the circuit at times $t_0 + \Delta T_1$, $t_0 + \Delta T_2$, $\cdots$. This is more efficient than evaluating the circuit at every time step.

In addition, since only *some nodes* change at times $t_0 + \Delta T_1$, $t_0 + \Delta T_2$, $\cdots$,

the simulator only has to reevaluate the states of *some devices* at these times. In particular, it must reevaluate devices in the fan-out of nodes whose value has changed.

These observations are used to formulate a general procedure for simulating circuits which only contain uni-directional devices, and whose nodes take on discrete values [39]. When inputs are applied to the circuit, the simulator schedules *events*, indicating which nodes must change states, and when. The simulator retrieves these events in order, and for each event executes the following steps:

1. Update the node's state.

2. Propagate the new state to the node's fan-out.

3. Re-evaluate each device in the fan-out.

4. Schedule one new event for each device output that changed.

Using this sequence of steps, device evaluation is only performed when one or more inputs to each device have changed. If only a small fraction of a circuit's nodes change state in a given time step, the above operations significantly improve performance over the "brute force" approach. A widely accepted estimate of typical activity rates is that only about 1% of the nodes in a typical circuit are active at any one time [39]. Using this figure, event-driven simulation is approximately 100 times faster than a "brute-force" approach, which evaluates every device at every time step.

2.3.3 Gate Primitives

Perhaps the most intuitive method for modeling simple, unidirectional digital devices is to build gate primitives into the simulator. In this scheme, the user specifies the circuit topology in terms of interconnected gates. When the simulator needs to evaluate a gate, it executes a procedure which evaluates that type of gate.

A typical algorithm for device evaluation is shown in Figure 2.5. In this algorithm, the *EvaluateGate* function examines the gate, and selects an appropriate function to calculate its new output value. The *EvalAND* function is one such function, used to evaluate AND gates.

The advantages of this model are two: it is simple and may readily be used in both compiled and interpretive simulators. In a compiled simulator, this model executes very rapidly since at most a few machine instructions are needed to replace the evaluation algorithm in Figure 2.5.

The gate-primitive model has the disadvantage that it only allows for a fixed set of device types. There is no way, short of modifying the simulator itself, for the user to add new device types. Furthermore, the gate-primitive model is awkward in cases where the target technology allows for devices which perform more complex tasks than the devices supported by the simulator. Such devices must consequently be modeled as a collection of gates, which degrades simulation accuracy and slows down execution.

2.3.4 Functional Block Primitives

Rather than providing a finite set gate primitives, we can assign truth tables to each type of gate. Since truth tables can just as easily be loaded at run time as when the

```
Inputs: gate type, number of inputs, array of input values
Outputs: new gate output value

Function EvaluateGate {
    switch(gate type)
        case AND: return EvalAND(Ninputs,inputs)
        case OR: return EvalOR(Ninputs,inputs)
        case NAND: return EvalNAND(Ninputs,inputs)
        case NOR: return EvalNOR(Ninputs,inputs)
        case XOR: return EvalXOR(Ninputs,inputs)
        case NOT: return EvalNOT(Ninputs,inputs)
    }


Inputs: the inputs to an AND gate
Outputs: new output value of AND gate

Function EvalAND {
    for( each input )
        if( input is equal to zero )
            return 0
    return 1;
    }
```

Figure 2.5: Sample gate evaluation algorithm

simulator is compiled, it is possible for the user to extend the library of primitive devices. Truth tables for some common Boolean functions are shown in Table 2.2. A typical device evaluation algorithm that uses this model is shown in Figure 2.6, where the truth table is stored as an array of integers. The *EvaluateTable* function converts an array of function arguments into a single integer, and returns the contents of the truth table at that location.

Table 2.2: Truth tables for some common gates

Input 1	Input 2	AND	OR	NAND	NOR	XOR
0	0	0	0	1	1	0
0	1	0	1	1	0	1
1	0	0	1	1	0	1
1	1	1	1	0	0	0

Inputs: A truth table, the number of inputs, and a pointer to an array of input values
Outputs: The function value for the given arguments

```
Function EvaluateTable {
    let offset=0
    for( each input )
        shift offset left 1 bit
        offset = offset or this input
    return table[offset]
    }
```

Figure 2.6: A typical truth-table evaluation algorithm

Using this model, we can easily build more complex primitives by specifying truth

tables for their logical functions. For instance, a binary adder would be specified by a pair of primitives: a sum-forming element and a carry-forming element, as shown in Table 2.3.

Table 2.3: Truth tables for a full adder

a	b	$carry_{in}$	Sum	$Carry_{out}$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

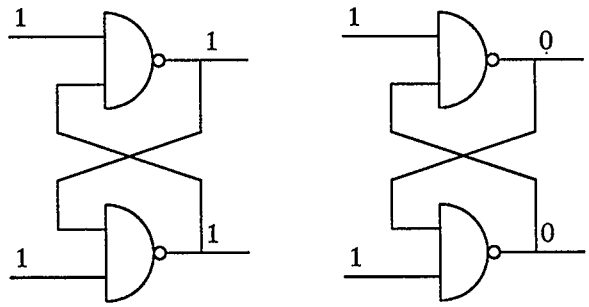
The disadvantage of this model is that there is no significant performance advantage to implementing truth tables as compiled code, so this model is only effective in an interpretive simulator.

2.3.5 Memory Elements

In the last two sections, methods for representing simple feed-forward devices were discussed. So far, none of the models explicitly specify how to simulate memory elements. In actual circuits, flip-flops, registers and latches are used to store data for one or more clock intervals. To be useful, a digital circuit simulator must provide a means of simulating such memory elements.

It is usually not practical to simulate memory elements as collections of gates. Consider the pair of NAND gates in Fig. 2.7(a). If the propagation delays of the two

devices are equal, the two outputs will simultaneously change to 0, back to 1, and oscillate indefinitely. This *erroneous* oscillation is due to the fact that, in practice, no two devices have exactly the same propagation delays, so such feedback loops settle at one state or the other. In the simulation, however, only nominal values of delays are known, so multiple devices with identical delays are common. While this instability problem is not unique to the small feedback loops found in flip-flops, due to their widespread use this is where the problem normally arises. [2]



(a) Unstable feedback state (b) Alternate state

Figure 2.7: Oscillation due to identical delays in feedback loops

In order to avoid this problem, and also speed up computation, most modern circuit simulators provide a special facility for simulating memory elements. A primitive memory element may store one or more binary digits. It may have one of the following configurations:

- A level-sensitive latch, which follows its input as long as it is enabled.
- An edge-sensitive flip-flop, which samples its input on either a rising or falling clock edge.

[2]Currently, no general method is known for dealing with this instability in circuit simulators [4].

Additionally, the memory element may or may not allow for asynchronous set and clear logic.

2.3.6 Higher Level Device Models

Rather than force the user to construct the circuit with gates or truth tables, some simulators provide a complete procedural language, such as *Very High Speed Integrated Circuit (VHSIC) Hardware Description Language* (VHDL) or Verilog. Using these languages, the user may specify his/her own models, and simulate the circuit at any desired level of abstraction. This approach is illustrated in Figure 2.8, which shows two possible ways to model an AND gate.

```
module and3_type1(y,a,b,c)
      if (a==0 or b==0 or c==0)
            y=0
      else
            y=1
endmodule


module and3_type2(y,a,b,c)
      y = a&b&c
endmodule
```

Figure 2.8: Two procedural models for 3-input AND gates

The procedural device model is useful primarily in cases where the behaviour of one or more modules in a larger system has been specified, but the module has not

yet been implemented. This type of construct is especially useful for mixed level simulation, in order to help test other modules which *have* been implemented. The main disadvantages of this model are that it may be inaccurate, and it is difficult to automatically synthesize a hardware design from a procedural description.

2.4 Event Scheduling

Regardless of the device and signal models selected, it is useful to associate delays with devices. There are many possible forms for the delays: they may be nominal values, statistical forms such as minimum/average/maximum, or physically related parameters such as rise and fall times. In addition, delays may be static or dynamic, reflecting the current state of a node's fan-out [2].

In order to ensure causal behaviour, the simulator must process events at all nodes in a time-increasing order. If this is not done, a simulator might evaluate one device, say device A, at simulation time t_0, and later in the sequence of evaluations it might calculate the state of another device, say B, at simulation time $t_0 - \Delta T$. If there is a signal path from any output of device B to any input of device A, this sequence may cause errors in the simulation.

Events may be scheduled in any order. For example, for the three time steps $t_2 > t_1 > t_0$, events scheduled for $\{t_0, t_1, t_2\}$ may be added to the queue in any order. Because they must be *retrieved* in the sequence $\{t_0, t_1, t_2\}$, a digital circuit simulator must provide a mechanism for sorting the event queue.

The task of adding events to a queue in arbitrary order, while retrieving them in time-increasing order is a classical problem in computer science [36]. Accordingly,

there are a number of available solutions. Two of these are briefly presented in this section.

2.4.1 Heapsort Scheduling

A classical solution to the scheduling problem is the heapsort algorithm [36], which uses a heap to represent a queue. The heapsort algorithm makes use of a special array **h**, called a heap, which satisfies the following condition:

$$\mathbf{h} = h_1, h_2, \cdots, h_N \tag{2.1}$$

$$\forall i, h_i \leq h_{2i} \text{ and } h_i \leq h_{2i+1} \tag{2.2}$$

Clearly, element h_1 must be the smallest element (earliest event) in the array. This structure lends itself well to scheduling, where we only retrieve the smallest element in the queue (earliest event). The heapsort algorithm resolves the scheduling problem by adding and retrieving events to and from the array **h**, while *maintaining the condition in Equation 2.2.*

Before considering how a heap works, we need some definitions: In a heap, the *parent* of element h_i is $h_{i/2}$. A *child* of element h_j is either h_{2j} or h_{2j+1}. A *path* through the heap is a sequence of the form: h_i, $child(h_i)$, $child(child(h_i))$, $\cdots$.

In scheduling, we may add a new element (an event in our case) to a heap by appending it to the array. After the element is appended, we examine its parent, and see if it is larger, i.e., if it is a later event. If so, we swap the parent and child elements, and repeat the procedure along a *path* until we reach an element that is *not* larger than the newly inserted element.

To retrieve an entry from a heap, i.e. to get the next event, we just retrieve h_1, since it is the smallest element, by virtue of Equation 2.2. After this is done, however, the empty space must be filled in such a way that the heap condition continues to be satisfied. This is done by moving the smaller of h_1's children to location h_1, and repeating the same process for that child along a path to the end of the array.

Since both the queuing and dequeuing procedures described above iterate along a single path in the heap, and since, for a heap containing n elements, the average path length is $\frac{1}{2}log_2(n)$, both queuing and dequeuing using a heap have a computational complexity of $O(log_2(n))$.

2.4.2 Time Wheel Scheduling

Since there are a finite number of devices in the circuit, each circuit has a maximum propagation delay, at the output of the slowest device. If a circuit uses only integer-valued delays, then there is only a finite set of possible delays. At most, this set consists of the integers $\{1, 2, \cdots, MAXDELAY\}$. We can take advantage of this observation by using an array to track pending events.

The time wheel algorithm represents the event queue as an array of *event lists*. The array length, N, is set to equal the longest delay in the circuit. The array is traversed in a circular fashion: events scheduled for time step i are stored at array position $i \ mod \ N$. Clearly, scheduling with a time wheel is an $O(1)$ operation: to insert an event at some delay P after the current timestep i, it is added to the event list at array position $(i + P) \ mod \ N$.

The basic concepts of time wheel scheduling are illustrated in Figure 2.9. A typical algorithm for accessing the event queue is shown in Figure 2.10. For the sake

of brevity, the code used to schedule inputs to the circuit, which may arrive later than the simulation time plus the maximum delay, is omitted.

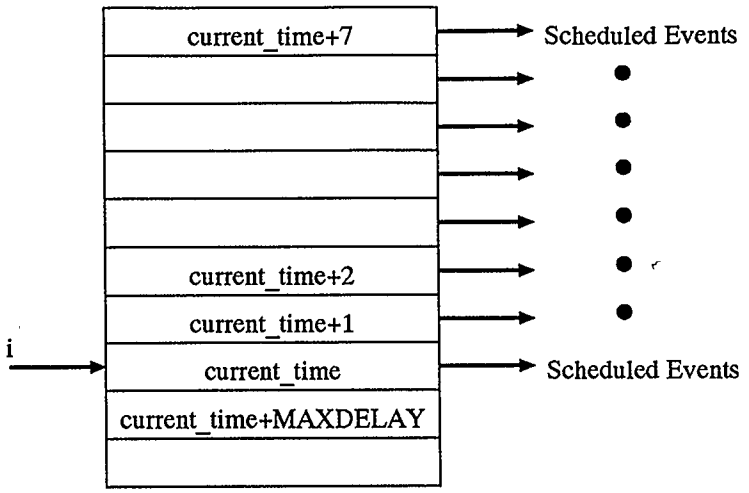


Figure 2.9: The time wheel scheduling paradigm

The primary advantage of the time wheel scheduling algorithm is that scheduling is very fast: Adding a new event to the queue is an $O(1)$ operation. As long as different devices in the circuit have many different delays, and there is a reasonably large amount of activity in the circuit, the **while** loop in the *GetNextEvent* function is rarely executed more than once, so event retrieval also averages $O(1)$.

The disadvantage of the time wheel algorithm is that for periods of low circuit activity, or for cases when most of the circuit activity is in devices with a single long delay, the **while** loop is executed frequently, thus slowing down event retrieval. Fortunately, the code in the **while** loop is short, so even this situation should not exact a heavy penalty.

Inputs: None
Outputs: The earliest pending event in the queue

```
let   list = an array of pointers to linked lists of events
      offset = an integer representing the current location in the time wheel
      current_time = the simulation time

Function GetNextEvent {
      /* while no events in this time step */
      while( list[offset] is an empty list )
            offset = (offset+1) mod MAXDELAY
            current_time = current_time + 1
      new = list[offset]                  /* remove first event in this list */
      list[offset] = new→next
      return new
      }
```

Inputs: A new event, and the delay after which it should be processed
Outputs: None

```
Procedure Schedule {
      let pos = (offset + delay) mod MAXDELAY
      new→next = list[pos]                /* add to appropriate list */
      list[pos] = new
      }
```

Figure 2.10: Time wheel scheduling algorithms

2.5 Circuit Initialization

Regardless of the device and signal models, we may represent the circuit to be simulated using a graph $G(E, V)$, where the edges $\{E\}$ represent devices, and the vertices $\{V\}$ represent nodes. In all but the switch model, the edges are directed – that is, signals flow from one vertice to one or more others.

Given that a circuit has been loaded into a graph $G(E, V)$ in memory, the simulator must set the initial state of the vertices $\{V\}$ in the graph. In practice, when a physical circuit is turned on, the initial state of the vertices $\{V\}$ is a complex function of the technology, layout, circuit topography and random thermal effects. Due to this complexity, there is no practical way for the simulator to accurately calculate the initial state of the circuit.

There are a number of methods for initializing a circuit in a logic simulator. The most common one is to set the value of each node to X (this presumes a four-valued signal model, or some variant). In this way, the simulator is making no undue initial assumptions. An alternate approach extends this by scheduling events to the ground and power nodes, so that the state of some of the devices in the circuit may be calculated before the simulation begins.

There are inherent problems with the method of setting the initial state of every member of $\{V\}$ to X. Consider a simple counter, made up of one or more flip-flops as shown in Figure 2.11. One of the possible uses of this counter is frequency division. In a practical circuit, a designer need not provide a reset signal to this network, since its function is independent of its initial state. As long as it has *some* initial state, the counter will correctly perform frequency division. The only characteristic of interest

in this device is the rate at which it progresses through states. If we initialize each node in this device to X, and never apply a reset signal, then the simulator will never change the state of the device, and the counter will fail. This is strictly a simulation problem, since in practice the counter starts at an arbitrary state, and functions correctly.

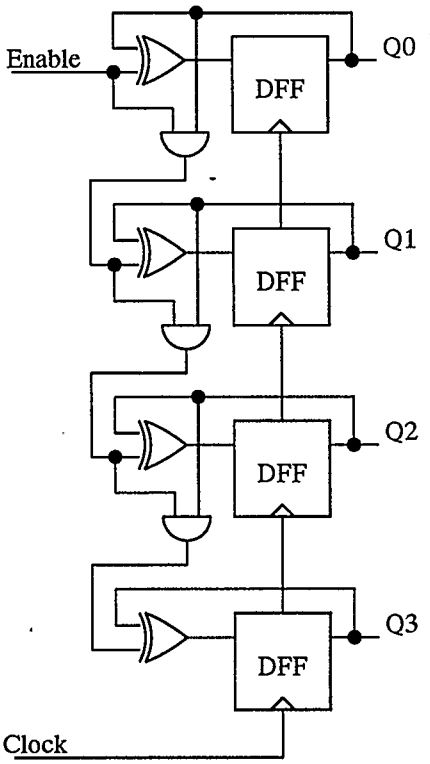


Figure 2.11: A frequency-division circuit, which does not have to be reset

Unfortunately, there is no known method to accurately initialize a circuit. In particular, it is impossible to guarantee the simulation results for any circuit which functions correctly regardless of its initial state, and whose initial state is not explicitly set.

Fortunately, this problem is easily solved by adding some small degree of flexibility

to the simulator. The user may either provide a secondary reset signal, which is only used to initialize "initial-state-independent" subcircuits. This signal can then be ignored by any hardware synthesis software. Alternately, the simulator may provide a facility for explicitly giving the initial state of nodes in the circuit. This calls for more care on the part of the user, but guarantees satisfactory simulation results.

2.6 Simulation Language Expressiveness

The final issue in simulator design is the expressiveness of the simulator language. The simulator language is used to describe the circuit, and in some cases device operation. The capabilities of different simulation languages vary enormously. The simplest languages, such as SPICE, are only capable of listing network topologies, while more complex languages, such as VHDL and Verilog, include complete procedural constructs.

Choosing a degree of expressiveness for a simulator language is a trade-off between performance and debugging capability. Adding high-level constructs is useful for modeling sections of a circuit before they are implemented. However, interpreting these constructs takes more time than interpreting a simple language, so the simulation of simple devices is somewhat slowed down.

Using only low level constructs makes it possible to ensure that it will always be possible to automatically synthesize a hardware design from the netlist. On the other hand, debugging large circuits is more difficult, since every module must be specified simultaneously.

Chapter 3

The TLSIM Digital Circuit Simulator

TLSIM stands for *Timing and Logic SIMulator.* It is an interpretive simulator, incurring low overhead for simulation start-up. Since building a circuit graph takes little time, simulation commences almost immediately after the TLSIM is executed. In addition its short delay, TLSIM provides rapid simulation by using an optimized device model. This model is capable of describing devices with complexity similar to that of small- and medium- scale integration discrete components.

This chapter addresses the major issues the development of the TLSIM digital circuit simulator. TLSIM uses a four-valued $(0, 1, X, Z)$ signal representation and a hybrid device model, with some features in common with both the truth-table and algorithmic models. This model results in excellent simulation performance.

3.1 Simulator Execution

TLSIM is a circuit simulator intended for *circuit design* rather *test generation.* In the design phase, designers specify the contents and topology of a circuit and apply some short tests to check their evolving design. This procedure is repeated frequently, so every effort should be made to shorten the turnaround time to a minimum.

In practice, simulators are used for two types of testing. In the first type of testing, the *circuit* is checked for correct operation. This testing is performed during circuit design, and is referred to as *conventional* simulation.

A second testing phase is performed in order to develop a set of input vectors for use in identifying faulty circuits after fabrication. To do this, some test vectors are applied to the circuit, and the simulation results are recorded. The same test vectors are then applied to a slightly modified circuit, where a manufacturing defect has been introduced. Simulation results from the two runs are compared, and if they differ the input vectors are recorded, since they have shown the ability to identify circuits with that particular manufacturing flaw. This testing phase is referred to as *fault* simulation.

Just as there are two types of testing performed on circuit designs, there are two types of simulators. The first, a compiled simulator, converts the circuit topology into source code, either in a high-level language like C, or in assembler. This code is then compiled into an executable file. This approach gives very rapid simulation at the cost of high overhead, due to the compile and link steps.

The second approach to simulation is to build data structures in memory to represent the circuit topology, and to change their state according to rules about device operation. This approach, called interpretive simulation, typically runs about an order of magnitude slower than compiled simulation, but has the significant advantage of very low overhead. That is, building a graph to represent a circuit takes very little time.

Since TLSIM is meant for circuit *design*, it uses an interpretive paradigm for execution. This means that for short simulation runs, i.e., typically not exceeding more than a few million input transitions, it is faster than compiled simulation, since it does not incur a compilation overhead. For exhaustive testing, however, a compiled simulator is preferable.

Using the interpretive paradigm, TLSIM execution proceeds in the following sequence of steps:

1. Load a library of Boolean functions, represented as truth tables. (By default, this is read from the file INTRINS.LIB.)

2. Load a library of device definitions, whose behaviour is specified using the above functions. (By default, this is read from the file DEVICE.LIB.)

3. Load a netlist of the circuit to be simulated.

4. Construct a graph in memory representing a flattened netlist, i.e., the hierarchy of network definitions in the netlist is converted into a flat representation consisting of only nodes and primitive devices.

5. Initialize the power nodes, Vdd and Gnd, and commence the simulation.

6. Load the first user-specified inputs into the circuit and simulate.

7. Continue simulating until there are no more inputs left, and the circuit has either settled or a time limit has expired.

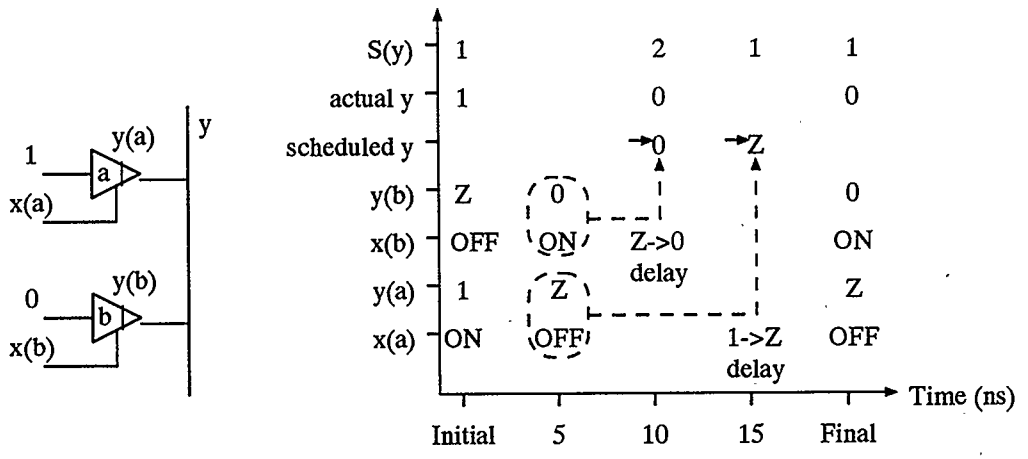
3.2 Signal Representation

At the logic design stage, the designer rarely knows what node capacitances in the circuit will be. By the same token, he/she does not yet know the driving strength of each transistor in the circuit. Normally, the circuit designer is more concerned with the logical operation of basic components, and how they interact to produce the overall circuit behaviour.

TLSIM achieves a significant performance improvement over switch-level simulators by neglecting node capacitance and signal strength information. This is done at no real cost in terms of simulation accuracy, since layout information is not normally available during logic design. By neglecting these details, it is possible to represent signals with a simple four-valued model, consisting of the values $\{0, 1, X, Z\}$.

In order to avoid errors introduced by transient signal conflicts, TLSIM extends the four-valued model with a *source count*: Whenever a device output changes from Z to one of the values $\{0, 1, X\}$, the source count at the associated node is incremented. Alternately, whenever a device output changes from one of $\{0, 1, X\}$ to $\{Z\}$, the source count at the associated node is decremented. Only when the source count becomes zero does a node's value become equal to Z.

Use of the source count guarantees correct simulation results, as long as no more than two signals are simultaneously applied to the same node. In particular, correct simulation results are guaranteed even if there are transient signal conflicts.



(a) A sample tristate node

(b) Using a source count to ensure correct behaviour

Figure 3.1: Correct simulation of transient signal conflicts

Figure 2.2 in Chapter 2 illustrated how indiscriminate use of the high-impedance state can cause simulation errors. The example in Figure 3.1 shows how TLSIM avoids errors caused transient signal conflicts by using the source count. In the figure, $S(y)$ is a source count, indicating how many device outputs are active at node y. Initially, only buffer a is on, and $S(y) = 1$. Due to the differing propagation delays, buffer b turns on before a can turn off, so for a short period $S(y) = 2$. However, when a does turn off, the transition sets $S(y) = 1$ and the transition to Z is ignored, so the final value of y is 0.

3.3 The UNIMOD1 Device Model

The device model used in TLSIM is key to its performance. *UNIfied MODel 1* (UNIMOD1) is capable of describing realistic circuit building blocks, in a way which helps accelerate simulation.

Most simulators offer one or more trade-offs between simulation performance, i.e., the time it takes to carry out the simulation, and accuracy. For instance, algorithmic models can be fast, since they describe the function of a large number of physical devices with simple high-level statements. Consider a multiplier; the circuit would occupy at least hundreds of gates, but only dozens of functional blocks, and a single high-level instruction. This makes the algorithmic model fastest to simulate. However, if the circuit is implemented using gates, then the gate-level model gives the designer an opportunity to test the circuit more accurately, and make predictions about timing characteristics of the circuit. This is clearly not possible with the algorithmic model.

TLSIM incorporates a *unified model* capable of modeling devices at multiple levels of abstraction. UNIMOD1 is a functional model with extensions for supporting some features of algorithmic models. It was designed with efficiency in mind, so it is as fast at evaluating gates as some dedicated gate-level simulators. The power of the UNIMOD1 model derives from the fact that it maps well to real circuit building blocks. For instance, an IC designer may use a device library that includes registers, adders, gates, etc. UNIMOD1 can model each of these blocks as single devices, whose behaviour is a relatively good approximation to the physical components. This means that a high degree of confidence may be reached that the final design implementation will conform to simulation results.

Since TLSIM uses a functional, *unidirectional* device model, it is incapable of modeling certain circuit structures. In particular, it cannot directly represent charge sharing or storage effects. These effects can only be simulated directy by switch-level or continuous simulators. However, since such effects are normally only used within macro-cells, which themselves *are* unidirectional, this restriction primarily affects the level to which TLSIM can break down a circuit, rather than limiting the number of circuits which it can simulate.

A block diagram of the UNIMOD1 device model is shown in Figure 3.2. The two truth table blocks (A and B) are able to evaluate complex Boolean functions, of the form $y_i = f(x_i, y_{i-1}, g(x_i, y_{i-1}))$, where i is the simulation time step. These blocks are used to calculate the state of a device from its previous state and data inputs. Memory elements (C) are used to model synchronous devices, such as flip-flops, latches and counters. They function by selecting when the Boolean function values ($o1$) are passed on to a device's output ($o2$). Multiplexors (D) model asynchronous

overrides, such as set and clear logic. They function by selecting either the enabled function values ($o2$) or some constants (0, 1 or X) to pass on to the device's output ($o3$). Finally, the delay elements model propagation delays, in the form of rise and fall times.

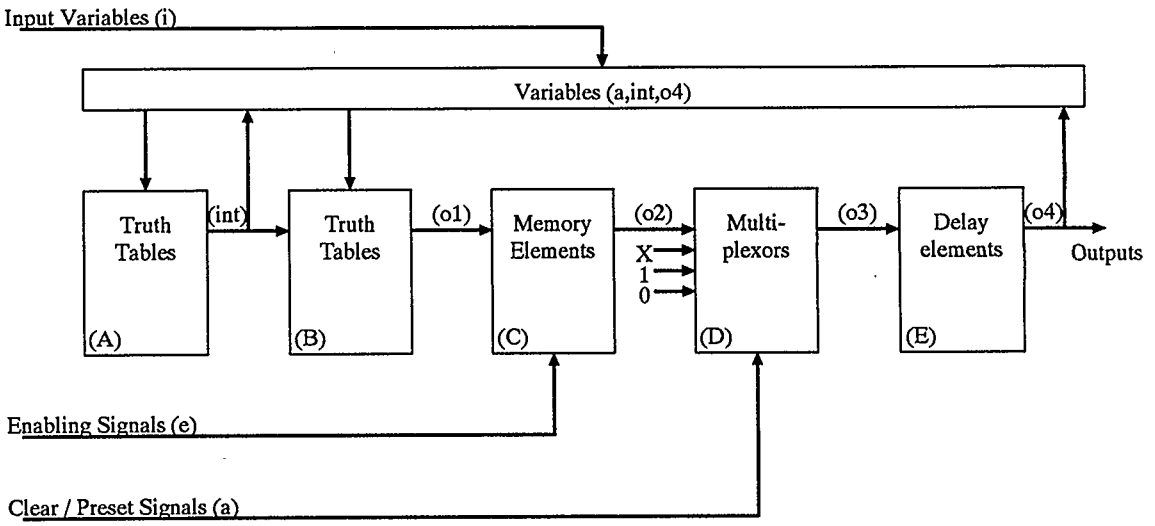


Figure 3.2: Complete UNIMOD1 device model

In the following several sections, starting with simple truth tables, and ending up with the complete model, we develop the UNIMOD1 device model in greater detail. The model is presented incrementally, in order to clarify the motivation behind every portion of the model.

3.3.1 Truth Tables

A simulator may use a simple truth-table model to represent combinational elements. A truth table may be evaluated using an efficient look-up procedure, as outlined in Figure 2.6, on page 22. For a device with m outputs, m arrays are used to represent device function (although the arrays need not be unique).

A device model consisting of only truth tables is illustrated in Figure 3.3.

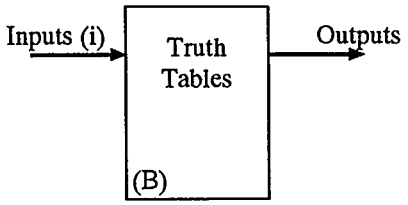


Figure 3.3: Simple Boolean device model

This model may easily be extended for use with four-valued signals. For four-valued signals, the array length becomes 2^{2n}, since two bits are needed to represent each input.[1] The truth table functions are evaluated using the same procedure, but the offset number is now formed using the equation:

$$\text{offset} = (b_1)|(b_2 << 2)|\cdots|(b_n << 2(n-1)) \tag{3.1}$$

In this equation, function arguments are represented by b_i, where i is the argument number. The arguments b_i take on the binary values $\{00, 01, 10\}$, which represent the signal values $\{0, 1, X\}$. Note that Z arguments are mapped to one of the other three values prior to function evaluation, so are omitted here.

This representation of function arguments as bit fields in an integer resembles the representation used by Rok Sosic et al. in their parallel Boolean function evaluation algorithm, Unison [35].

[1] For four-valued signals the *construction* of the array is more complicated. Chapter 4 describes this issue in detail.

3.3.2 Device Memory

The simple truth table model suffices for gates, but it has no provision for devices with memory. It cannot directly represent devices which have state, such as flip-flops, latches or counters. Memory elements are incorporated into UNIMOD1 by adding a control element after the functional block.

While in physical circuits node voltages decay over time, this is not *inherently* the case in a simulated node. Since the simulator must store the state of each node in a variable, simulated nodes have built-in memory. The simple device model presented in the previous section does not take advantage of this property, however, since every change in a device's function values is immediately applied to the device's output nodes.

UNIMOD1 extends the simple truth-table model to include memory by controlling *when* a device's output values are copied to its output nodes.

The modified device model is shown in Figure 3.4. UNIMOD1 allows for any signal value or transition (i.e., $0, 1, 1 \rightarrow 0$, or $0 \rightarrow 1$) to be cause a device's output values to be copied to their nodes. Making the enabling signals sensitive to signal transitions ($1 \rightarrow 0$ or $0 \rightarrow 1$) makes the device edge-sensitive, while using signal values (0 or 1) makes the device level-sensitive.

The function of the enabling block may be stated formally as follows:

$$o_2^i = \begin{cases} o_1^i, & \text{if } e \text{ is active} \\ o_2^{i-1}, & \text{if } e \text{ is inactive} \end{cases} \tag{3.2}$$

where superscripts indicate a timestep, and variable names are as shown in Figure 3.2.

This model of storage elements has the benefit of improving simulator performance. Consider a device input changing value while some or all of the device's

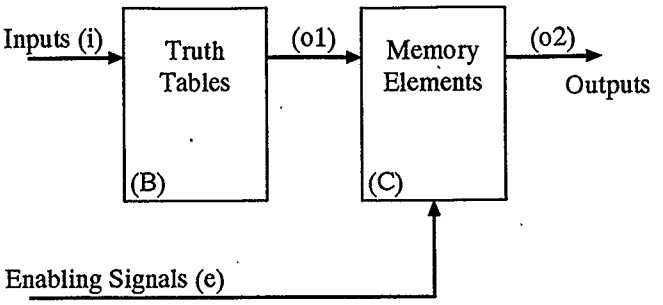


Figure 3.4: Device model with memory added

outputs are disabled. With this model, the simulator does not have to recalculate the disabled device outputs. Since several such input transitions may occur while outputs are disabled, the simulator may save a substantial amount of computation. In general, if n input transitions occur while a device output is disabled, $n-1$ recalculations are skipped. The single recalculation is performed when the output is re-enabled.

3.3.3 Asynchronous Inputs

The device model presented in the last section is essentially a model for synchronous devices. They have data and clocking inputs, and an arbitrary number of outputs. In practice, we may also want to be able to set and reset outputs *asynchronously.* This ability may be added to the model by adding a layer of multiplexing elements between the memory blocks and output nodes. When the device is operating normally, the memory blocks' outputs are transferred directly to the output nodes. When an output is set, its multiplexor passes a 1 signal to its output nodes. When an output is reset, a 0 signal is passed to its output node. The ability to pass an X signal is added for completeness, although it probably has no practical application.

The addition of asynchronous overrides to the device model is illustrated in Figure 3.5. In this figure, for each output, a multiplexor in block D selects either the enabled function value from block C, or a Boolean constant to pass on to the output node.

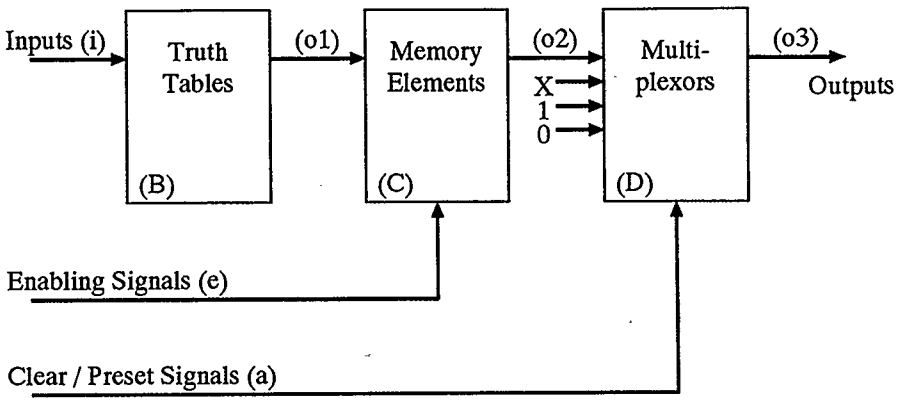


Figure 3.5: Device model with asynchronous overrides added

As with the clocking inputs, TLSIM can make asynchronous overrides active on any signal state or transition. The ability to set and clear device outputs on an override *transition* is included for completeness, rather than with any particular application in mind.

The function of the asynchronous override (Multiplexor) block may be stated formally as follows:

$$o_3 = \begin{cases} K, & \text{if } a \text{ is active} \\ o_2, & \text{otherwise} \end{cases} \tag{3.3}$$

where K may be one of the constants $\{0, 1, X\}$, and any level or transition can make a active.

The asynchronous override blocks (D) are similar to the memory element blocks (C) in that they can also reduce the computation required for simulation. As long

as an output is set or reset, there is no need to recalculate the associated function, even though inputs to the device may change. As before, if n transitions occur at a device's inputs while an output is forced to some state, only $n - 1$ function recalculations are required.

3.3.4 Internal Nodes and Feedback

While circuits built using the model in Figure 3.5 can represent most physical devices, the model cannot represent state machines *internally.* With this model, state information can only be fed back to a device's function block (B) through the surrounding circuit, outside of the device. To get feedback, a device must include state variable outputs, as well as state variable inputs. These must then be connected to one another. A useful modification to the device model, then, is to add *an internal feedback path.*

A second feature which is sometimes useful is to factor Boolean functions into subexpressions, so that several outputs in a device may reuse intermediate results. With this modification, we have three sets of variables for use in the Boolean functions: Input pins $(p_1, \cdots, p_P)$, internal nodes $(i_1, \cdots, i_I)$ and output nodes $(o_1, \cdots, o_O)$, where a device has P inputs, I internal nodes, and O outputs. Relating this nomenclature to Figure 3.2, input pins enter the device from the left, internal nodes are the outputs of block A, and output nodes are the outputs of block B. Using this terminology, the outputs may be calculated using the following equations:

$$
\begin{aligned}
i_1 &= f_1^i(p_1, \cdots, p_P, i_1, \cdots, i_I, o_1, \cdots, o_O) \\
&\vdots \\
i_I &= f_I^i(p_1, \cdots, p_P, i_1, \cdots, i_I, o_1, \cdots, o_O)
\end{aligned} \tag{3.4}
$$

$$
\begin{aligned}
o_1 &= f_1^o(p_1, \cdots, p_P, i_1, \cdots, i_I, o_1, \cdots, o_O) \\
&\vdots \\
o_O &= f_I^o(p_1, \cdots, p_P, i_1, \cdots, i_I, o_1, \cdots, o_O)
\end{aligned} \tag{3.5}
$$

In the above equations, f_j^i is the Boolean function for the jth internal node, and f_k^o is the Boolean function for the kth output node. To simplify these equations, we may define a vector notation:

$$
\begin{aligned}
\mathbf{P} &\equiv [p_1, \cdots p_P]^T \\
\mathbf{I} &\equiv [i_1, \cdots i_I]^T \\
\mathbf{O} &\equiv [o_1, \cdots o_O]^T \\
\mathbf{f}^i &\equiv [f_1^i, \cdots, f_I^i]^T \\
\mathbf{f}^o &\equiv [f_1^o, \cdots, f_O^o]^T
\end{aligned} \tag{3.6}
$$

Equations (3.4) and (3.5) may then be rewritten as:

$$
\mathbf{I} = \mathbf{f}^i(\mathbf{P}, \mathbf{I}, \mathbf{O}) \tag{3.7}
$$

$$
\mathbf{O} = \mathbf{f}^o(\mathbf{P}, \mathbf{I}, \mathbf{O}) \tag{3.8}
$$

Equations (3.7) and (3.8) may be thought of as the state equations of a device. They determine a device's outputs and next state from its inputs and present state.

By adding compound functions, as shown in Equations 3.7 and 3.8, and internal feedback to the model in Figure 3.5, we finally get the complete UNIMOD1 device model, illustrated in Figure 3.2.

3.3.5 Examples

In order to further clarify how UNIMOD1 works, some sample devices are modeled in this section. The simplest examples are gates, while the most complex illustrate how UNIMOD1 can represent complete arithmetic-logic units.

Gates

Gates are trivially defined in TLSIM by omitting the internal, output enabling and asynchronous override blocks in the device model (blocks A, C and D). Due to its use of truth tables, TLSIM requires that a separate truth table be provided for gates with different numbers of inputs. For instance, a 2-input NAND gate would be defined with one truth table, and a 3-input NAND gate would use another.

Flip Flops

A D flip-flop may be readily implemented in TLSIM using the Boolean identity function, enabled by a falling (or rising) edge on a clock signal. This is illustrated in Figure 3.6.

A JK flip-flop may be implemented using a special truth table, as shown in Table 3.1.

The inverted output $\overline{Q}$ can be formed either by an inverter outside the flip-flop, an inverting function within the flip-flop, or a second truth table, with outputs negated to the ones in Table 3.1, also within the device. The choice of which, if any, of these

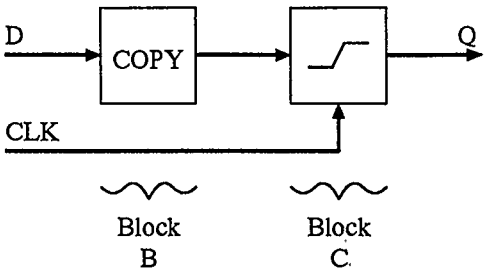


Figure 3.6: A DFF implementation using UNIMOD1

Table 3.1: Truth table for a JKFF implementation

J	K	q	Q
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

to use in a device library should be made based on the characteristics of the physical devices being modeled.

An implementation of the JKFF using the Boolean function in Table 3.1 is shown in Figure 3.7. Note that this particular representation does not generate an inverted output $\overline{Q}$.

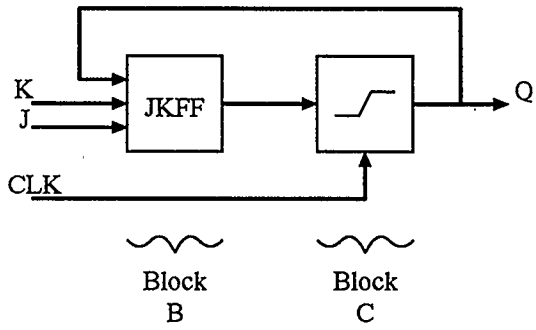


Figure 3.7: A JKFF implementation using UNIMOD1

Counters

A unidirectional counter, which follows the sequence $\{0, 1, \cdots, 2^n, 0, 1, \cdots\}$, may be modeled as a single device in TLSIM. Consider that in the state transition of a counting process, a bit will change from 1 to 0 or 0 to 1 if and only if all preceding bits are equal to 1 in the previous state. Accordingly, internal nodes may be used to calculate how many contiguous bits, starting with the least-significant bit, are equal to 1. Given the results of these internal calculations, the truth table in Table 3.2 can then be used to calculate each counter output. This arrangement is shown in Figure 3.8. Note that the truth table turns out to be simply the XOR function.

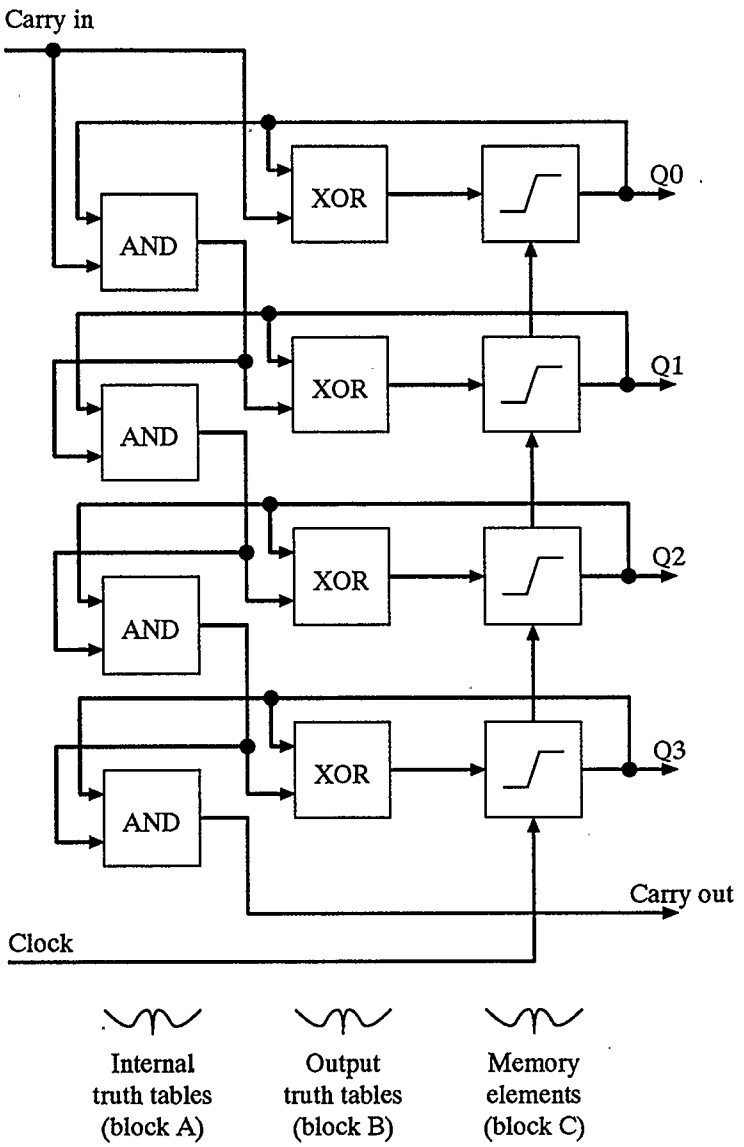


Figure 3.8: A counter implementation using UNIMOD1

Table 3.2: Truth table for a counter implementation

all less significant bits equal to 1?	q	Q
0 (no)	0	0
0 (no)	1	1
1 (yes)	0	1
1 (yes)	1	0

Adders

Binary adders may be modeled as either single monolithic devices or arrays of smaller adders. For instance, a 16-bit adder (with two sets of 16 data inputs, a carry input, 16 sum outputs and a carry output) could be implemented as a monolithic device, a pair of eight-bit devices, four four-bit devices, etc. Selecting a level of abstraction depends on two variables – accuracy, i.e., the model should reflect the actual implementation, and performance.

As shown in Figure 3.9, as the complexity of a UNIMOD1 device grows, it absorbs a growing amount of circuit function, so less event scheduling is required to account for inter-device communication. At the same time, the computational effort required to evaluate each device increases. Clearly, increasing device complexity reduces the number of required device evaluations, but it also lengthens the time required to perform *each* evaluation. The net result is that while the event scheduling effort shrinks, the device evaluation effort increases. The best simulation performance is achieved by selecting a device complexity which roughly balances device evaluation against event scheduling.

Returning to the 16 bit adder mentioned above, assume that the technology in which the circuit is to be implemented provides 4-bit adder macro-cells. In this

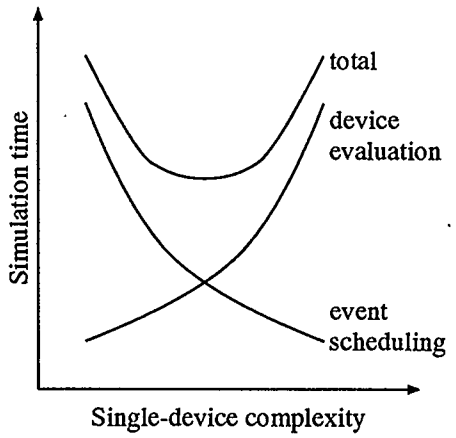


Figure 3.9: Total computational effort as a function of device complexity

environment, TLSIM would model the 16-bit adder as a cascade of 4-bit adders, as shown in Figure 3.11. Using a pair of truth tables for carry and sum functions, a four bit adder could be modeled as a single device as shown in Figure 3.10.

3.4 Four-Valued Truth Tables

The use of four-valued truth tables complicates the function evaluation procedure in Figure 2.6. In order to represent four values, we must use two bits per argument. Furthermore, the simulator must evaluate functions whose arguments are set to X or Z in a reasonable manner.

Since in general it is not known how a Z input will be interpreted by a physical circuit, Z inputs are replaced by X during function evaluation. However, in some technologies Z *does* behave like 0 or 1 when it is applied to a device input, so TLSIM allows the user to override the default mapping of $Z \rightarrow X$ with $Z \rightarrow 0$ or $Z \rightarrow 1$.

When a function with one or more X arguments is evaluated, its value may or may not equal X. Consider two functions – Boolean AND and OR, with various

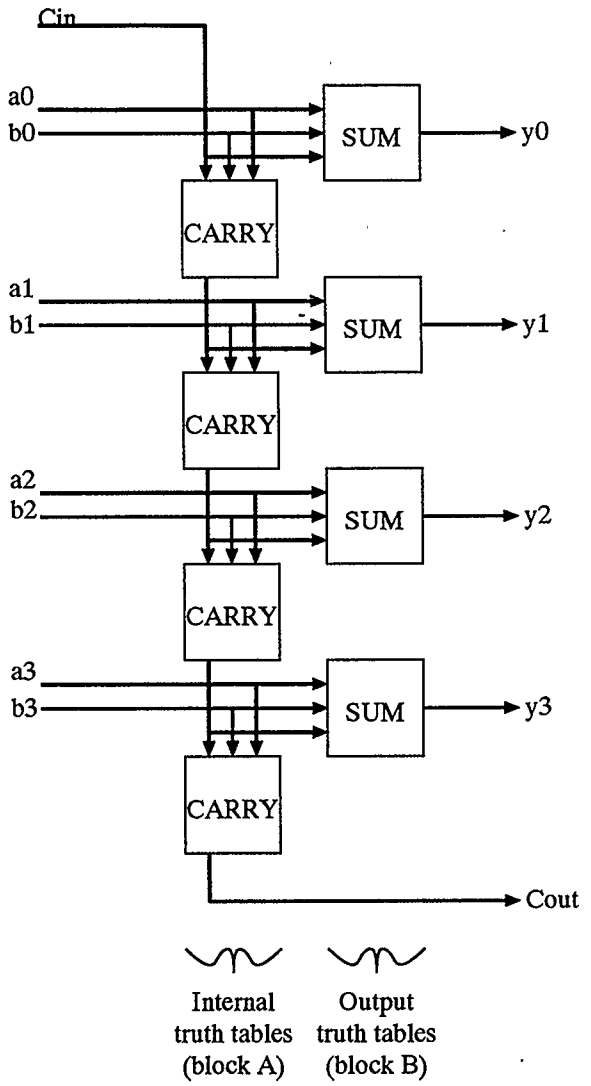


Figure 3.10: A 4-bit adder model

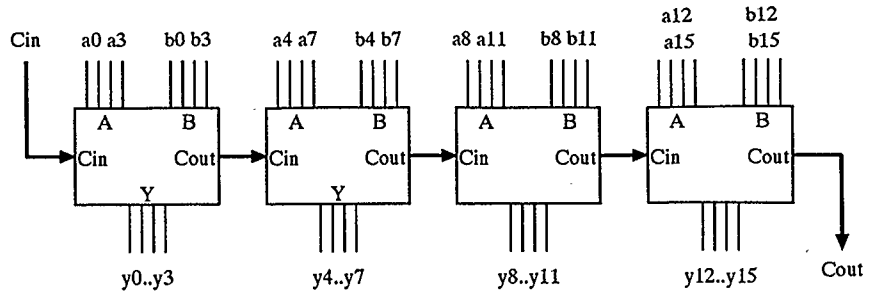


Figure 3.11: Cascading 4-bit adders to make a 16-bit adder

input combinations, as shown in Figure 3.12, on page 54. Clearly there are cases where at least one argument is unknown (X), and the function value is not X (i.e., it is 0 or 1).

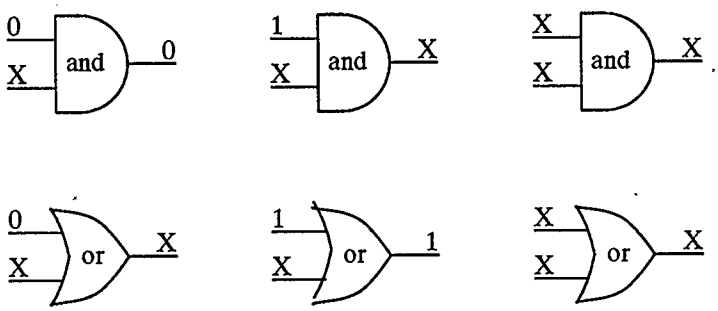


Figure 3.12: AND and OR gates with X inputs

To address the problem of correctly evaluating Boolean functions with X arguments, we begin by reformulating a function's arguments, or *input vector* as an equivalent set. If the vector contains no X entries, then its equivalent set has a single member – the original vector. However, if the vector contains X entries, the set is larger. In general, an equivalent set representing an original input vector will have 2^n elements, where the input vector contains n X entries. Some examples of the relationship between input vectors and their equivalent sets are given in Table 3.3. This equivalence relation is defined rigorously in Chapter 4.

Table 3.3: Sample input vectors and equivalent sets

Input vector	Equivalent set of explicit vectors
1111	{1111}
11X1	{1111, 1101}
X1X1	{1111, 1101, 0111, 0101}

If a given Boolean function has the same non-X value for *each member of the equivalent set of an input vector*, then the function value for that input vector is not X.

In order to avoid the need to enumerate X inputs and calculate equivalent sets, whenever a function is evaluated, TLSIM calculates the value of each truth table for *every possible input vector* when its loads the truth table library, before simulation begins. For instance, a two-input Boolean OR function might be specified by a truth table such as the one in Table 3.4. Assuming that Z inputs are mapped to one of $\{0, 1, X\}$, there are five input vectors which the original table does not directly specify. These are: (X, X), $(0, X)$, $(1, X)$, $(X, 1)$ and $(X, 0)$. During initialization, TLSIM calculates the equivalent sets for each of these input vectors, and updates the truth table appropriately. During simulation, any of these inputs can be applied to the truth table directly, and a look-up procedure similar to the one in Figure 2.6 is used to evaluate the function rapidly.

3.5 Event Scheduling

TLSIM uses a time wheel to manage the event queue. This model was chosen after both the time wheel and heap scheduling algorithms were compared using a number

Table 3.4: Truth table for a Boolean OR function

Input a	Input b	a OR b
0	0	0
0	1	1
1	0	1
1	1	1

of practical circuits. In all cases, the time wheel algorithm performed better, but only marginally.

In a time wheel, delays are necessarily integer. Accordingly, a device library is specified using some fundamental unit of time; typically nanoseconds or tenths of nanoseconds. Device outputs are assigned rise and fall delays, specified as integer multiples of this fundamental unit. The longest delay in the device library is then used to set the length of the time wheel.

Event scheduling in TLSIM is somewhat more complex than the classical time wheel model presented in the background chapter. This is so for two reasons: to ensure correct simulator operation, and to improve performance. In order to ensure correctness, TLSIM employs an *event cancellation* mechanism. In order to improve performance, TLSIM uses a fixed number of *event slots*, or allocated event data structures. Event allocation is then performed using a *stack of pointers* to these event slots. The net result is that both event allocation and deallocation are $O(1)$ operations.

3.5.1 Event Allocation and Deallocation

In practice, a simulator spends much of its time allocating and deallocating space for events. In fact, due to operating system overhead, as much as 50% of the total simulation time may be taken up by memory allocation routines. Although a large number of events are allocated during the course of a typical simulation run, only a few event data structures are in use at any one time.

TLSIM takes advantage of the fact that only a relatively small number of events are needed simultaneously by implementing a two-tier memory allocation scheme for events. This scheme is illustrated in Figure 3.13. TLSIM first allocates a reasonably large number of event data structures. These structures are stored in a static array, whose size may be increased as necessary during the course of the simulation. Next, TLSIM allocates a stack of pointers to these data structures. Initially, there is a one-to-one relationship between the *pointer stack* and the *event array.* Whenever TLSIM requires an event data structure, it pulls a pointer from the stack. After an event has been processed, its address is pushed back onto the pointer stack. Since both *pull* and *push* stack operations require an $O(1)$ effort, event allocation and deallocation are both $O(1)$.

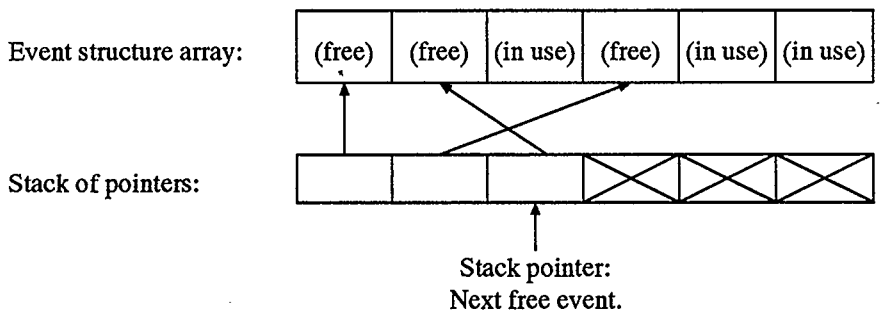
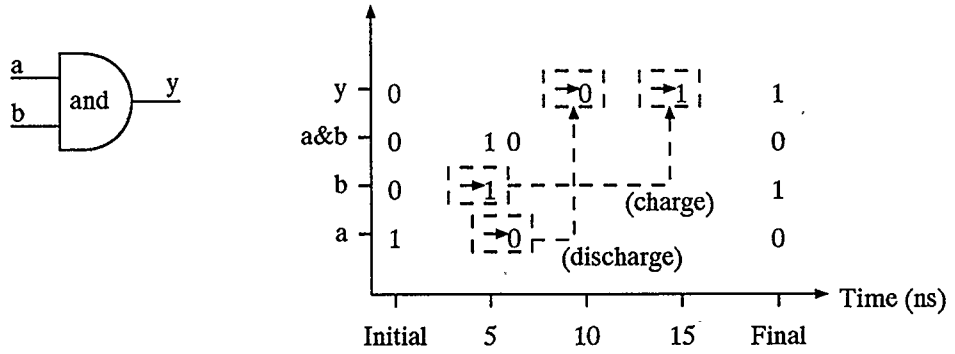


Figure 3.13: Event memory allocation scheme

3.5.2 Event Cancellation

Consider the circuit of Fig. 3.14(a), and the sequence of events at its inputs, as shown in Fig. 3.14(b). Assume that the AND gate has a rise propagation delay of 10ns, and a fall propagation delay of 4ns. Let the initial inputs be $a = 1$ and $b = 0$. At t=5ns, a transition occurs, setting $b \rightarrow 1$. Since both inputs of the AND gate are now equal to 1, an event is scheduled for t=15ns to set $y \rightarrow 1$. Next, at t=6ns, a second transition occurs, setting $a \rightarrow 0$. Since the current function value is 1, a second event is scheduled, to set $y \rightarrow 0$ at t=10ns.



(a) Labelled AND gate (b) Event sequence with an error in the final state

Figure 3.14: The need for event cancellation

If the two events at node y are processed in sequence, then the final state of the gate will be: $a = 0$, $b = 1$ and $y = 1$. Clearly, this is wrong.

In order to alleviate this problem, and others like it, it is necessary to differentiate between the sequence in which events are *scheduled*, and the sequence in which they are *retrieved*. TLSIM guarantees correct steady-state response by ensuring that, for every node, only the last *scheduled* event is applied to the node. For instance, a number of inputs at a gate might change simultaneously, causing a sequence of events

to be scheduled at the gate's output node. TLSIM tags each event with a unique ID number, reflecting the order in which it was *scheduled* and stores the event ID at the node referred to in the event. The node then "knows" which event to "expect." When a spurious event is retrieved from the queue, its ID does not match the event ID at the target node, and it is discarded.

Referring once again to Figure 3.14, the event setting $y \rightarrow 1$ would be given an ID number, say '1.' At the same time, the event ID '1' would be stored at node y. Since it is *scheduled* later, the event to set $y \rightarrow 0$ would get a later event ID, say '2,' and the event ID at node y would also be set to '2.' When the $y \rightarrow 0$ event is retrieved from the queue, its ID will match the one at node y, and it will be processed. However, when the $y \rightarrow 1$ event is retrieved from the queue, its ID will *not match* the ID at node y, so it will be cancelled.

3.6 Circuit Initialization

TLSIM assigns an initial value of X to every node in a circuit. When simulation begins, TLSIM schedules events to set the value of the ground node to 0 and of the power node to 1. This is done in order to propagate the power nodes' constant values to device inputs before any other signals are applied. Furthermore, it ensures that devices whose *only* inputs are power nodes are evaluated at least once, even though good designs should have no such devices. This must be done because the simulator can make no assumptions about reasonable circuit design – its function is to identify *poor* designs.

3.7 Summary

Overall, TLSIM is designed to give good simulation performance and accuracy by using a device model which simultaneously is efficient to evaluate and accurately represents device blocks used in real circuits.

The UNIMOD1 device model is fairly expressive: it can easily describe the major modes of device operation. These are feed-forward, i.e., simple Boolean functions, clocking, to control when outputs are asserted on their nodes, and asynchronous. Using this scheme, TLSIM can represent the operation of devices ranging from gates to small arithmetic-logic units (ALUs) using a single, unified model, without the performance degradation associated with modeling low-level components in an algorithmic language.

In addition to its powerful device model, TLSIM provides accurate simulation using its four-state signal representation, and a fast event cancellation mechanism, which ensures correctness in complex scenarios involving rapid sequences of input transitions. TLSIM does not suffer a performance penalty for the four-state signal model, as it uses a novel variation on the classical truth table look-up procedure to rapidly evaluate Boolean functions even when some of their arguments are equal to X.

Finally, TLSIM achieves good execution performance. This is done in several ways. Using UNIMOD1, TLSIM can discard many events prior to device evaluation. Using efficient algorithms for memory allocation, function evaluation, event cancellation and treatment of the high-impedance state, TLSIM can efficiently process the remaining events. For instance, a major time savings is achieved by reusing memory

allocated for event data structures. Similar measures are taken to rapidly allocate memory for devices and nodes during circuit initialization.

Chapter 4

Basic Algorithms in TLSIM

This chapter describes the algorithms used to implement the various mechanisms in the TLSIM simulator. It describes their function, and how they address issues in simulator design. The main focus in each section is on implementing various components of a simulator *efficiently*, so as to reduce execution time. This is a feature of primary importance in the TLSIM simulator.

Section 4.1 discusses how truth tables are constructed, and how they are used to evaluate Boolean functions. Together, the algorithms from these two sections make it possible to use the four-valued signal model without any significant performance degradation as compared to the binary-valued model.

Section 4.2 describes how devices are evaluated when the state of one or more of their inputs changes. This is significant, because TLSIM spends most of its time evaluating devices. Every small improvement in performance here is multiplied out by the number of device evaluations carried out during execution, so can yield a large savings in overall simulator run time.

Section 4.3 gives a detailed description of the scheduling mechanisms in TLSIM. There are two components to TLSIM's scheduling mechanism – one that loads transitions from an input file, and another that schedules internally-generated events. This section illustrates how these components are integrated.

Section 4.4 describes the event-cancellation mechanism in TLSIM. It emphasizes how a potentially difficult problem is averted with no computational cost. In fact,

event cancellation in TLSIM can yield a performance *improvement.*

Section 4.5 illustrates how the problems which arise from the use of a high impedance (Z) state are addressed in TLSIM. The source count mechanism in TLSIM ensures correctness while incurring only minor overhead.

Section 4.6 extends the ideas presented in Section 4.2 to illustrate how memory elements are evaluated. There are some important points to consider here, in order to avoid feedback on data lines in RAM chips, and to maintain a correct count of the number of signal sources active at every node.

Section 4.7 describes the memory management algorithms used in the simulator. Since the simulator must allocate a large number of small blocks of memory to represent devices and nodes in the circuit, and later must perform a large number of memory allocation and deallocation operations, efficient memory management makes a large contribution to simulator performance.

Figure 4.1 shows the relationship between procedures in TLSIM. It should serve as a useful reference while reading this chapter and Chapter 5.

4.1 Four-Valued Boolean Functions

TLSIM uses four-valued logic to represent device functionality. This is an extension to classical Boolean algebra, which adds the values $\{X, Z\}$ to the classical $\{0, 1\}$.

TLSIM uses truth tables to evaluate Boolean functions rapidly. In order to yield reasonable results for function evaluations where the function arguments range over $\{0, 1, X, Z\}^n$, rather than just $\{0, 1\}^n$, some care must be taken both in truth table construction and evaluation. Sections 4.1.1 and 4.1.2 show how TLSIM constructs

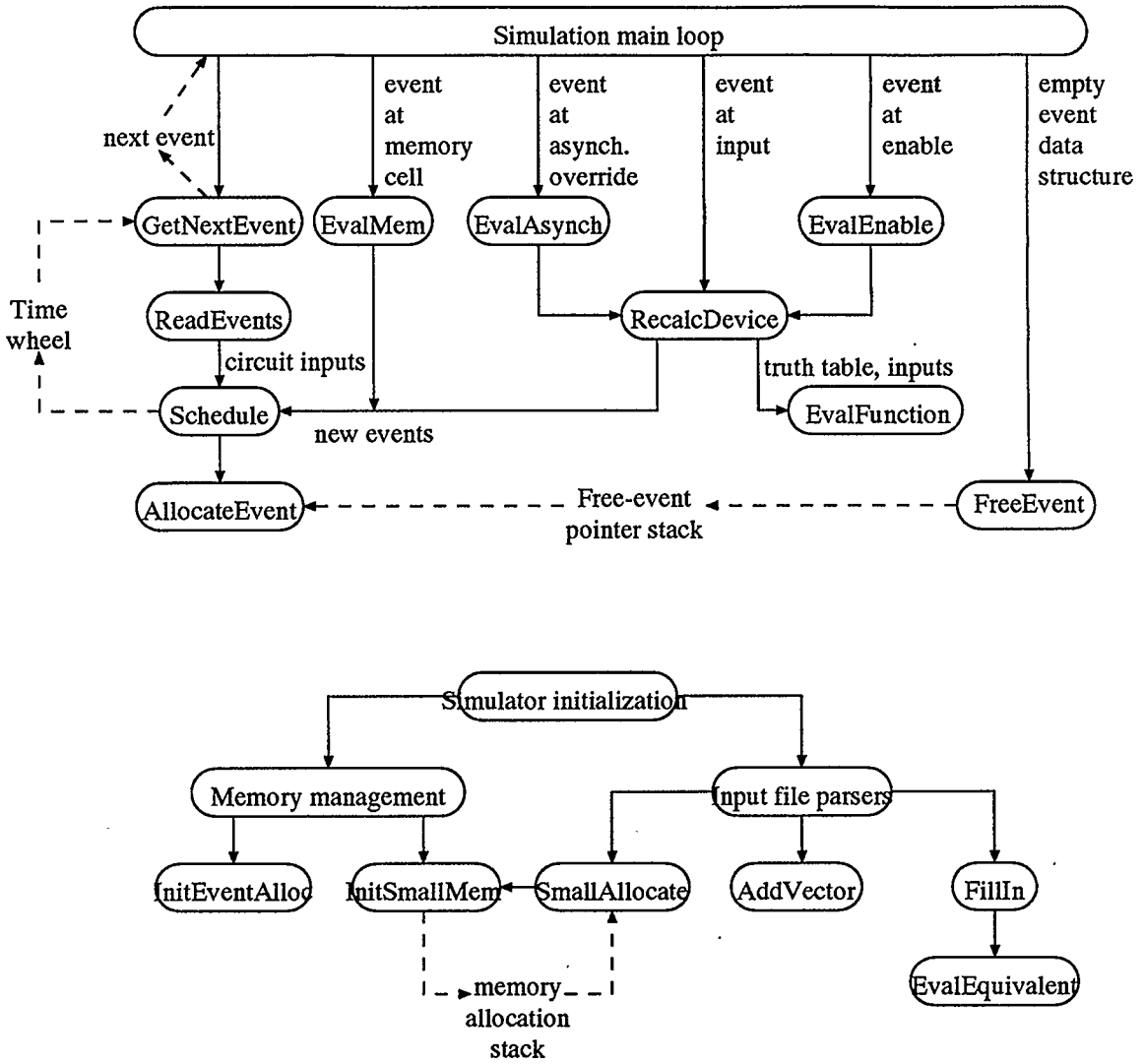


Figure 4.1: TLSIM function and procedure hierarchy

truth tables and it uses them to evaluate four-valued Boolean functions.

4.1.1 Building Four-Valued Truth Tables

TLSIM uses four-valued truth tables to represent Boolean functions. These tables are stored in memory as arrays of integers, where each integer represents the function value for a different combination of inputs. A truth table is evaluated by forming an offset into the table, where each two bits in a binary representation of the offset number represent an input variable. The function value for that combination of inputs is then given by the integer at the offset position in the table.

Building four-valued truth tables is a non-trivial task for two reasons. First, the table may be specified in terms of input vectors containing X elements, as illustrated in Figure 4.2. Second, the table may have to be evaluated for cases when one or more of the inputs are unknown (X). This separate problem is illustrated in Figure 3.12, on page 54.

Boolean OR

a	b	y
X	1	1
1	X	1

2-Input MUX

s	d_0	d_1	y
0	0	X	0
0	1	X	1
1	X	0	0
1	X	1	1

Figure 4.2: Truth tables specified using X entries

In order to more clearly illustrate how TLSIM solves these two problems, we must

first consider some definitions. Arguments to a Boolean function may be taken to be fixed-length vectors, where each element of the vector takes on one of the values $\{0, 1, X\}$. Using this notation, four-valued Boolean functions may be represented as $f(v) \in \{0, 1, X, Z\}$, where $v \in \{0, 1, X\}^n$, for some n.[1] A vector v is classified as an *explicit* vector if and only if $v \in \{0, 1\}^n$. Otherwise, a vector is classified as an *implicit* vector – i.e., it contains at least one element which is equal to X.

Next, we must consider the concept of equivalent sets of vectors. An explicit vector v_e is equivalent to the set $\{v_e\}$. An implicit vector v_i is equivalent to the set $\{v_{i1}, v_{i0}\}$, where v_{i1} and v_{i0} are formed by replacing an element in v_i which was equal to X with 1 and 0, to give two new vectors v_{i1} and v_{i0}. This equivalence relation may be applied repeatedly to find larger equivalent sets to v_i.

In order to evaluate a Boolean function for an *explicit* argument, we simply look up the value of the truth table at the offset derived from its vector. To evaluate a Boolean function for an *implicit* argument, we first find its largest equivalent set all of whose elements are explicit vectors, and then evaluate the function for each member of that set. If each of the vectors in this set yields the same function value, then that is the function value for the original implicit argument. Otherwise, the function evaluates to X.

Since TLSIM allows truth tables to be specified in terms of implicit vectors, the following steps must be taken to initialize each truth table:

1. Allocate a table, and set all elements to 0.

2. For every implicit vector in the function definition, generate two entries by

[1] Elements of a vector which are equal to Z are mapped to one of $\{0, 1, X\}$ before function evaluation.

selecting an element in the vector whose value is X, and replacing it with both 0 and 1. Repeat step 2 for each new vector.

3. Set the table at the offset derived from each resulting explicit vector.

This procedure is illustrated in Figure 4.3.

```
Inputs: The truth table (table), array of arguments (arguments), and the function value (value)
Outputs: None

Procedure AddVector {
    /* expand out implicit vectors: */
    for( each argument, i )
        /* if implicit, substitute 1 and 0 and repeat: */
        if( argument_i = X )
            let argument_i = 0
            recurse: AddVector(table,arguments,value)
            let argument_i = 1
            recurse: AddVector(table,arguments,value)
            let argument_i = X
            return
    /* assign function values to explicit vectors: */
    offset=0
    for( each argument, i )
        shift offset left 2 bits
        offset = offset or argument_i
    let table[offset] = value
    }
```

Figure 4.3: Algorithm to expand implicit vectors to their equivalent sets

The algorithm in Figure 4.3 only initializes positions in the table which correspond to *explicit* argument vectors. TLSIM must then *fill in* the table in order to

assign correct function values to all implicit vectors. To do this, TLSIM checks every possible input vector and, if it is implicit, calculates the function value for that vector by checking its equivalent set. If the function yields the same value for each member of the equivalent set, then that is the implicit vector's function value. Otherwise, the function value for the implicit vector is set to X. This procedure is illustrated in Figure 4.4.

The effect of these two algorithms on a sample truth table is illustrated in Figure 4.5, for a two-input multiplexor function. The *AddVector* algorithm removes any information from table entries representing implicit vectors, while the *FillIn* algorithm restores these, as well as any other implicit vectors.

4.1.2 Evaluating Four-Valued Truth Tables

Given that a truth table has been filled in and expanded as shown in Figure 4.5, TLSIM may use a trivial truth table evaluation algorithm. This algorithm simply forms an offset into the table, and returns the table's contents at that offset. This procedure, which is oblivious to implicit and explicit vectors and therefore quite efficient, is illustrated in Figure 4.6.

4.2 The Device Evaluation Algorithm

Whenever inputs to a device change, TLSIM must reevaluate the state of the device and, if outputs change, schedule transitions at the nodes connected to the device's outputs. In order to reduce the time required to recalculate the state of devices, TLSIM attempts to identify device states where changing some of the inputs will

Inputs: The truth table (table), and the number of arguments (Narguments)
Outputs: None

Procedure FillIn {
    for($v = 0$ *to* $2^{2Narguments}$)
        /* v represents a function argument vector */
        if(v *contains no* Z *elements* **and** v *is implicit*)
            table[v] = *EvalEquivalent*(table,Narguments,v)
}

Inputs: The truth table (table), number of arguments (Narguments), and argument vector (v)
Outputs: The function value

Function EvalEquivalent {
    /* recursively handle implicit vectors: */
    for(*every element* k *in the vector* v)
        if(*element* $k = X$)
            let i = *EvalEquivalent() with the same arguments, except that element* k *is set to* 0.
            let j = *EvalEquivalent() with the same arguments, except that element* k *is set to* 1.
            if($i = j$)
                return i
            else
                return X
    /* explicit vectors need no special treatment */
    return table[v]
    }

Figure 4.4: Algorithm to assign values to implicit vectors in a truth table

Inputs			User	After	After
s	d_0	d_1	Specification	*AddVector*	*Fillin*
0	0	0	-	0	0
0	0	1	-	0	0
0	0	X	0	-	0
0	1	0	-	1	1
0	1	1	-	1	1
0	1	X	1	-	1
0	X	0	-	-	X
0	X	1	-	-	X
0	X	X	-	-	X
1	0	0	-	0	0
1	0	1	-	1	1
1	0	X	-	-	X
1	1	0	-	0	0
1	1	1	-	1	1
1	1	X	-	-	X
1	X	0	0	-	0
1	X	1	1	-	1
1	X	X	-	-	X
X	0	0	-	-	0
X	0	1	-	-	X
X	0	X	-	-	X
X	1	0	-	-	X
X	1	1	-	-	1
X	1	X	-	-	X
X	X	0	-	-	X
X	X	1	-	-	X
X	X	X	-	-	X

Figure 4.5: The effect of the AddVector and FillIn algorithms

Inputs: The truth table (table), an array of arguments (arguments), and the number of arguments (Narguments)
Outputs: The function value

```
Function EvalFunction {
    let offset = 0
    for( i = 0 to Narguments - 1 )
        shift offset left 2 bits
        offset = offset or argument_i
    return table[offset]
    }
```

Figure 4.6: Algorithm for evaluating four-valued truth table functions

not affect the device outputs. When inputs change while a device is in such a state, TLSIM postpones device evaluation, thus reducing execution time.

There are two cases where TLSIM does not reevaluate the state of device outputs. The first is when outputs are disabled, and the second is when outputs are under the control of an asynchronous override. Consider the example of a D-type flip flop. The DFF has a single data input (D), a clock input (CLK) active on its falling edge, an asynchronous clear input (CLR), which is active low and an output (Q). A transition on the D input will never cause an output recalculation. Furthermore, Q need not be recalculated as long as CLR is set to 0. The only transition that may cause Q to be recomputed is a $1 \rightarrow 0$ transition on the CLK input *while CLR is set to 1* (inactive).

In general, devices have more complicated output functions than the simple $Q = D$ operation of a DFF. By avoiding the recalculation of many output nodes (block

B of Figure 3.2, on page 40), TLSIM gains a measure of efficiency.

The following sections describe the operating principles in TLSIM's evaluation-cancellation mechanism. The final section puts these ideas together to show the device evaluation procedure.

4.2.1 Output Enables

TLSIM's output enables are analogous to clocking pins on a device. They are used to model both level- and edge-sensitive inputs, which control when the value of a device output is updated. TLSIM allows for at most one enable per output.

Associated with each output in a device is a flag indicating whether it is currently enabled. When the output function is considered for reevaluation (i.e., when some input to the device has changed), TLSIM checks this flag to see if the output is active. If it is not, the evaluation is skipped. Whenever an enable input to a device changes, TLSIM updates all the appropriate flags, and if the state of an enable input changed from inactive to active, each affected output is re-evaluated.

The procedure for deciding whether or not to recalculate any subset of a device's outputs when an enable input changes is shown in Figure 4.7. Given a device, the number of the enable that changed, and the new enable value, this procedure updates flags inside the device, and if necessary calls the device evaluation procedure.

4.2.2 Asynchronous Overrides

TLSIM allows for a number of asynchronous override inputs to be used in a device. These inputs model asynchronous clear and preset logic in otherwise synchronous devices. They are also useful in gating circuitry – for instance, a row of AND gates,

Inputs: The device (device), the input that changed (pin) and new value (l)
Outputs: None

```
Procedure EvalEnable {
    switch( active transition type for this enable pin )
        /* active-high */
        case HI:    if( l = 1 )
                        set enable flag to True
                        recalculate affected output functions in the device
                    else
                        set enable flag to False

        /* active-low */
        case LO:    if( l = 0 )
                        set enable flag to True
                        recalculate affected output functions in the device
                    else
                        set enable flag to False

        /* rising-edge triggered */
        case LOHI: if( l = 1 )
                        set enable flag to True
                        recalculate affected output functions in the device
                        set enable flag to False

        /* falling-edge triggered */
        case HILO: if( l = 0 )
                        set enable flag to True
                        recalculate affected output functions in the device
                        set enable flag to False

}
```

Figure 4.7: Algorithm to process transitions at device enable inputs

with a function as shown in Figure 4.8, could be modeled as a single device with eight inputs, eight outputs, and a clearing asynchronous override. This model is more efficient than using eight separate gates, since it requires less computation, with no loss in precision.

$$\begin{aligned} y_1 &= AND(d_1, gate) \\ y_2 &= AND(d_2, gate) \\ &\vdots \\ y_8 &= AND(d_8, gate) \end{aligned}$$

Figure 4.8: Gate circuit with eight data lines

Asynchronous overrides differ from enable inputs in two ways. First, asynchronous overrides specify not only when an output is active, but also what value it takes on when it is inactive. Second, more than one asynchronous override may be applied to the same output (e.g., clear and set in a flip-flop). This second property presents a problem – what happens when more than one asynchronous override is applied to an output? The solution is to keep track of *how many* overrides are active on each output, in order to properly determine when to reactivate it.

The algorithms for processing asynchronous overrides are simple. Whenever an override becomes active, all the outputs it controls are set to the appropriate logic value, and each of their override counters is incremented. As with enable inputs, before a device output is recalculated, the override count is checked. If the count is zero, then the output is considered to be active, and the output function is evaluated. Otherwise, the evaluation is *skipped.* Whenever an asynchronous override is deactivated, the override counter on each affected output is decremented. *For each*

output whose counter becomes equal to zero, the output function is evaluated.

This procedure for dealing with multiple, simultaneously applied asynchronous overrides unfortunately has the property that the last override asserted on a device output is the one retained until the output is re-enabled. Consider the input sequence: {*Set on, Clear on, Clear off, Set off*}, applied to asynchronous overrides in one device. In this sequence, a device output should follow the sequence $a \rightarrow 1 \rightarrow X \rightarrow 1 \rightarrow b$, where a is the initial output function value, b is the final output function value and device operation when both set and clear are active is undefined. However, using the TLSIM model, the actual sequence will be $a \rightarrow 1 \rightarrow 0 \rightarrow 0 \rightarrow b$. This may not reflect the operation of the device in reality. However, since device behaviour is in general undefined when multiple overrides are applied to devices, this is only a minor issue. Although a simple algorithm could be used to give better model behaviour under multiple overrides, it is probably not worth the additional computation.

The algorithm for processing events applied to asynchronous overrides is shown in Figure 4.9.

4.2.3 Propagation Delays

Physical devices have propagation delays due to capacitive charge and discharge. Each electrical point inside a device has associated rise and fall times, which reflect the time it takes to charge and discharge the capacitance between that node and ground. Furthermore, charge and discharge delays are caused by the capacitance of wire routing between devices. Typically, the delays at device *outputs* dominate, because wire capacitance is much larger than the capacitance of active areas within

Inputs: The device, the asynchronous override number and its new value
Outputs: None

```
Procedure EvalAsynch {
    /* level sensitive override became active */
    if( (active high and value = 1) or (active low and value = 0) )
        for( each affected output )
            increment the asynch count
            if( override value ≠ old output value )
                schedule a transition to the override value

    /* level sensitive override became inactive */
    else if( (active high and value = 0) or (active low and value = 1) )
        for( each affected output )
            decrement the asynch count

    /* momentary override */
    else if( (falling edge triggered and value = 0) or
        (rising edge triggered and value = 1) )

        /* note that the asynch. override count is unaffected */
        for( each affected output )
            if( override value ≠ old output value )
                schedule a transition to the override value
}
```

Figure 4.9: Algorithm to handle signal transitions at asynchronous overrides

devices [26]. Accordingly, TLSIM makes the approximation that device delays are lumped entirely at their outputs (i.e., $t_{wire} + t_{device} \simeq t_{wire}$ since $t_{wire} \gg t_{device}$).

Whenever the value of an output node in a device changes, due to changes in the device's inputs, TLSIM schedules an event at the node in the circuit which is driven by that output. This event represents a transition which is set to occur at the current time, *plus some propagation delay.* Devices have propagation delays defined individually for each output, with rise and fall times specified separately. TLSIM looks up the appropriate delay for each output transition before scheduling an event.

4.2.4 Input Transitions

Whenever a device *input* changes value, TLSIM must decide which truth tables in the device to reevaluate. In general, it must recompute the value of every internal node (reevaluate block A in Figure 3.2, on page 40). This is the case for two reasons – there is no way to temporarily disable an internal node (as is possible with outputs), and TLSIM has no *convenient* way to calculate the dependency relationship between device outputs (which may be disabled) and internal nodes.

After evaluating internal function, TLSIM checks each output in the device. In case it is currently active, i.e., it is enabled, and no asynchronous override is asserted on it, the output function is evaluated. If the new function value differs from the node value, a transition is scheduled to change the node's value after some delay.

4.2.5 Summary

Combining the ideas of the previous sections, we get the algorithm for device evaluation, as shown in Figure 4.10. Note that the algorithm shown is somewhat simplified,

in that it does not show how all the flags in each device are initialized.

4.3 Event Scheduling

TLSIM is an event-driven simulator. This means that inputs cause events to be scheduled, which the simulator processes in a non-decreasing time sequence. When the simulator processes an event, it changes the state of the affected node, and reevaluates every device in the node's fan-out. If any device outputs changed, new events are scheduled at the nodes connected to those outputs.

From the above description of simulator execution, it is evident that TLSIM has three repeating execution phases: event scheduling, retrieval and device evaluation. Accordingly, any improvement in event scheduling leads to significant performance improvement.

To manage an event queue efficiently, TLSIM uses the time wheel algorithm, as outlined in Section 2.4.2, on page 28.

The time wheel algorithm derives its performance from the fact that scheduling events is a trivial operation – they are simply inserted into the linked list at array position

$$i_{insert} = (i_{current\ time} + T_{delay}) \bmod N, \tag{4.1}$$

where $i_{current\ time}$ is the position representing the current time step, T_{delay} is the device propagation delay, and N is the array size.

Similarly, retrieving events from the array is trivial – just increment $i_{current\ time}$ until an occupied array element is reached, and process the list of events at that element. If activity in the circuit is reasonably high, and different devices have

Inputs: The device, the pin number type where the transition occurred
Outputs: None

```
Procedure RecalcDevice {
    /* internal evaluation can't be skipped when inputs change */
    if( device has internal nodes and an input just changed )
        evaluate each internal node

    for( each output )
        /* disabled due to asynchronous override? */
        if( at least one asynchronous override is active )
            continue to next output

        /* input or enable changed to affect the device? */
        if( any input changed and the output is enabled or an enable
            became active and it controls this output )

            /* evaluate the output function */
            new = output function( current inputs )
            node = node which this output drives

            /* schedule changes with the right delay */
            if( new ≠ value( node ) )
                if( new = Z and output is currently on) Δsources = -1
                else if( new ≠ Z and output is currently off) Δsources = 1
                else Δsources = 0

                if( new = 0 )
                    Schedule a transition at node, to new, with a propagation
                    delay t_fall, and add Δsources to the source count.
                else
                    Schedule a transition at node, to new, with a propagation
                    delay t_rise, and add Δsources to the source count.
}
```

Figure 4.10: Device evaluation algorithm

different propagation delays, then the distribution of events over the array will tend to be uniform, and the index $i_{current\ time}$ won't have to be incremented much before a new group of events is found.

The algorithm for scheduling events in the time wheel is shown in Figure 4.11. The algorithm used to retrieve events from the time wheel is shown in Figure 4.11. Note that these algorithms are somewhat more efficient than the classical time wheel algorithm [39] shown in Figure 2.9, shown on page 29, in that they avoid having to make N iterations when the time wheel is empty by keeping count of how many events are still pending in the queue.

These two algorithms suffice for scheduling events produced within the simulator, where propagation delays are bounded. However, inputs applied to the circuit may arrive at any time, with an unbounded delay. This can cause the time wheel algorithm to fail. TLSIM handles externally generated events by keeping track of the time of the next pending *input event.* When TLSIM runs out of internal events to process, or reaches the time for which an input event is pending, it retrieves all the input transitions for the next time step, and updates the 'pending time' variable to indicate the next set of inputs.

The overall procedure for event retrieval, from both the input file and time wheel, is shown in Figure 4.12. It first checks to see if there are input events pending for the current time step, and if so reads them into the queue. Next, it attempts to dequeue a single event. If this fails, it attempts to read *any* events from the file. If this also fails, it ends the simulation. Assuming the previous steps produced a valid event, it is returned.

The procedure for reading input transitions from the stimulus file is shown in

Inputs: The new event (e), and a delay after which the event should be processed
Outputs: None

```
let     currentoffset = the current position in the time wheel.
        array = the time wheel array of linked lists.
        pendingevents = the number of events in the queue.

Procedure Schedule {
        position = ( current_offset + delay ) mod N
        /* link e to the array at position */
        e→next = array[position]
        array[position] = e
        increment pending_events
        }
```

Inputs: None
Outputs: The next event in the queue

```
Function Retrieve {
        if( no events pending in the queue )
                return NULL

        while( array[current_offset]=NULL )
                current_offset = (current_offset + 1) mod (array length)

        /* next time, continue with the next event at this position */
        ptr = array[current_offset]
        array[current_offset] = ptr→next
        return ptr
        }
```

Figure 4.11: Time wheel scheduling and retrieval algorithms

Figure 4.13. This procedure is designed to read *every event for a single time step* into the queue. First, it checks for an end-of-file condition, and quits if this is the case. Next, it reads a single event from the stimulus file (by default called `input`). If this is the first event to be read, or it occurs at the same time as the last read event, the new event is added to the queue. Otherwise, it is returned to a buffer, to be reread later.

```
Inputs: None
Outputs: The next event (e)

let   time = the current simulation time.
      nextinput = the time of the next event in the stimulus file.

Function GetNextEvent {
      /* if there are pending events for this time step, read them */
      if( time ≥ nextinput )
            ReadEvents()

      Retrieve event e from the queue
      if( no event )
            ReadEvents()
            Retrieve event e from the queue
            if( no event )
                  End the simulation

      return e
      }
```

Figure 4.12: Algorithm to retrieve the next event, from either the time wheel or stimulus file

Inputs: None
Outputs: None

```
let     nextinput = the time of the next event in the stimulus file, a global variable.
        thistime, t = time variables

Procedure ReadEvents {
        thistime = -1;
        while( True )
                if( end of file )
                        /* wind down simulation */
                        nextinput = current time + 1000
                        return

                Read: the next event (e) in the file

                /* if not first event and later than last event read */
                if( thistime ≠ -1 and time( e ) ≠ thistime )
                        return e to input stream (to be reread later)
                        nextinput = time( e )
                        return

                Schedule( e )

                thistime = time( e )
        }
```

Figure 4.13: Algorithm to read transitions from stimulus file

4.4 Event Cancellation

During the course of simulation, a rapid sequence of transitions may arrive at a device's inputs. Such a sequence may cause a naíve event-driven simulator to produce incorrect steady-state results. This problem is illustrated in Figure 3.14 on page 58, where a sequence of input events causes a two-input AND gate to reach an incorrect steady-state value.

TLSIM addresses this problem by adding an event-cancellation mechanism to the scheduler. Of n events pending in the queue, all at node N, only the last *scheduled* event is applied to node N. This event may well be different from the *latest* of the n events. To do this, whenever an event is scheduled, the node at which the transition is to occur is marked with an event ID, which identifies that event uniquely. When events are retrieved from the queue, they are only processed if their ID matches the ID stored at their target node. This has the effect that spurious events are queued and processed by the event-scheduling routines, but when they are retrieved spurious events are cancelled.

The event cancellation mechanism identifies events uniquely using a global counter, which is incremented every time an event is allocated. When scheduling an event, TLSIM follows the steps:

1. Allocate an event.
2. Initialize the event's data structure with a node, time and value.
3. Set the event's number to the counter, *global_event_number*.
4. Set the target node's pending event number to *global_event_number*.

5. Increment *global_event_number*.

6. Add the event to the time wheel.

When an event is retrieved, the following steps are taken to remove spurious events:

1. Retrieve an event.

2. Compare the event ID with the number in the node.

3. If they differ – return to step 1 (i.e., cancel this event).

4.5 The Source Count High-Impedance Model

TLSIM gives the high impedance (Z) state special treatment. In many practical circuits two signals may be active on the same node for a brief period, before one of them is disabled, as shown in Figure 2.2 on page 13. Clearly, if the last event to be dequeued in Figure 2.2 is $y \rightarrow Z$, a simulation error will occur. In order to overcome this problem, TLSIM only changes the state of a node to Z when the number of signals asserted on a node becomes equal to zero. To do this, TLSIM must monitor device outputs turning on and off, and count the number of active signal sources at each node.

In order to track transitions to and from Z, when TLSIM schedules events it includes a *source change* variable, $\Delta sources$, in the event data structure. This variable indicates whether the transition at the device output in question was *to* the Z state (in which case $\Delta sources = -1$), *from* the Z state (in which case $\Delta sources = 1$), or

did not involve the Z state ($\Delta sources = 0$). When an event is retrieved from the queue, $\Delta sources$ is immediately added to the source count at its target node. In case the source count of the node then becomes equal to zero, the node is set to Z.

4.6 Simulating Memory Blocks

Memories in TLSIM are simulated using a regular array of binary digits, where the size of the address space is limited by the memory available on the host computer, and the wordlength is limited by the wordlength of the host computer. Memory blocks have three types of connections – data lines, address lines, and a $read/\overline{write}$ input. Address lines are strictly inputs, data lines are bidirectional, and the $read/\overline{write}$ line is an input. Memory blocks are meant to be used as *building blocks* when modeling real memory circuits, such as RAM and ROM chips, so they have no propagation delay. Propagation delays are normally included in the interface circuitry, as shown in Figure 4.14. Furthermore, TLSIM does not distinguish between read-only and read-write memories. There is, however, a facility for setting the initial contents of a memory, so read-only memories are modeled by initializing the contents of a memory block, and fixing the $read/\overline{write}$ line to a logic high value ($read$), as shown in Figure 4.14.

In accordance with the source count model presented in Section 4.5, TLSIM must ensure that the source count at nodes connected to data lines is accurate. Since the direction of signal flow reverses between read and write operations, TLSIM decrements the source count at nodes connected to data lines whenever a memory is set to *write*, and increments the same source counts whenever the memory is set to

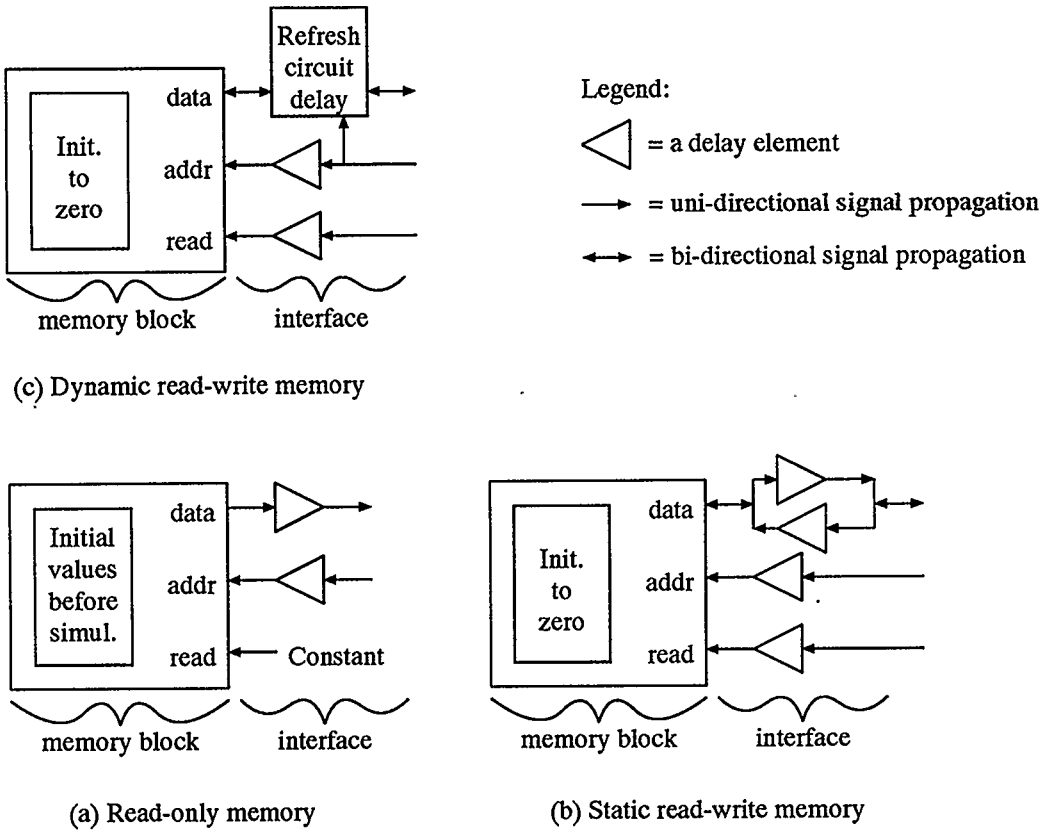


Figure 4.14: Modeling different types of memory in TLSIM

read.

A problem presented by the bidirectional nature of data lines is that the memory block appears in both the fan-in and fan-out of nodes connected to the data lines, as shown in Fig. 4.14(b) and 4.14(c). To avoid unnecessary signal propagation, where signals are passed out of and back into a memory block, whenever a memory block is set to *read*, transitions on the data lines are ignored by the memory block, since by definition that is where they originated.

The procedure for processing events at memory block boundaries is shown in Figure 4.15.

4.7 Memory Allocation

TLSIM allocates a large number of memory blocks, most of which are quite small. Initially, it must allocate data structures representing function and device definitions. Next, it must allocate and initialize a graph of the circuit. Finally, during simulation TLSIM must allocate and deallocate a large number of event blocks. In many simulations, the number of allocated blocks can run into the millions. Clearly, providing a mechanism for fast memory allocation makes a large contribution to simulator performance.

TLSIM contains two separate algorithms for rapid allocation of small memory blocks. The first is an algorithm that allocates large amounts of space from the operating system, and then hands out memory in small blocks to various functions in TLSIM, as required. The performance of this algorithm rests on the assumption that all these small memory allocation units will remain in use until the end of the

Inputs: memory block (mem), pin type, new value
Outputs: None.

```
Procedure EvalMemory {
    switch( pin type )
        case DATA:
            /* ignore data lines when reading */
            if( reading ) return
            data_word = data_word with new bit changed
            data[address_word] = data_word

        case ADDR:
            address_word = address_word with new bit changed
            if( writing ) /* update array at new location */
                data[address_word] = data_word;
            else /* copy new word to data lines */
                for( each data line )
                    if( value( node ) differs from data bit )
                        Schedule an event at this data line
            return

        case READ:
            if( now reading ) /* change to writing */
                data_word = data[address_word]
                for( each data line )
                    if( value( node ) differs from data bit )
                        Schedule an event at this data line
                    increment source count at that node
            else /* change to reading */
                data[address_word]= data_word;
                for( each data line )
                    decrement source count at that node
}
```

Figure 4.15: Procedure to process transitions at memory block inputs and outputs

simulation, so do not need to be deallocated individually. A second algorithm is used to rapidly allocate and deallocate events. The performance of this algorithm is based on the observation that the number of events queued at *any given time* is much smaller than the total number of events required during the course of the simulation. Accordingly, relatively small number of event data structures is *reused* throughout the simulation.

4.7.1 Small Block Allocation

When loading function definitions (by default from the file INTRINS.LIB) and device definitions (by default from the file DEVICE.LIB), as well as the circuit netlist, TLSIM uses a large number of small, dynamically-allocated memory blocks. Node names, fan-ins and fan-outs, truth tables, device definitions, etc. are stored in these blocks. Rather than incur the overhead of calling the operating system's memory allocation function for every small memory block that's needed, TLSIM allocates large blocks of memory, and incrementally uses up the space they provide.

Initially, TLSIM allocates a large stack in memory. Whenever TLSIM requires a small amount of memory, it is pulled from the stack. When the stack is empty, TLSIM allocates a new stack, and the process continues. The procedures for allocating and using the memory stack are illustrated in Figure 4.16.

4.7.2 Event Allocation

During execution, TLSIM uses a large number of events to represent circuit activity. However, only a relatively small number of events are required *simultaneously.* In order to reduce its memory requirements and improve performance, TLSIM uses a

```
Inputs: None
Outputs: None

let    memptr = a static pointer to a large block of memory
       sp = be a static integer used as an offset into this block

Procedure InitSmallMem {
       let memptr = allocate( a large block of memory )
       let sp = 0
       }


Inputs: Number of bytes required (size)
Outputs: A pointer to a free block of memory

Function SmallAllocate {
       if( sp+size > the size of the memory block )
              InitSmallMem()
       /* remember this location and increment sp to reflect the allocation */
       i = sp
       sp = sp + size
       return memptr + i
       }
```

Figure 4.16: Algorithms for allocating small memory blocks

static array of event data structures and a stack of pointers to these structures, as shown in Figure 3.13, on page 57. In case an event is required when the stack is empty, there is a facility for allocating more event data structures, and resizing the stack. The event allocation and deallocation algorithms are shown in Figure 4.17.

```
Inputs: None
Outputs: None

let   eventblock = a pointer to a block of event data structures
      estack = a pointer to a block of pointers to event data structures
      esp = an integer used as an offset into the estack array

Procedure InitEventAlloc {
      let eventblock = allocate( a large number of event data structures )
      let estack = allocate( a large number of event structure pointers)
      let esp = 0
      for( i = 0 to the length of the eventblock array )
            estack[i] = eventblock + i
      }


Inputs: None
Outputs: A pointer to an empty event

Function AllocateEvent {
      if( esp = the length of the eventblock array )
            enlarge the eventblock and estack arrays.
      ptr = estack[esp]
      esp = esp + 1
      return ptr
      }


Inputs: A pointer to an event
Outputs: None

Procedure FreeEvent {
      esp = esp - 1
      estack[esp] = ptr
      }
```

Figure 4.17: Algorithms for managing event allocation

Chapter 5

Algorithm Analysis

This chapter gives an analysis of the execution time incurred by the most frequently called functions in the TLSIM simulator. It does *not* give an an analysis of the various input-file parsers or circuit initialization routines. Furthermore, it makes no attempt to analyze the overall execution time of the simulator, due to its dependence on the following factors:

- Circuit size
- Average fan-out
- Tendency of devices to *propagate* signal transitions
- Number of input vectors

These factors are random and unpredictable, so there is little point in calculating overall execution time, as the resulting formula would be unusable.

On the other hand, it is helpful to calculate the computational complexity of many of the algorithms in TLSIM, as this gives some indication of where the simulator spends much of its execution time, and how simulator performance may, in general, be improved.

The following sections give a brief analysis of each of the major algorithms in Chapter 4.

5.1 The `AddVector` Procedure

This procedure, shown in Figure 4.3 on page 67, is used to add a vector, supplied by the user, to the truth table used to represent a Boolean function. A special algorithm is needed for this purpose since implicit vectors are allowed in the function definition. Essentially, if a vector is explicit, the table at that location is duly initialized. Otherwise, the implicit vector supplied by the user is recursively expanded out into its largest equivalent set, and the same assignment is performed for every resulting explicit vector.

If the user-supplied vector contains n bits set to X, then the size of the equivalent set is 2^n explicit vectors. Since most of the execution time in a short recursive algorithm like this is spent in function-call overhead, it is sufficient to find the number of function calls required to generate each member of the equivalent set.

Consider that a single function call initially starts `AddVector`. Furthermore, if there is an X entry in the input vector, it is replaced with two vectors, so `AddVector` is called again for each. Clearly, this pattern continues, forming a binary tree of function calls with the explicit vectors at its leaf nodes. For a binary tree with m leaf nodes, there are $2m - 1$ edges (or function calls). In our case, for n bits set to X in an implicit vector, $m = 2^n$, and `AddVector` is executed $2(2^n) - 1$ times. Accordingly, its requires $O(2^n)$ time to process a vector with n bits set to X. In practice, vectors seldom contain more than 6 elements, so $n \leq 6$.

5.2 The FillIn Procedure

This procedure, shown in Figure 4.4 on page 69, is used to assign values to every implicit vector that might be used as an argument to a Boolean function. In the **for** loop, every vector is tested to see if it is both valid and implicit. For a function with n arguments, using 2 bits per argument, there are 2^{2n} possible input vectors, so the loop is repeated 2^{2n} times.

A vector is valid if it is of the form: $\{0,1,X\}^n$. Using 2 bits per input, this means that no element in the vector is equal to Z. The probability that a given vector is valid is then $\Pr[\text{valid}] = (\frac{3}{4})^n$.

Similarly, a vector is explicit if it is of the form $\{0,1\}^n$, for some n. Given that it is valid, the probability of a vector being explicit is just $(\frac{2}{3})^n$. If a vector is not explicit, it must be implicit, so the probability that a vector is implicit is $1-(\frac{2}{3})^n$.

Whenever a vector is valid and implicit, the function `EvalEquivalent` is called. In the same way as `AddVector` in Section 5.1, `EvalEquivalent` requires a total of $2(2^l)-1$ recursions, where l is the number of X elements in its initial argument. We must therefore calculate the average value of l, $\bar{l}$, for implicit vectors in order to find the average time complexity of the `EvalEquivalent` function, and thus the `FillIn` procedure.

Consider a vector of length n bits, of the form $\{0,1,X\}^n$. The probability that the number of X elements in the vector is k is given by:

$$Pr[k] = C_n^k(\frac{1}{3})^k(\frac{2}{3})^{n-k} = \frac{n!}{(n-k)!k!}(\frac{1}{2})^k(\frac{2}{3})^n \tag{5.1}$$

The average number of X elements in a completely random vector is thus given

by:

$$\bar{l} = \sum_{k=0}^{n} k(Pr[k]) = \sum_{k=1}^{n} k(Pr[k]) \tag{5.2}$$

Substituting, we get:

$$\bar{l} = (\frac{2}{3})^n n! \sum_{k=1}^{n} (\frac{1}{2})^k \frac{1}{(n-k)!(k-1)!} \tag{5.3}$$

After some manipulation, this result can be reduced to:

$$\bar{l} = \frac{n}{3} \tag{5.4}$$

An implicit vector of length n can be formed from a random vector of length $n-1$ by placing a X in one of the implicit vector's positions, and using the random vector to fill all the remaining elements. Accordingly, we can use the previous results to find the average number of X elements in an *implicit* vector:

$$\overline{l_{implicit}} = 1 + \frac{n-1}{3} = \frac{n+2}{3} \tag{5.5}$$

Applying the above results, `EvalEquivalent` is executed, on average, $2(2^{(n+2)/3}) - 1$ times per implicit vector. The probability of a vector being implicit is $(3/4)^n(1-(2/3)^n)$. The overall computational complexity of `FillIn` is then:

$$2^{2n}(\frac{3}{4})^n(1-(\frac{2}{3})^n)(2(2^{(n+2)/3}) - 1) \tag{5.6}$$

For large n, we can let $1-(2/3)^n \simeq 1$, and $2(2^{(n+2)/3}) - 1 \simeq 2^{(n+5)/3}$. The computational complexity of the `FillIn` algorithm then becomes:

$$2^{(n+5)/3}3^n \tag{5.7}$$

This expression may be rearranged as follows:

$$\exp(\frac{n+5}{3}\ln 2)\exp(n\ln 3) \tag{5.8}$$

$$\exp(\frac{n+5}{3}\ln 2 + n\ln 3) \tag{5.9}$$

For $n >> 1$, we factor out the n terms, to get the approximate computational complexity of `FillIn`:

$$\exp(n\ln(3.78)) = 3.78^n \tag{5.10}$$

Overall, then, the computational complexity of `FillIn`, for a truth table with n arguments, is $O(3.78^n)$. As before, it is fortunate that n is limited to at most 6 in practice.

5.3 The `EvalFunction` Function

This function, shown in Figure 4.6 on page 71, is used to evaluate Boolean functions when device inputs change. `EvalFunction` uses a table lookup approach to find the value of a Boolean function for a given set of inputs. An offset vector into the truth table is formed from the function arguments, and the table's value at that offset is returned.

For a function with n inputs, it takes n steps to form the offset vector, so the `EvalFunction` algorithm requires an $O(n)$ computational effort.

5.4 The `Schedule` Procedure

This procedure, shown in Figure 4.11 on page 81, is used to add events to the queue. In the time-wheel paradigm, this is just a matter of calculating the insertion point, and attaching the event to the approrpiate linked list. Accordingly, scheduling requires only an $O(1)$ computational effort.

5.5 The `RecalcDevice` Procedure

This procedure, shown in Figure 4.10 on page 79, is used to update the state of a device, following a change in the state of one of its data inputs, enables, or asynchronous overrides.

Consider a device with d data inputs, i internal nodes, and o outputs. In principle, every Boolean function could have as many as $d + i + o$ arguments. As outlined in Section 5.3, recalculating the state of a Boolean function with m arguments is an $O(m)$ operation. Accordingly, the loop for evaluating every internal node requires an $O(i(d + i + o))$ effort.

Next, a **for** loop checks the state of each output. If every output function is calculated, the **for** loop requires an $O(o(d + i + o))$ effort.

Together, then, for d inputs, i internal nodes and o outputs, the `RecalcDevice` procedure may require as much as an $O((i + o)(d + i + o))$ effort. If d, i and o are comparable, then we may define a new quantity, n, to be the total number of nodes in the device, such that $n = d+i+o$. In this case, we may say that the `RecalcDevice` algorithm requires an $O(n^2)$ effort. *Per device output*, this is approximately $O(n)$.

5.6 The `EvalEnable` Procedure

This procedure, shown in Figure 4.7 on page 73, is used to decide whether or not a device evaluation must be performed after a change in the state of one of a device's enables. Since it just makes some decisions and possibly calls the `RecalcDevice` function once, it requires the same order of computational effort as `RecalcDevice`: $O(n^2)$, where the device has n nodes.

5.7 The `EvalAsynch` Procedure

This procedure, shown in Figure 4.9 on page 76, is used to decide whether or not device evaluation must be performed after a change in the state of one of a device's asynchronous overrides. For a device with o outputs, as many as o outputs may be affected by the state of the asynchronous override that triggered `EvalAsynch`'s execution. Since a loop is executed to update the asynchronous override count at each output, after which the `RecalcDevice` procedure may be called, the computational complexity of `EvalAsynch` is $O(n^2) + O(n) \simeq O(n^2)$, for a device with n internal nodes and outputs.

5.8 The `EvalMemory` Procedure

This procedure, shown in Figure 4.15 on page 89, is used to process every transition that arrives at the boundaries of a memory cell. It actually consists of three mutually-exclusive segments of code, which handle transitions at data lines, address lines, and the read line, respectively. As outlined at the start of this chapter, it is impossible

to predict the relative frequency with which these will be executed in practice.

In case the transition occurred on a data line, `EvalMemory` executes a simple, $O(1)$ code segment.

In case the transition occurred on an address line, one of two things may happen. If the memory is in *write mode*, an $O(1)$ operation is performed to update the contents of the buffer. Otherwise, in *read mode*, up to d transitions may have to be scheduled, for a memory cell with d data lines. Accordingly, in this case the `EvalMemory` procedure requires an $O(d)$ effort.

In case the transition occurred on the read line, one of two **for** loops must be executed. Both, however, are executed once per data line, so an $O(d)$ effort is required.

Overall, then, evaluating a memory cell requires either an $O(1)$ or an $O(d)$ effort.

5.9 The `Retrieve` Function

This function, shown in Figure 4.11 on page 81, is used to dequeue events from the timewheel. This is done by checking every position, starting with the current position in the time wheel, to see if it is empty. At the first location in the time wheel which contains one or more events, this function removes one event from the linked list.

Clearly, the order of this algorithm depends on the number of iterations in the **while** loop. This number, in turn, depends both on the level of activity in the circuit, and the distribution of different propagation delay values over devices in use in the circuit.

For instance, there may be heavy activity, but every device may have a fixed propagation delay of 50 time units. In this case, the **while** loop is executed either zero or 50 times, depending on whether all events at the current position in the timewheel have been exhausted yet.

Alternately, the circuit may have many devices with different propagation delays. In this case, assuming there is sufficiently heavy activity, the events will be spread out more evenly over the timewheel, and the **while** loop will be executed either zero or one times per call to **Retrieve**.

In practice, it is hoped that users will give devices realistic (and therefore varied) propagation delays. This will lead to an even distribution of events over the timewheel, and thus an average computational complexity of $O(1)$ for the `Retrieve` function. However, the worst case performance of this algorithm remains $O(n)$, where n is the size of the time wheel.

5.10 The `GetNextEvent` Function

This function, shown in Figure 4.12 on page 82, is used to retrieve the next event, be it in the queue or pending in the input file. To do this, it compares the time of the next input event with the time of the next event in the queue, and decides from which source it should fetch an event.

Since this function has no loops, its computational complexity depends on that of any functions it might call. **GetNextEvent** calls only queue scheduling and retrieval events, which all average $O(1)$, so its own computational complexity is $O(1)$.

5.11 The `ReadEvents` Procedure

This procedure, shown in Figure 4.13 on page 83, is used to read a set of events, all of whom are scheduled to occur simultaneously, from the stimulus file. Its only loop iterates as many times as there are simultaneous events, so if there are an average of n events per time step in the input file, its computational complexity is $O(n)$.

5.12 Memory Management

The memory allocation and deallocation routines used to accelerate TLSIM's execution contain no loops, and call no other functions, so they all execute in $O(1)$. This is crucial since they are called so frequently.

Chapter 6

Experimental Results

6.1 Correctness

In order to verify the correct operation of the TLSIM circuit simulator, a number of circuits were implemented. For each design, the circuit was simulated and the simulation results were manually verified. This testing procedure was carried out over a 12 month period, until no new errors in simulation results were found.

Among the circuits used to test TLSIM were:

- A small 8-bit CPU
- A 4-bit adder implementing carry look-ahead
- A serial-line interface circuit
- Several small ALU's and counters
- An 8-bit by 8-bit multiplier
- A pure-tone sound synthesizer
- A distributed-arithmetic matrix multiplier

In addition, the full complement of ISCAS'85 [8] and ISCAS'89 [7] benchmark circuits were simulated with TLSIM using randomly generated test vectors. To verify correct operation, a translator from the TLSIM network description language to the

Verilog [18] netlist language was written. The benchmark circuits were translated to Verilog, and resimulated using that program. The simulation results from the two programs were compared graphically. In each case, the results matched.

6.2 Performance

In order to test the performance of the TLSIM circuit simulator, both TLSIM and Verilog were used to simulate the circuits mentioned above. Identical randomly generated test vectors were applied to each program. Note that all simulations were performed using a Sun SPARCStation 2 computer.

In order to demonstrate that TLSIM executes rapidly, benchmarking was performed in two steps. First, to show that TLSIM performs well even when the expressive capability of the UNIMOD1 device model is *not* utilized, the ISCAS'85 and ISCAS'89 benchmark circuits were simulated with both TLSIM and Verilog. Since these circuits use only gates and individual flip-flops, this test demonstrated that *at the gate level*, TLSIM executes at least twice as fast, on average, as Verilog. Next, a small number of test circuits were simulated both at the gate level and using more complex devices. Simulating circuits using more complex devices was approximately three times faster than using gates, indicating that a potential factor of six speed improvement can be achieved by using TLSIM with high-level devices instead of using Verilog with gates only.

6.2.1 Verilog Simulations

In order to make the timing results that follow more meaningful, some information about Verilog is required. The version of Verilog used for all the following simulations is Verilog 1.6.0.1, from Cadence. Verilog is a multi-level simulation language, in that it can simulate everything from switches to procedural constructs. Clearly, this very flexibility causes some overhead in its execution. Nonetheless, many people use Verilog as a gate-level simulator, although it is capable of much more. Accordingly, comparison of gate-level simulation results with Verilog are reasonable.

The ISCAS'85 and ISCAS'89 circuits are distributed as flat netlists. They were translated to Verilog using an automatic translator, which produced modules such as the one in Figure 6.1.

6.2.2 ISCAS Benchmarks

Benchmark circuits from the 1985 and 1989 International Symposium on Circuits and Systems (ISCAS) were used to test TLSIM. The ISCAS'85 benchmark circuits are combinational circuits, consisting only of AND, OR, XOR and inverter gates. Circuit size parameters are shown in Table 6.1.

The ISCAS'89 circuits are sequential, in that they also include D-type flip-flops. Size parameters for these circuits are given in Table 6.2. Note that inverters are listed separately from other gates in the documentation supplied with these benchmarks.

Performance results from the benchmark tests are summarized in two figures: Figure 6.2 shows the execution time required to simulate each circuit in the ISCAS'85 benchmark set. Figure 6.3 gives this information for the ISCAS'89 benchmark circuits.

```
primitive dff(q,d,clk,r);
   input d,clk,r;
   output q;
   reg q;
   table
   // d    clk   r    : q : q+
      ?    ?     0    : ? : 0;
      ?    ?     x    : 0 : 0;
      0    (10)  1    : ? : 0;
      1    (10)  1    : ? : 1;
      ?    (01)  1    : ? : -;
      (??) ?     ?    : ? : -;
      ?    ?     (??) : ? : -;
   endtable
endprimitive


module testnet(clear,clock,g3,g0,g1,g2,g17);
  input clear,clock,g3,g0,g1,g2;
  output g17;
  wire g5,g6,g7,g8,g9,g10,g11,g12,g13,g14,g15,g16;
  supply0 f;
  supply1 t;
   nor #(10,10) device1(g13,g2,g12);
   nor #(10,10) device2(g12,g1,g7);
   nor #(10,10) device3(g11,g5,g9);
   nor #(10,10) device4(g10,g14,g11);
   nand #(10,10) device5(g9,g16,g15);
   or #(10,10) device6(g16,g3,g8);
   or #(10,10) device7(g15,g12,g8);
   and #(10,10) device8(g8,g14,g6);
   not #(10,10) device9(g17,g11);
   not #(10,10) device10(g14,g0);
   dff #(10,20) device11(g7,g13,clock,clear);
   dff #(10,20) device12(g6,g11,clock,clear);
   dff #(10,20) device13(g5,g10,clock,clear);
endmodule
```

Figure 6.1: Sample Verilog test circuit: ISCAS'89/s27

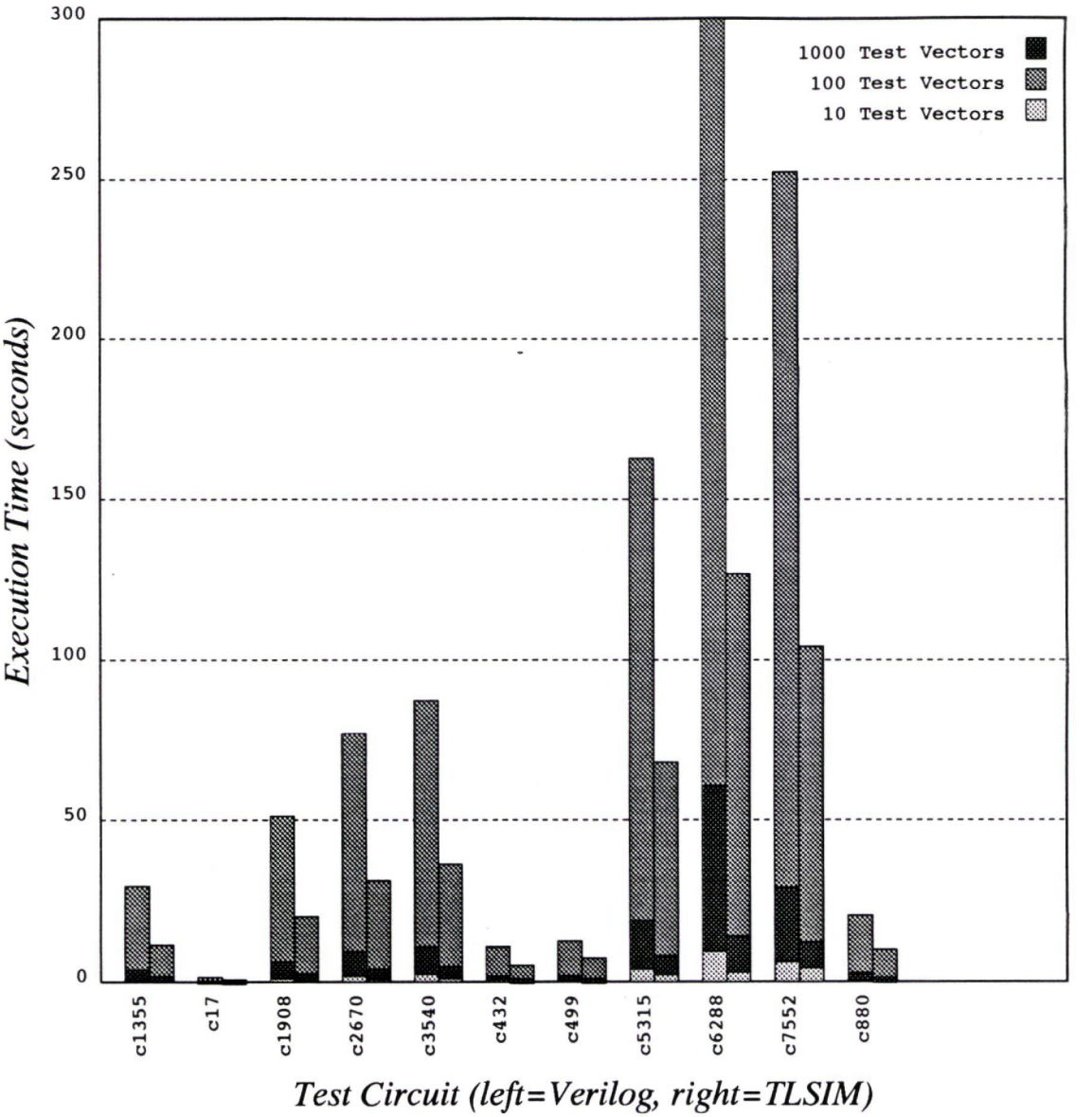


Figure 6.2: ISCAS'85 benchmark circuit simulation performance

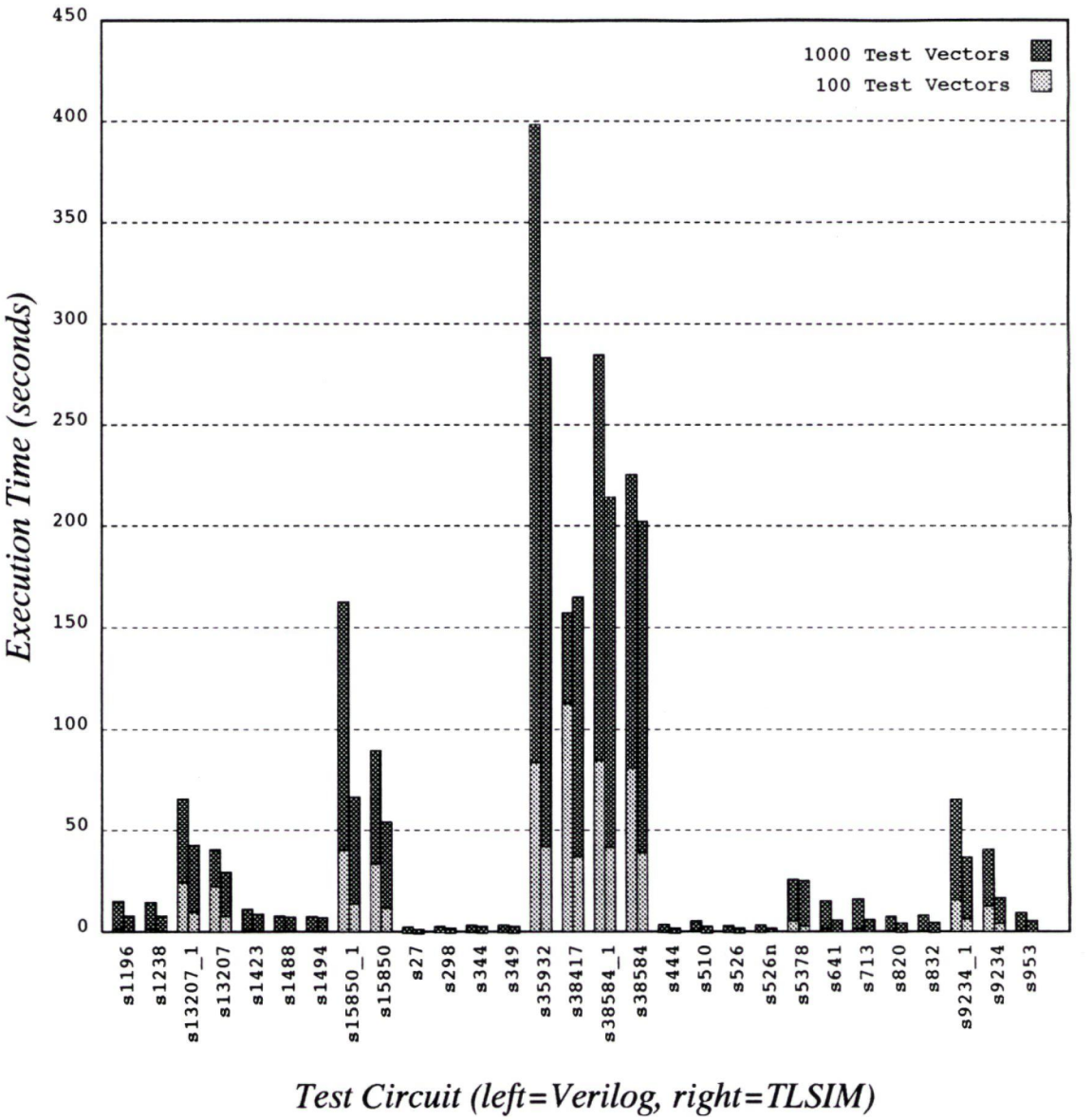


Figure 6.3: ISCAS'89 benchmark circuit simulation performance

Table 6.1: ISCAS'85 benchmark circuit parameters

Circuit Name	Number of Gates
c17	6
c432	160
c499	202
c880	383
c1355	546
c1908	880
c2670	1193
c3540	1669
c5315	2307
c6288	2416
c7552	3512

From these figures, it is evident that TLSIM is approximately twice as fast as Verilog when performing gate-level simulation. Note that the ISCAS'89 circuits showed much higher activity counts, so the overhead incurred by scheduling used up a greater fraction of the execution time. This accounts for TLSIM's smaller performance gain in simulating these circuits, since the advantages of UNIMOD1 do not come into play to the same degree.

To illustrate the balance between the scheduling and device evaluation portions of TLSIM, execution profiles were calculated using some of the ISCAS'85 and ISCAS'89 circuits. These profiles are shown in Figure 3.9, on page 52. The *Recalc-Device* function is used to evaluate the state of devices in the circuit. *Eval-Node* is a function for evaluating truth tables, and the functions *Add-Heap*, *Schedule* and *Pull-Heap* are all associated with processing events in the queue.

Note that the largest portion of the execution time is taken up by the *Recalc-Device* function. This function must check each device output, determine whether or not to evaluate it, and if so it must call function-evaluation and scheduling pro-

Table 6.2: ISCAS'89 benchmark circuit parameters

Circuit Name	DFFs	INVs	Other gates	Total gates
s27	3	2	8	10
s208	8	38	66	104
s298	14	44	75	119
s344	15	59	101	160
s349	15	57	104	161
s382	21	59	99	158
s386	6	41	118	159
s400	21	58	106	164
s420	16	78	140	218
s444	21	62	119	181
s510	6	32	179	211
s526	21	52	141	193
s526n	21	54	140	194
s641	19	272	107	379
s713	19	254	139	393
s820	5	33	256	289
s832	5	25	262	287
s838	32	158	288	446
s953	29	84	311	395
s1196	18	141	388	529
s1238	18	80	428	508
s1423	74	167	490	657
s1488	6	103	550	653
s1494	6	89	558	647
s5378	179	1775	1004	2779
s9234	228	3570	2027	5597
s9234.1	211	3570	2027	5597
s13207	669	5378	2573	7951
s13207.1	638	5378	2573	7951
s15850	597	6324	3448	9772
s15850.1	534	6324	3448	9772
s35932	1728	3861	12204	16065
s38417	1636	13470	8709	22179
s38584	1452	7805	11448	19253
s38584.1	1426	7805	11448	19253

cedures. This involves more computation than any of the other functions listed.

6.2.3 Functional Device Modeling

Three circuits were implemented at both the gate and functional levels. These circuits are described briefly below:

Sound Synthesizer This circuit uses frequency division techniques to approximate any of the 96 pure tones in the musical scale. It uses a number of counters of different types (loadable, incrementing, decrementing and bidirectional), as well as some logic to generate count-down delays, and to drive seven-segment displays (a simple user interface). Due to its nature, this circuit contains a large amount of sequential logic, in the form of counters, flip-flops, etc. The functional level description of this circuit made use of multiplexors, counters and multi-bit latches in addition to simple gates.

8×8 **Bit Binary Multiplier** This circuit is composed of four 4-bit by 4-bit multipliers, and some logic to add up the results. The 4×4 multipliers are, in turn, made up of more adders. No special carry-propagation circuitry was used in this multiplier. The circuit was implemented at the gate level, using only AND, OR and NOT gates, and at a functional level, where small adders were represented as monolithic devices. Adders with 2, 3, 4 and 5 inputs were modeled (e.g., the 5-input adder has three outputs, and can yield an answer as high as 101).

Distributed Arithmetic Matrix Multiplier This circuit forms the product of an 8×8 matrix by an 8×1 vector, generating all eight elements of the product

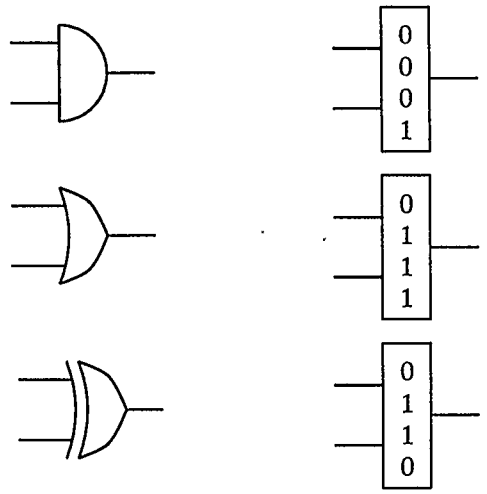
in parallel. For simplicity, all input quantities were taken to be 8 bits wide, thus giving 19-bit product terms. This circuit was implemented using 1-bit latches and NAND gates for the gate-level simulation. For the functional simulation, multiplexors, shift registers and latches were modeled as monolithic, multi-bit devices. Furthermore, the adders were modeled using 4-bit slices.

Some size parameters for these circuits are given in Table 6.3.

Table 6.3: Circuit parameters for custom circuits

Circuit	Number of Nodes		Number of Devices	
	Gate-level	Behavioural	Gate-level	Behavioural
Distributed-Arithmetic Matrix Multiplier	5530	858	5396	80
Sound Synthesizer	277	108	264	80
Multiplier	1123	191	1107	64

Appropriate test vectors were made up for each circuit, and the three circuits were simulated on a Sun SPARCStation 2. The results of these simulations are shown in Figure 6.4. Note how in all cases the functional model performed much better than the gate-level representation. This is of significant importance, since practical VLSI circuits are often designed using macro cell libraries, rather than just gates. This means that the functional model can provide *improved* simulation fidelity while at the same time reducing execution time.



(a) Gate-based Model (b) Truth-Table Model

Figure 6.4: Performance improvement with functional modeling

Chapter 7

Discussion

This chapter reviews the significance of the major contributions of the TLSIM simulator to the methodology of circuit simulation. It focuses on three key areas: The application of interpretive simulation, the UNIMOD1 device model, and the algorithms for rapid evaluation of four-valued Boolean functions.

7.1 Interpretive Simulation

One of the key points of this thesis is to reinforce the idea that interpretive simulation is very useful in circuit design. Recently, the focus in the literature has been on compiled simulation, which is limited primarily to use in fault simulation (for automatic generation of test vectors).

Compiled simulators are, in practice, limited to fault testing because they can easily require 10 or more minutes [40] just to compile the circuit, before simulation can even begin. This is unacceptable when the simulation run itself only takes several seconds.

Although interpretive simulation is inherently slower than compiled simulation, often by approximately an order of magnitude, it is nonetheless more suited to circuit design, since the latency in compiled simulation dominates the amount of time required when simulation runs are short.

If interpretive simulation is to be used, then the task remains to reduce the gap

between the execution speed of interpretive and compiled simulation, while retaining the low overhead inherent in the interpretive paradigm. This thesis offers some solutions to this problem: Notably, a computationally efficient device model and algorithms for rapid evaluation of four-valued Boolean functions.

7.2 The UNIMOD1 Device Model

The main method by which TLSIM attempts to accelerate the simulation process is by using a device model which evaluates rapidly. UNIMOD1 makes it possible to extend the classical event driven concept[39] to the internal operation of devices. By defining clocking and asynchronous override inputs separately from data inputs, UNIMOD1 makes it possible to minimize the amount of computation required to recalculate the state of a device's outputs.

This extension to the classical event-driven paradigm is only possible when modeling devices using functional blocks. Little acceleration is achieved if a circuit is modeled using primitive elements such as gates or switches. This is only a minor constraint, however, as larger circuit elements, such as registers, latches and counters are often used as circuit building blocks. This is true both in the design of digital circuits made up of discrete components, and in the design of ASIC and semi-custom VLSI circuits.

7.3 Rapid Evaluation of Four-Valued Truth Tables

The second method by which TLSIM accelerates the simulation process is the rapid evaluation of Boolean functions. Since a large portion of the simulation is spent

evaluating these functions, it is critical to make function evaluation as efficient as possible.

TLSIM uses a four-valued signal model, in order to alleviate the need for determining signal strengths. This is useful at the circuit design stage, when routing information is rarely available, since without this information it is impossible to predict node capacitances.

The use of a four-valued signal model complicates the function evaluation algorithm, however. Even though the four possible values of function arguments can be reduced to three, by mapping the Z state to one of states: $\{0, 1, X\}$, the question of how to handle X inputs remains.

In TLSIM, Boolean functions are represented as arrays. This has the primary advantage that function evaluation is very rapid (an $O(n)$ operation, for n-input functions). However, in order to accommodate the possibility that X arguments might be applied to these functions, TLSIM must "fill in" the arrays for *all possible input vectors* before they are used. This "fill-in" function is performed when initializing the truth tables.

Since function initialization is performed only once per function, before simulation begins, it incurs very little overhead. Function evaluation *during simulation* is therefore very rapid.

Chapter 8

Conclusion

8.1 Accomplishments

This thesis describes the development of a digital circuit simulator. This simulator is by no means unique, in that many software circuit simulation tools are commercially available. However, this simulator does offer some interesting features: It is optimized for digital system *design*, and tuned to model realistic primitive components.

This simulator embodies UNIMOD1: a *UNI*fied device *MOD*el, designed specifically for modeling small- to medium-scale macro cells such as those available in VLSI standard cell libraries. Unlike many other packages, this simulator attempts to address a single issue only: the simulation of netlists at the level of gates and functional blocks. TLSIM gains a measure of efficiency by avoiding other levels of abstraction such as the switch-level and algorithmic models.

8.2 Future Work

The number of features that could be added to a simulator such as TLSIM, to make it more useful as well as easy-to-use, is nearly boundless. The addition of such features could be carried out for the entirety of the software's useful life span. Of these features, some of the most useful might include the following:

Pessimism could be added to the simulator in several ways. For instance, when

a node undergoes a transition, its value could be set to X rather than held fixed during the transition period. Another place where the simulator could make more "conservative" predictions is in cases where more than one signal is simultaneously asserted on a node (such as in a transient signal conflict on a tri-state bus). Currently, TLSIM applies new values to nodes as they arrive, but it could set a node's value to X whenever more than one device attempts to drive it simultaneously.

Another way in which TLSIM could be made more flexible is by improving its delay model. Currently, it only models rise and fall times for device outputs. These delays are associated with each device, but may be overridden by the user for any node. A more detailed model would give $4 \times 3 = 12$ different delays, from every signal value to every other signal value. Better still would be a delay model that in some way takes into account the dynamic loading that appears at each node as devices in its fan-out change state.

Other features that could be added are likely not as generally useful. For instance, direct support for multi-port memories might be added. Currently, memory blocks in TLSIM have a single address and a single data port, but this could be changed to allow for multiple data and address ports to a single storage block.

As it stands, however, TLSIM is reasonably complete.

Bibliography

[1] Dan Adler. SIMMOS: A Multiple-Delay Switch-Level Simulator. In *23rd ACM/IEEE Design Automation Conference*, 1986.

[2] Dan Adler. Switch-Level Simulation Using Dynamic Graph Algorithms. *IEEE Transactions on Computer Aided Design*, 10, No. 3, March 1991.

[3] Prathima Agrawal and Vishwant D. Agrawal. Can Logic Simulators Handle Bidirectionality and Charge Sharing? *IEEE International Symposium on Circuits and Systems*, 1990.

[4] Prathima Agrawal and William J. Dally. A Hardware Logic Simulation System. *IEEE Transactions on Computer-Aided Design*, 9, No. 1, January 1990.

[5] Z. Barzilai. SLS – A Fast Switch Level Simulator for Verification and Fault. In *23rd ACM/IEEE Design Automation Conference*, 1986.

[6] D. T. Blaaw. Automatic Generation of Behavioral Models from Switch-Level Descriptions. In *26th ACM/IEEE Design Automation Conference*, 1989.

[7] F. Brglez, D. Bryan, and K. Kozminski. Notes on the ISCAS'89 Benchmark Circuits. *MCNC*, October 1989.

[8] F. Brglez and H. Fujiwara. A neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in FORTRAN. *IEEE International Symposium on Circuits and Systems*, June 1985.

[9] R. E. Bryant. A Switch-Level Model and Simulator for MOS Digital Systems. *IEEE Transactions on Computers*, C-33, No. 2, February 1984.

[10] Eduard Cernay, John P. Hayes, and Nicholas C. Rumin. Accuracy of Magnitude-Class Calculations in Switch- Level Modeling. *IEEE Transactions on Computer-Aided Design*, 11, No. 4, April 1992.

[11] Basant R. Chawla, Hermann K. Gummel, and Paul Kozak. MOTIS – An MOS Timing Simulator. *IEEE Transactions on Circuits and Systems*, CAS-22, No. 12, December 1975.

[12] Chorng-Yeong Chu and Mark Alan Horowitz. Charge-Sharing Models for Switch-Level Simulation. *IEEE Transactions on CAD*, CAD-6, No. 6, November 1987.

[13] E. D. Fabricius. *Introduction to VLSI Design.* McGraw-Hill, 1990.

[14] G.E. Flores and B. Kirkpatrik. Optical Lithography Stalls X Rays. *IEEE Spectrum*, 28-10, October 1991.

[15] J. Gu. Research and Development of VLSI CAD Systems. Private Communications, 1990-1993.

[16] J. Gu and K.F. Smith. A Structured Approach for VLSI Circuit Design. *IEEE Computer*, 22(11):9–22, Nov. 1989.

[17] D. A. Hodges and H. G. Jackson. *Analysis and Design of Digital Integrated Circuits.* McGraw-Hill, 2nd edition, 1988.

[18] Cadence Design Systems Inc. *Verilog-XL Reference Manual*, March 1991.

[19] Brion L. Keller, David P. Carlson, and William B. Maloney. The Compiled Logic Simulator. *IEEE Design and Test of Computers*, March 1991.

[20] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall Inc., New York, 1978.

[21] G.A. Korn and J.V. Wait. *Digital Continuous-System Simulation.* Prentice-Hall Inc., New York, 1978.

[22] David M. Lewis. A Hierarchical Compiled Code Event-Driven Logic Simulator. *IEEE Transactions on Computer Aided Design*, 10, No. 6, June 1991.

[23] Wojciech Maly. Prospects for WSI: A Manufacturing Perspective. *IEEE Computer*, 25-4, April 1992.

[24] Peter M. Maurer. Scheduling Blocks of Hierarchical Compiled Simulation of Combinational Circuits. *IEEE Transactions on Computer-Aided Design*, 10, No. 2, February 1991.

[25] Peter M. Maurer and Zhicheng Wang. Techniques for Unit-Delay Compiled Simulation. In *27th IEEE/ACM Design Automation Conference*, 1990.

[26] Carver Mead and Lynn Conway. *Introduction to VLSI Systems.* Addison-Wesley Inc., 1980.

[27] Steve Meyer. A Data Structure for Circuit Net Lists. In *25th ACM/IEEE Design Automation Conference*, 1988.

[28] Masayuki Miyoshi. Speed Up Techniques of Logic Simulation. In *22nd Design Automation Conference*, 1985.

[29] V. Nagasamy, N. Berry, and C. Dangelo. Specification, planning, and synthesis in a VHDL design environment. *IEEE Design and Test of Computers*, 9, No. 2, June 1992.

[30] Lissa F. Pollacia. A Survey of Discrete Event Simulation and State-of-the-Art Discrete Event Languages. *Simulation Digest*, 20-3, Fall 1989.

[31] Vijaya Ramachandran. An Improved Switch-Level Simulator for MOS Circuits. In *20th ACM/IEEE Design Automation Conference*, 1983.

[32] M. H. Rashid. *SPICE For Circuits And Electronics Using PSpice.* Prentice Hall, 1990.

[33] Arturo Salz and Mark Horowitz. IRSIM – An Incremental MOS Switch-Level Simulator. In *26th ACM/IEEE Design Automation Conference*, 1989.

[34] I. Shoham and J. Gu. UNIMOD1: A New Device Model for Digital Circuit Simulation. Submitted for Publication, 1993.

[35] R. Sosič, J. Gu, and R. Johnson. The Unison Algorithm: Fast Evaluation of Boolean Expressions. *Communications of ACM*, Accepted for publication, 1990. To appear.

[36] Stubbs and Webre. *Data Structures with Abstract Data Types and Pascal.* Brooks / Cole, 2nd edition, 1989.

[37] K. Subramanian and M. R. Zargham. Distributed and Parallel Demand Driven Logic Simulation. In *27th IEEE/ACM Design Automation Conference*, 1990.

[38] Robert Tjarnstrom. Switch-level simulation based on local decisions. *INTEGRATION, the VLSI journal*, 9, No. 3, July 1990.

[39] E. G. Ulrich. Exclusive Simulation of Activity in Digital Networks. *Communications of the ACM*, 12, No. 2, February 1969.

[40] Zhicheng Wang and Peter M. Maurer. LECSIM: A Levelized Event Driven Compiled Logic Simulator. In *27th IEEE/ACM Design Automation Conference*, 1990.