VLSI Systems Simulation

Michael T. Pope

A thesis submitted to the faculty of Engineering
for the degree of Doctor of Philosophy

Department of Electrical and Electronic Engineering
The University of Adelaide
South Australia

January 1991

Contents

1 Introduction **1**

1.1 Simulation 1

1.1.1 The Transform and Filter Brick 2

1.1.2 Communications Link 7

1.1.3 Lessons and conclusions 8

1.2 Overview 10

2 Literature Review **11**

2.1 Introduction 11

2.2 Taxonomy of Modes 11

2.2.1 Architectural 12

2.2.2 Structural 13

2.2.3 Functional 14

2.2.4 Digital 15

2.2.5 Analog 16

2.2.6 Device 17

2.2.7 Timing 18

2.2.8 Fault 18

2.2.9 Mixed and Hybrid 18

2.3 Functional and Structural Modes 19

2.3.1 Introduction 19
2.3.2 Hardware Description Languages 20
2.3.3 Low Level Modelling 23
2.3.4 Timing Analysis 28
2.3.5 Goals 32
2.4 Digital Modes 33
2.4.1 Introduction 33
2.4.2 Logic States 34
2.4.3 Logic Strengths 37
2.4.4 Digital Simulation Algebras 41
2.5 Bidirectionality 42
2.5.1 Delay Modelling 44
2.5.2 Other Features 44
2.6 Analog Modes 48
2.6.1 Introduction 48
2.6.2 Analog Circuit Equation Formulation 48
2.6.3 SPICE 51
2.7 Speeding up SPICE 54
2.7.1 A Philosophical Digression 57
2.7.2 Alternative Analog Approaches 61
2.8 Mixed Modes 65
2.9 Hybrid Modes 71
2.10 Summary 75

3 Design of Loge **77**
3.1 Introduction 77
3.2 Structure 77
3.2.1 HDL issues 78

3.3 Scheduling 81
3.3.1 Scheduling Algorithms 82
3.3.2 Event Suppression 84
3.4 Simulation Objects 87
3.4.1 Nodes 89
3.4.2 Ports 101
3.4.3 Modules 109
3.4.4 Devices 129
3.4.5 Analog Devices 148
3.4.6 Example 149
3.4.7 Summary 150
3.5 Simulation 151
3.5.1 Simulation control 152
3.5.2 Tools 152
3.5.3 Verification 157

4 Results **167**
4.1 Introduction 167
4.2 General Performance Tests 167
4.2.1 Ring Oscillator 168
4.2.2 Shift Register 170
4.2.3 Adder 171
4.2.4 Binary Trees 177
4.2.5 Barrel Shifter 185
4.2.6 Memory 186
4.2.7 Simulator performance 188
4.2.8 Summary 190
4.3 GaAs technology 192

4.3.1 MESFET current models . . . 192
4.3.2 Simulations . . . 195
4.3.3 Summary . . . 198
4.4 Design of a Combinator Engine . . . 203
4.4.1 Introduction . . . 203
4.4.2 Functional Programming and Combinators . . . 203
4.4.3 Combinator Compilation Example . . . 205
4.4.4 Basic Architecture . . . 206
4.4.5 Tokens . . . 208
4.4.6 Argument Counting . . . 213
4.4.7 Combinator Execution Example . . . 214
4.4.8 Stream . . . 215
4.4.9 Control . . . 216
4.4.10 Combinator Processor Design . . . 217
4.4.11 Combinator Processor: A Loge Functional Description 221
4.4.12 Combinator Processor: Hardware Partitioning . . . 233
4.4.13 Other stream elements . . . 235
4.4.14 Stream Group . . . 235
4.4.15 Top Level . . . 237
4.4.16 Simulations . . . 238

5 Conclusions **251**

Abstract

The complexity of modern digital systems, particularly VLSI systems, continues to grow quickly. Simulation has become entrenched as a vital technique for dealing with this complexity, in a variety of ways. In simulation there is a perennial tradeoff between the conflicting goals of accuracy of the results and speed of simulation (turnaround). It is highly desirable that simulation tools exhibit some variability in this regard.

This thesis details the design, implementation and performance of Loge—a simulation tool which while developed for MOS VLSI is quite adaptable to related technologies and digital systems generally. The core of the design philosophy of Loge is that it support a fast moving investigative style of development, where turnaround receives greater emphasis than fine points of accuracy (good simulators with the opposite emphasis are readily available).

Loge is a mixed-mode, variable speed/accuracy system. The modes implemented form a continuum of abstraction, from a functional level (based on a powerful hardware description language which allows the specification of generic modules), through a choice of hybrid modes, down to a conventional analog device modelling mode. All modes interface cleanly with each other within the limits imposed by different levels of abstraction. Flexibility and responsiveness to user control are featured throughout.

Note

This thesis contains no material which has been accepted for the award of any other degree or diploma in any University and to the best of my knowledge and belief contains no material previously published or written by another person without due reference being given.

I consent to this thesis being made available for photocopying and loan if accepted for the award of the degree.

Michael T. Pope

Acknowledgements

Undoubtedly, this project would never have occurred but for the infectious enthusiasm and stimulating instruction of my supervisor, Kamran Eshraghian. Thank you for all the encouragement, support and for creating an environment full of opportunities to wander through interesting fields of learning.

Integrated Silicon Design deserve mention for appearing just at the right time with an offer of a quick contract programming job... little did either party know that it would be the first step or a longish trek towards a Ph.D. thesis.

Alex Dickinson can not escape special note for some highly disruptive suggestions— while happily trying to cut a path through the jungle with a pair of scissors, Alex would suddenly hand one a chainsaw.[1] Thanks are also due for the architecture used in the functional simulation case study.

It is still a mystery how Mike Liebelt found the patience to deal with an endless stream of awkward requests and foolish questions while keeping the computers running reliably. Many thanks Mike for all the times "The Guru was In".

DSTO Salisbury have been very helpful in providing equipment, comment and financial support— especial thanks to Peter Drewer for the encouragement and wry but apposite remarks.

Other assorted technical help came from people like Ian Dall, Alf Grasso, John and Simon Rockliff, Bruce Tonkin, Richard Wagner, Susan Wu, Greg Zyner and the massed TFB project members. The DCFL flip flop design is due to Derek Abbott.

Many other people deserve appreciation for positive contributions to ones

[1]Sometimes I took a little while to realize how to use it...

working environment. I would like to acknowledge the participants at tea time and Friday night dinners (especially the Vice-President for Business Dinners, and the Whitbreads for regularly turning their house into a restaurant substitute), Amber, J.S.Bach, Communica, the Australian Broadcasting Corporation's classical FM network, Bitey, W.Yendor, and all past, inactive, and active members of the Flat Earth Society.

Finally I would like to thank my extremely patient family, especially my parents, to whom this thesis is dedicated.

Chapter 1

Introduction

1.1 Simulation

Among the attractions of VLSI design over previous technological paradigms is that it gives the designer great freedom. Freedom to directly build interesting new structures and to solve problems with an elegance approaching that of mathematical proofs... but for the inevitable intrusion of technological details and practical limits such as die size. Throughout all engineering disciplines such a gap exists between the theoretical world of the provably correct and the practical world of the flawed implementation. Simulation is a major technique used to bridge this gap in the field of VLSI, and indeed in digital systems generally.

Simulation is useful throughout the entire process of design, in a variety of roles—

Design Implementation For the present, the most intensive simulation effort tends to occur in excitation of models of a system under active development where the model has been *extracted* from an independent implementation representation. Here, the role of the simulator is to

reveal errors in functionality.

Speculative Investigation Alternately, a simulator can be a model building tool— models may be created in some abstract format with no direct connection to an implementation. These models are then be refined and experimented with in a exploratory fashion. This role could be labelled— "simulator as whiteboard". It is the theoretical complement to the practical design implementation role.

Functionality Investigation Given a model, the addition of *instrumentation* to support studies of its internal operation— perhaps to identify bottlenecks, collect performance statistics, or for exhaustive checking of special cases. This role embraces the previous two— the system being simulated could equally be an extracted implementation or a speculative model.

Comparison Many models of the same logical system are possible, either as independent abstract representations or extractions of different implementations. Some may be correct and others faulty— comparison of their behaviour under identical test conditions is likely to be instructive. This is a special case of functionality investigation.

Unsurprisingly, most existing simulators focus on the demands of simulation for design implementation. There is however good reason to desire a wider ranging simulation system, as the following experiences suggest.

1.1.1 The Transform and Filter Brick

The Transform and Filter Brick (or TFB for short) [Eshraghian+85] was an innovative signal processing architecture developed in the mid-eighties by a

team of academics, postgraduates and undergraduates lead by Dr. Kamran Eshraghian at the University of Adelaide, Department of Electrical and Electronic Engineering (the author's principal responsibility in the project was in design and implementation of the memory subsystems).

The TFB architecture is quite complicated, due to the inherent demands of the problems it was designed to solve, which require multiple functional units (ALUs and memory) operating in parallel on a shared but segmented ring bus. Thus TFB encountered the design tradeoffs common to other multiple functional unit systems, such as the VLIW (Very Long Instruction Word) processors [Cohn+89].

From the beginning of the project, it was realized that a serious obstacle to its progress was the conceptual barrier that complexity imposes on a design and implementation group. The research of one TFB team member directly tackles this problem— Alex Dickinson's thesis *Complexity Management and Modelling of VLSI Systems* [Dickinson88]. Amongst the results in this thesis is a discussion of the modelling tool *Pink*, which is derived from an earlier system called *TICTOC*, which was developed in parallel with the development of TFB. It was intended that an extensive overall simulation of the TFB architecture be made— to convince the design team of its correctness, and to provide a more precise semi-formal specification of the system than could be supplied by a simple written report.

Unfortunately, for various resource and manpower-related reasons the overall simulation was never fully realized [Schomburgk84], which was cause for some discouragement. With the benefit of hindsight, it appears that the TFB group erred in placing to much emphasis on implementation at the expense of simulation.

Even in the implementation of subsystems, complexity was a major obstacle. However, here quite strenuous efforts were being made to overcome it.

Some difficulties with the subsystems were imposed by the constraints of the implementation technology. TFB was conceived to be a CMOS single chip processor, which could be collected together in large arrays to attack problems of interest to the signal processing researchers in the group. Given that the TFB architecture (figure 1.1) included four ALUs (including divider), four small memory blocks, an Input and an Output processor, a large program memory, decoder, plus considerable routing and control, the problem of squeezing the required functionality onto the die was an ambitious project for the time. Certainly it was the largest and most complex VLSI design and implementation project attempted by our department, albeit a logical development from participation in a number of multi-project chip designs [Clarke82].

The subsystems of TFB, although mostly simple in structure, were nevertheless sizable objects. The combination of a growth in module size with a fairly static set of software tools and a fixed small amount of computing capability resulted in badly degraded tool performance, sometimes to the point of failure. In particular, one of the most adversely affected tools was an analog simulator [Int87]. This was the main tool used to check the correctness of implemented designs, thus the resulting reduction in the amount of simulation that could practically be performed damaged the credibility of much of the implementation work.

Nevertheless, subsystem implementation proceeded to completion of layout. At this point we could conclude that—

- The subsystems worked in isolation for a small test set.
- No one had noticed anything in the specification of the overall architecture that was obviously wrong.

However we were unable to draw any stronger conclusions, such as, for

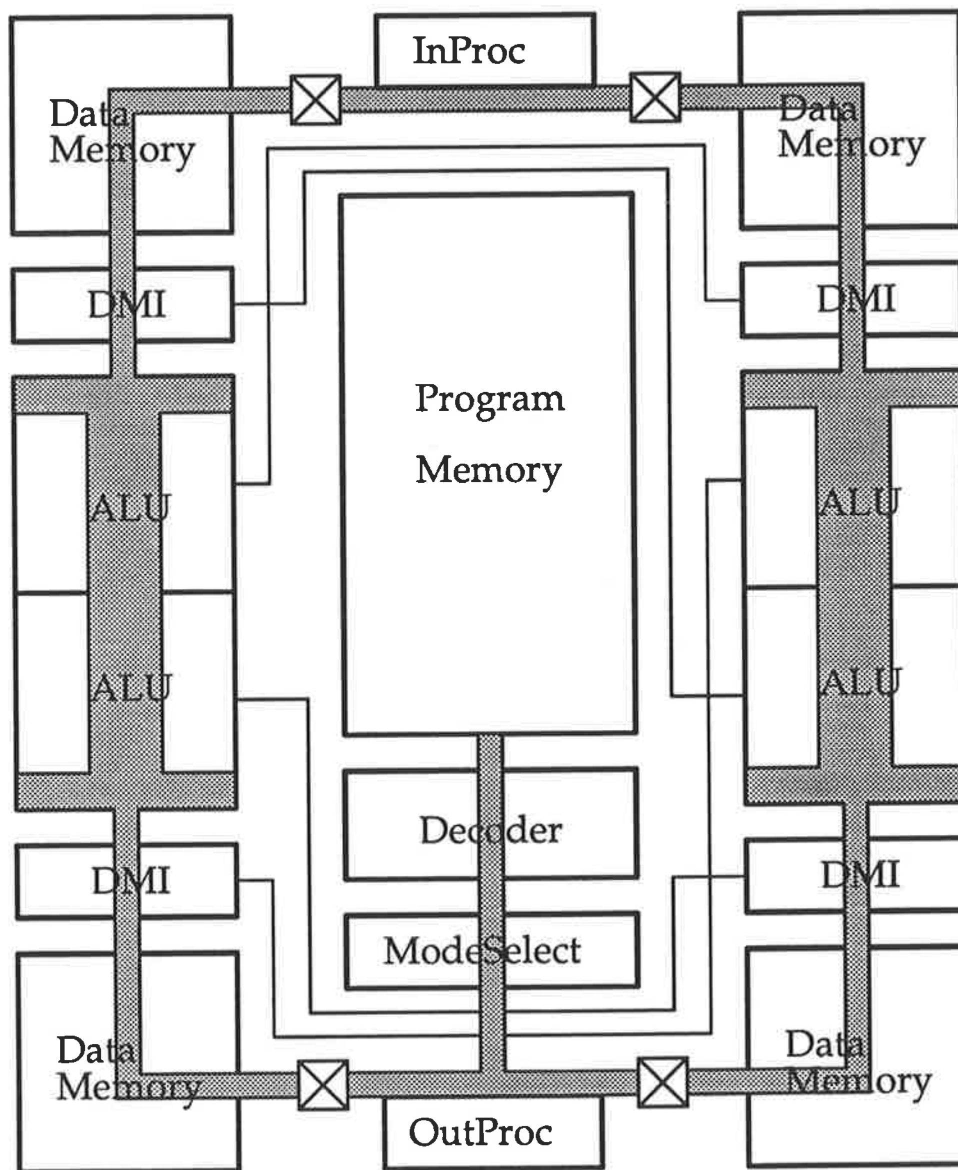


Figure 1.1: TFB Basic Architecture

example— that an ALU would be able to read data from its local memory over the ring bus under microcode control if everything was connected together. Clearly the existing simulation systems were unlikely to be able to reassure us given that they were struggling to handle problems ten times smaller than the situation being contemplated. In particular, it was estimated that the time to create the extracted description of the whole system would exceed the mean time between shutdowns of the computing facilities available to us. Yet it was important to investigate such details as—

- How tolerant was the architecture to control signal skew?— particularly on control lines that lead to the isolating gates between different sections of the bus.
- How accurate were our estimates of the ring bus capacitance? Would a series of iterations of resizing the bus drivers be necessary?
- How accurate were our estimates of subsystem power consumption? — specifically with respect to concern about metal migration and number of supply rail pins necessary.

— for which we had no answers that seemed (at least to the author) to be any better than folklore.

Thus once the task of integrating the subsystems was reached the inadequacy of the simulation strategy taken to that point became very clear. There remained the possibility that a dedicated person could hand-edit a net list, leaving only the interface logic— but this was recognized to be a very time consuming and error prone option, to be avoided if possible. It was perhaps merciful that at this point the TFB project wound down through lack of funding— the last stages would have been both tedious and difficult, due to the earlier compromises.

1.1.2 Communications Link

The lessons of the TFB project were reinforced in an unrelated commercial project, but with a brighter outcome. Briefly, the problem was to devise and implement an interface between a bidirectional two-megabit per second serial data link (connected to microwave hardware) and a bidirectional sixteen bit parallel port. The system would be configured identically at both ends of the link as shown in figure 1.2. The two-megabit link was to be considered unreliable, but no data entering through the parallel ports was to be lost.

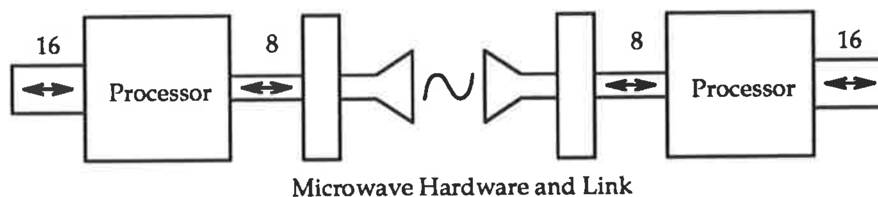


Figure 1.2: Communications Link

Following a fairly conventional path, the system was built from a commodity microprocessor, memory, communications controller and checksum generate/check hardware. The controlling software provided a simple packetizing scheme (a remote relative of the widely implemented Kermit protocol [da Cruz87]). Development proceeded relatively smoothly, but stalled at a point where the system worked most of the time, but would occasionally unexpectedly lose packets.

Happily, simulation saved the day. The errors were too infrequent and ill-defined to be caught in the physical system, so a high level language simulation of the hardware/software system was quickly developed. This allowed arbitrarily flexible tests to be performed, and in a short time a few marginal special case failure conditions were identified and corrected, resulting in

a reliable link, and a conviction that one should have done the simulation sooner.

1.1.3 Lessons and conclusions

The experience of the TFB project heavily emphasized the importance of both adequate initial investigative simulation, and high quality implementational simulation. One noticed a feeling of paralysis when designing with the knowledge that testing would be difficult to impossible.

The communications link project highlighted the advantages a flexible simulation model has over a less malleable physical piece of hardware or VLSI design representation. In particular it brought home—

- The relative ease with which one can instrument a computer model for functionality investigation.
- The usefulness of comparative simulation, particularly in comparing a steadily refined *reference model* with a near complete but fault-ridden implementation.
- Attention to accurate modelling is extremely important— each time one increased the level of detail of simulation, another level of marginal error could be found.

The last item above exposes the great weakness of relying extensively on simulation to find errors and characterize performance of a system— one must trust the simulator and models to some degree. Such trust can never be absolute outside the domain of proofs of correctness, and develops under stimuli of rather mixed quality. Comparative simulation is a good method of improving confidence in a model or group of models, but with the dilemma

that if two models disagree, which is wrong? — is there a mistake in the high level specification of the reference model, or a simple miswiring of the implementation, an error introduced by the extraction process, or are both models wrong?

These concerns notwithstanding, the main conclusion reached from the preceding experiences is that to design and build systems of significant size or novelty it would be most helpful to be able to use a wide ranging simulator capable of acting in all roles listed in section 1.1 (the speculative role in particular is of interest to the author for the purpose of investigating novel computer architectures). Development of such a simulator thus became the goal of this research.

Having decided to develop a simulator, one must consider the question of what it should simulate. Despite the *VLSI Systems* of the title, in practice the scope of the project broadened towards technological independence— the final result is almost technology neutral, making it suitable for the simulation of general purpose digital systems. Similarly while it was initially intended to apply the classic approximation that signals are mainly restricted to the set $\{Hi, Lo\}$ (the *digital model*), in practice this attempted narrowing of scope failed to hold— analog phenomena are quite capable of compromising the digital model, therefore analog quantities are modelled everywhere (although the digital model is more readily apparent to the user). Nevertheless, there is no intention or capability to compete with highly detailed analog simulators (at the level of SPICE [Nagel75]).

Another intentional limitation is that parallel operation was not considered a major design criterion. Interestingly though, several practical considerations arose that resulted in a surprisingly parallelizable design. This is clearly an area for future work. The original limitation was due to practical constraints such as the then relative scarcity of parallel hardware, and be-

cause in a cost-conscious world, uniprocessor systems are likely to persist as hosts for simulations for a considerable period of time.

1.2 Overview

Having introduced the background and motivation for this thesis in this chapter, chapter two continues with a combination of literature review, philosophical discussion and development of more specific design criteria than those given so far. A taxonomy of simulation modes and clarification of general terminology appears. Chapter three is a detailed description of the design, implementation and operation of the general purpose digital simulator— Loge. In chapter four performance is summarized and case studies presented. Conclusions appear in chapter five.

Finally, in what way is this work a novel contribution to the field? Briefly, the notable features of Loge include—

- Clean by design inter-mode interfaces— existing simulators are prone to restrictions introduced by loss of information at mode and/or module boundaries.
- A concise, powerful and extremely flexible hardware description and functional modelling language— specified modules may be *generic*.
- Variable accuracy simulation modes.
- Fully event-driven operation throughout.

— resulting in a high quality digital simulator and architectural modelling tool.

Chapter 2

Literature Review

2.1 Introduction

The overall intention of this chapter is to steadily clarify the requirements for a good digital simulator through a review of the extensive literature on simulation of integrated circuits from the major and more accessible conferences and journals. As important design criteria arise they will be noted in the form—

Design for *(some significant design criterion)* *(0)*

2.2 Taxonomy of Modes

The most logical way of decomposing the field of integrated circuit simulation is by *simulation mode*. Each simulation mode (or level) is charactcrized by the assumptions made in abstraction from reality. There are many modes, forming a continuum of abstraction.

Unfortunately, the names of the modes vary from researcher to researcher. Figure 2.1 clarifies the terminology of this document, defining the main sim-

ulation modes in decreasing order of abstraction. Synonyms and submodes are not shown, but will be discussed under their parent mode.

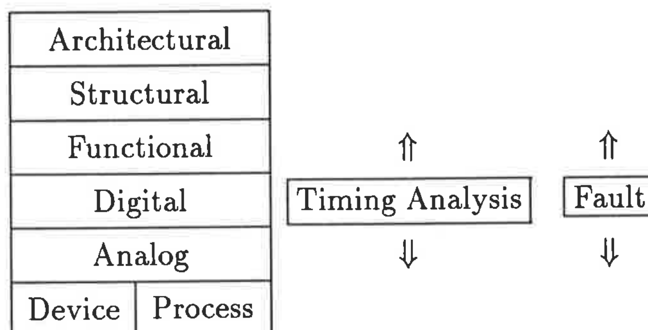


Figure 2.1: Mode Hierarchy

2.2.1 Architectural

Architectural mode simulation is the abstract mental process of design. It is often associated with analysis of requirements, for example "This device is intended to compute inner products— therefore the arithmetic unit/s need addition and multiplication, but not division." In other words, architectural mode is concerned with the broad functionality of the system.

Modelling this type of conceptual process is an open problem in artificial intelligence, and beyond the scope of this research, which is concerned with less abstract modes. However the very fluidity of architectural simulation presents a major challenge to the lower levels in abstraction— which must be highly expressive and variable.

2.2.2 Structural

Structural level is the point of abstraction where a hierarchy of specific *modules* are decided upon, and function assigned to them. Connectivity between modules is also a structural issue. A module corresponds to the classical engineering idea of the *black box*— its function and connections are known, but details of its internals are hidden, or equivalently to use of the technique of *information hiding*. Methodologies for partitioning have been discussed at length in both computer science [Yourdon+75] and circuit design contexts [Mead+80], in particular with respect to the top-down style of design.

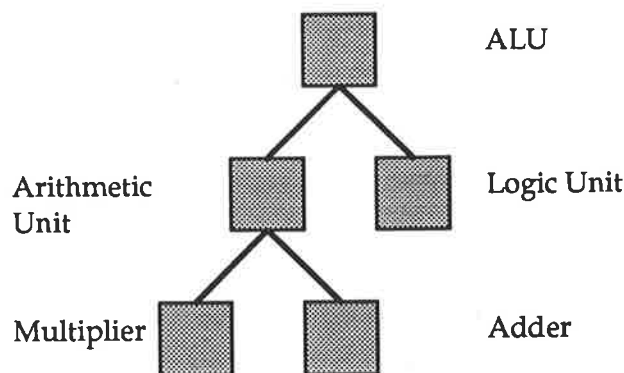


Figure 2.2: Structural Decomposition

Describing a structural decomposition as a simulation mode may appear initially to be rather odd. The justification is this— given that there exist more specific modes, it is possible to simulate a module which contains submodules as the aggregation of the simulations of its submodules in their various modes.

2.2.3 Functional

Functional mode is often known perhaps more descriptively, as behavioural or algorithmic mode. This stage deals with detailed specification of the behaviour of each module.

In developing a multi-mode simulator, functional mode is probably the point at which the greatest "design stress" occurs. On the one hand, it is desirable that the functional description to be simple, elegant, easily comprehendable, and short— attempting to match the architectural abstraction in the designer's mind. On the other hand, it is necessary that the functional mode be able to adequately duplicate all the special case conditions of lower level simulation modes or the technologies that they model. Where the line between these two requirements should be drawn is always likely to vary from design to design— it is to be hoped that both aims are not totally mutually exclusive. The approach eventually taken here is reminiscent of the spirit used in the development of the X Windowing System [Scheifler+86]— to provide functionality but to avoid dictating policy, as summarized by the dictum "Tools not Rules".

It is natural to develop functional descriptions for modules derived from the structural decomposition. Cases where the structural hierarchy (chosen for the purposes of construction) and the functional hierarchy (chosen for the purposes of simulation) are not well matched are rare, and indeed in the field of VLSI such a situation would tend to be viewed as an act of perversity on the part of the designer ([Sussman+80] discusses a system for handling non-isomorphic hierarchies). Because of this close relationship these two modes will be considered together from here on.

The functional level introduces the question of communication between modules. In most cases there is no better solution than to directly model

reality, and provide communication with *nodes*. Every distinct connection between at least two modules defines a node. Each node has an associated value, which may be accessed or modified in various ways by the modules connected to it, depending on the structure and/or function of the module— see figure 2.3. Links between modules and nodes may be directional.

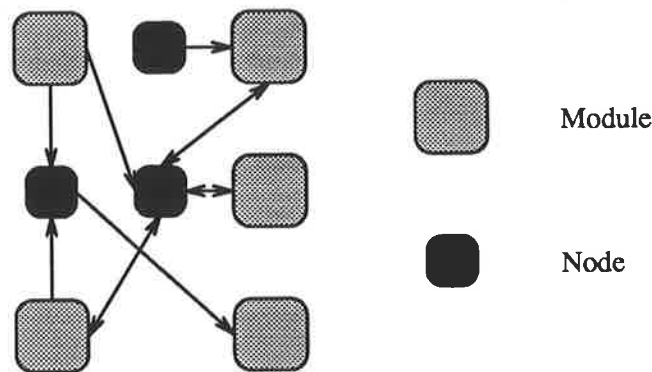


Figure 2.3: Modules and Nodes

Alternately one may think of the nodes as *registers*— which gives rise to a specific submode commonly known as *Register Transfer Level* (RTL). RTL is a subset of functional level as RTL-mode modules typically perform only *Read* and *Write* operations on the nodes. A fully general functional mode may plausibly define arbitrarily many other types of interactions.

2.2.4 Digital

The general digital model makes the characteristic assumption that the nodes take on a highly constrained set of values— "High" and "Low" or *Hi/Lo* or 1/0, in the purest form of the digital abstraction. This is a highly efficient choice for systems whose function is easily expressed in terms of boolean

logic— hence the common but imprecise use of the term *Logic* simulation.

Starting from this basic abstraction, much variation is possible. One of the most common extensions is to allow a third value (often called X) which indicates that the node is in an intermediate state between *Hi* and *Lo*— in support of the threshold voltages defined for logic families such as TTL. A well known submode is *gate-level* simulation, in which the system is dominated by simple logic gates. Another submode is *switch-level* simulation, where decomposition has reached the underlying devices which are simulated as switches.

Ain
Bin

Figure 2.4: Gate Level

2.2.5 Analog

Analog modes abandon the abstraction of the digital model, and deal in continuous quantities. By this stage of modelling accuracy one is dealing with real voltages, currents, resistances and capacitances, rather than an abstract concept of "value". Similarly, the modular decomposition has progressed down to the raw devices. Common terms for this level are circuit and timing

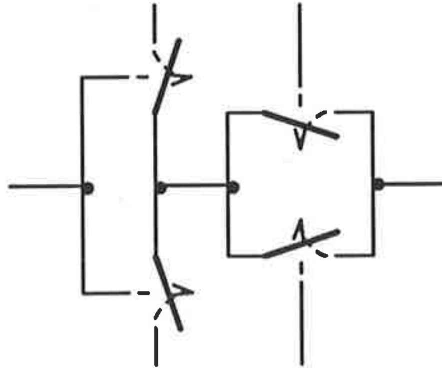


Figure 2.5: Switch Level

mode.

Diversity of features within analog mode is large. Physical devices have many varied and interesting special case modes of behaviour, resulting in a busy and productive field of research. In the literature one senses a drive towards ever greater modelling accuracy competing against a desire for simple computationally efficient models. At the extreme end of the scale, the horizon collapses to the size of a single device—

2.2.6 Device

—leading to the esoteric field of device simulation. Here the engineer and computer scientist give way to the physicist, and consideration of the overall system is lost, therefore device simulation is beyond the scope of this thesis. A related area is *process* simulation, where the input to the simulator is in terms of fabrication equipment parameters, and the output is the device characteristics required by a device simulator.

2.2.7 Timing

Apart from referring to a type of analog simulation, the word *timing* is attached to the mode of simulation best described as Timing (or Delay) Analysis. In this mode, the object is to determine delays along signal carrying paths within the system without recourse to complete knowledge of node values. Given a mode-specific algorithm to determine delay, Timing Analysis can interface to Functional through Analog modes, and is thus orthogonal to the main mode hierarchy.

2.2.8 Fault

Fault simulation is likewise orthogonal. This is another rich field that studies the effects of errors in a system and how to minimize or efficiently test for them. While fault simulation primitives could be added to Loge[1] they have been omitted for the time being, as has detailed consideration of the fault simulation literature.

2.2.9 Mixed and Hybrid

Many simulators are said to be *mixed* mode. The common meaning of this is that they are capable of simulating individual modules in different modes. In some cases, however, the use of the term "mixed" is intended to imply the use of a mode that contains features from two other common modes (for example, a mode that reports only digital values, but which uses an internal analog representation). Such modes are better described as *hybrid*. One must observe that while many excellent individual modes exist, the problem of cleanly interfacing Functional, Digital and Analog modes has rarely if at

[1] A node controllability/observability statistic generator is a future project.

all been achieved with elegance.

Design for *Multiple interchangeable modes* *(1)*

2.3 Functional and Structural Modes

2.3.1 Introduction

It is difficult to say what is the first instance of functional simulation of integrated circuits. Is a subroutine library for a common high level language sufficient? Or the capability to build "macros" of simple AND/OR type gates into larger groupings within a simple gate-level boolean evaluator? Progress in functional simulation has been incremental, but growing steadily in ambitiousness with advances in programming language design and computing capacity. The origin of structural mode simulation is similarly obscure.

Does functional simulation justify its existence? Or, to quote the seminal [Szygenda+73]—

> The past work on functional simulation can be characterized as generally inadequate. Questions, such as: "Can functional simulation accurately reproduce circuit behaviour?", "Is there any savings realized, in time and storage, when using functional simulation?", or "Can we have gate-level and functional descriptions of a module and interchange them during simulation?" have not been answered.

One would hope that nearly two decades later we are in a position to answer these questions. Sadly however, while it is fairly clear that there are real benefits available from functional simulation, and that systems exist that allow on-the-fly interchange of simulation modes, the question of accuracy is still open amongst some commentators— notably amongst critics of VHDL.

Design for *Fidelity of modelling* *(2)*

2.3.2 Hardware Description Languages

The central focus in functional and structural simulation is on the representation language or languages— the *Hardware Description Language* (HDL). An important design choice immediately arises— either to develop a special purpose circuit modelling language (as indeed appears in [Szygenda+73]), or to embed simulation capability into an existing language, as in [Hill+79] where the *base language* was Pascal.

What are the tradeoffs? Special purpose languages allow the developer to retain tight control of language features, they can be carefully optimized for the simulation task and avoid excess syntactic or executional baggage, but must be built from scratch. On the other hand, embedded languages are often easier to develop and write models with, since a rich existing framework is already provided.

Embedded simulation languages are typically implemented with a preprocessor that emits base language code, which is linked with a special purpose library— thus compilation and linking may take significant time, and executable code may be large. Alternately, either type of language may be interpreted— trading speed for flexibility or ease of implementation and extension. Indeed there is a continuum between pure compiled and interpreted approaches— an example of an intermediate case is [Armstrong+81] and [Armstrong84] in which functional behaviour is described in an assembly language (GSPASM), which is compiled into a microcode form, to be interpreted by the simulator.

Some simulators are of such scope as to allow both approaches— an especially obliging system appears in [Doshi+84] which supports both a specific

RTL, and functional models in C, Fortran, Pascal and PL/1! The external language models must of course be compiled, but the RTL is semi-compiled to an interpreted RPN form— giving considerable freedom of choice to its users.

Compiling an entire system is tedious, and makes the addition of new modules rather a chore. In FUNSIM [Des Marias+82] some work is saved by implementing dynamic loading of compiled module object code. The language FML is difficult to categorize as either special purpose or FORTRAN-embedded, as the intent appears to be that native FML constructs be used almost exclusively, despite FORTRAN code being produced as an intermediate step.

Language theorists have commented on the phenomenon whereby language may affect thought processes. A facetious computer-language illustration of which is to recast the proverb "When all you have is a hammer, everything looks like a nail." as "When all you have is FORTRAN, everything looks like an array." The relevance of this to functional simulation is in highlighting a pitfall of embedded languages— does the base language constrain the thinking of the implementor (and thus the functionality of the simulation system), and/or the process of developing functional models? [Pilotny+82] describes the CONLAN project— in which a family of languages are derived from Standard Pascal. The adequacy or otherwise of Standard Pascal for general purpose programming has been a source of controversy— in the case of CONLAN one is left with the impression that something of a struggle with the limitations of Pascal occurred. [Maissel+82] is a similar case, using APL.

Real hardware is inherently parallel, thus for uniprocessor simulation a non-procedural language could be thought to be a natural choice, as for example in [Sakuma+83]. Another example is [Brown+83], for which the functional

specification is rule-based, supporting an exploratory style of development at the expense of performance.

An even more appropriate choice of software technology for functional and structural simulation is object-oriented design [Booch90]. The most important decisions in the use of object-oriented design is the choice of the object partitioning. In the case of functional simulation, a one-to-one correspondence between the module hierarchy and the object partitioning is an easy, natural solution. A fine example of this style is [Lathrop+85] (with some extra objects, such as buses). This functional simulator shows considerable generality, largely derived from the base language (Lisp/Flavors [Weinreb+81]). A notable feature is the use of *versional blocks*— several versions of the same module, which receive the same input, and whose outputs are compared to detect errors or modelling inconsistencies (an approach familiar from fault-tolerance/reliability). [Wolf89] is a similar system where object-oriented Lisp was used in prototyping, as preparation for a C++ production version.

No survey of functional simulation can credibly avoid mention of VHDL [Lipsett+86], [Shahdad86], [Ins88]. This U.S. Department of Defence backed HDL will doubtless become the standard that it is intended to be, which is certainly desirable for reducing fragmentation and isolation amongst the development community. The technical merits of VHDL (as a whole or individual features thereof) have been hotly debated by many experts, however it is revealing that it has been criticized both for being too closely associated with Ada, and also for not including enough Ada features [Nash+86]. Certainly VHDL does not lack features in comparison with other HDL/simulator systems of its vintage [Aylor+86]. One can not help feeling that having gone a long way along the road to Ada that it was regrettable for the design of VHDL that it stops short of full integration therein. Similarly, proponents

of small languages criticize VHDL for creeping featurism, which is at least partially attributable to the influence of the large Ada language.

More detailed discussion of the implications of various base languages can be found in [Katzenelson+86] which compares the use of Pascal, Simula, CLU, and Enhanced C as hosts. Overall it appears that there is pressure to provide an expressive and powerful HDL, and that the more power there exists in a base language, the easier the embedding process and the development of models.

Design for *Flexible, powerful HDL* *(3)*

Having provided the HDL to define the system, there is still a requirement for a means of controlling a simulation of it. This *control language* is often built into the functional HDL, or relegated to a one-off command line interpreter type interface. This is a minor point, however [Terman83] comments positively on the utility of having a powerful language (Lisp) with which to write test programs and directly process results.

As a note of caution, one must not become distracted by the intricacies of HDLs from the fact that they are merely a means to an end. Perhaps the most important design criterion for an HDL is that it be easy to read and write.

Design for *Easy to use HDL* *(4)*

2.3.3 Low Level Modelling

A subsidiary concern of functional simulation is its provision for detailed modelling of low level effects. This issue is sometimes deliberately ignored in published systems— for if a rigid top-down design style is assumed then the functional level has no theoretical reason to model small scale behaviour. Regrettably however, if a higher level mode provides excessively coarse models

with respect to the next level down, verifying the equivalence of two representations at these levels is severely complicated. One must beware of allowing the simulator to dictate methodological policy.

Taking the remarks above to the extreme, it would be desirable for all modes to be closely integrated, allowing faithful comparison of equivalent designs in arbitrary modes. The specific practical upshot of which is— high-level modes like functional mode must interface cleanly to the very lowest-level modes, such as analog mode. This requirement strongly suggests a common communication strategy is required.

Design for *Clean interfacing between all modes* *(5)*

Modules interact by communication through nodes. Nodes driven by an analog mode will contain analog voltage waveforms. How then do functional modes deal with analog waveforms read from their input nodes? As functional modes usually make the digital assumptions, analog inputs are converted to digital levels by thresholding. Well behaved implementations attempt the courtesy of returning non-trivial analog outputs (for example by providing an exponential waveform spanning a change in logic level), but many simply convert logic outputs directly back to analog levels, resulting in a discontinuous signal. Assumption of the digital model for functional simulation has potential implementational benefits such as high level abstraction of module input/output ports as integer-valued variables, reduction in code complexity, and reduction in storage requirements. On the debit side, the ability to model unusual or explicitly analog modules (for example a sense-amplifier such as appears in a well known static RAM design ([Weste+85], page 364), has been lost. Ideally, one would hope that both types of behaviour were available.

Given that some scheme has been devised to convert between time varying analog quantities and discrete digital changes, one must consider the timing

of digital outputs— how long from change of input does it take to produce output? Once again there are various approaches— in FML [Des Marias+82] delays are assignable with constructs such as—

```
WHEN CLOCK RISES MAKE A = A + 1 WITHIN 10 NSEC;
```

Similar notations abound, such as in [Foyster86].

Another approach is to make delay a property of the links between modules (especially in systems that assume top-down design). There is wonderful potential for chaos here if links are allowed to have zero (or negative!) delay. A zero delay link between the terminals of an inverter is the functional simulation equivalent of an infinite loop. In defence of the decision to allow zero delay links, [Lathrop+85] states "It is possible to write an infinite loop in any programming language..." Despite this hazard, in a purely prototyping situation with no catastrophic positive feedback, universal use of non-delaying links is analogous to the situation within common clocked systems where— the clock phase changes, much circuit activity follows, eventually stabilizing before the next clock cycle. In this situation a link is effectively a contract that modules can perform arbitrary amounts of communication and stabilize within single clock cycles. Such omissions may be acceptable in early stages of development to avoid the overhead and trouble of making the clock control explicit. A similar case occurs where all delays are unitary, which implies synchronous communication between neighbouring modules.

Because digital transitions can be said to occur at a specific time instant, they are often referred to as *events* or *messages*. Highly efficient simulation can be achieved within a message-passing paradigm. Unfortunately, there are some serious traps hidden in the semantics of event scheduling and delivery, which are clearly explained in [Luckham+86] from which the following

example is drawn.

Design for *Speed* ⇒ *event driven simulation* *(6)*

Consider a digital inverter, with *Lo*→*Hi* delay of 10ns, *Hi*→*Lo* delay of 14ns. If it receives a pulse of width 3ns, the correct pure digital behaviour would be for the output to remain *Hi* as the *Hi*→*Lo* transition will not complete before the inverter is driven *Hi* again. In event driven terms, two events will be generated from the input events— Output→*Lo* at $t = t_0 + 14$ and Output→*Hi* at $t = t_0+3+10$. With a simple minded event scheduler that merely executes events in time order, these events will become transposed resulting in a erroneous final output state of *Lo*— as shown in figure 2.6.

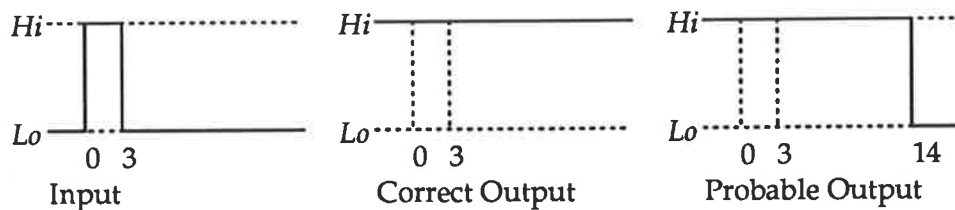


Figure 2.6: Event Scheduling

To prevent events "catching-up" like this, the scheduler must adopt preemptive semantics— when an event for a particular node is scheduled for a time t_i, then if there are other event/s scheduled for that node at time/s t_j where $t_i < t_j$ then the later events are preempted (that is, cancelled). Sadly this approach while safe, may be unnecessarily pessimistic— the cancelled event/s may actually have been quite correct. In particular, preemption constrains functional descriptions to generate events in strict chronological order, which may be an unnatural restriction.

The compromise presented in [Luckham+86] is to qualify the event transition over an interval, for example—

```
when inport=low =>
  outport := high after 10;
when inport=high during 4 =>
  outport := low after 10;
```

—protecting the simulator from generating a transition to *Lo* unless sufficient time has passed that no transition to *Hi* could preempt it.

The solution above is quite sensible, but one wonders whether it was worth the effort. The source of the event catchup problem is the attempt to enforce the strict digital model. If the functional mode was capable of analog output, the events could be directly scheduled in time order, with the result shown in figure 2.7. Thus both an event scheduling problem and an interfacing problem can be simplified by introducing the design criterion—

Design for *Communication with analog voltages* *(7)*

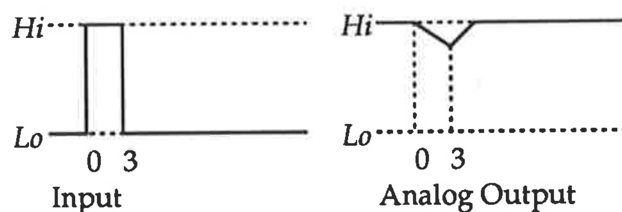


Figure 2.7: Analog Event Scheduling

An impressive approach to the modelling of digital events in time appears in [Heydemann+88]. Here the use of a rigorous mathematical formalism allows high efficiency modelling of modules as state-machines, with achieved performance of the order of 50,000 events/second per MIPS.

2.3.4 Timing Analysis

It is appropriate at this point to consider the subfield of timing analysis. The goal of timing analyzers is to estimate the propagation time of a signal through some network, which in the general case can include diverse elements such as raw devices through logic gates up to behavioural models— covering the spectrum of circuit simulation primitives. Timing analysis is a useful adjunct to analog simulation, particularly as a tool for directing performance optimization, however another important application is in hoisting the abstraction level of a circuit, (especially from out of the analog domain and into the digital)— the analyzer is run to estimate the delays that characterize a low-level network, the results of which are then used in an explicit delay specification at a higher abstraction level (figure 2.8). This methodology is strongly advocated in [Newton81]. Of course, overall performance of a high level system with specified delays is also of interest— an early example is [McWilliams80] which analyzes synchronous clocked systems of logic gates and blocks with assigned delays. Practical experience gleaned from [Elder+84] suggests that good delay estimation tools, applied from the start of the design cycle result in swifter delivery of systems performing to specification.

In general terms, timing analyzers must enumerate paths through the system under test, looking for the critical path between a specified input event and the arrival of its consequent outputs. Clearly, circuits with large branching (and reconnect) complexitics will exhibit swift growth in the number of paths with respect to circuit size, making exhaustive enumeration costly. The review [Hitchcock82] notes this fact, and suggests that the user should guide the search by labelling paths with tags such as "critical", "marginal", "normal" and "trivial". Alternately, [Nomura+82] argues that excessive searching

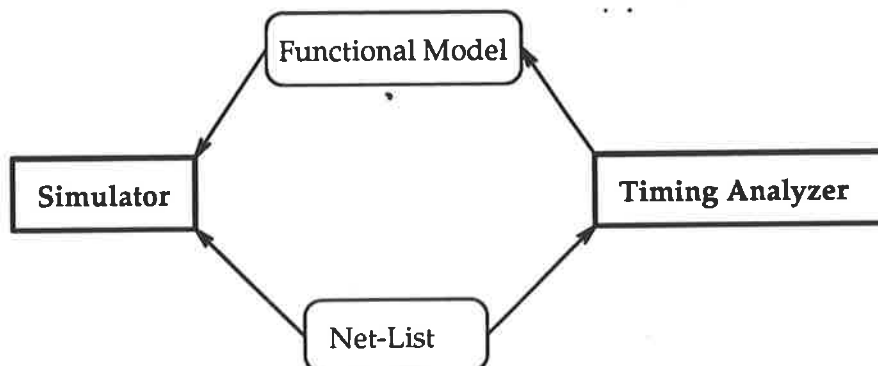


Figure 2.8: The Role of Timing Analysis

can be avoided with reference to the hierarchical structure of the system— for example to avoid checking several instances of bitwise equivalent paths [Jouppi83a].

A practical point made in [Bening+82] is that just finding the one worst path is probably a wasteful approach— a more useful form of output would be the N worst paths in the system, with the ability to mark paths for omission once the user is satisfied that no further improvement can be made on them. Another concern is that there are other statistics of interest apart from just "maximum time to traverse a path"— the average time, and the minimum time are as relevant if one wishes to check *clock skew*— see [Dagenais+86] for an implementation. A potential time saving measure is rather than analyzing for path delays, attempt to prove that minimum clock periods and duty cycles are met ([Cherry88])— allowing early termination on violation.

As usual, feedback configurations are a complication. The immediate practical question is how does one avoid looping when there are cycles between inputs and outputs? The easy detection of cycles has been cited as a reason for using a depth-first search for path enumeration [Agrawal82].

However if the number of cycles is small, depth-first search may be slower then breadth-first search, which in the seminal TV [Jouppi83a], [Jouppi83b] is projected to be linear in the number of devices. This is a controversial point, as another influential timing analyzer, Crystal [Ousterhout83], uses depth-first on the grounds that the paths are short, thus backtracking is nominal and justified by the algorithmic simplifications.

Perhaps as a consequence of breadth-first searching, TV includes much analysis of the circuit with the aim of finding the purpose of each device—see figure 2.9. With such knowledge and the assumption that devices have unidirectional information flow, many invalid paths can be eliminated from consideration. In [Jouppi87], this analysis reaches an impressive accuracy of around 99.9%.

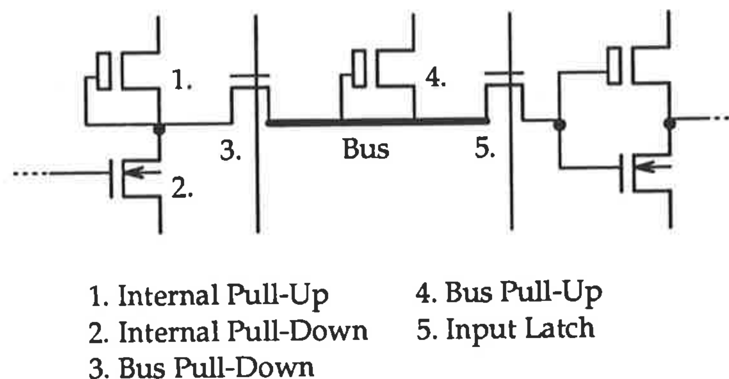


Figure 2.9: TV Device Labelling

The important result in Crystal, is introduction of delay sensitivity to the shape of the input signal and its interaction with loading effects (in detail in [Ousterhout85]). The Crystal model is refined in [Matson85] where the consequences of different sized devices are included, and further in [Hwang+86], and again in [Overhauser+88] which allows for dynamic loads (nodes that are

partially charged, overlapping inputs, etc). The delay analysis methods in Crystal have been used to extract circuit parameters from SPICE waveforms [Rathmell86].

It is well known that many technologies show different delay characteristics for rising and falling signals. For a complete analysis then, a separate accounting must be kept of delays for a rising input and a falling one. Furthermore, given the system under test contains objects such as functional mode modules or gates, the "unateness" (the inversion characteristics) of these objects must be known [Ng+81].

Like other modes, timing analysis has tracked technological development, changing focus from gates to devices— a typical timing analyzer allows user specified block delays, and includes an RC-tree model— for example [Murphy+85]. Accurate models of gates appear in [Okazaki+83], [Etiemble+84] with the specific case of inverters treated in [Sakurai88]. Much work on RC-tree analysis has occurred, flowing from the seminal [Penfield+81] and its refinement [Rubinstein+83], which present useful upper and lower bounds on delay for RC networks. Refinements such as [Chu+87] (for charged shared networks) and [Martin+88] (cyclic networks) continue to appear, but are showing a trend towards accuracy at the expense of computation time. This trend appears likely to be unavoidable in the case of very fast systems— where large buses begin to resemble transmission lines rather than simple nodes or wires. A simple, efficient transmission line model is a difficult problem— see [Canright86].

The explicit split between RC-tree (or general low level element) and high level block models is a blemish that may weaken timing analyzer performance. Attempts to unify these submodes appear in [Brocco+88] and [Wallace+88]— a very general system with no explicit notion of time or delay.

An important theoretical paper is [Lin+84], which details a general and comprehensive model for delays through a system of two-port modules characterized by parameters R (series resistance), C (total capacitance), D (internal delay), Q (stored charge), $D*$ (internal delay due to stored charge), and how this extends to the general RC tree case. Notably, the rules for combining the parameters of characterized modules imply that it is possible to continue to rise up through the structural hierarchy, accumulating delay information.

Thus to summarize, timing analysis is a useful simulation tool applicable to the main primitive elements found in digital systems. Good estimates of average, upper and lower bounds on delay times through a system can be found. These estimates are useful in the development of higher level models, which provides justification for the delay modelling system used in many functional and gate-level simulators.

2.3.5 Goals

Some mention must be made of the goal of automatically synthesizing hardware from a functional description— the point where simulation and compilation meet. Silicon compilation, and general hardware synthesis (for example [McFarland86]) are busy fields in both research and development, and deservedly so. Possibly the most desirable development in VLSI design tools would be a system in which a single high level description can be both reliably compiled to hardware and subject to simulation that adequately reflects the behaviour the compiled version, rather than the more conventional case where the simulator operates on an extraction of the compiled hardware— see figure 2.10. Such systems are considered unlikely to appear for at least another two years according to [Murphy90]. The reverse operation of "decompiling"

a hardware description into a functional description ([Blaauw+89]) appears to be more immediately tractable, especially for simple boolean modules and cases where regularity and/or hierarchy can be detected and exploited.

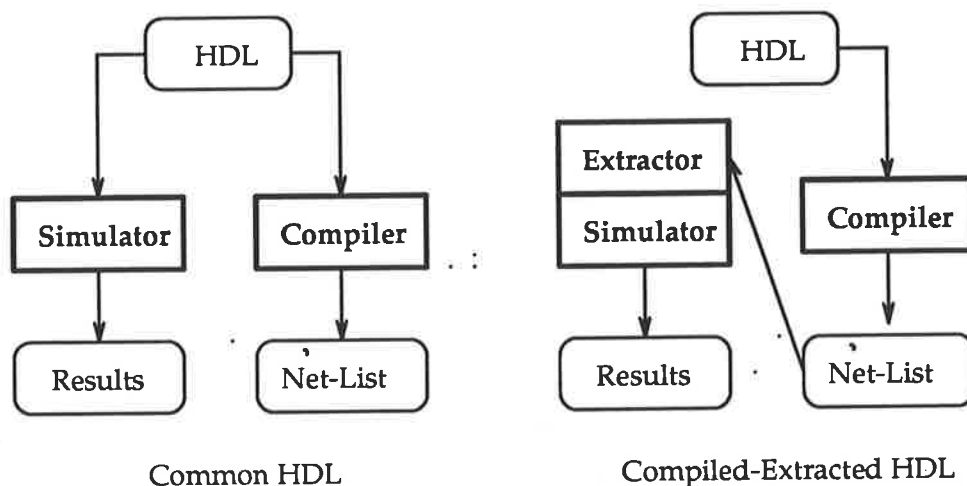


Figure 2.10: Simulation and Compilation

2.4 Digital Modes

2.4.1 Introduction

Anyone attempting to generate random numbers by deterministic means is, of course, living in a state of sin.

— John Von Neumann

Digital simulation of integrated circuits began in earnest at roughly the same time as the introduction of TTL. See [Seshu+62] for a venerable example. From the start, the digital simulation literature has been dominated by

attempts to broaden the digital model to faithfully handle special cases, the most prominent of which are unknown states, delays, bidirectionality and feedback. It appears that adopting the digital assumptions is something of a state of sin similar to the use of pseudo-random numbers.

The development of the field is clearly documented in review papers such as [Bening79], [Ruehli81], [Breuer+81], [d'Abreu85] (gate-level), [Smith86], [Bryant87a] (switch-level), and [Soulé+88] (parallel algorithms). For an example of an early seminal system, see [Szygenda72]. [Breuer+81] observes a notable feature of logic simulators— the steady proliferation of logic *states* and *strengths*.

There is something of a discontinuity between gate and switch level simulators— simulation of gates whose inputs uniquely determine their outputs is inherently simpler than simulation of devices which (being modelled as switches) may either connect or disconnect nodes. The heart of the problem is that the direction of flow of information across a switch is ambiguous.

2.4.2 Logic States

A circuit consisting of logic gates connected such that no two gate's outputs are connected together and with no cyclic connections, can be adequately simulated using the pure digital model of node values being either High or Low. Breaking the non-cyclic connection requirement allows unstable positive feedback as mentioned previously, but more importantly it allows the creation of a simple static memory element— the flip-flop. Once flip-flops are present, the question arises as to what their initial state is— the unsatisfactory nature of assigning either *Hi* or *Lo* to such nodes lead to the introduction of an "Unknown" state— often abbreviated U, or I specifically for unknown initial states (as in [Hirakawa+82]), or X.

Similarly, if outputs of gates may be connected, the possibility exists that two such gates will attempt to drive in opposite directions— creating another use for the unknown state, or perhaps even a distinct *conflict* state. Usually though, these different distinct motivations for separate states are combined, leaving the states *Hi*, *Lo* and X— the *Ternary Logic* model.

Introducing a new state requires consideration of its semantics as an input. In gate-oriented systems this is fairly clear— for example a two input AND gate with an unknown input will have unknown output unless the other input is *Lo*. Nevertheless, [Breuer72] presents a telling example where the semantics of a specific X state is sufficiently different from "either *Hi* or *Lo*" such that erroneous simulation would occur. The same example is still likely to cause recent digital simulators problems. Fortunately this evidence of the theoretical inadequacy of the digital model is not a serious practical difficulty— the problem of unknown initial states can be trivially avoided by assigning all initial states to a specific value (probably requiring more effort from the designer). Alternately, if the system under test is designed sufficiently robustly, X states may be tolerated during simulation, but their continued presence at the end of the test (or at important checkpoints such as the end of a clock period) can be considered as evidence of a design error— this is in fact a useful test.

Thus the X state is a relatively benign addition to a gate-level simulator. Its impact on the simulation of switch-like networks of transistors is rather more severe. Firstly, X is much more likely to be generated since *Hi* and *Lo* node values collide more frequently in transistor networks (often by design). Secondly, the implications of a X on a MOSFET gate node is much less obvious than for logic gates. Thirdly, no simple conflict resolution strategy exists— when an X is connected to a *Hi* or *Lo*, there are two obvious choices, either ignore the X (which is hopelessly optimistic) or allow the X to over-

ride the other value (which has the disasterous property of "polluting" the system with unwanted X states). These difficulties can be resolved only by complicating the digital model further— for example [Flake+83] uses a fifteen state logic algebra with the specific goal of arresting pessimistic propagation of X states.

Further criticism appears in [Bryant84], where the practice of assigning X to unknown initial nodes is deprecated for not obeying the Law of the Excluded Middle to which real systems are subject. In [Stevens+83] a practical attempt to improve this practice is made— the initial system is labelled with two special tagged X values which retain inversion information (that is, if one type of X is the input to an inverter, the other type will be placed on its output)— thus whole strings of X states may be removed if any one of them is identified (this is a simple example of the use of *symbolic simulation* [Bryant90]). Useful though this heuristic may be, Bryant remarks further that rigorous modelling of a valid-but-unknown state is equivalent to the problem of boolean satisfiability— an $\mathcal{NP}$ complete problem [Chang+87]. The $\mathcal{NP}$ completeness property provides a strong motivation for the use of higher level descriptions— see [Chandra+89].

A common idiom in digital design is the use of a bus to allow read/write communication between several modules. Each module with write capability must be able to "disconnect" itself from the bus [2]— or more precisely to set its bus connection to a sufficiently high impedance so as to pass negligible current. With additional control logic it is possible to design well-behaved systems wherein only one module at a time attempts to write to the bus. When simulating such systems it is highly desirable to distinguish cases where two or more modules are contending for the bus from the case where a single

[2] Commonly known as the *tristate* condition.

module is overwriting the data placed on the bus by a previous module which has relinquished its connection. Another issue is that if the bus data is not written for some time it may decay and become unreliable. These concerns lead to the introduction of the Z or High Impedance state, to be used for nodes that are "undriven", as for example in [McDermott82].

Z exists in various forms—

- As a transistor state. An "off" transistor is sometimes labelled Z.
- As a value specifically for decayed undriven nodes.
- In three subforms, *Z0*, *Z1*, *ZX* indicating a node with a particular logic value which is qualified as undriven.

The semantics of a basic Z are similar to its unqualified form X. For the more precise model with *Z[01X]*, these state's semantics are similar to their unqualified forms.[3] In both cases the difference is that driven forms take precedence over undriven forms. One could say that the driven forms have greater *strength.*

Some other states have been proposed— for example [Jea+79] includes U and D for nodes undergoing upward and downward transitions respectively. These have in general not found much favour as they introduce more complexity than they return in improved fidelity of modelling.

2.4.3 Logic Strengths

The previous section on logic states showed how the various Z states were introduced to handle cases where correct operation of a system could be achieved if some states were considered stronger than others. In fact, there

[3]The plain Z is equivalent to ZX

is no real reason to consider Z et al as logic states, but rather to return to the earlier Hi, Lo and X logic state space, with an independent strength attribute.

Strength has a more important use than for high impedance modelling. Returning to the question of the X state, it is clear that generation and propagation of X conditions will be reduced by using multiple strengths—since many logic state conflicts will be resolved by a difference in strength. Notably in the case of nMOS networks, multiple strengths are necessary to have any hope at all of successful simulation.

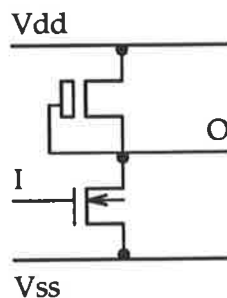


Figure 2.11: nMOS Inverter

For the simple nMOS inverter of figure 2.11, the upper "pull-up" device (depletion mode FET) is always "On", continually driving the output node to Hi. Without a strength model, the lower "pull-down" device (n-type enhancement mode FET) will be unable to change the output to anything but X. If however the depletion mode device is considered to propagate a "weaker" Hi than the Lo propagated by the enhancement mode device, then the inverter can be adequately modelled.

The example of the nMOS inverter illustrates the physical significance of the strength abstraction. The strength of a logic state is a measure of

the current driving capacity supporting the logic value of the node. An undriven bus has no supporting current maintaining its state, therefore it has the minimum strength. An externally supplied voltage rail has a probably unknown but presumably very large supporting current capacity, giving it a very high strength.

Node strength then is either an implicit property of a node (in the case of voltage rails, and perhaps other external connections), or derived from the device/s connected to the node. Indeed it appears that devices too have strength. Some quantitative modelling of device strength is necessary to clarify situations where several devices are competing— supposing the nMOS inverter had another pull-up in parallel with its existing one— are their combined strengths sufficient to overcome the pull-down? This leads to the complication of different device sizings— in the case of MOSFETs identical parallel devices are logically equivalent to a single device with gate region width to length ratio equal to the sum of the ratios of the original devices.[4] In systems using *ratioed logic* there will be multiple device strengths present.

The difficult question about strengths that it asked of every digital simulator designer is— "How many strengths is enough for accurate modelling?". [Sangster+83] uses three, [Int87] four, [Stevens+83] five, [Hodgson84] seven, [Adler86] thirty-two. Axiomatically, given $N + 1$ types of devices, each with different driving capacities, it is possible to construct a simple network that will defy accurate simulation by a simulator which can model up to N strengths. Here we have a theoretical objection to digital simulation with non-trivial practical implications. The simulator designer must make a choice for N, which can never be guaranteed to be completely adequate,

[4]This is only approximately true, but adequately so for most purposes under digital simulation.

and which may even result in inefficient simulation if the bulk of simulated systems require significantly fewer distinct strengths.

Bryant estimates that for most circuits a mere two strengths are adequate [Bryant81b]. Nevertheless, the presence of ratioed logic is a justification for requiring arbitrary many devices of different current driving capacity. Another supporting complication is the property of some devices to propagate different logic states with different strength— for example an n-type enhancement MOSFET passes *Lo* well, but *Hi* in an attenuated form. Certainly, the general trend in reported simulator implementations has been for increasing numbers of strengths.

Little mention has been made so far of any systematic procedure for propagating logic states and strengths throughout a system. This was in deliberate attempt to evade consideration of certain problems, of which bidirectionality is the most dire. To delay this reckoning a little longer, assume that all devices have an *unambiguous direction of information flow*, or equivalently, that the sign of the difference in strength of the nodes on opposite sides of a switch can be determined by static analysis. As an example consider a simple CMOS Dynamic Shift Register Cell (figure 2.12), in which the devices can be unambiguously labelled as transferring information in one direction, (and often with an easily computable range of (state, strength) pairs).

This circuit could be trivially simulated by analyzing each device's gate input to determine whether it is conducting or not, and if conducting then propagating the source state to the drain state. There is however some ambiguity in strength propagation— does a device propagate the strength of the source node or its own strength or some combination thereof?

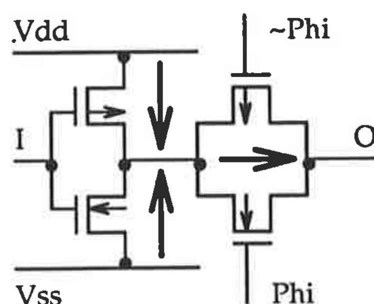


Figure 2.12: CMOS Dynamic Shift Register Cell

2.4.4 Digital Simulation Algebras

Several researchers have proposed algebras for modelling switch and gate level systems. [Brzozowski+79] is a general mathematical treatment of ternary logic, while [Jea+79] formulates the problem in terms of allowable state transitions. Hayes has been particularly influential in this area— with connector/switch/attenuator networks in [Hayes82], a detailed treatment of ternary logic with strengths in [Hayes86a], and with greater refinement in [Hayes86b] which allows for discretized node voltages and bidirectionality. General mathematical equivalence between various modelling techniques is shown in [Barros+83].

Particularly important papers in this field are those by Bryant, in which switch-level simulation was first given definitive form (with the simulator MOSSIM and its descendants). Bryant uses Hasse diagrams (figure 2.13) to define logic state and strength algebras [Bryant81a], [Bryant81b], [Bryant84]. The diagrams define a precedence of states and strengths, with collisions resolved using a least upper bound operation— for example nodes with same strength and different state generate X at the same strength, but where the strength differs, the corresponding state prevails.

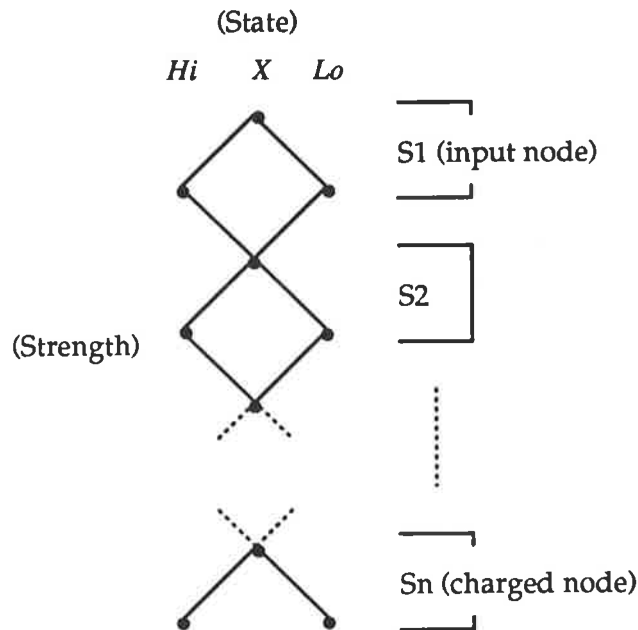


Figure 2.13: Hasse diagram of an arbitrary strength logic algebra

2.5 Bidirectionality

One of Bryant's complaints about digital simulators is that they often artificially assign a direction of information flow to fundamentally bidirectional devices— see for example [Sherwood81]. Modelling bidirectionality has always caused trouble to simulators, and has given rise to complex special case solutions that are often inefficient. One of the better alternatives is seen in [Holt$^+$81] which models bidirectional elements as two oppositely directed unidirectional elements— the resulting overhead is considered acceptable as it is asserted that true bidirectionality is rare in real systems.

MOSSIM is claimed to be more natural— devices are modelled purely as inherently bidirectional switches, which is a good match to MOS technology. Internally MOSSIM determines the strongest *rooted path* to each node— the

path through the network of switches and nodes with the greatest strength according to the strength algebra. No special case consideration of bidirectional effects is needed— however bidirectional configurations (DC connected devices/devices connected together source to drain) may require a number relaxation steps to stabilize.

MOSSIM was an impressive system, lending weight to the arguments of its supporting papers. Nevertheless, some subsequent work has continued to use unidirectional devices. An obvious reason is efficiency. Examples include [Sangster+83] (which has an explicit model for a common digital idiom— the "wired" bus), and [Almeida+84] which uses an eight-valued algebra in bidirectional areas which reduces to a five-valued case elsewhere, with bidirectional devices modelled as cross-coupled AND gates.

Similar to MOSSIM is [Stevens+83], where on an input event, a wavefront of change is propagated through the system, with wavefronts combined according to the least upper-bound algebra. Another MOSSIM derivative is [Adler86] which extends MOSSIM to handle stages driven by gate-level primitives, adds an RC delay model (requiring an event scheduler), and uses a thirty-two strength algebra.

[Dumlugöl+83] presents a system that takes the common step of partitioning the system into its DC groups. A strong $O(n)$ relaxation algorithm is applied to produce a steady state configuration free of X states. This system is refined in [Dumlugöl+87] to perform the partitioning dynamically at boundaries defined by non-conducting devices. Special handling of feedback loops is required.

When considering the difficulty caused by bidirectionality and feedback, the question arises— "Are they worth supporting?" In [Ramachandran83] only *well designed* systems are considered— for which the control graphs must be acyclic in sections delimited by clocks (allowing explicit ordering of

the devices), and race conditions may not occur at any of the nodes (that is, data values may not vary with delay). In return for adopting this design discipline, the designer is promised a faster simulator with the ability to reliably detect hazards.

2.5.1 Delay Modelling

Digital mode shares many of the delay modelling concerns of functional mode, (mainly as both share the digital assumptions, but also due to the correspondence between a gate and a functional module). Delay models have often been added as an afterthought to existing simulators, with all the problems this approach implies.

Many delay systems have been proposed— [Jea+79] provides a range of delay services— zero or unit delay, rise/fall, min/max, min/max rise/fall (for instance, rise and fall times have distinct minima and maxima). A more electrically based, but thorough, case is [Hirakawa+82] in which the delay model includes handling of fanout.

[Bryant83] is a system for race detection. It makes the point that analog simulation does not show that a circuit will work correctly independent of delays. Delay sensitivity can be proven with a two-phase ternary logic simulation— in phase one all *Hi* or *Lo* to *X* transitions are performed, then in phase two all *X* to *Hi* or *Lo* transitions. If any *X* states remain a race is present.

2.5.2 Other Features

In the absence of connecting themes, this discussion of digital simulators continues with an arbitrary collection of useful features. The list is by no means exhaustive. A miscellaneous paper is [Miyoshi+85], which details sev-

eral methods for speeding up gate-level simulators— such as merging gates, ignoring gates that can not change (equivalent to the constant propagation stage in an optimizing compiler), and using zero delay elements wherever it is safe to do so.

Node discharge Nodes in MOS systems have the property of slowly discharging through the substrate. This effect is provided in [Sherwood81] via a "node time-out" event.

Latency [Leinwand81] takes a high view of a system, considering localized "processes"— whereby events in different modules are isolated from each other (each module has its own event list and clock). This has the desirable properties of increasing locality of reference, reducing event queue operation overhead, and making the simulator more amenable to multiprocessor operation.

Demand driven An alternate method of avoiding doing unnecessary work is given in [Smith+87], where requests for signal values are propagated backwards through the system— avoiding evaluating gates whose outputs are ignored. [Subramanian+90] is a recent system of this type adapted for multiprocessor operation.

Flexibility It is undeniably desirable for there to be available some ability to vary the severity of modelling. [Hodgson84] for example allows several stages of algebraic complexity up to six-state seven-strength.

Technological applicability In a rare departure from the dominance of MOS, [d'Abreu+84] is capable of handling bipolar devices.

Analog compatibility In section 2.3.3 the event catchup problem was presented, with the simple solution of using analog waveforms. Diverse

systems such as [Jea+79] and [Schaefer85] use analog ramps to model transitions.

Incremental operation First generation algorithms would usually revalidate their state at each iteration. Second generation algorithms preserve a modicum of state, and only propagate changes thereto, gaining impressive speed improvements. An incremental algorithm appears in [Bryant84], and a variant in [Adler88]. Similarly, the system of [Dumlugöl+83] is reworked for incremental operation in [Sundblad+87]. A continuation of the trend to trade state space for speed is the use of *waveform relaxation* techniques, as in [Dumlugöl+87]. In for example [Salz+89], incremental changes of the structure of the circuit are treated similarly, which yields up to three orders of magnitude reduction in total simulation during that phase of design where a circuit is tested and modified until correct.

Low Overhead [Appel88] exploits hierarchy to reduce the amount of state per node down to one bit. This is an impressive achievement, to which one can only jealously comment that perhaps other parts of the design suffered as a result. Obviously this feature is in conflict with those in the previous paragraph, and with the backward propagation in [Smith+87].

Compilation Recently, several systems have appeared that statically analyze switch networks and compile them into an efficient form that executes typically orders of magnitudes faster than previous simulators. The literature shows a steady increase in the amount of work done in compilation— from the early [Cerny+85], through [Wang+87] and [Bryant+87b] (in which the preprocessor emits C code, the compi-

lation of which takes 70% of the total translation time) to [Hansen88] and [Choi+88]. Compilation to a high level language is a common approach, although [Barzilai+88] compiles direct to assembly language (doing a coarse granularity MOSSIM II-style evaluation). Incremental capability is added in [Beatty+88], and hierarchical incremental (a threaded-code implementation) in [Lewis89].

Parallelism Digital simulation tasks are reasonably amenable to parallel implementation— as shown by the results of [Frank86] where simulations of a 64-processor dataflow machine for switch-level simulation showed a speedup of 16 to 24. In [Bryant88], multiple parallel simulations with varying data are performed, either by exploiting bit level parallelism of logic operations on uniprocessor machines, or using fine-grained parallel hardware— yielding overall improvements in the range 20–30 and then another 2–3 orders of magnitude respectively. See also [Kravitz+89].

Partial Ordering The technique of ordering strengths has been very successful, but there are pathological circuits where it is difficult to assign strengths automatically— user intervention is needed to resolve ambiguity. Another problem with strength ordering is that large numbers of distinct strengths can adversely impact simulator performance. In [Agrawal+88] these problems are tackled by only requiring *partial ordering* of sets of strengths, and by minimizing the total number of strengths required to correctly simulate a particular circuit.

In [Chamberlain+86] [Wong+86] the authors study the behaviour of a switch-level simulator which has been instrumented to allow data to be collected about its performance and notably the distribution of work amongst

various simulation subtasks— the results here are rather surprising— it appears that for their system, the event-queuing, functional evaluations, net-list operations and other task are quite well balanced with no one task predominating in general.

So, to summarize, digital simulation is in wide use and is unquestionably a useful technique despite a number of pitfalls and theoretical inadequacies.

2.6 Analog Modes

2.6.1 Introduction

Pure analog simulation is the most honest and mathematical of the modes seriously considered for Loge. It is concerned in the main with real analog quantities, avoiding the quirks of the digital model. Much research in analog simulation is driven by the ambition to model lower level effects, which steadily leads into detailed models of specific device types (see reviews such as [Engl+83], [Sheu+88]) and finally into device simulation itself. This is obviously a worthy aim, but the price is heavy— progressively decreased simulation turnaround for the designer, and increased technology dependence. It is a conscious decision that Loge should not take this approach— the engineering tradition is to avoid full mathematical rigour where it is found not to be cost-effective. Additionally, much analog simulation research is sufficiently advanced so as to require its practitioners to be expert numerical analysts.

2.6.2 Analog Circuit Equation Formulation

Before discussing the literature, this section presents a formulation of the classic analog network equations derived from the comprehensive review in

[Newton+84]. The terminology herein will be followed in later sections even where it conflicts with that presented in the papers being discussed.

Firstly, some simplifying assumptions—

- There exists a reference node, of constant voltage and measurable capacitance to all other nodes.
- All circuit elements can be modelled as current sources, where the current is a function of the node voltages ($I = f(\bar{V})$).
- Inputs to the system are in the form of an applied voltage with respect to the reference node.

Applying Kirchhoffs Current Law to a system of N nodes yields—

$$G(V(t))V(t) + C(V(t))\frac{dV(t)}{dt} + \sum I(V(t)) = 0 \tag{2.1}$$

—where $V(t)$ is the vector of node voltages at time t, $G(V(t))$ an $N \times N$ matrix of internode conductances, $C(V(t))$ an $N \times N$ matrix of internode capacitances and $\sum I$ is a vector dependent on the node voltages which contains the sum of the currents charging each node at time t. (Note that the diagonal terms of the capacitance matrix represent capacitance from a node to the reference node.)

Some obvious simplifications suggest themselves— it is highly desirable to minimize the effect of the dependence of G and C on $V(t)$. Therefore these matrices may be split into constant and varying parts—

$$C(V(t)) = C_{const} + C_{vary}(V(t)) \tag{2.2}$$

— or a further restriction placed on acceptable circuits, such that these time varying elements are disallowed. The practical result of this restriction is

often a significant reduction in simulator code complexity, and thus improved performance.

A second simplification is to disallow "floating" elements— all internode conductances and capacitors connected between nodes other than the reference node. Obviously this seriously restricts the types of circuit topologies that can be simulated, however for VLSI systems where deliberate placement of such capacitors is very rare (legitimate cases would be bootstrapping circuits, attempts to model device inter-terminal parasitics and bus crosstalk) the simplification is more justifiable. Similarly, internode conductances are not usually present by design as circuit elements, but are usually included as part of the device modelling process— thus the choice is sometimes made to encapsulate the effect of G within the calculation of $I(V(t))$. These reductions result in a null G matrix and diagonal C matrix, allowing the equations to be rewritten such that—

$$\forall i \ C_i \frac{dV_i}{dt} + \sum I_i(\bar{V}) = 0 \tag{2.3}$$

Thus it can be seen that the critical issues in analog simulation are—

- The choice of approximation for dV_i/dt (a numerical analysis problem).
- Finding models for the individual components of $I(V(t))$ (a device modelling problem).
- Efficiently solving the overall system of equations (a mathematical and computer science problem).

The next step is to discretize the system of equations in time, such that analysis begins at a time point $t_0 = 0$, and proceeds in discrete steps from t_i to t_{i+1}. At each time point, previous information is used to predict a new solution (thus it is assumed that $\bar{V}(0)$ is known). Several techniques

have been used to perform the discretization— the Backward Euler and the Trapezoidal Rule integration methods are common examples. The intention is to reduce the system to the form—

$$\bar{V}(t_{n+1}) = f(\bar{V}(t_n), \bar{V}(t_{n-1}), \ldots) \tag{2.4}$$

—or alternately as far as—

$$g(\bar{V}(t)) = 0 \tag{2.5}$$

— (being a system of nonlinear, algebraic equations).

For example, if all simplifying assumptions are made such that the capacitance matrix is diagonal, and the most direct of integration formulas applied (Forward or Explicit Euler—

$$\frac{dx(t_n)}{dt} = \frac{x(t_{n+1}) - x(t_n)}{t_{n+1} - t_n} \tag{2.6}$$

) then the system reduces to—

$$V_i(t_{n+1}) = V_i(t_n) + \frac{1}{C_i} \sum_{j,j \neq i} I_{j,i}(\bar{V}(t_n)) \Delta t_n \tag{2.7}$$

— for each node i. For most nodes the $I_{j,i}(\bar{V}(t_n))$ terms are almost all zero, except where node j is connected to node i through a device.

At this point it is appropriate to return to the literature, where the variations on this overall theme will be cited.

2.6.3 SPICE

High accuracy analog simulation in VLSI is nearly synonymous with SPICE [Nagel75]. Since its development in the early seventies, SPICE has become accepted as the standard by which the accuracy of other analog simulators tends to be judged. Similarly, it is a popular choice for performance comparison, perhaps partially due to the fact that SPICE is guaranteed to be slower than newer systems!

The relatively slow response of SPICE is inevitable on the following grounds—

- SPICE is an extremely general tool, containing models for all manner of analog componentry, from resistors to transformers to junction field-effect transistors. Thus its modelling techniques are subject to a lowest common denominator effect. Additionally, SPICE performs other forms of analysis than the time domain transient analysis beloved of the integrated circuit designer.

- SPICE is now an old program, which although it has been carefully revised and improved, inevitably reflects design decisions made when integrated circuit device counts were of the order of hundreds rather than millions. There is no easy solution to this problem— SPICE is a captive of the bugbear of software development— Backward Compatibility.

How then does SPICE and its descendants work? [White+87] describes a four step process (which follows the general formulation presented in section 2.6.2)—

i) An extended form of the nodal-analysis technique to construct a system of the differential equations from the circuit topology.

ii) Stiffly stable implicit integration methods, like the backward-difference formulas, to convert the differential equations which describe the system into a sequence of nonlinear algebraic equations.

iii) Modified Newton methods to solve the algebraic equations by solving a sequence of linear problems.

iv) Sparse Gaussian elimination to solve the system of linear equations generated by the Newton method.

— and continues by making the observation that for systems containing n devices, the process quickly becomes limited by the speed of the matrix solution, which is of $O(n^a)$, with a observed to be generally in the range $1.2 < a < 1.4$, depending on the individual circuit and quality of the sparse matrix routine implementation.

This description exposes another reason for the success of SPICE— it uses conservative but robust techniques, which have undoubtedly lead to it being trusted by its users.[5] Another revealing comment from [White+87] is—

> ...at one major IC house SPICE is run more than 10,000 times per month. At another major IC house, more than 70% of an IBM 3090 is devoted to circuit simulation. Circuits containing as many as 10,000 active devices have been simulated with circuit simulators. For some of these circuits, running times on the order of one hour of IBM 3081 CPU *per time point* (!) have been reported.

With such a demand, it is not surprising that considerable research and commercial effort has been expended on the development of more computationally efficient analog simulators. Unfortunately, for some time the pure complexity of integrated circuits has outstripped the ability of the best analog simulators to cope with them, even though simulators benefit from being run on ever faster machines built of the very components they struggled to model in the previous generation.

[5]The other main force behind the widespread adoption of SPICE is that it costs almost nothing to educational institutions.

2.7 Speeding up SPICE

The easiest way to build a faster SPICE is to abandon its generality and tune performance for a particular technology. This has been done very often for MOS technology, especially as MOS systems have two highly desirable properties—

- MOSFETs have minimal coupling between their gate node and both source and drain nodes— $I_{gs} \approx I_{gd} \approx 0$.
- Partially as a consequence of the above, except in exceptional circumstances the capacitance between a circuit node and the ground node is much greater than any other of its internode capacitances ($C_{i,gnd} \gg C_{i,j \neq gnd}$) (figure 2.14).

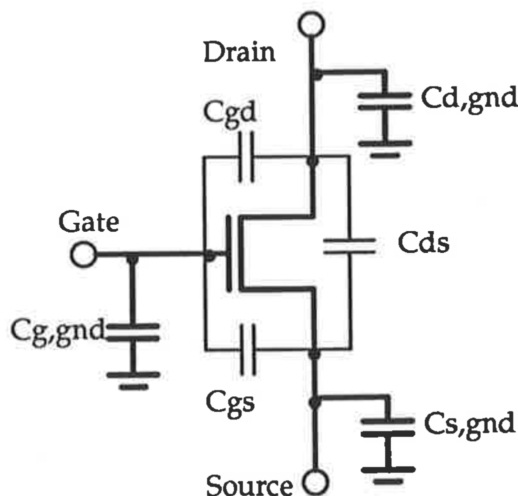


Figure 2.14: MOSFET Capacitances

The use of the ground node as reference node is so obvious and widespread as to need no comment. The capacitance property has the useful consequence

that the general capacitance matrix $C(V(t))$ is almost certainly diagonally dominant, which mercifully implies the existence of a bounded inverse.

The first optimized MOS analog simulator was the influential MOTIS [Chawla+75]. The above properties allow efficient use of *relaxation methods*, which resulted in a dramatic increase in the speed accuracy product of MOTIS with respect to its contemporaries— essentially by substituting SPICE's direct attack on the system of equations with an iterative solution. In recognition of the performance boost, MOTIS was designated a "timing" simulator, as it now allowed detailed verification of the time domain properties of a system previously unavailable under gate-level simulators, and allowed swifter analysis of large systems than provided by the direct techniques used in "circuit" simulators like SPICE.

MOTIS uses a Backward Euler method for discretization (

$$\frac{dx(t_n)}{dt} = \frac{x(t_n) - x(t_{n-1})}{t_n - t_{n-1}} \tag{2.8}$$

). Backward Euler is perhaps the natural choice for this application, as—

- It is A-stable rather than merely convergent like Forward Euler.
- It is quite simple to implement.
- It requires little overhead per node— only $V(t_n)$ and $V(t_{n-1})$ need be stored, while other methods tend to need more "history". (Admittedly Forward Euler requires only $V(t_n)$).

Certainly Backward Euler is very commonly used in other simulators.

Another useful technique in MOTIS was the use of look-up tables to contain the analog device models— this optimization is still highly useful on hardware where the arithmetic performance is relatively weak. The obvious disadvantage of look-up tables is that their memory requirements grow with

the power of the number of variables as accuracy is increased— thus simple MOSFET tables which typically model $I_{ds} = f(V_{gs}, V_{ds})$ exhibit square law growth.

In [Agrawal+80] the early MOTIS became part of a mixed-mode simulator. The name lives on in later systems— [Lo+83], [Chen+84], [Tsao+85]. A related simulator, EMU (from the Mulga design suite [Ackland+81]) has served as the basis of work into application of parallel techniques to analog simulation [Ackland+85], and in [Ackland+86] where the circuit is partitioned onto a multiprocessor with one coroutine per drain-source connected region.

Having presented time discretized equations, one must consider the question of the choice of time points. If the discretization method is convergent then arbitrarily accurate solutions may be obtained by making the difference between successive t_n small enough. However the smaller the time step, the greater the amount of computation to simulate a desired interval T_{sim}. Thus the problem is to choose the t_n as large as possible while retaining an acceptable bound on accuracy. This is the reason why A-stable integration methods are particularly desirable— A-stability guarantees that the error due to discretization decays independently of the size of the time step (even in stiff systems of equations such as are typically present in MOS circuit formulations [Sakallah+85b]), whereas a non-A-stable method must place an upper bound on the time step or the results will become unstable.

Given that an A-stable method is used, the time step can be chosen with a granularity that closely matches the activity within the circuit. Especially in clocked MOS systems, the amount of switching within the system rises dramatically at a clock transition, then decays to a steady state, until the next transition. By simple heuristics that detect the amount of current flowing in the system the time step can be dynamically altered such that during switching transients a high degree of precision is maintained but as the sys-

tem quietens down the time step is extended, avoiding much unnecessary computation. This is a major optimization, an early case of which appears in [Rabbat+79].

2.7.1 A Philosophical Digression

Research in analog simulation is characterized by mathematical rigour and a conservative attitude towards simplifications that may reduce accuracy. This tendency always to emphasize accuracy ([Vogel85]) is a design choice from which the author must beg to differ— just as the digital model assumptions appear to be too generous on the one hand, on the other much work on analog simulation appears too precise given the great desirability of fast turnaround. In support of this doubly heretical position, here are a few philosophical remarks...

In [Newton+84], Newton makes the fair comment—

> "A circuit designer soon loses confidence in a program that occasionally gives an incorrect answer!"

However in reply, I contend that there exist many such programs in common use, which continue to be used despite being known to contain faults. A trivial example is almost any compiler for a moderately complex high level programming language— take for example a compiler descended from the Portable C Compiler (pcc) [Johnson+78]. Pcc based compilers now boast more than ten years of evolution involving widespread access to the source code and an enormous user community— and a correspondingly steady yield of newly discovered errors. One might argue that pcc was only tolerated in the absence of readily available yet less error prone competition, but this fails to explain the acceptance recently achieved by the GNU C Compiler [Stallman90]. Gcc has been widely adopted by users and manufacturers as

a replacement for pcc based compilers, for although gcc demonstrably contains more errors, it produces faster object code. A more directly simulation relevant example can be found in any analog simulator which allows control of the default time step— giving coarse control over the speed/accuracy tradeoff, with the result that excessive speed is often chosen despite the less accurate results.

(*Aside:* One might perhaps argue that simulators should not allow their users to specify unstable configurations. Unfortunately this requirement may not be efficiently realizable as instability is a complex phenomenon, depending heavily on system under test, simulation algorithms used, input data, and so on. More practicable, and desirable from the users point of view, is that such instability induced errors be detected and some action taken— such as printing a warning. This issue is an instance of a very common dilemma in applications programming design— should the program "hold hands" and prevent the presumably naive user from making mistakes, or should it assume that the user knows best and do exactly as it is told? Often, a combined approach is taken, by which programs have both "Novice" and "Expert" modes. One must add that human users are much more flexible than computer programs— this mismatch suggests that writing unnecessarily restrictive programs is ill-advised.)

User confidence then is not necessarily lost through the mere presence of errors, regrettable though they are. Consider the following classes of errors—

Undetected Error: Cases where the user is presented with incorrect results that are consistent with expectations and which are then trusted. The error may be discovered at some later time, at which point it becomes one of the following—

Unexplained Error: An error is detected but the cause is not clear. If,

subsequently the cause is discovered, the error is now a—

Clear Error: An error is detected, and its cause is evident.

I believe that confidence is most seriously shaken by the discovery of a previously undetected error, particularly as these may potentially undermine a large body of accumulated results. Unexplained errors are much less unpleasant by comparison, and may often degenerate to clear errors. In contrast, users are reasonably tolerant of clear errors— provided there is some means to correct them. To clarify further, here are some simulation related examples of these types of error—

Undetected: A functional simulator that occasionally loses an event (for example see section 2.3.3)— perhaps concealing potentially serious glitches.

Unexplained: A digital simulator which is unable to model circuits that depend on charge sharing effects.

Clear: An analog simulator which produces wildly unstable output waveforms if the user increases the default time step too far.

—obviously the undetected error is unacceptable, the clear error is regrettable but simple to recover from, and the unexplained error while initially serious may be converted to a clear error once the user is aware of the phenomenon.

Now if the above three simulators were separate products available to a group of designers, and assuming the turnaround available was in the normal inverse order of accuracy, one would predict that after a period of familiarization the functional simulator would be used rarely if at all, the digital simulator would be used most, with the analog simulator in use only where

the digital mode was inadequate, and then often driven very close to the point of instability for most of the simulations it performs. In other words, experienced designers will search for the fastest turnaround for which errors are at least detectable.

Such observations, coupled with evidence of massive computational loads (such as White's quote), lead one to suggest a style of simulation practice in which high-speed modes are employed initially, and as the design is refined, greater accuracy is gradually introduced. Even within the confines of analog simulation such a methodology has been strongly supported in the SPECS system [Nguyen+89], for which it is recommended that the user perform *training* simulations with detailed parasitic modelling disabled, with the intention of both revealing gross errors of functionality and highlighting areas where detailed modelling would be appropriate. The authors note that surprisingly often training simulations are sufficiently accurate that further simulation is unjustified. SPECS is notable for providing an accuracy measure as feedback to the user— I believe such features have strong positive benefit in establishing user confidence in a simulation.

What then is the likely error-interaction with this methodology? One must recall that in VLSI simulation most errors encountered are due to gross functional flaws in the system under test— in an analogous manner to syntax errors in a computer program. These are encountered early in the design process, and thus there is a major benefit in using the highest available speed at this point— which is entirely practical as the gross errors are unlikely to be unnoticed. High accuracy can then be reserved for the very final characterization stages— where there is no substitute for SPICE or a relative thereof.

Having presented some reservations about the emphasis of much work in analog simulation, we now return to the discussion of the literature, but

with new terms of reference derived from the general methodology presented above. Such a shift in thinking allows a narrower consideration of this wide field.

2.7.2 Alternative Analog Approaches

The field of analog simulation is dominated by a long stream of papers and simulators originating from the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. Roughly one third of the work cited in this section originates therefrom, and of the remainder, secondary connections often exist, such as in the case of MOTIS, as a second generation MOTIS called MOTIS-C appears in another Berkeley paper [Fan+77]. MOTIS-C was notable for exposing interesting variations in accuracy depending on the order the equations were solved in— (the Gauss-Seidel relaxation method iterates over $\bar{V}$, such that at a particular point in (real) time some V_i will be in their k-th iteration and some in their $k+1$-th iteration at (simulation) time t. If nodes are operated on in ascending order, a particular V_i is then is solved for in an environment where—

$$V_j = \begin{cases} V_j^k & j > i \\ V_j^{k+1} & j < i \end{cases} \tag{2.9}$$

). This effect is unsurprising given the implicitly ordered nature of many circuits— for example chains of directional gates. One would expect that at the very least that there should be variation in speed of convergence depending on equation ordering. Systems for finding optimal device and/or equation orders in both digital and analog domains have appeared— a matrix-level example is [Yang85].

Event-driven equation ordering was an innovation of the SPLICE simulator [Newton79], which with improvements in numerical methods evolved

into a system described as performing Iterated Timing Analysis. SPLICE also avoided unnecessary integration of inactive nodes [Newton+84]. See also [De Micheli+83] for algorithm analysis and [Deutsch+84] for parallel solutions.

A most interesting result was the discovery that relaxation techniques are applicable over a waveform of non-trivial length. In [Lelarasmee+82b] and [Lelarasmee+82a] the resulting *Waveform Relaxation* method is described. Unfortunately, despite being encouragingly faster than SPICE, early simulators of this design tended to converge slowly on certain circuits. In [Saviz+88] it is observed that slow convergence is usually due to tight coupling— leading to a system which avoids uses Newton/Raphson iterations in such places and waveform relaxation elsewhere, giving the best of both worlds.

Another problem with waveform relaxation is that the storage requirements to retain long waveforms quickly become prohibitive with large circuits.

[White+85] shows some ways of improving waveform relaxation performance, notably again with careful choice of step size, and use of parallelism.

Waveform relaxation has attracted much attention, for example being adapted for incremental operation in [LeBlanc+85]. However, a particularly useful adaptation appears in [Hennion+85], where by rearranging the order of the loops in the waveform relaxation algorithm such that the "time" loop is outermost, the desirable properties of reduced storage requirement and interruptability are provided. (Because standard waveform relaxation gradually improves a whole waveform towards correctness, if the process is interrupted the intermediate result is usually not useful— whereas if the time loop is outermost, an interrupted simulation will yield results valid to the point of simulation time at which the interruption occurred.) A recent promising variant appears in [Erdman+89].

Returning to the simpler matrix based techniques— given the non-linear growth of matrix solution time, a sound procedure for speeding up this task is to reduce the size of the matrix— a classic application for *divide and conquer* algorithms. [Engl+82] and [Zwolinski+84] give a strong recommendation to use of modularity to achieve this goal.

However, modular partitioning may not be enough in the presence of strong intermodule coupling according to [Mokari-Bolhassan+85]— in which instead of treating modules as black boxes, the partitioning overlaps the extremities of a module. Overlapped sections are repeatedly solved as long as they are sensitive to relaxation of other modules that may drive them.

A notable series of simulators appears under the name SAMSON, SAMSON2, ... [Sakallah+85b]. The aim of these simulators is to allow a more event driven approach to traditional analog simulation— the system is partitioned into blocks of varying activity for which the solution time points (the events) are independently chosen. Much emphasis is placed on the accurate modelling of the system in its *dormant* state between event time points (where it is described as *alert*) (the dormant model is refined in [Sakallah85a] to remove a theoretical inadequacy). Time step control is managed with a priori estimates which are then accepted or rejected on the basis of a posteriori estimates of the local truncation error. In contrast, [Cox+88] returns to using a single time step for the whole system, but continues to place strong emphasis on dormancy as the important feature from which improved performance can be obtained. The independent time step feature probably appeared first in [Chen+84].

As an indication of the state of the art of analog simulators, consider XPSim [Bauer+88][6] , which is reported to be only 4–5 times slower than the

[6]XPSim is difficult to unambiguously categorize— it is perhaps better described as a hybrid simulator.

timing analyzer Crystal [Ousterhout83]. This system employs both static and dynamic partitioning of the circuit, allows floating capacitors and leakage resistors, and models waveforms as exponential curves. A design goal of XPSim was to allow simulation of "whole" circuits— a 100K device example is mentioned— requiring 70 Megabytes of memory for representation.

The device/memory figures above reveal a important practical detail about integrated circuit simulation. Because of the large numbers of objects (nodes, devices) being modelled, vast amounts of memory may be consumed by large simulations. For example— consider a hypothetical minimal analog simulator, where the modelling requirement for a node is only a voltage and a capacitance (floating point quantities), and for a device only pointers to up to three nodes and a device type field, then a lower bound on storage requirement is—

$$2 \times \text{sizeof(float)} \times N + 4 \times \text{sizeof(pointer)} \times D (\text{bytes})$$

—where N is the number of nodes and D is the number of devices. Then on a conventional thirty-two bit processor with, say 100K nodes and 150K devices this results in a memory bill of at least four megabytes. As a comparison, at the medium-abstract gate level, a survey of commercial simulators showed that gate element records consume around 20–40 bytes [VLS85], and the timing analyzer TV takes 80 bytes per device and 104 bytes per node.)

Sophisticated systems such as XPSim trade storage for speed and have greater requirements than such a minimal representation. In all cases however, eventually these storage requirements will become a significant burden to the computing facilities available. In the common current case, where a reasonable sized compute server has around thirty-two megabytes of main memory and a virtual memory system, large simulations become significantly impacted by the speed of the paging subsystem.

These practical considerations suggest several design criteria—

- High storage management overheads militate against the desirability of simulating a whole circuit at analog level. Hierarchical capability is again shown to be highly desirable.
- The payoff in reduced overhead from adopting a concise circuit representation is more and more significant with increased circuit size.
- Paging should be minimized by using a circuit representation that maximizes *locality of reference.* (Unfortunately, although the interconnections between devices and nodes may appear to exhibit a fair degree of locality, in practice the sheer weight of device and node numbers tends to spread the representation out across available memory.)

Design for *Hierarchical capability* *(8)*
Design for *Concise circuit representation* *(9)*
Design for *Locality of reference* *(10)*

2.8 Mixed Modes

This section discusses issues specific to mixed-mode simulators— by definition those where at least two distinct modes of simulation are *simultaneously* possible.

Early work in this field is dominated by cases of existing systems being extended to handle an extra mode— for example [Thompson+80] where hard-coded models were added, or [Nash+80] where an RTL level was added, in both cases to a previously designed gate-level simulator. This sub-optimal implementation method has become less common as the field matures. There is general agreement that all modes provided by a simulator should be equal

partners, present by design, which is a "difficult, but comprehensive" goal ([Chadla+88]).

Since functional and digital modes are more closely compatible than any pairing with analog mode, the bulk of simulators that qualify to be called mixed-mode under the preceding definition possess these two modes— see [Raeth+81], [Hirchhorn+81], [Sakai+82], etc. Therefore the DIANA simulator [Arnout+78], which is the first analog and digital mode simulator, is particularly noteworthy. At the other extreme is [Mayaram+88] which details a mixed analog and device level tool.

The task of interfacing analog outputs to digital inputs has lead to various *thresholding* techniques. The simplest of these uses two fixed absolute analog voltage levels $V_{low_threshold}$ and $V_{high_threshold}$ where—

$$V_{ss} < V_{low_threshold} < V_{high_threshold} < V_{dd}$$

—and—

$$V_{logic} = \begin{cases} Lo & V_{analog} < V_{low_threshold} \\ Hi & V_{analog} > V_{high_threshold} \\ X & \text{otherwise} \end{cases} \qquad (2.10)$$

This simple solution presents little computational overhead, however it suffers from a tendency to either sustain X conservatively if the thresholds are wide apart, or to smooth noisy signals if the thresholds are relatively close (of course, this may actually be desirable in some cases). A more sophisticated thresholding method appears in [Agrawal+80] (notable for presenting possibly the first analog, digital and functional simulator)— here further parameters V_{cross} and T_{settle}[7] are introduced, where $V_{ss} < V_{low_threshold} < V_{cross} < V_{high_threshold} < V_{dd}$ and $T_{settle} > 0$. The result is redefined to be—

[7]The nomenclature used here differs from that in the original paper.

$$V_{logic} = \begin{cases} Lo & V_{analog} < V_{low_threshold} \\ Hi & V_{analog} > V_{high_threshold} \\ \text{else} & \\ X & V_{low_threshold} < V_{analog} < V_{high_threshold} \text{ for } \geq T_{settle} \\ X & \text{or if } dV_{analog}/dt \text{ changes sign} \\ \text{else} & \\ Lo & V_{low_threshold} < V_{analog} < V_{cross} \\ Hi & V_{cross} < V_{analog} < V_{high_threshold} \end{cases} \tag{2.11}$$

—which has the desirable property of suppressing the generation of X in normal $Hi \Rightarrow Lo$ and $Lo \Rightarrow Hi$ transitions where V_{analog} passes quickly through the region $[V_{low_threshold}, V_{high_threshold}]$ without changing the sign of its derivative. The consequences of X-pollution are sufficiently severe to justify the extra computational burden of this method. Unfortunately, it is somewhat unnatural— consider the simple exponential signal shown in figure 2.15, and its possible output for a particular choice of parameters.

The sequence of logic transitions Lo, Hi, X, Hi includes at least one spurious state— either the first Hi, or both it and the X. Assuming the input is sufficiently slow that generation of X is desirable, attention focuses on the first Hi. This transition comes about due to the algorithm predicting that the input will swiftly reach $V_{high_threshold}$, whereas in practice this proves not to be the case. This is an awkward problem, when one considers the ubiquity of exponential analog signals. Fortunately, it seems to be possible to minimize its occurrence by careful choice of the four threshold parameters. Interesting approaches available to an event-driven system include not propagating X states until they are resolved, or using an event-unrolling procedure to remove spurious states.

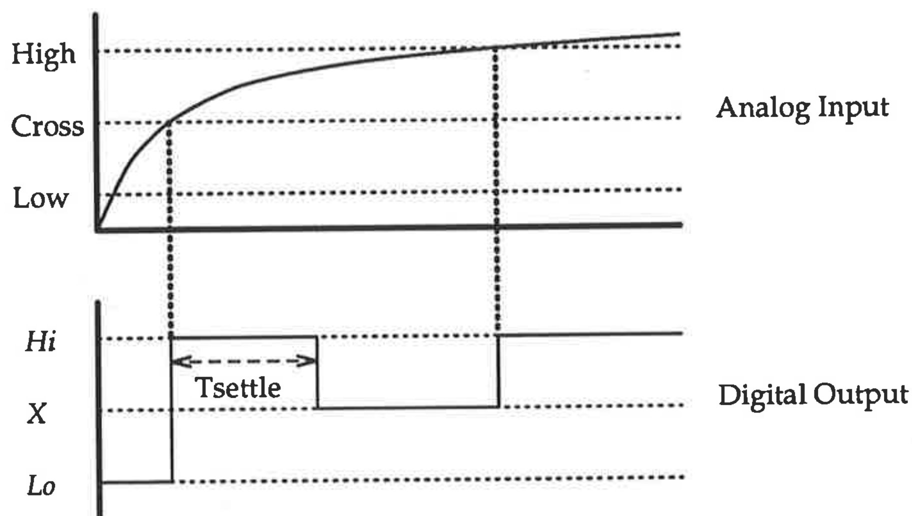


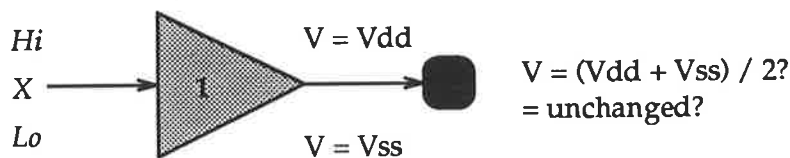
Figure 2.15: Thresholding Problem

That the analog ⇒ digital interface should be troublesome is hardly surprising as there is a loss of information travelling across this boundary. The converse case is simpler, with the main alternatives shown in figure 2.16. Digital values can either be directly converted to analog levels (1), or treated as defining linear current sources ([Arnout+78] introduced this technique as *boolean controlled switches*) with X state outputs corresponding to a zero current (2), or even to insulate the raw logic levels with a layer of real device modelling— for example by inserting a simple buffer circuit (3).

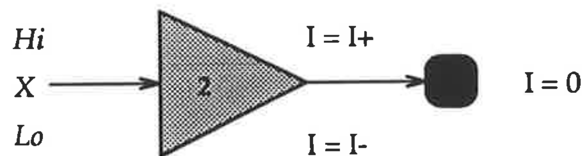
A common practice in mixed-mode simulation is to decouple the structural information from the functional by implementing distinct languages for the two— for example [Sasaki+80]. Reasons for this split include the possibility of separate compilation, and also because isolating the structural information allows it to be quickly accessible to other tools (such as a cell-builder) without requiring extra parsing. In opposition to this technique is the desirability of keeping all the information on a module in one place— an-

noying mistakes can arise if the structural and functional descriptions become inconsistent.

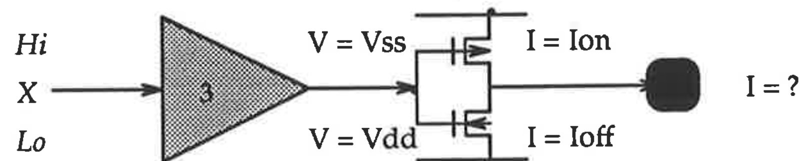
Feedback and bidirectional elements continue to cause trouble in mixed-mode. It is common to place restrictions on the system under simulation— such as in [Lieberherr83] which requires all cyclic paths to be clocked and bidirectionality not to be visible at the structural level— that is, all bidirectional elements must be hidden inside leaf modules. Such restrictions have been found to be widespread in commercial simulators (see [Walker88]). However, the alternatives are similarly unpleasant— for example in [Borrione+83] feedback is handled at the expense of much special case scheduling and mod-



Digital to Analog Voltage Conversion



Digital to Analog Current Conversion



Digital to Inverse Analog Voltage,
then analog device models

Figure 2.16: Digital to Analog Interfaces

elling complexity. The IDSIM2 system of [Overhauser+89] is exceptional in its dynamic handling of feedback— IDSIM2 categorizes modules as either analog or digital on the basis of several tests, one of which is for feedback configurations. Because feedback is often conditional on input data, IDSIM2 is able to dynamically reclassify modules during simulation, as feedback paths are made or broken— the intention is to automate the choice of the instantaneously most efficient simulation mode.

The major attraction of mixed-mode is the promise of being able to test a whole large system, with areas of particular interest simulated in great detail, with other areas handled by more efficient modes. The main method for achieving this goal is exploitation of the hierarchical structure of the system under test, with the provision that one must always trust the functional simulation of a parent composition module to accurately reflect the aggregate behaviour of its child modules. (The importance of using the hierarchy is stressed in [Saab+88], which reports a successful development of a functional model for a Motorola MC68000 microprocessor). Satisfying this constraint is a major design goal of the hierarchical simulator presented in [Chen+83] (and its derivative [Lin+86]), in which a uniform representation of the interface mechanism at all levels of modelling is cited as the key factor.

The general problem of verifying absolute equivalence of supposedly functionally identical descriptions prohibitively difficult. A more practical goal is to attempt to measure equivalence between the run-time behaviour of modules under a particular set of input test vectors. Again this suggests use of a mode-uniform event-based interface representation— as a means to limit the amount of information that would need to be analyzed. Verification is of course a large field— detailed consideration of various verification strategies is beyond the scope of this review.

2.9 Hybrid Modes

The previous sections on digital and analog modes have mentioned problems with each. Several researchers have therefore attempted to extract the best features of both modes and produce a superior hybrid. A typical early effort appears in [Nham+80], where from analysis of analog device characteristics and load capacitances, a distinct rise and fall time for each device is calculated. This analog-derived parameter controls the rate of propagation of digital state changes through each device.

A related approach is the seminal RSIM [Terman83], which treats devices as a variable resistor— with a precomputed "on" values (two dynamic values for $Hi \Rightarrow Lo$, $Lo \Rightarrow Hi$ transitions, and a static value for steady state calculations), infinite resistance when "off" and when the device is driven by a X takes the unusual step of using interval arithmetic— modelling the resistance as the range $[R_{on}, \infty]$. This abstraction proves to be quite powerful in terms of reducing algorithm complexity and thus delivering simulation speed, while accuracy mainly remains within 30% of SPICE. Unfortunately, charge-sharing effects (especially in networks of pass-transistors) are handled poorly, requiring special treatment.

An interesting generalization of the hybrid-mode paradigm is given in [Dewilde+85]. Here the state of a node is controlled by an approximating spline function, and the process of simulation thereby becomes a matter of propagating changes in node function parameters (figure 2.17). This model is clearly attractive for event-driven implementation. Another benefit is that with a rich set of function control events, it becomes straightforward to provide multiple simultaneous simulation modes.

As was shown in the section on analog simulation, many simulators operate by discretizing time, and solving for voltages. Inevitably, the dual method

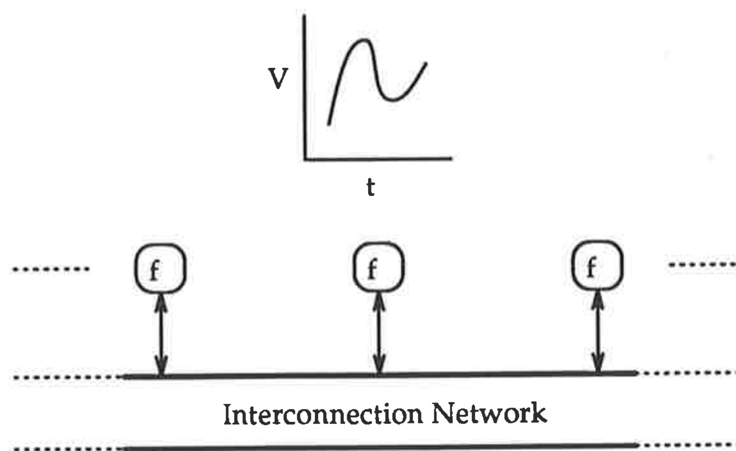


Figure 2.17: Parametric Simulation

of discretizing voltage and solving for time was tried (the first instance being [de Geus84])— leading to some impressive systems. Once voltage is discretized one is firmly on hybrid ground, as the digital model embodies the ultimate (useful) voltage discretization.

If one solves for time rather than for voltage, event-driven operation of an otherwise basically analog simulator become practical almost for free. Given that a discrete set of voltages has been chosen, the basic simulation algorithm at T_i involves solving for the time T_{i+1} at which a given node currently with voltage V_j will reach an adjacent voltage $V_{j\pm1}$— such occurrences define the events (figure 2.18). Equally fortuitous is the property that by limiting calculation to inter voltage-set steps, excessive voltage variations that can occur in conventional simulators that allow the time step to become too large are automatically prevented.

A direct and effective implementation of this technique appears in the CINNAMON system of [Vidigal+86]. Here voltage-change events (changes above a supplied threshold) give rise to solution of the few device equations

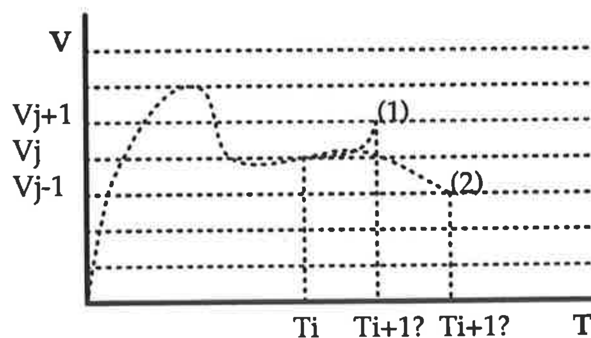


Figure 2.18: Discretized voltages and event generation

relevant to that node, or in the case of simple uncoupled node/device configurations an exponential approximation for the expected voltage characteristic is substituted. User control of the voltage-change threshold provides a turnaround/accuracy tradeoff. The basic response with a threshold of 100mV shows two orders of magnitude improvement over SPICE. Similar techniques are used in [Odryna+86] (iteration is introduced by scheduling events to re-evaluate neighbouring nodes) and in [Kim+89] where group of algorithms are presented under the collective name ELogic. ELogic is an adaptation to hybrid-mode of the lessons learned in previous simulators from the University of California, Berkeley, and shows typical thoroughness in modelling awkward analog effects. Event-EMU ([Ackland+89]) reverses the trend of such systems to schedule node transitions and returns to the proven technology of static partition into tightly coupled regions with scheduled periods of iteration.

[Beckett86] presents another high performing system based on voltage discretization but with the simplification that the inter event behaviour of voltages is assumed to be piecewise linear. The advantage of the linear approximation is that it reduces the complexity of the event-time calculation—

which is now the main area of numerical overhead. Interestingly, the resulting discontinuous voltage waveform data is accumulated, and used as input to a curve fitting procedure!

Further evidence for the general usefulness of voltage discretizing simulators is given by [Kao+88] which models ECL/CML devices, however some performance issues remain to be addressed, particularly regarding tightly coupled configurations— see [Visweswariah+89].

Having discretized voltages successfully, one is lead to ask "What else can be discretized?" In [Ruan+85], it is the turn again of device currents— however rather than considering devices to be resistors as in RSIM, devices are limited to producing only three currents $I_2, I_1, I_0 = 0$ as determined by the general device characteristics shown in figure 2.19. In this formulation, the event points are given by the time that the terminal voltages of a device require a change of current region. Obviously choice of the critical voltages and currents is crucial to the accuracy— the approach taken in [Ruan+88] optimizes for two common modes of device operation.

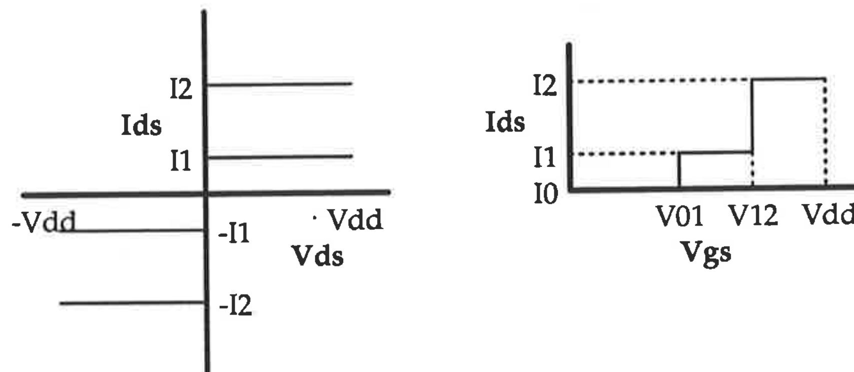


Figure 2.19: Discretized Current Characteristics

These systems all provide positive evidence for the benefits of an hybrid

simulation mode. In fact, so impressive are the results that one is lead to the conclusion that hybrid mode simulation is sufficiently powerful to replace separate analog and digital modes.

Design for *Hybrid mode* *(11)*

2.10 Summary

To summarize the review, assembled below is the list of major design criteria identified in this chapter—

1. Multiple interchangeable modes
2. Fidelity of modelling
3. Flexible, powerful HDL
4. Easy to use HDL
5. Clean interfacing between all modes
6. Speed $\Rightarrow$ event driven simulation
7. Communication with analog voltages
8. Hierarchical capability
9. Concise circuit representation
10. Locality of reference
11. Hybrid mode

—these criteria determined the shape of Loge, and underpin all the discussion of the following chapter. The list could easily be much longer— so many criteria and useful techniques arise from the literature that decisions of what to omit were frequent and vexing.

Chapter 3

Design of Loge

3.1 Introduction

The structure of this chapter loosely follows the order of the development process. Broadly speaking discussion proceeds from a preparatory investigation of functional simulation through scheduling issues to detailed examination of the primitive simulation objects, which leads to hybrid mode simulation and finally some examples. Concern for mode interfacing, fidelity of modelling and HDL issues is interwoven throughout.

3.2 Structure

The broad specification for a simulator is that it should accept models written in a hardware description language or languages, and be able to simulate the time behaviour of these models in a controllable environment. Loge follows a conventional pattern of user interaction that arises from this specification—

% loge [<*input-files*>]

```
loge>  input commands
output results
:
loge> (quit)
%
```

— that is, a simple command interpreter. The basic user interface to Loge is thus quite spartan— and deliberately so, the intention being to offload the burden of a more powerful interface onto special purpose programs that either invoke Loge or are themselves invoked through HDL expressions. This relatively easy decision focuses attention on a more difficult problem of specifying the HDL.

3.2.1 HDL issues

The matter of model representation and HDLs has been previously discussed in section 2.3.2. Given the requirement for a powerful HDL, then of the two main HDL implementation techniques of embedded language and special purpose language it appears that building onto an existing language is both less work and can guarantee a known minimum degree of expressivity. Thus the question for Loge became— "Which host language?".

This issue is complicated by the different style of input to the simulation modes— the input to most analog, logic or hybrid mode simulators is simply a list of electrical elements— capacitors, devices, etc, in a format such as EDIF or a SPICE "deck", whereas functional mode simulator input tends to be mostly modelling code. Nevertheless it was decided to build one unified language for all modes, as this is less restrictive to the user.

Prototypes

Build one to throw away... you will anyway.

— Fred Brooks

Choice of the Loge host language was aided by a prototyping phase, as were the resolution of many other initially unclear language-related design issues—

Signal typing Can strong typing of signals be enforced (for example, preventing the connection of two "Write-Only" signals), and if so is it useful?

Genericity How difficult is it to implement *generic* modules— modules whose structure is to some degree variable and not fully known until the module is instantiated?

Connections A common problem with functional simulators is that specifying connections between modules is tedious. Can this be improved?

Hierarchical construction How does one construct a hierarchy of modules? What are the tradeoffs between amount of storage, efficiency of access, locality of reference and genericity?

A functional and structural mode prototype was written in Franz Lisp, using a Flavors package to provide an object oriented environment, while prototype hybrid and analog modes were developed in Pascal and C++ as extensions to an existing simulator ([Int87]). The main conclusions were—

1. That signal typing was useful and could be checked easily.

2. Generic modules were possible, but required the full flexibility of Lisp. Simplified forms (such as allowing variable width bus connections) would be necessary if a less flexible implementation language was used.

3. Specifying connections between modules is inherently tedious. Generic modules help, but one must expect a specification of complex wiring to look complicated.

4. Hierarchical systems must strive hard to minimize storage. With the exception of device-level systems, it is difficult in general to improve locality of reference beyond that already supplied by the module boundaries. Control of layout of simulation objects in memory is necessary in either event.

Conclusions 2) and 4) typify the conflicting requirements of structural versus hybrid modes— to provide both generic modules and efficient memory layout requires a very wide ranging implementation language. In characteristic fashion for this project, the solution was to provide a system that is a hybrid of two languages— Lisp and C++ [Stroustrup86] (it was subsequently heartening to note the work of [Wolf89] in which these languages individually proved to have strong features applicable to digital systems simulation). C++ is a particularly appropriate choice as its development was triggered by a desire for high performance successor to Simula67 [Dahl+70], which has given long service to the simulation community.

Loge is thus a C++ program, which manipulates simulation-specific objects. In particular, computationally expensive tasks such as event scheduling and hybrid mode evaluations are pure C++ code. The actual HDL however is an interpreted Common Lisp subset with simulation-specific extensions, which allows specification of generic modules with arbitrarily complex interconnections.

The performance penalty inherent in the use of an interpreter is only significant when building a module instance hierarchy and when running functional code. The first case is unavoidable, but the second may be too great a burden in cases where Loge is used for top-down specification of experimental architectures, in which modules will only operate in functional and structural modes (as in the case study of section 4.4). To allow efficient simulation in these cases an interface to user-supplied object code is provided, giving the best of both worlds.

3.3 Scheduling

Loge is an event driven simulator, using a single central event scheduler. The basic functions of the scheduler are—

Enqueue Schedule an event for an object at some later time.

Dequeue Fetch the object with the earliest pending event.

Preempt Cancel a pending event.

The general algorithm for event driven simulation with a central scheduler is—

Algorithm 3.1 *Event Driven Simulation*
Enqueue initial events
repeat
  Dequeue an event
  Run event-specific service routine
    (possibly enqueues or preempts more events)
until out of events or explicitly terminated

For efficient simulation it is important that the scheduling functions be reasonably inexpensive, as they are pure overhead to the real work of simulation which occurs in the event service routines.

Implementation Note: Time The representation of time is vitally important to a scheduler. Given a discrete event paradigm, it is natural and efficient to represent time as an integer quantity. Unfortunately, the most natural size of integer available on most contemporary computers is only thirty-two bits, giving a range of nine orders of magnitude. Since one may conservatively expect a simulator for VLSI systems to be able to resolve picosecond events over a total run time of seconds, twelve orders of magnitude is a minimum time range requirement— thus an integer time representation can not be used directly, although it is still possible if one is prepared to use scaling and time-origin shifts or expensive long integer formats. Eventually larger fast integer formats will become more readily available, however for the present Loge uses floating point quantities with time origin shifts to represent time, following the example of [Sakallah+85b].

3.3.1 Scheduling Algorithms

What is the most suitable algorithm for an event scheduler in a general purpose digital simulator? Algorithms may be compared on the basis of their best, average and worst case performances, but for such comparisons to be relevant, some knowledge of the event time distribution is necessary. Regrettably, event time distributions vary widely across the simulation modes and systems under test. In very broad terms, some typical cases are—

- Low level simulation of circuits with analog behaviour: Difficult to predict, especially if feedback is present, but often characterized by

continuous large numbers of closely spaced events.

- Low level simulation of digital systems: High densities of events following a clock input change, decaying fairly quickly to some minimal level.
- Functional simulation of simple digital primitives: Event times are relatively sparsely distributed, with a tendency to *cluster* onto specific time points (again this is related to the clocking used).
- Functional simulation of complex modules with user supplied code: Often similar to the previous case, but potentially the most wildly varying situation of all.

With such variability it is highly unlikely that one single algorithm will be optimal for all cases.

Timing Wheel

The classic *timing wheel* [Ulrich80] (or "temporal hashing") is perhaps the most frequently used algorithm in schedulers for analog simulators. In terms of the number N of events in a timing wheel, and the number of slots M in the wheel the enqueue and preempt operations require an average of $N/2M$ comparisons, while dequeue requires only a single test when the event density is high. Unfortunately, under sparse event times, the average cost of dequeue rises towards M comparisons and if clustering is present the worst case cost of enqueue and preempt tends towards N comparisons (some of the effects of clustering can be reduced by placing distant events on a simple list, while the imminent events continue to be placed on the wheel, and at intervals the events on the list are resubmitted for potential placement on the wheel, however this procedure is not without overhead).

Timing wheels are indeed well suited to analog simulation, but degrade under conditions characteristic of a class of functional simulations.

Simple Heap

Performance of a simple heap [Sedgewick88] of events (embedded in a fixed array) has the advantage of being less sensitive to variations in event distribution. Heap operations appear to have the advantage of asymptotically superior performance ($O(\lg N)$) to those of a timing wheel if enqueue and dequeue events occur in similar numbers, however this superiority will only occur under a very high event density unless M is quite small.

Nevertheless, the heap algorithm is a robust choice for a general purpose simulator. It is the default in Loge, with the option of using a timing wheel if the user sees fit. The particular heap implementation in Loge is optimized in favour of the common situation where an event is preempted and immediately reinserted at a slightly later delivery time— rather than fully removing and replacing such events, they may be directly moved down the heap (typically requiring a very small number of steps).

3.3.2 Event Suppression

Although it is worth taking care in the design of the event scheduler, the best opportunities to improve overall simulator performance are in application of techniques that reduce the total number of events required for accurate simulation. Many event suppression techniques are specific to the objects under simulation and will be discussed in later sections. However, a generally applicable technique is to order events at the same time point. Consider a functional module M, connected to node N and many other nodes and modules, for which both M and N have events due at time T (figure 3.1).

If servicing an event can cause messages (dashed lines) to be generated, and the receipt of messages can trigger further messages (*ricochets*), then the total number of events and messages can fluctuate wildly between different event orderings, notably depending on the fanout of the components. To illustrate this more concretely, assume a hypothetical event/message scheme where—

- When servicing an event or receiving a message a module sends "Current" messages to nodes, and then schedules its next event.

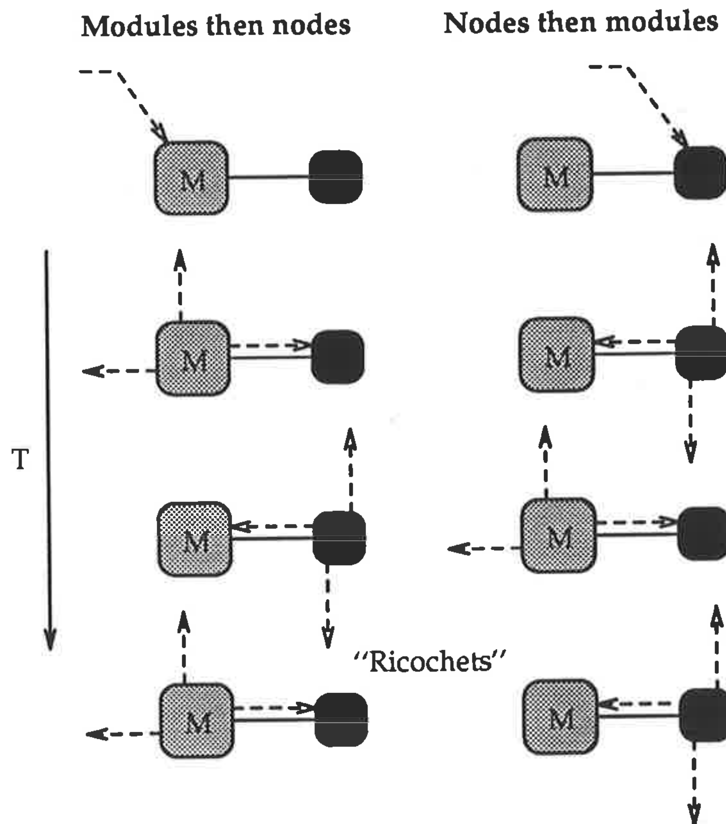


Figure 3.1: Events between module M and node N

- When servicing an event a node sends "Voltage" messages to modules, and then schedules its next event.
- When receiving a "Current" message a node may preempt its next event, but sends no ricochet messages.

— in such configurations it is always worth servicing nodes before modules, for if the module is serviced first it may shortly require extra processing should it receive a "Voltage" message from a neighbouring node.

To accommodate such optimizations, the event scheduler is supplied with an analysis function which compares two time equivalent events and returns indication of which should take priority in the queue. An extension of event ordering is the rejection of duplicate events, which may occur in functional simulations of bus structures (figure 3.2).

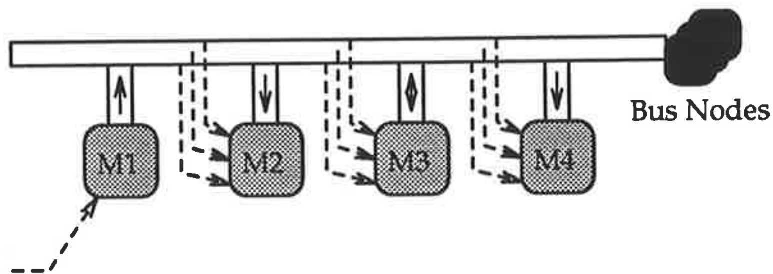


Figure 3.2: Bus events

Supposing the event discipline is changed such that nodes instead of sending "Voltage" messages, schedule events for the modules that are connected to them at time points when the node is expected to be stable. In such cases a write to a K bit bus by module M1 results in K events (dotted lines) being generated for the modules M2, M3, and M4 that are capable of reading the bus. If the bits of the bus are identical (as they are prone to be in top down

functional investigations where no electrical parameters have been specified), then all events will be scheduled at the same time. By simply rejecting the duplicates the number of events to be serviced is reduced from $3K$ to 3 — very good value from a trivially implemented optimization.

Bus structures are so common that it is almost certainly worth making a similar optimization available in the case where the bus nodes generate somewhat different event times from the same write. Consider a module that reads from a K bit bus, and computes a complicated function of the bus bit values which is written to an output node— if all input nodes change state at different times, K computations could be performed where one would possibly suffice. Worse still, if preemptive semantics are applied to events queued for modules, later events may be incorrectly ignored. What is needed is a means to *structure* nodes into a single entity at the module interface (while still allowing nodes to have separate electrical parameters for the purposes of analog simulation), such that the module is only contacted when all nodes in the bus have been stable for some period— a hysteresis effect. This can not be done purely by the event scheduler, but is easy to implement by defining a new type of object— a node concentrator, or *port*, which isolates modules and nodes (figure 3.3).

3.4 Simulation Objects

This section describes the four main simulation objects— modules and ports (specific to functional mode), devices (specific to low level modes) and nodes (common to all modes). In each case, one must consider what physical quantities the object is expected to model, and whether the quantity is a unique property of each *instance* of the object, or a property of a *class* of object— for example, the instantaneous voltage at a node is the private

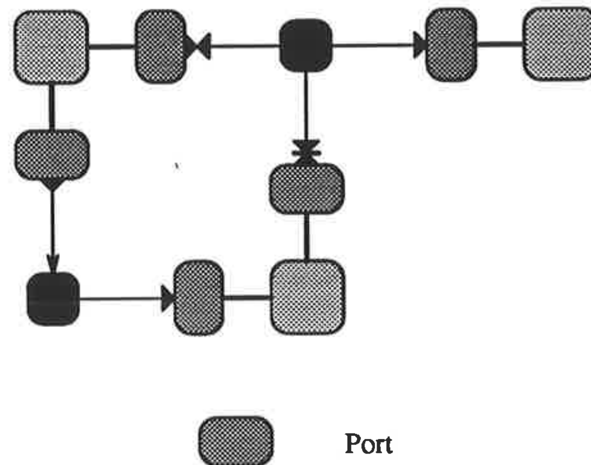


Figure 3.3: Modules, Ports and Nodes

property of that node instance, but there may be a large number of nodes that all take an initial voltage of 0V — perhaps defining a class of nodes. As there will generally be considerably more object instances than classes, to avoid high storage costs it is vital to minimize size of the per instance properties.

There are some properties shared by all objects. Firstly, an object instance must contain a reference to its class. Secondly, the requirement for a hierarchical representation means that each instance of an object is part of a tree of instances— the leaves will be nodes, ports and devices, while the trunk will contain modules. It is fundamentally important to be able to traverse this tree, thus modules must contain a list of their child instances, and all objects must contain a link back to their parent instance.

Other properties shared by all object instances are its *local time*, and a pointer to any currently scheduled event— allowing quick invalidation of events when preemption is required. The structure thus far is shown in fig-

ure 3.4. A less tangible property of the simulation objects is the set of messages they send and/or receive. The discussion that follows aims to derive a broad representative set of message types for use in various specific algorithms. Note that in the discrete event/message passing paradigm adopted for Loge, the terms "event" and "message" are used semi-interchangeable— strictly an event occurs on execution of a service routine when a object reaches the head of the scheduler queue, while a message is an instantaneous (small) data transfer between objects. However in many cases sending a message causes similar activity in the recipient object to servicing an event[1], so the distinction is blurred. A convenient piece of terminology used hereon is that when an object receives an event or message it is said to be *awakened*.

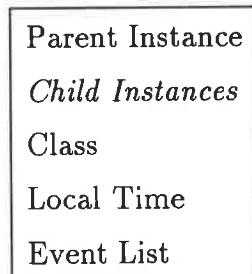


Figure 3.4: Simulation object instance common internals

3.4.1 Nodes

One of the design criteria for Loge requires analog voltages as fundamental quantities. Since voltages are properties of circuit nodes, we recall the per node form of the simplified nodal analysis equations, equation 2.3—

$$C_i \frac{dV_i}{dt} + \sum I_i(\bar{V}) = 0$$

[1]Or even for the recipient object to preempt and schedule itself for immediate service.

This equation is in terms of per node (i) quantities C_i, dV_i/dt and $\sum I_i(\bar{V})$, where $\bar{V}$ is the vector of the V_i. V_i and dV_i/dt are clearly per node variables and belong to the node instance, although their initial values need not. While C_i is constant and may be common amongst several nodes, and thus potentially a class-level property, capacitance figures heavily in the procedure for updating node voltages, therefore it too is stored in every node. The current term is not a property of any one node— it belongs collectively to those elements that are connected to the node i.

Implementation Note: Many object properties are subject to numeric computation. While it is desirable to use integer arithmetic if possible, the decision to adopt a floating point representation of time defeats most of the potential speedup afforded by integers, as time is present in many if not most of the computations performed by Loge. Therefore, with considerable regret, all physical analog properties such as voltage, current, resistance, etc are stored and manipulated in floating point.

Another property of a node instance is the set of other objects (such as ports) that are connected to it as part of the physical "wiring" of the system. There is a useful duality here between modules and all other objects including nodes— because modules can not be leaves of the object instance hierarchy, *their connections are all hierarchical* — being either parent-to-child or child-to-parent. On the other hand, nodes et al are always leaves, with a single hierarchical child-to-parent link, but no parent-to-child connections. These leaf objects do however have "wiring" connections between themselves, whereas modules do not. Figure 3.5 illustrates this difference, showing in abstract form the instance tree of a non-inverting buffer module which consists of two cascaded inverter modules.

Implementation Note: The practical upshot of this non-overlap of connection type is that the same slot in the general object structure can be used

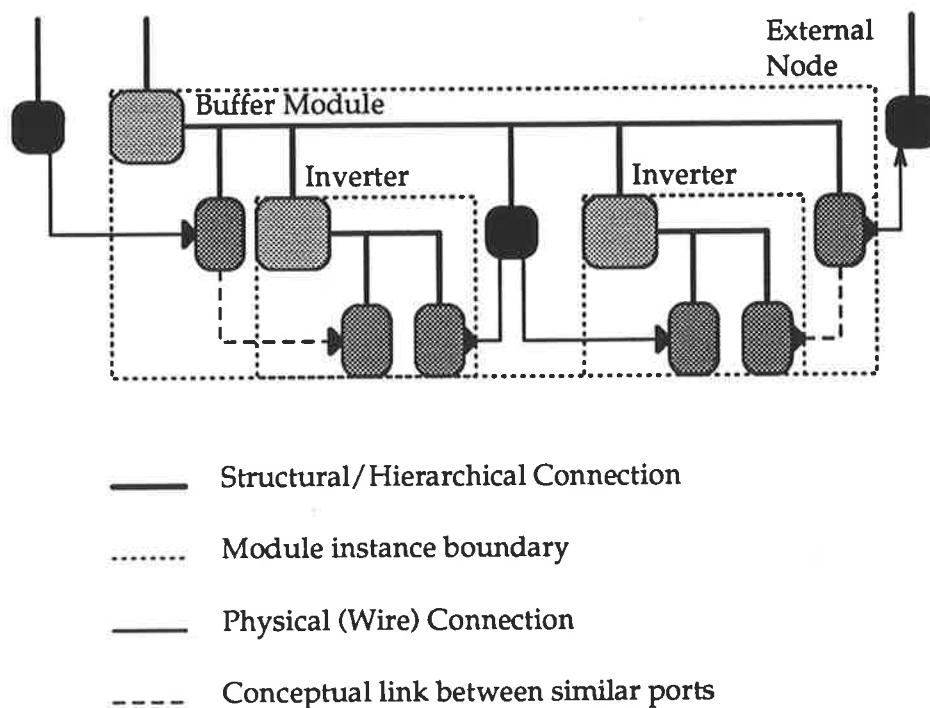


Figure 3.5: Simulation object connections

for leaf object interconnections as is used for module parent-to-child links. A more general name for this slot is the *fanout* property. The properties of a complete node instance appear in figure 3.6.

Implementation Note: At present Loge represents voltage internally in normalized form $V_{norm} = V_{actual}/V_{supply}$, thus all voltages are the range $[0.0, 1.0]$. This allows a number of useful optimizations and internal consistency checks, at the expense of complicating detailed simulation of mixed technology systems (those with multiple V_{supply}).

Having collected the tangible properties of a node, the following sections focus upon the event and message interactions of nodes in both hybrid and functional modes, and show why it is useful for local time to be a property

Parent Module
Fanout
Class *(Node)*
Local Time
Event List
V_i
dV_i/dt
C_i

Figure 3.6: Node instance

of nodes and other simulation objects.

Hybrid Mode

A commonly encountered feature of many hybrid simulation modes is that the time rate of change of node voltage is constant over the interval between events for that node. Also common is the restriction of voltages to a finite set— $Voltages = \{ Voltage_j$

—where—

$$\forall i,j \in [0,N];\ i < j \Rightarrow Voltage_i < Voltage_j$$

Under these assumptions there are a number of feasible calculation and messaging disciplines, such as that of algorithm 3.2 and figure 3.7 which could be called *broadcast-and-sum.*

Algorithm 3.2 *Broadcast-and-sum*

On awakening at time t_k

$V_i \leftarrow V_{next}$

Send $V_i(t_k)$ *messages*

Receive $I_{f,i}(t_k)$ *messages*

$dV_i(t_k)/dt = \frac{1}{C_i}\sum_f I_{f,i}(t_k)$

Given $V_i(t_k) = Voltage_j$

$$V_{next} = \begin{cases} Voltage_{j+1} & dV_i(t_k)/dt > 0 \\ Voltage_{j-1} & dV_i(t_k)/dt < 0 \end{cases}$$

$t_{next} = t_k + (V_{next} - V_i(t_k)) \ / \ dV_i(t_k)/dt$

Schedule wakeup at t_{next}

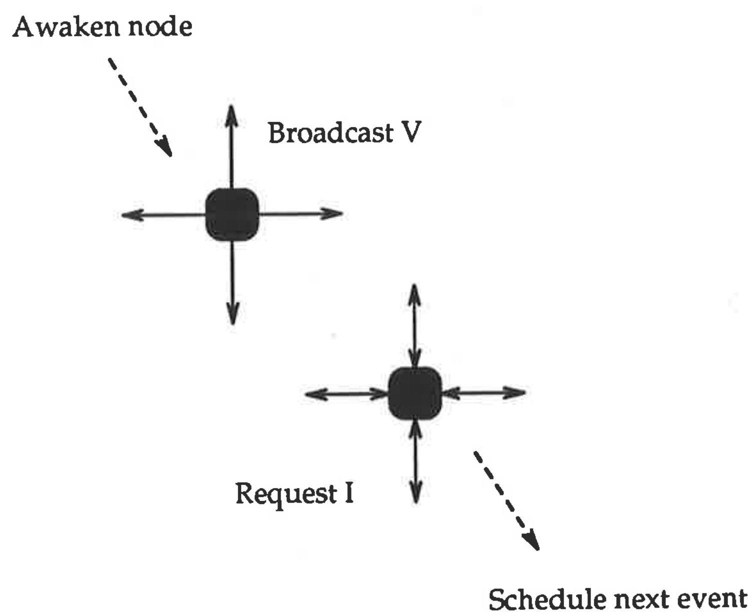


Figure 3.7: Node event/messages

This method is potentially workable, but as stated deliberately contains some instructive flaws. Firstly, given that current is only sampled occasionally, serious inaccuracy may result if a large current change occurs between awakenings of the node— an all too plausible example of which could be

with a CMOS inverter that is changing state and sampling occurs during the short period when the total current into the output node is small (giving a small $dV_i(t_k)/dt$ which implies a long wait till t_{next}), by which time the devices could be fully on/off again (worse still, $dV_i(t_k)/dt$ may actually be zero). These difficulties may be avoided by requiring that there be an interval t_{step}, which serves as an upper bound on the rescheduling time. Sadly, this means that truly inactive nodes will still require regular processing. A better approach is to notify nodes of important changes in current when they occur, rather than waiting for the node to reawaken— however this raises a second difficulty...

The broadcast-and-sum scheme arises out of an aggressive attempt to minimize storage consumption— an implementation could reduce the per node information to a number j which implies $V_i(t_k) = Voltage_j$ and a number $x \in \{-1, 0, 1\}$ which implies $V_{next} = Voltage_{j+x}$. Unfortunately, this is at the expense of recalculation of $dV_i(t_k)/dt$ (a potentially lengthy computation if node i is part of a well-connected bus) and the restriction that nodes are only ever awakened in response to self-scheduled "awaken for voltage transition" events. This inflexibility prevents nodes being awakened by current change messages, as a node may receive such messages while it is in transition between $Voltage_j$ and $Voltage_{j+x}$— at which point the node *does not necessarily contain enough information to determine its instantaneous voltage.*

Clearly, this is unacceptable. Nodes, and indeed all simulation objects *should not rely on assumptions about the messaging discipline to preserve internal information.*

Certainly in the case of nodes, the most natural semi-minimal set of the information necessary for a hybrid mode simulation is V_i, dV_i/dt, and the *local time at the node* t_i. With these, if a node is "unexpectedly" awakened

at time t_{msg}, a simple recovery procedure can be applied—

Algorithm 3.3 *Node voltage recovery*

If $t_{msg} > t_i$ *then*

$V_i \leftarrow V_i + (t_{msg} - t_i)\ dV_i/dt$

If $V_i < Voltage_0$ *then* $V_i \leftarrow Voltage_0$

If $V_i > Voltage_N$ *then* $V_i \leftarrow Voltage_N$

$t_i \leftarrow t_{msg}$

—establishing an invariant condition for the node. In fact, a local time property is useful in varying degree to all the simulation objects, usually for this same purpose of maintaining information independently to the event + message discipline.

A significant advantage of retaining the per node dV_i/dt value is that rather than completely recalculating it at every event, it may be updated *incrementally*— if fanin objects send $\Delta I_{f,i}$ messages containing the change in current along the object's branch to node i. This scheme is now better named broadcast-or-receive (algorithm 3.4).

Algorithm 3.4 *Broadcast-or-receive*

On awakening at time t_k

Recover voltage

If message is from oneself then

Send $V_i(t_k)$ *messages*

Else a $\Delta I_{f,i}$ *message has arrived*

Receive $\Delta I_{f,i}$

$dV_i/dt \leftarrow dV_i/dt + \frac{1}{C_i}\Delta I_{f,i}$

(given $Voltage_l < V_i < Voltage_u$*)*

If $dV_i/dt > \epsilon$ *then*

$t_{next} = t_i + (\,Voltage_u - V_i)\ /\ dV_i/dt$

Else if $dV_i/dt < -\epsilon$ *then*

$t_{next} = t_i + (\,Voltage_l - V_i)\ /\ dV_i/dt$

Else

$t_{next} = never$

Schedule wakeup at t_{next}

Note a slight refinement of broadcast-or-receive— nodes are considered *quiescent* when ($-\epsilon < dV_i/dt < \epsilon$ where ϵ is "small") and are not scheduled to be reawakened. *Implementation Note:* In Loge, ϵ is under user control, allowing quiescence to be matched to the properties of a specific simulation— one could for example set ϵ to mean "less than 1mV change in a clock period".

Broadcast-or-receive represents the middle ground in node messaging activity. Reductions in the number of broadcasts of $V_i(t_k)$ may be trivially achieved by reducing the size of the *Voltages* set— or possibly by discarding *Voltages* completely and *never scheduling self-awakening events for nodes.* For this extreme scheme to work, nodes must still do some broadcasting of the value of dV_i/dt whenever it changes. These events are noted by all a node's fanin and fanout objects, which use the information to schedule self awakenings— further consideration of this method will be postponed until the structure of these objects has been discussed.

In summary, having considered various hybrid mode event/message passing schemes, the following message types have arisen—

◁*Awake*▷ Received by nodes, from themselves, after a time interval.

◁*V=*▷ Broadcast by nodes to node fanout objects, stating the current node voltage.

$\triangleleft dV/dt{=}\triangleright$ Broadcast by nodes to node fanout objects, stating the current node time rate of change of voltage.

$\triangleleft I?\triangleright$ Broadcast by nodes to node fanout objects, requesting an $\triangleleft I{=}\triangleright$ be sent to the node at once.

$\triangleleft I{=}\triangleright$ Received by nodes, from their fanin objects, stating the current from object to node.

$\triangleleft \Delta I{=}\triangleright$ Received by nodes, from their fanin objects, stating that the current from object to node has changed by this amount.

$\triangleleft dV/dt?\triangleright$ Received by nodes, from their fanin objects, requesting that a $\triangleleft dV/dt{=}\triangleright$ be sent at once.

Functional Mode

The events and messages required for functional mode are relatively straightforward. Functional models must read and write digital (and perhaps even explicit analog) values from/to a module port. Reading will require a "request for voltage" message, unless node voltage broadcasts can be relied upon and be cached in the module/port structure. Writing can be performed with either $\triangleleft I{=}\triangleright$ or $\triangleleft \Delta I{=}\triangleright$ messages, but for full generality it is desirable that node properties be directly modifiable by modules (this allows, for example, experimental analog device models to be written as functional modules).

Also, to support event driven functional modelling, provision must be made such that a module can be awakened when a node changes logic state—specifically when its voltage crosses a logic threshold. This may be trivially achieved in a system where nodes broadcast $\triangleleft V{=}\triangleright$ messages by placing the logic thresholds in the *Voltages* set, or more specifically to *use a simplified*

Voltages set $\{Lo, V_{low_threshold}, V_{high_threshold}, Hi\}$. Such a node would generate $\triangleleft V{=}\triangleright$ messages of greater average relevance to functional components. Furthermore, there exist hybrid-mode algorithms that do not require nodes to generate $\triangleleft V{=}\triangleright$ messages at all— equivalent to the case where the *Voltages* set is empty. Similarly, there are nodes (notably the supply rails) that have constant voltage and need never broadcast. These requirements are met be defining node subtypes where algorithmic differences occur, and by making the *Voltages* set a node class property (allowing for example a very fine grain set to be used in vital sections of a circuit).

Thus the additional messages from functional mode are—

$\triangleleft V?\triangleright$ Received by nodes, from fanin objects, requesting that a $\triangleleft V{=}\triangleright$ message be sent at once.

$\triangleleft V{\leftarrow}\triangleright$ Received by nodes, from fanin objects, assigning the node voltage.

$\triangleleft dV/dt{\leftarrow}\triangleright$ Received by nodes, from fanin objects, assigning the node time rate of change of voltage.

Miscellaneous extensions

A trivial extension applicable to all node types is to model the substrate leakage current $I_{i,b}$. Two techniques of contrasting speed and accuracy suggest themselves—

1. For all nodes receiving insufficient supporting current schedule two events— one that degrades a *Hi* to X, and one that degrades X to *Lo* (after [Sherwood81]).

2. Explicitly model $I_{i,b}$ by modification to dV_i/dt— modelling accuracy is now dependent on the *Voltages* set.

A far more demanding problem is that of the breakdown of the assumption that a node is a atomic uniform object, as is becoming increasingly common in high speed interconnect nodes, particularly in aggressive technologies such as gallium arsenide. When transmission line effects begin to be significant, the computation necessary for accurate modelling becomes very heavy— an experimental node model based on techniques in [Canright86] was partially developed, but abandoned mainly due to discouragingly poor performance.

The other hindering factor was the immaturity of the available circuit extraction technology. What is needed are extractors that recognize nodes likely to exhibit transmission line effects, and emit some concise representation its the parameters— which is an awkward problem when typical bus nodes may have tens of arbitrarily shaped taps and corners.

Good modelling of fast distributed nodes is an open problem. The obvious scheme that may give good performance is to replace a distributed node by small nodes at each connection point, connected through a special multiport module which contains a matrix of characteristic delays between each terminal node (figure 3.8), by analogy with the S parameter matricies of field theory. These delays would be used to schedule events that effectively determine when a current change at point A on the distributed node begins to be felt at point B. How the elements of such a matrix are to be extracted and indeed whether such an approximation is reasonable is however unclear at this point.

Node Class

To summarize consideration of nodes, the node class-level properties are listed in figure 3.9.

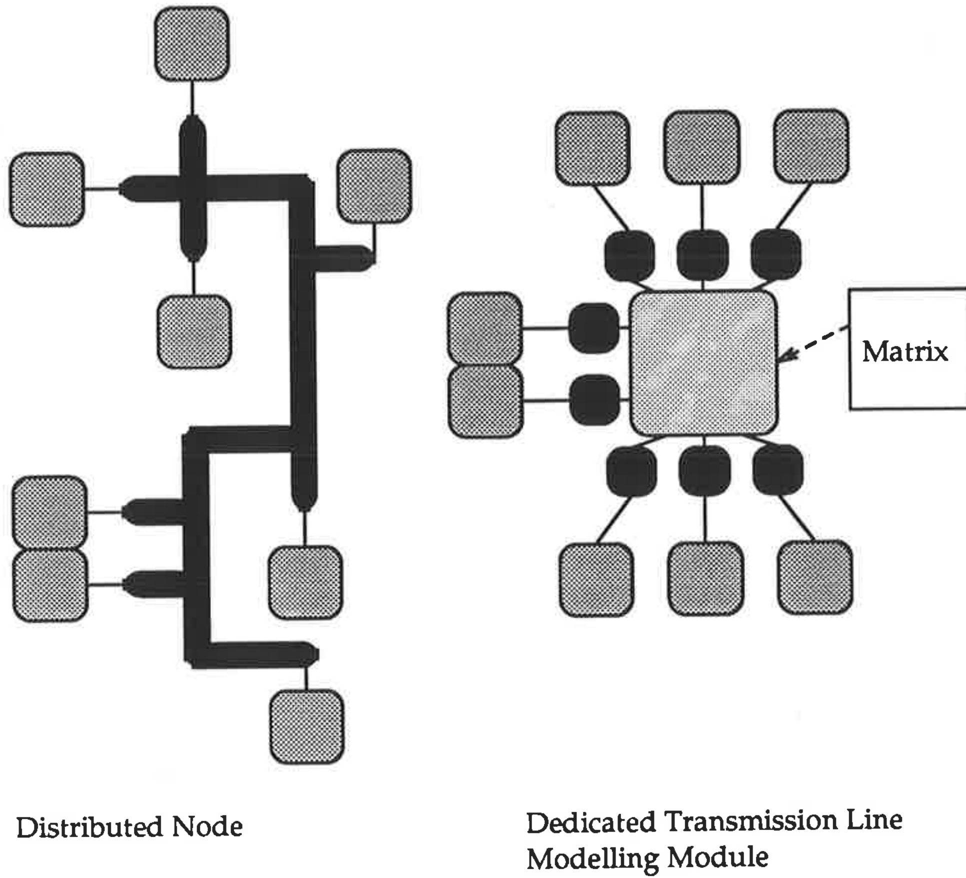


Figure 3.8: Distributed node model

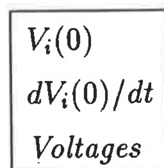


Figure 3.9: Node class

3.4.2 Ports

Ports are a combination of node concentrator, message filter, and a convenient "handle" for modules to read from and write to. The most important property of a port is its connections— both from port to parent module, and from port to connected nodes. Connections are per instance properties which are stored in the same manner as for nodes.

Unlike nodes however, the number of connections a port can have is fixed at its point of specification— this is the port *width* property. Port width could potentially be a class-level property, but this would be rather inefficient as there are many operations on ports of the form "for all bits of the port, do...", thus width is best placed within the port instance for swift access.

Signal typing

Ports are the obvious place to implement signal typing, as at a port there are information is transferred between modules and nodes with a clear *direction.* An information flow property leads to four fundamental *port types*—

Null No information may flow through a null port.

Input Node→Module transfers allowed.

Output Node←Module transfers allowed.

Ioput Information may flow through an ioput port in either direction.

With every port instance tagged with one of these types, all functional mode operations on ports can be checked— for example attempts by a module to write to an input port can be disallowed. Connections too may be checked— for example, when constructing nodes it is simple to check that every node is connected to *at least one input-capable port and exactly one*

output-capable port.[2] This check could be described as "Kirchhoff's Information Law", following a similar usage in [Jouppi83b]. However this initial form of node connection checking is too rigid, as although disallowing more than one output-capable port connection will help identify potential mistakes in functional models, it also disallows the common idiom of the bus with N writers. In practice such configurations are made safe by making the multiple output ports conditionally disconnectable with tristate circuitry. Loge follows this practice by adopting **Tristate** as a port type qualifier (that is, a port may be of type **Output-Tristate**— see figure 3.11), and the connection check becomes—

Every node must have connections to—

- At least one input-capable port (information sink).
- At least one output-capable port, either—
  - Exactly one pure **Output** port (unconditional information source).
  - Or at least one **Output-Tristate** port (conditional information source).

It is also worth performing some trivial checks on port connections— every bit of a non-**Null** port should be connected to a node, while no bits of a **Null** port should be connected. This makes a useful guarantee to the runtime system, and strengthens the value of **Null** ports as markers for "Not Yet Implemented" sections of models.

[2]These rules can be simply extended to allow for devices.

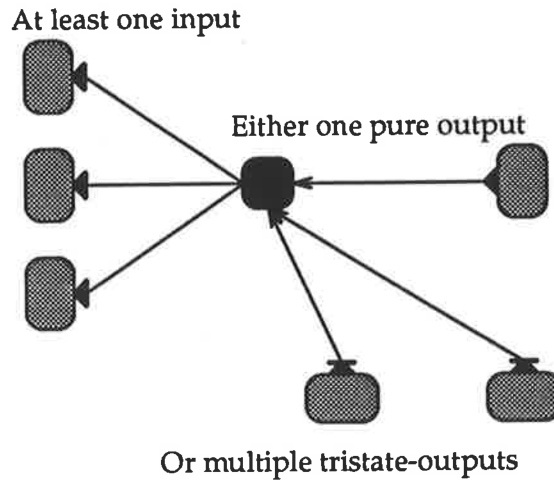


Figure 3.10: Kirchhoff's Information Law

Before considering other port properties an important restriction must be made. Ports are intended to provide a module with a uniform bitwise interface to potentially many nodes— it is assumed that *the properties of this interface are the same for all the bits* with the exception of the "value" of each bit. Thus, while it is possible to write *Hi* to bit 0 and *Lo* to bit 1 of a two bit port, it is not possible that bit 0 be of ioput type and bit 1 of output with tristate type— the type property applies to the whole port.

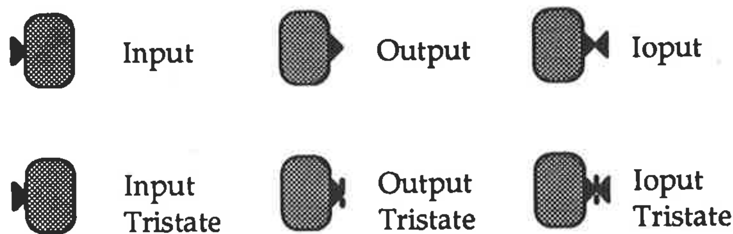


Figure 3.11: Port types

This restriction (while suggested by implementation considerations of time and space efficiency) intentionally strengthens the type system to impress upon the model author that ports should consist of physically similar bits— if the properties of two bits are significantly dissimilar then they should be placed in separate ports.

Current modelling

Since nodes contain voltage information it is natural that ports should contain current information. Detailed analog current modelling is supported with primitives to assign a specified current to a particular bit of a port— although such modelling would be better done at the device level. For the more common digital applications, the following simpler model is adequate— a read-only port is defined to draw no current (a reasonable assumption in the case of MOS implementation technology), while the currents generated by output ports are constrained to a set of constant currents corresponding to the standard logic states $\{I_{Hi}, I_{Lo}, I_X, I_Z\}$. Of these, I_Z is trivially zero amps, while I_{Hi} and I_{Lo} are user specified constants supplied through the port class.

I_X is inherently awkward— one might choose to implement it as a random current $I_{Lo} \leq I_X \leq I_{Hi}$. However for the purposes of functional simulation this is rather unusual— functional models tend to write specific quantities. Therefore Loge disallows writes of X, (without loss of functionality, as the same result can be achieved with the explicit use of a random number generator within a functional model).

In some cases, notably when parameters of the implementation technology are known, it is more convenient to specify the data-driving currents in terms of *port resistances* R_{01}, R_{10} where—

$$I_{Hi} = (V_{dd} - V_{ss})/R_{01}$$
$$I_{Lo} = (V_{ss} - V_{dd})/R_{10}$$

—following the example of [Terman83]. A further simplification is the case where $R_{01} = R_{10} = R$, where R is an average resistance for the port. Indeed this will probably be more the rule than the exception in functional simulation where the physical structure of the port is unknown.

In functional mode, a more natural method of deriving these currents is by specifying the maximum delay between logic transitions on a typical node. Given that logic levels are defined by thresholds (equation 2.10) and assuming that modules are notified of changes in logic state when attached node voltages cross the thresholds $V_{low_threshold}$ and $V_{high_threshold}$, then the currents are—

$$I_{Hi} = C_{delay}(V_{high_threshold} - V_{ss})/T_{01}$$
$$I_{Lo} = C_{delay}(V_{low_threshold} - V_{dd})/T_{10}$$

—where C_{delay} is a "typical" or worst-case node capacitance, and the T_{**} are the desired maximum delays. For functional mode models this is superior to the previous scheme as it focuses on the transition points where message generation can occur.

Ports and events

Given the mapping of logic level to current, writes to ports are translated to currents at the individual bits. A tradeoff occurs here between simply

sending $\triangleleft I{=}\triangleright$ messages on every write, or retaining enough state information to detect non-zero changes in currents, which give rise to $\triangleleft \Delta I{=}\triangleright$ messages. The inefficiency of $\triangleleft I{=}\triangleright$ mechanisms have been demonstrated, but the state overhead (which must be an instance-level property) is non-trivial (width $\times$ sizeof(current) bytes), although this only applies to output-capable ports. The balance is decisively tipped towards the state-retaining approach by the extra opportunity it gives for event suppression— for example if the current numeric output of an N bit port is $0\ldots00_2$, if this is changed to $0\ldots01_2$ then the resulting message generation may be reduced from N $\triangleleft I{=}\triangleright$ messages to a single $\triangleleft \Delta I{=}\triangleright$ message from bit 0.

A similar filtering is possible in the opposite direction. Functional modules are usually intended only to be sensitive to changes in logic level, thus while input-capable ports may receive $\triangleleft V{=}\triangleright$ messages, they need only notify their parent module of events that cross a logic threshold. Once again there is a storage overhead (applicable to input-capable ports only), and a counter-argument on the basis of message suppression. In this case it was decided to conserve storage as these messages can be suppressed by using the modified nodes of page 98.

Even more events can be suppressed, if whole ports can be *dynamically ignored*. Consider the case of the module M in figure 3.12, which has one quickly varying input I, an "enable" input E, and an output O that mirrors I only when the module is enabled. During the long periods when E disables O, the value of I is irrelevant to the function of the module— O can not change until E is asserted, therefore it is highly wasteful for the port I to be continually notifying M of transitions. Thus if it is possible for M to advise port I to cease notification until further notice (an *ignore* operation), the next event the module will receive will correspond to the eventual assertion of E, at which point port I may be *watched* again.

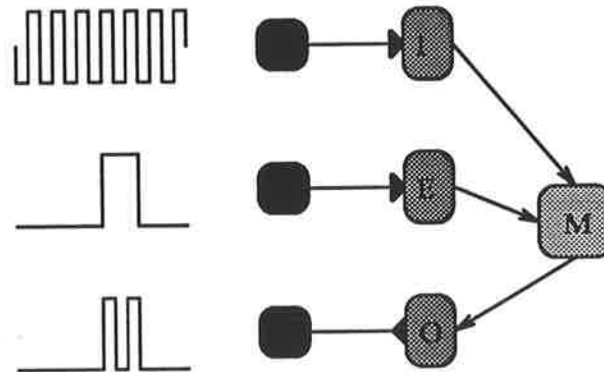


Figure 3.12: Dynamic event watching

The consequence of this is that port instances must contain a *watch / ignore* flag. This handles the case of logic transitions, which are communicated to the port by ◁*V*=▷ messages. Some lower level modules may also need to selectively receive notification of ◁*dV/dt*=▷ messages (for example a module which monitors a node and plots its waveform in time— which thus requires notification of changes in the slope of the node voltage).

Section 3.3.2 mentions the desirability of collapsing several closely occurring node logic transitions into a single event at the module interface. This can be achieved by the port preemptively scheduling an event for itself after a short delay whenever a non-ignored transition occurs on one of its bits. The preemption effect will allow further transitions within the delay (known as the *hysteresis delay*) to cause the port event to be repeatedly rescheduled until finally no transition occurs before the event arrives (in the form of a ◁*Awake*▷ message to the port). On receiving this message, the port's parent module is at last notified of a change of input. Note that if the module is awakened by some other means during a sequence of transitions on a port, any reads from the port will be almost certain to find some bits in the X

state— this condition can be trivially detected, revealing a potential data hazard.

Modules and ports

A deliberate ambiguity in the discussion so far is the communication method between modules and their ports. The obvious technique of defining more message types exclusive to this pair of simulation objects is quite workable, but at least for a uniprocessor implementation this is unnecessary as all module-to-port and most port-to-module communication is inherently instantaneous (a port is very much a part of its parent module, unlike nodes and devices). Therefore no interaction with the scheduler is required, and the port↔module communication reduces to a procedure call.

The complete list of module↔port interactions is—

- Module changes port watch/ignore state.
- Module writes a number to the port. The number is disassembled into bits and may give rise to $\triangleleft \Delta I {=} \triangleright$ messages.
- Module writes to a single port bit (may generate $\triangleleft \Delta I {=} \triangleright$).
- Module disconnects from the port— enters tristate mode (may generate $\triangleleft \Delta I {=} \triangleright$).
- Module writes an analog current to a single port bit (may generate $\triangleleft \Delta I {=} \triangleright$).
- Module writes the analog voltage or its derivative of the node beyond a port bit (generates $\triangleleft V {\leftarrow} \triangleright$ or $\triangleleft dV/dt {\leftarrow} \triangleright$).
- Module reads a number from the whole port in response to awakening ($\triangleleft V? \triangleright$ messages unless voltage caching is present).

- Module reads a number from a port bit (a $\triangleleft V?\triangleright$ message).
- Module reads an analog current value from a single port bit.
- Module reads the analog voltage or its derivative of the node beyond a port bit (generates $\triangleleft V?\triangleright$ or $\triangleleft dV/dt?\triangleright$ messages and replies).
- Port awakens a module with new data.

Summary

To conclude the discussion of ports, the structure of port instances and classes is shown in figures 3.13 and 3.14.

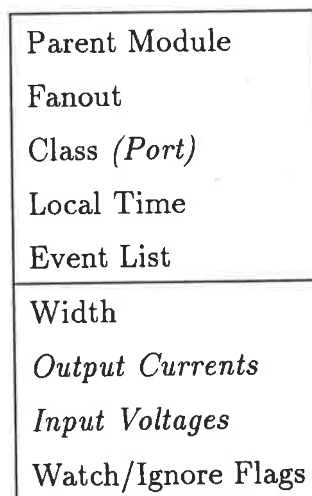


Figure 3.13: Port instance

3.4.3 Modules

Instances of modules have two roles, corresponding to the functional and structural modes. Their functional role is as the focus of all HDL execution,

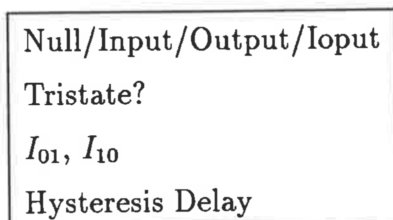


Figure 3.14: Port class

while structurally a module is the container object that forms the trunk of the instance hierarchy tree. Module definitions are likewise split between HDL code definition, and specification of how to construct the module's child instances. However there is considerable blurring between these categories— for example it is desirable for functional models to contain per module local variables, the declaration and storage of which is a structural issue; and again— when making a structural specification of the ports, nodes and other objects that a module contains, it is useful to give each object a distinct name for functional code to refer to it by. Interdependent issues like these helped to prevent the Loge HDL from splitting into inelegantly distinct functional and structural sublanguages and justify the initial decision to provide a unified HDL.

From the point of view of a module instance, the progress of a simulation is a succession of executions of functional code within its private context, interspersed with periods of inactivity while other modules execute. This is recognizable as a coroutine or continuation [Abelson+85] paradigm— although in the case of Loge, *context switching* is perhaps the most appropriate description, where a module's context is its set of instance properties.

Implementation Note: Modules v Lisp

There are a number of places where a Lisp-based HDL can lead to performance degradation beyond that purely due to interpretation. Consider the module instance level properties, which fall into three main categories—

- The module's connections, to its parent (if any) and to contained instances.
- Such module local variables as the functional model declares— for example a counter module must have somewhere to store its current count.
- Simulation related variables, such as the module local time.

Of these, only the local variables present any difficulty when context switching— because they are Lisp variables a series of interpreter internal symbol un/bindings must be made. Effort has been taken to make this operation quick (in particular, to avoid consing on context switch— otherwise garbage collection becomes a major overhead).

Functional modelling code must be able to refer to the components of the module, such as its ports. This can be most simply implemented by Lisp extensions whereby the four main simulation object types are also Lisp object types, whereupon the internal instances of a module may be handled the same as any other module local variable.

Unfortunately, this would bring the instances within the clutches of the Lisp garbage collector. As garbage collection causes Lisp objects to be moved around in memory—

- Any locality of reference that derives from the module hierarchy would be progressively diluted.

- Simulation objects are implicitly highly interconnected. Therefore every time an object is moved, a potentially large number of pointers would need to be updated (objects with large fan-in such as bus nodes are a pathological case).

— which leads to the conclusion that a direct equivalence of simulation objects and Lisp objects is impractical.

The problems with garbage collection of instances do not apply to class objects as they are relatively few, and not highly interconnected. Thus the HDL includes extended Lisp types for class definitions of nodes, ports, modules and devices.

The hierarchy of simulation object instances should be "compiled" into one place and left there for the duration of a simulation. If so, a Lisp object to simulation-object correspondence can be supplied by a special *reference* Lisp object that simply contains a constant pointer to a fixed simulation object, or better still, to a group of related objects— such as a vector of modules or array of nodes, all of which are referred to together through a single named Lisp symbol. See figure 3.15. The reference object can of course be garbage collected cheaply— although given that module instances may contain pointers to Lisp objects (local variables, child instance references, parent class) some secondary collection must occur (only once per module instance though). What has been lost is a quick means to perform the reverse mapping from simulation object to Lisp object— the need for which is minimal.[3]

[3]Currently only in the "find Lisp name of this object" routine.

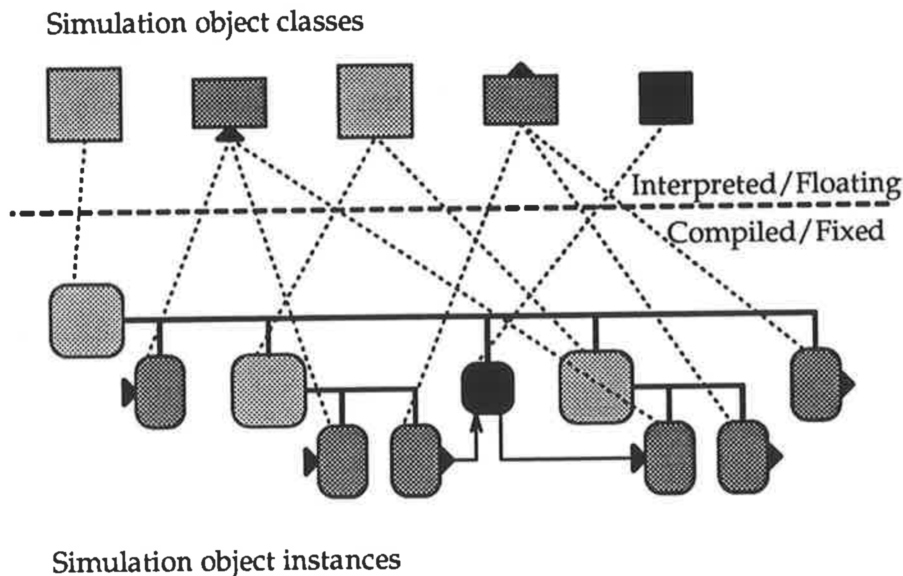


Figure 3.15: Lisp class types, fixed object instances

Building modules

Highly flexible module construction requires parameterized models— ranging from a simple integer parameter N used to define port widths, on through parameters containing module type specifiers, to even more exotic cases. Since parameterization causes module definitions to resemble functions, this leads to the forms—

(defmodule *<name>* *<lambda-list>* *<forms>*...)

(instantiate *<module-defn>* *<number>* *<arguments>*...)

—which are somewhat analogous to Lisp's DEFUN and FUNCALL.

In fact, `INSTANTIATE` also accepts port, node and device class definition objects as its second parameter— it is the fundamental primitive that compiles simulation object instances from their defining class objects. Typically, `INSTANTIATE` is called once to create the top level module instance, and indirectly some arbitrary number of times as submodules are recursively created, using depth first traversal (depth first instantiation combined with a simple sequential placement of instances in memory preserves such locality of reference that the module hierarchy supplies).

When instantiating a port, the *<arguments>* it expects are a list of nodes to connect to, and the list must be the right length. Devices are similar. The *<number>* parameter to `INSTANTIATE` is a special case, it may either be `NIL`, a number `N`, or a list of two numbers `(N1 N2)`, which indicates a single instance, a vector of N instances, or an array of $N1 \times N2$ instances is to be created respectively. Ports are an exception— ports should always have a specified width, and as vectors and arrays of ports do not appear to be very useful, only the number `N` form is accepted where N is the port width rather than a vector length.

Still unspecified are the *<forms>* accepted by `DEFMODULE`. There are at several pieces of modelling code that must be supplied—

Build code The code that controls instantiation of all internal objects of the module.

Init code Code to run every time a simulation is started— this is useful, for example, to initialize module local variables.

Run code The actual code that models the run time behaviour of the module. This is usually the most complicated section.

Done code Code to run when a simulation is finished— for example, code to close module specific log files.

Thus with the addition of a little syntactic sugar, a two input AND gate can be specified as—

```
(defmodule and2 (A0 A1 O)
  (build
    (A0 I-port 1 A0)
    (A1 I-port 1 A1)
    (O  O-port 1 O))
  (init
    (watch-ports A0 A1)
    (ignore-ports O))
  (run
    (port-write O
      (logand (port-read A0) (port-read A1))
    ))
  (done)
)
```

BUILD is currently a macro that effectively converts its argument lists—

From:

(<*name*> <*defn*> <*number*> <*args*>)

...

To:

```
(setq
```

```
    <name> (instantiate <defn> <number> <args>)
    ...)
```

— such that at build-time instantiations occur sequentially within the module, and preorder overall. Some simple type checking is performed, including checking that <*name*> is present on the module's lambda list.

Note the convention of using the parameters `A0 A1 O` to module `AND2` to do double service— on instantiation they are single member lists of nodes, but as the build code executes, the gate ports are instantiated with each parameter in turn first serving as an argument list and then being rebound to the instantiated port (where it may later be used for port read/write operations in the run code). This convention is not mandatory— separate parameters `A0-PORT` and `A0-NODES` could have been supplied— however it is a simple trick to reduce the length of argument lists, and will be used frequently in later examples.

Modules and events

Real systems exhibit a variety of functional behaviour, but can be broadly classified as asynchronous, synchronous, self-timed and mixtures thereof. This section contains simple example modules which manipulate the event system to achieve these modes of operation.

The AND gate is an example of a purely asynchronous model— new data arrives at a port, the module is awakened and executes its run code which possibly generates new output. Interestingly, purely synchronous models have similar run code— consider an N bit pre-loadable clocked counter model—

```
(defmodule counter (n out clk count &aux lim)
  (build
```

```
    (out O-port n out)
    (clk I-port 1 clk))
  (init
    (watch-ports clk)
    (ignore-ports out)
    (setq lim (1- (ash 1 n))))
  (run
    (if (= (port-read clk) 1)
      (port-write out
        (setq count (if (>= count lim) 0 (1+ count)))
      )))
  (done)
)
```

Many purely synchronous modules are recognizable by init code of the form—

```
...(watch-ports <clock-ports>)
  (ignore-ports <all other ports...>)...
```

The run code similarity between the AND gate, the counter and indeed any fully synchronous module is due to the fact that the function of these modules as specified by the run code is insensitive to the activity of its ports—the AND gate "does not care" which of its ports caused it to be awakened, while the synchronous modules can only ever be awoken by their clock port. If however it was known that the inputs to the AND gate would change at vastly different rates (as was the case with the system of figure 3.12) the following is a more efficient implementation—

```
(defmodule fast-slow-and-2 (fast slow out)
  (build
    (fast I-port 1 fast)
    (slow I-port 1 slow)
    (out O-port 1 out))
  (init
    (watch-ports fast slow)
    (ignore-ports out))
  (run
    (let (
          (slow-in (port-read slow))
          (fast-in (port-read fast)))
      (if (equalp slow (the-port))
        (if (= 0 slow-in)
          (ignore-ports fast)
          ;else
          (watch-ports fast)))
      (port-write (logand slow-in fast-in))))
  (done)
)
```

This illustrates the more typical case, where the function of a module varies with the port that awakened it— as supplied by the function THE-PORT.

Modules may also need to be self timed— implying that the module performs its own scheduling. For example, here is a modification to the N bit counter such that it produces output at regular intervals without recourse to a clock input.

```
(defmodule counter-source (n out count period &aux lim)
  (build
    (out O-port n out))
  (init
    (setq lim (1- (ash 1 n)))
    (ignore-ports out)
    (awaken-after 0))
  (run
    (write-port out
      (setq count (if (>= count lim) 0 (1+ count))))
    (awaken-after period))
  (done)
)
```

Notes: (awaken-after <*time interval*>) is the preemptive scheduling primitive. This modelling mode can be combined with the general multi-port style as THE-PORT returns NIL when a module is awoken by a self-scheduled event. The &AUX form defines LIM to be a module local *non-parameter* variable.

Hierarchical construction

The examples so far are all semi-leaf modules containing nothing but nodes and ports. Submodules are specified with a similar syntax— here is a four bit AND gate, using the two input AND gate of page 115 as a submodule—

```
(defmodule and4 (A0 A1 O &aux m0 m1 m2 no)
  (build
```

```
    (A0 I-port 2 A0)
    (A1 I-port 2 A1)
    (O  O-port 1 O)
    (no node   2)
    (m0 and2   ()
      (list (node A0 0) (node A1 0) (node no 0)))
    (m1 and2   ()
      (list (node A0 1) (node A1 1) (node no 1)))
    (m2 and2   ()
      (list (node no 0) (node no 1) (node O 0)))
    )
  (init)
  (run)
  (done)
)
```

Notes: The connections of the AND2's ports are specified from the overlying *alias* ports of AND4 through explicitly constructed parameter lists for each submodule. The NODE form extracts a node from node references, lists of nodes, or by tracing port connections.

Implementation Note: Parameter passing

The parameter passing from parent to child module instance in the four bit AND example is quite straightforward, but there only single submodule instantiations are performed. When instantiating a vector of modules a list (or vector) of argument lists must be supplied, and for an array of modules, a list of lists (or array) of argument lists. These complete submodule

parameter expressions can easily become excessively tedious to write— fortunately the underlying interpreter allows the definition of helper macros such as MODULE-VECTOR—

```
(module-vector <index variable>
  ...
  at <n> <argument list>
  over <n1> <n2> <argument list>
  ...)
```

The form of MODULE-VECTOR is intended to exploit the regularity of connection usually seen in vectors of modules. Often it is possible to write a general expression in terms of an index variable for the connections of most if not all of the submodules instantiated as a vector. Even for quite complicated module types, the most common case is that only the first and last module instances have special case connections. A characteristic example of this effect is an N bit adder with carry chain. Assuming an adder cell type—

```
(defmodule adder-cell (IN0 IN1 CIN SUM COU)
  (build
    (IN0 I-port 1 IN0) ; data input
    (IN1 I-port 1 IN1) ; data input
    (CIN I-port 1 CIN) ; carry input
    (SUM O-port 1 SUM) ; data output
    (COU O-port 1 COU) ; carry output
    ...)
```

—then an N bit adder built of ADDER-CELLs is specified—

```
(defmodule adder-N (n I0 I1 CI O CO &aux vn a)
  (build
    (I0 I-port n I0)    ; data input
    (I1 I-port n I1)    ; data input
    (CI I-port 1 CI)    ; carry input
    (O  O-port n O)     ; sum output
    (CO O-port 1 CO)    ; carry output
    (vn node (1- n))    ; internal nodes
    (a  adder-cell n (module-vector i
        at 0 (           ; LSB with carry in
          (node I0 i) (node I1 i) (node CI)
          (node O i)  (node vn i))
        over 1 (1- n) (; General case
          (node I0 i) (node I1 i) (node vn (1- i))
          (node O i)  (node vn i))
        at (1- n) (     ; MSB with carry out to port
          (node I0 i) (node I1 i) (node vn (1- i))
          (node O i)  (node CO))
      )))
  ...)
```

Note how the special case first and last cells correspond to an AT clause of the MODULE-VECTOR form, while the general middle case is covered by the OVER clause. There is an analogous form for arrays of modules.

Module modes

In the four bit AND gate example, the module run code was null— just `(run)`. This is effectively an admission that this module is purely a structural module— it is inherently never a leaf module as it contains submodules, which are to be used in aggregate to provide the functional behaviour of this module, which is one way of stating that `AND4` *always operates in structural mode.*

Implications of structural mode on port connectivity An unsophisticated method of realizing structural module operation would be to detect cases of null run code and internally replace it with awakenings of the underlying submodules. This technique is flawed firstly in that the order that submodules are awakened may introduce undesirable artifacts into the simulation, and secondly in that it requires that submodule ports be somehow prevented from receiving events— either by the existence of an extra level of connection between parent port and submodule port (figure 3.16), or by a disabling mechanism.

The internal port connection method of figure 3.16 above stems from a misguided attempt to strictly enforce hierarchical encapsulation such that no submodule is connected beyond the boundary of the parent module except through a parent port. A more natural solution is for module and submodule ports to always be connected directly to their nodes (figure 3.17), but to use the mode of each module to *activate / deactivate* its child ports. Purely structural modules such as `AND4` never need be awakened (indeed it is wasteful if they are), therefore a structural mode module always deactivates its ports, causing them to ignore all events. Active functional modules such as the `AND2` instances will always activate their ports.

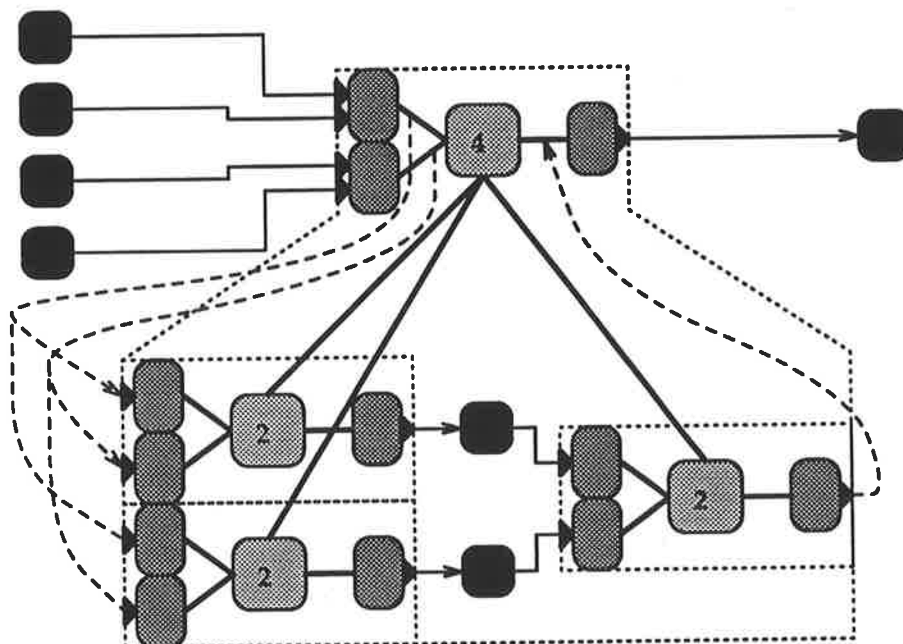


Figure 3.16: Submodule port internal connection

Of course, modules with submodules need not be purely structural— they may have run code allowing them also to operate in functional mode. For example—

```
(defmodule and4 (A0 A1 O &aux m0 m1 m2 n)
  (build
    ...)
  (init
    (watch-ports A0 A1)
    (ignore-ports O))
  (run
    (port-write O
      (if (= 3 (logand (port-read A0) (port-read A1)))
```

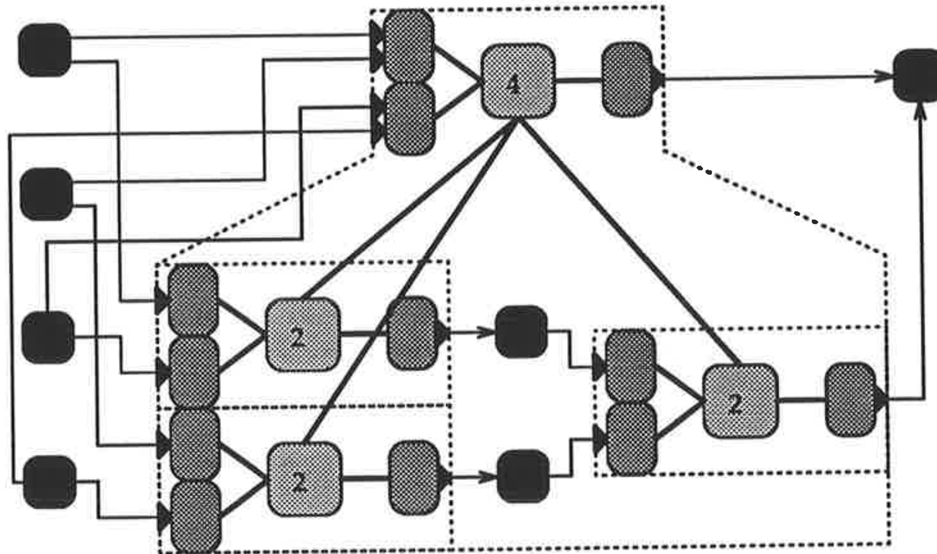


Figure 3.17: Submodule port direct connection

```
        1 0)))
  (done)
)
```

The simulation wavefront

With the existence of modules capable of multi-mode operation some primitives to manipulate the mode are required. There are now three modes to distinguish between— structural, functional and the state of being an inactive child module to an active functional parent module— such modules are labelled inactive or *non-structural* (as it would make no difference whether the child is present or not). Hybrid mode fits neatly into this set as a special case of functional mode if one considers a device to be a predefined module. Thus in any valid simulation, within the module instance hierarchy there will

be a (possibly empty) layer of structural mode modules, above a (non-empty) layer of functional modules, above a (possibly empty) layer of non-structural modules. The layer of active functional modules is known as the *simulation wavefront*— figure 3.18 shows an instance hierarchy for an eight bit AND gate built from `AND4` and `AND2` modules, operating in an assortment of modes.

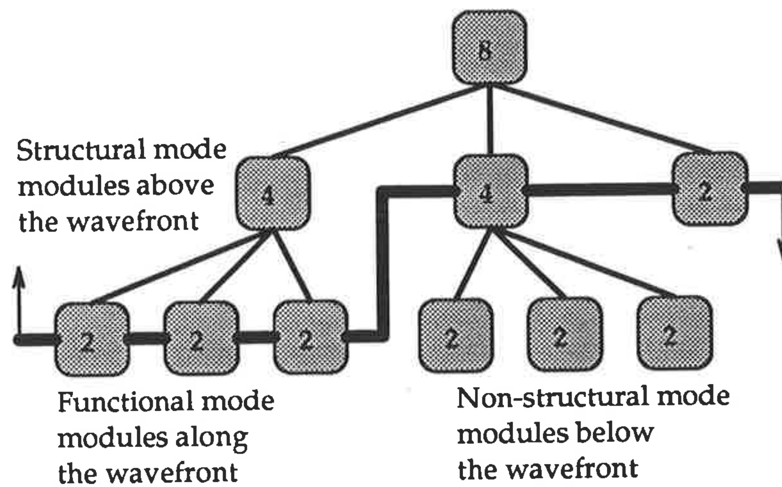


Figure 3.18: Module modes and the simulation wavefront

There are constraints on the possible set of modules that may make up a valid simulation wavefront. One way of expressing these constraints is that—

Root The root of the module instance tree must be either in structural mode, or functional mode (a degenerate case).

Structural If a module is in structural mode, its child modules must be in either structural or functional mode, and there must exist at least one such child.

(Consequence to **Structural**: If a module is in structural mode, its parent if any is also in structural mode.)

Functional If a module is in functional mode, its children if any must be in non-structural mode.

(Consequence to **Structural** and **Functional**: If a module is in functional mode, its parent if any is in structural mode.)

Non-structural If a module is in non-structural mode, its children if any must be in non-structural mode.

(Consequence to **Functional** and **Non-Structural**: If a module is in non-structural mode, its parent is either in functional mode or non-structural mode.)

(General consequence: The leaves of the module instance tree are in functional or non-structural mode).

The constraint set allows the definition of rules by which the simulation wavefront set may be changed. Note that the rules fire recursively up and down the module instance tree stopping only when either no change need be propagated or a previously encountered instance is reached.

When set Structural Fail if this module has no children. Otherwise set all children to functional mode, set parent to structural mode.

When set Functional Fail if this module has null run code. Otherwise set all children to non-structural mode, set parent to structural mode.

When set Non-structural Fail if this module has no parent. Set all children to non-structural mode. Set parent to functional mode.

With these rules, the act of changing one of the non-structural `AND2` modules of figure 3.18 to functional mode would cause its parent to become structural, and thus for the other non-structural `AND2` to become functional.

With simulation mode thus firmly entrenched as a module instance property, its access and modification are built into the HDL as a Lisp SETF-able form—

```
(setf (module-mode <module>)
  [ :structural | :functional | :inactive ])
```

As an added precaution, pure structural modules can use the function `(structural-module-only)` as run code, which will cause an error when executed, trapping a poor run time placement of the simulation wavefront. Module modes may be changed "on the fly" during simulation.

Init and Done

Typical tasks performed in the init code of a module are—

- Set module mode
- Set Watch/Ignore status of ports
- Write an initial output value to a port
- Self driven modules queue their initial event
- Initialize local variables

Of these, setting the module mode can cause difficulties if there are many such attempts from various places in the instance hierarchy— the best solution is often to only set module modes in the init code of the root instance. Nevertheless, if multiple mode settings are desired, the process will at least be deterministic given that initialization occurs on preorder traversal of the

instance tree. To complement the init code, done code is executed via depth first traversal.

Done code is less frequently used. It can be thought of as a module's private version of the Lisp `UNWIND-PROTECT` form, and thus has similar applications, such as guaranteeing files are closed, and other general postprocessing tasks.

Instance and class contents

The properties of module instances and classes are now complete in figures 3.19 and 3.20.

Module instance
Parent Module
Fanout
Class *(<type>-Module)*
Local Time
Event List
Mode
Lisp variables

Figure 3.19: Module instance

3.4.4 Devices

The final type of simulation object is the device. There are several subtypes of devices, which are subdivided firstly by simulation algorithm, and secondly by technological differences. Devices may be thought of as hard-coded special purpose modules with their ports retracted inside the module periphery,

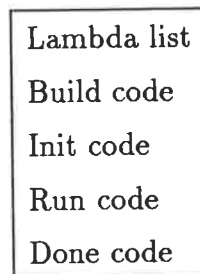


Figure 3.20: Module class

thus they connect only to nodes (figure 3.21). MOS technology is assumed throughout this chapter, however as will be seen in section 4.3 much of the discussion is relevant to gallium arsenide technology too.

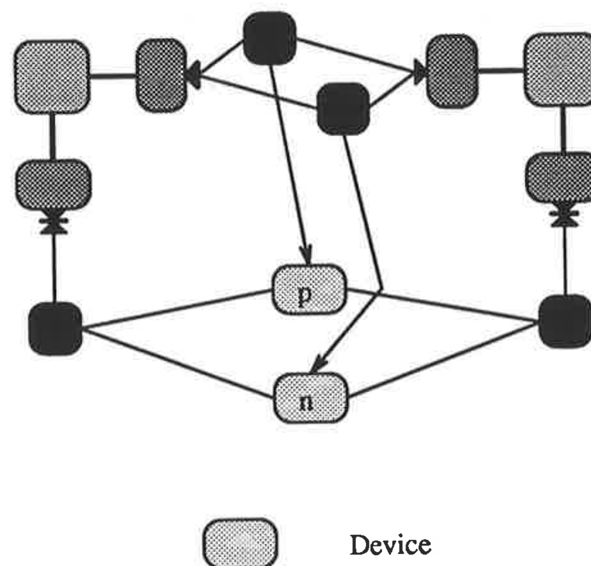


Figure 3.21: Device connections

Devices interact with the simulation wavefront exactly as if they were modules, except that as devices must be leaves of the instance tree they will

never be in structural mode. Thus, devices become active (that is, effectively in functional mode) when their parent module enters structural mode.

Device Properties

Since devices subsume the functions of ports, they must contain inter-node current information. In MOS technology the strong isolation of the FET gate reduces this requirement to the single current I_{ds}. If $\triangleleft \Delta I = \triangleright$ events are generated, similar opportunities for message suppression are available as applied to ports— sufficiently small changes in current may be ignored.

Implementation Note: Also common between ports and devices is the question of whether to cache node voltage values that arrive in $\triangleleft V = \triangleright$ messages. In this case the tradeoff is clearer— in a system containing devices, they are likely to occur in large numbers, and as extra storage for just three voltages would approximately double the size of a device instance it is thus far better for devices to rely on $\triangleleft V ? \triangleright$ messages to access terminal voltages.

The other property common to all devices is "width"— not in the sense of a number of bits as was the case with ports, but a measure of the physical shape of the device where this is relevant to its current driving capacity. *Implementation Note:* Width could be a class level property, but as it is likely to be deeply embedded in current calculations it is implemented as an instance level property for efficiency.

Each of the algorithmic subtypes of devices have their own specific instance and class level properties. These subtypes will now be individually discussed.

Evaluated Devices

Evaluated devices are straightforward models that implement the standard hybrid-mode algorithm as revised from the flawed form of algorithm 3.2. They are further subdivided by technology and technology-specific class level parameters (for example a CMOS circuit may contain devices of a class `evaluated-pMOS-1.5u`).

All evaluated devices respond to $\triangleleft V{=}\triangleright$ messages by gathering the new instantaneous values of their external voltages, and performing some calculation of I_{ds} as a function of the voltages. If the new value of I_{ds} has changed sufficiently, $\triangleleft \Delta I{=}\triangleright$ messages are sent to the drain and source nodes. Events are never directly scheduled for evaluated devices— they rely entirely on $\triangleleft V{=}\triangleright$ traffic to awaken them. Thus the speed/accuracy of an evaluated device is largely a function of the *Voltages* set of its terminal nodes— voltage waveform detail can be increased semi-arbitrarily with finer thresholds, with a proportional cost in generated events.

The exact details of the I_{ds} calculation are technology dependent, and also vary within particular instances of a technology. This variation is achieved by hard-coded routines for each specific type of device (for example *<simple nMOS>*) wherein some of the parameters of the calculation are class level properties of the device (for example, one type of nMOS device may have a different threshold voltage to another).

MOS The baseline device technology used in Loge is MOS, particularly CMOS. A convention that subsequently proves useful is to treat the gate as the reference node in the device equations— this simplifies the inherent ambiguity between source and drain nodes that bedevils the simulation of MOS devices. Thus the standard first-order MOS equations—

$$I_{ds} = \begin{cases} 0 & V_{gs} - V_{thr} \leq 0 \\ \frac{\beta}{2} V_{ds}(2(V_{gs} - V_{thr}) - V_{ds}) & 0 < V_{ds} < V_{gs} - V_{thr} \\ \frac{\beta}{2}(V_{gs} - V_{thr})^2 & 0 < V_{gs} - V_{thr} < V_{ds} \end{cases} \quad (3.1)$$

—become—

$$I_{ds} = \begin{cases} 0 & V_{sge} \geq 0 \\ \frac{\beta}{2}(V_{sge}^2 - V_{dge}^2) & V_{sge} < V_{dge} < 0 \\ \frac{\beta}{2} V_{sge}^2 & V_{sge} < 0 < V_{dge} \end{cases} \quad (3.2)$$

—where $V_{sge} = V_s - (V_g - V_{thr})$ and $V_{dge} = V_d - (V_g - V_{thr})$.

Assuming constant V_{thr} and $\frac{\beta}{2}$, a calculation of simple MOS I_{ds} costs two or three tests ($V_d > V_s$, $V_{dge} > 0$, $V_{sge} > 0$), two to four subtractions, and zero, two or three multiplications (ignoring the effects of device size), which still amounts to a significant computational load when applied to large numbers of devices.

Evaluated devices therefore provide accuracy that is ultimately limited only by the fidelity of the device equations implemented, at the expense of slow turnaround even for the above simple MOS equations.

Explicit Regional Devices

Use of V_{sg} and V_{dg} suggests that the operation in time of MOS devices can be visualized as a pointer drawing dots on the (V_{sg}, V_{dg}) plane. In a pure time-step based simulator the path would be nearly continuous, while in an event driven simulator larger jumps are made. At every point in this plane (except the impossible regions $V_{dg} - V_{sg} > 1$ and $V_{sg} - V_{dg} > 1$) there will be a characteristic value of I_{ds} (figure 3.22)— thus the pointer may alternately be visualized as following the contours of a three dimensional surface defined by points $(V_{sg}, V_{dg}, I_{ds} = f(V_{sg}, V_{dg}))$.

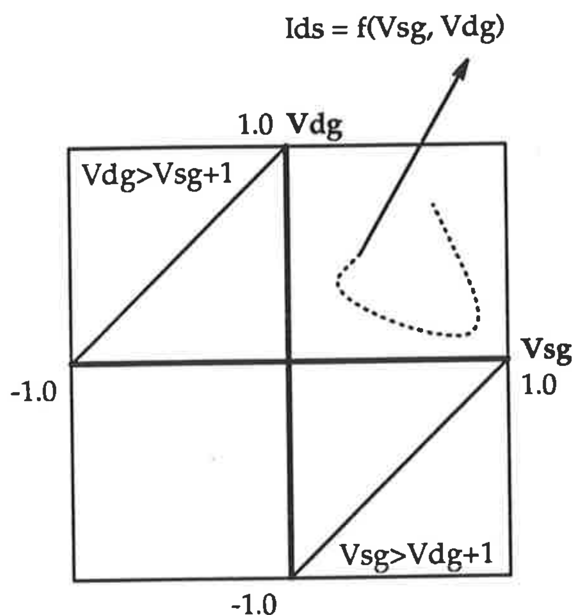


Figure 3.22: The V_{sg}, V_{dg} plane

For most I_{ds} functions there will be *regions* in the plane where I_{ds} has common behaviour— for example the saturation, linear and cut-off regions defined in the MOS I_{ds} equation (figure 3.23).

The MOS device has regions within which I_{ds} varies considerably. For sufficiently complex I_{ds} equations, evaluation may be simplified or expedited if each device records which region it is currently operating in. Changes of region can be handled by explicitly scheduling an ◁*Awake*▷ event for a device that is approaching a region boundary. For this procedure to be worthwhile, the speedier access to I_{ds} must outweigh the overhead of handling region changes, thus there is strong pressure for regions to be geometrically simple. This technique is not immediately applicable to evaluated MOS devices, as it would amount only to replacing three tests by an eight-way branch— this type of regional mode will in general only benefit technologies with an I_{ds}

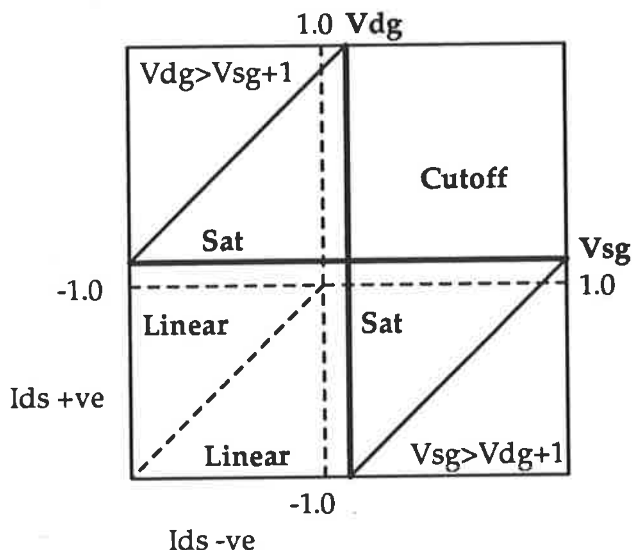


Figure 3.23: nMOS evaluated device regions

equation whose computational requirements vary wildly across its regions.

If however, one is prepared to divide the (V_{sg}, V_{dg}) plane into regions in which variation in I_{ds} is ignored such that a single, precomputed value for I_{ds} may characterize the whole region, then the region technique becomes much more generally applicable. This is effectively the method used in [Ruan+85], where the basic I_{ds} for each device is constrained to the set $I_{ds} = \{-I_2, -I_1, 0, I_1, I_2\}; I_1 < I_2$, depending on the terminal voltages. In terms of the (V_{sg}, V_{dg}) plane, this amounts to the five region system of figure 3.24 (translated from figure 2.19). Similarly the single on/off current model of [Terman83] is a effectively a three region scheme.

With the assumption of constant current within regions, the procedure to calculate the time when a device leaves a particular region is now independent of I_{ds}, except inasmuch as I_{ds} will influence dV_d/dt and dV_s/dt. In general, if a device is in region r defined by n edges in the form—

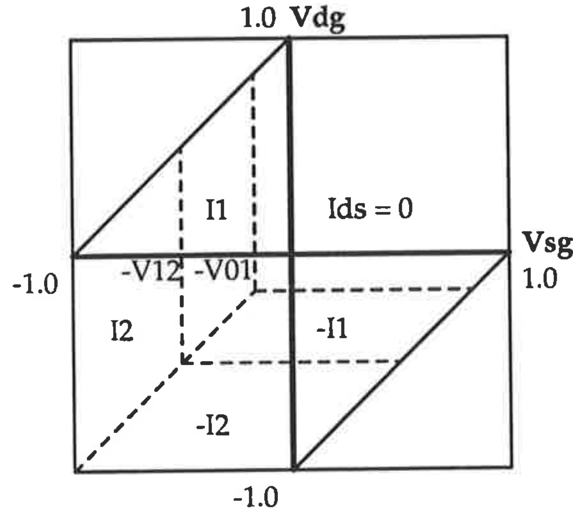


Figure 3.24: Five current region explicit model

$$k_{r,1,j}V_{dg} + k_{r,2,j}V_{sg} + k_{r,3,j} = 0 \tag{3.3}$$

—then the time at which the device next reaches a region boundary t_{next} is—

$$t_{next} = t_i + \min_{j\in[1,n]} -\frac{(k_{r,1,j}V_{dg} + k_{r,2,j}V_{sg} + k_{r,3,j})}{(k_{r,1,j}dV_{dg}/dt + k_{r,2,j}dV_{sg}/dt)} \tag{3.4}$$

In practice it is seldom necessary to do more than two calculations of this form— usually a quick test of the signs of dV_{dg}/dt and dV_{sg}/dt will uniquely identify one or two potential boundaries. Computation can also be short-circuited if the composite dV/dt values (the denominator term) has too small an absolute value, implying quiescence and a ridiculously long delay to t_{next}. Another simplification is that it is unnecessary to test for the possibility of leaving the encompassing ($-1.0 \leq V_{sg} \leq 1.0, -1.0 \leq V_{dg} \leq 1.0$) region or entering impossible regions. Also, the k coefficients defining region

boundaries are often degenerate—

$$\forall_{x\in\{1,2,3\}}\, k_{r,x,j} \in \{0, \pm 1\}$$

—thus the complexity of a t_{next} calculation is typically around three add/subtracts, two multiplications and (sadly, unavoidably) one division. Only the division separates the complexity of a t_{next} calculation from that of a MOS I_{ds} evaluation.

Note however that t_{next} may be invalidated by changes in dV_g/dt, dV_d/dt or dV_s/dt— a region-based device should recalculate its region crossing time on receipt of $\triangleleft dV/dt\!=\!\triangleright$ messages. Nevertheless, since a dV/dt changes in response to changes in an I_{ds}, which in turns requires that somewhere a device has changed region, such reschedulings should not be significantly more frequent than region changes except where node fanout is high. *Implementation Note:* An effective practical technique is that when an explicit device receives a $\triangleleft dV/dt\!=\!\triangleright$ message, it preempts any distant events and reschedules itself to be awakened after some fixed short delay (a class level property). On awakening, it will discover that it is still in the same region so no $\triangleleft \Delta I\!=\!\triangleright$ messages need be sent, but the normal region exit calculations may proceed. This scheme provides damping against pathological situations where storms of $\triangleleft dV/dt\!=\!\triangleright$ events appear on one or more device terminals (this is common when interfacing to sections of circuit composed of analog devices, for example).

Assuming constant I_{ds} across a region clearly reduces simulation accuracy, usually in near direct proportion to the size of the region. In return, event traffic is aggressively minimized to a similar degree. Indeed it is difficult to see how any further reduction is possible beyond that achievable with explicit regional devices with very small numbers of regions— in figure 3.25 the a brief simulation of a CMOS inverter composed of explicit six-region

devices is shown. This simulation is functionally correct, in seven device events—

1. t=0: The pull-up device enters its high current region.
2. t=1.25ns: The pull-up switches off, having pulled the output voltage high.
3. t=2.05ns: The input began to rise at 2.0ns, so now the pull-down enters its low current region.
4. t=2.06ns: With the continued rise in input the pull-down enters its high current region.
5. t=2.09ns: By now the output has fallen a little, but as the input is still fairly low the pull-up device returns to its low current region.
6. t=2.14ns: The input is now high, so the pull-up switches off again.
7. t=3.07ns: The output is now low, and the pull-down switches off.

"Bucket-Brigade" Oscillation Unfortunately, restriction of I_{ds} exacerbates the the difficulties caused by bidirectional devices. For example in the CMOS NOR gate of figure 3.26, consider the event sequence when initially input B is high (thus nodes N and O are low), and input A changes from high to low, switching on device P1. Node N will begin to rise, and soon the device P2 will switch on and current will flow through to the output node. Unfortunately, with simple device regions, it is now likely that node O is rising *faster* than node N, and as soon as $V_O = V_N$ device P2 will switch off again, whereupon only node N will continue to rise, causing P2 to switch on... This oscillatory "bucket-brigade" effect can occur wherever

drain-source connected groups of devices occur, and is an inevitable consequence of the combination of gross explicit regions and bidirectional devices. (In [Ruan+85] a palliative scheme where connected node groups are "collapsed" together is used. Unfortunately such schemes require extra storage in the node and/or device instances.)

Despite this oscillatory flood of events, these simple modes still give quick yet functionally correct results for a wide range of circuits. Circuits that rely on precise charge sharing effects are the most likely to fail, but this is hardly surprising, since the degree of modelling abstraction is approaching that of

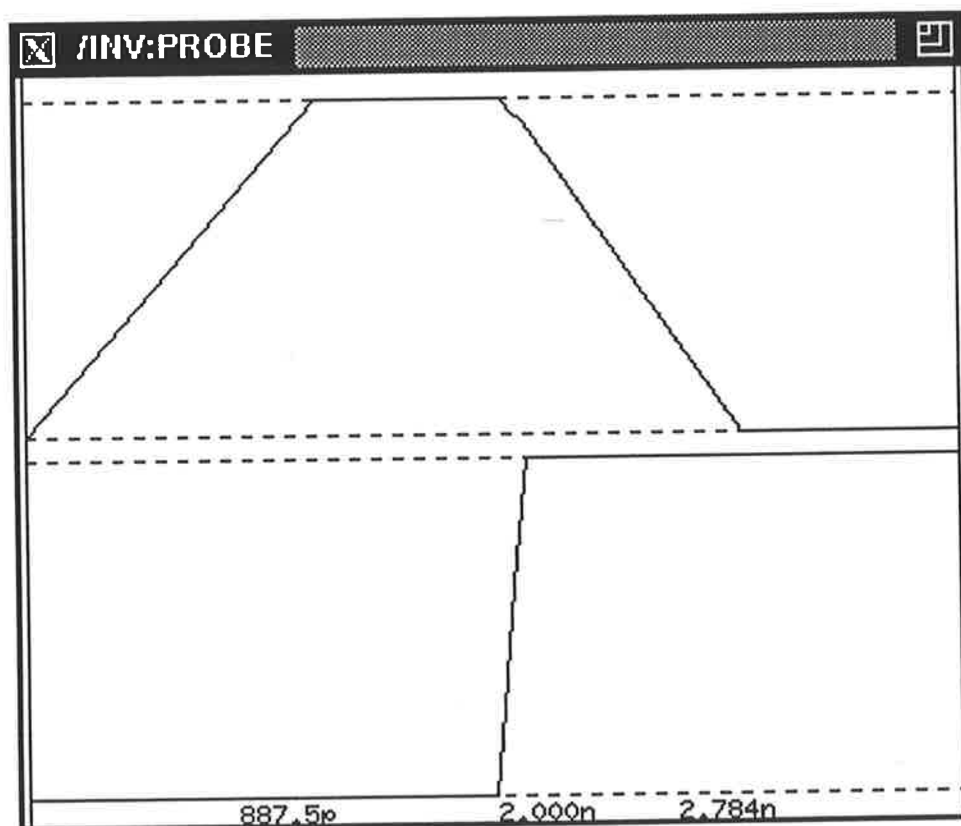


Figure 3.25: CMOS Inverter Simulation (explicit six region devices)

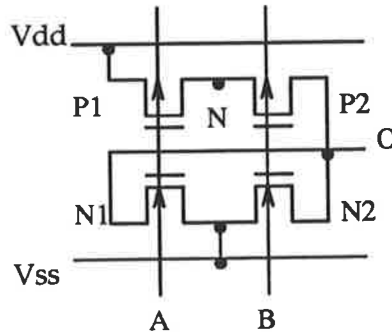


Figure 3.26: CMOS NOR gate

digital simulation where such failures are inevitable.

Fortunately, bucket-brigade oscillation may be reduced significantly by dynamically shifting the region boundaries such that the edges of the current region seem to recede by some amount (effectively by changing $k_{r,3,j}$)—causing all t_{next} values to increase. This adds a degree of *overshoot* such that region changes are noticed some time after they actually occur. The practical upshot of this is to reduce the frequency of bucket-brigade oscillations, at the expense of spurious superimposed node voltage oscillations of greater amplitude. For example in the NOR gate simulation (figure 3.27) 223 device events occurred, 175 of which were for device P2(!), with the voltage overshoot parameter set to 10mV. Increasing overshoot to 20mV decreased the P2 events to 92 (in 138) without discernible effect on the output waveform.

In nMOS logic the presence of continually conducting load devices can cause an active pull down device to exhibit bucket brigade oscillation. Once again, the amount of oscillation can be reduced, but unlike the previous case, the current being transferred does not cause any change in external conditions which eventually allow the oscillation to cease. Without a solution to this problem, applicability of hybrid device algorithms to such technologies

is seriously reduced. Unfortunately, all solutions require yet further per object storage and inelegant subclassing of nodes and/or devices (for example one could have a separate model for explicit six region devices connected to V_{ss} and a node with pull up). Worse still, this subclassing is insufficiently general— convert the inverter to a NAND gate and the problem reappears.

One of the less ugly solutions is to create another special type of node— a *clamped node.* This node differs from other nodes in that it "lies about its voltage" to some devices— the node voltage never appears to fall below a clamped value sufficiently above the V_{ss} rail to keep explicit devices from

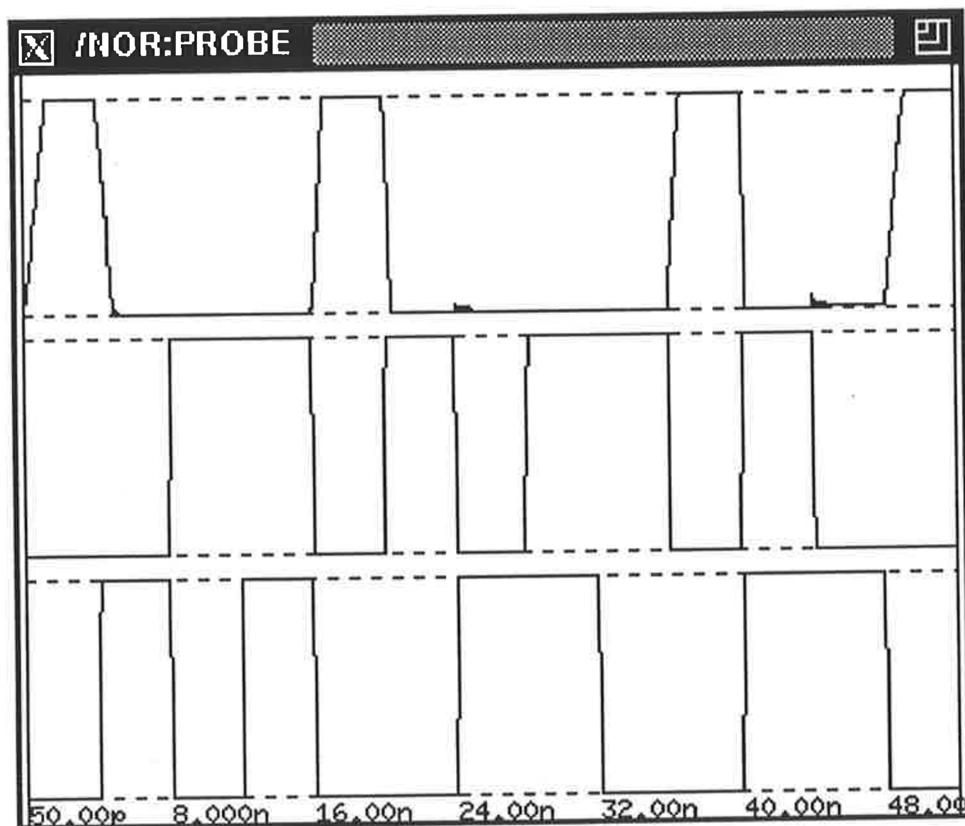


Figure 3.27: CMOS NOR Simulation (explicit six region devices)

switching off. With some care in the choice of this value, many devices can be persuaded to cease oscillating between regions, although event generation may still be fairly high.

Six region devices While much of the preceding discussion has referred to the five-region model of figure 3.24, the presented simulations are all of a six region model. The reason for this procedure is that there is a practical flaw in five region devices— without meticulous attention to accuracy in t_{next} calculations, five region devices are prone to oscillate between the adjacent regions of positive and negative current without ever passing through a state where current is zero, producing nonsensical results. There are two reasons for this problem—

- In general with regional devices it is wise not to allow great variations in I_{ds} between adjacent regions. The larger the difference in I_{ds} across a boundary, the greater the chance of oscillations developing there.
- When calculating t_{next} with the aim of exactly reaching a region edge, it is possible that due to the approximations inherent in floating point arithmetic the t_{next} value will be slightly too small— thus on the next awakening the region will not quite have changed (so an event has been wasted), and the distance to the region edge is now very small, such that the difference between the local time t_i and a newly calculated t_{next} will be *extremely* small, stretching the adequacy of floating point yet again... and so on until t_{next} is equal to t_i. To avoid this numerical collapse, a simple, robust solution is to insist that some small but non-zero amount of overshoot is used.

Thus five region devices have boundaries with high variation of I_{ds}, and are absolutely guaranteed to overshoot from one to another in a robust im-

plementation. This prompted the development of a six region model, where a narrow diagonal region of $I_{ds} = 0$ is added to separate the regions of opposite current (figure 3.28).

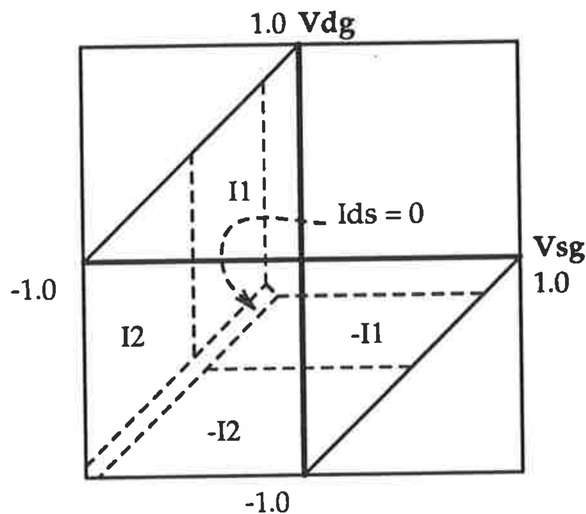


Figure 3.28: Explicit six region devices

The width of the new region is a class level property of the explicit device. In the simulations presented so far it slightly more than 20mV. Clearly this region is an inherently fictional construct— real MOS devices do not totally cease to conduct when $0 < | V_{ds} | \leq 10$mV although I_{ds} need not be large. A sad consequence of this inaccuracy is that it may potentially cause voltage drops of up to 10mV across devices where no difference is expected— therefore this parameter should be small. On the other hand, the width of this region is an effective upper bound on the amount of overshoot that may be used to combat bucket-brigade oscillations— for example in the CMOS NOR gate example, increasing the overshoot to be greater than the width of the diagonal region causes device P2 to "punch through" between positive

and negative current regions— exactly what was to be avoided. So, for good control of oscillations, a wide diagonal region is desirable. There is no clear optimal solution to this conflict— therefore overshoot and region boundaries are user-controllable.

Summary Explicit region devices are computationally cheapest of all hybrid mode devices, and the least accurate. While capable of extremely low event generation, the combination of heavy approximation and bidirectional devices can cause event storms, although these may be controlled to some extent. An important point in favour of explicit devices is that they concentrate the scheduling and current calculations purely within the scope of the device, rather than relying on regular messages from external nodes as is the case with evaluated devices. They are also quite adaptable to alternate technologies— events may be minimized by carefully choosing the shape of the device regions.

Sampled Devices

Evaluated and explicit regional devices are the speed and accuracy extremes of hybrid mode simulation. Sampled devices provide semi-continuous variation between these two extremes. Given an I_{ds} equation as used in evaluated devices, the (V_{sg}, V_{dg}) plane is divided into a mesh, samples of I_{ds} taken at each mesh intersection, about which square (or rectangular) regions are constructed— figure 3.29 shows a sampled device only slightly more complex than a explicit six region device.

As sampled devices still use regions of constant I_{ds}, simulation follows exactly the same algorithm as for explicit devices (stated shortly as algorithm 3.5), except in the routine for determining the instantaneous device region— with explicit devices this can usually be most efficiently determined

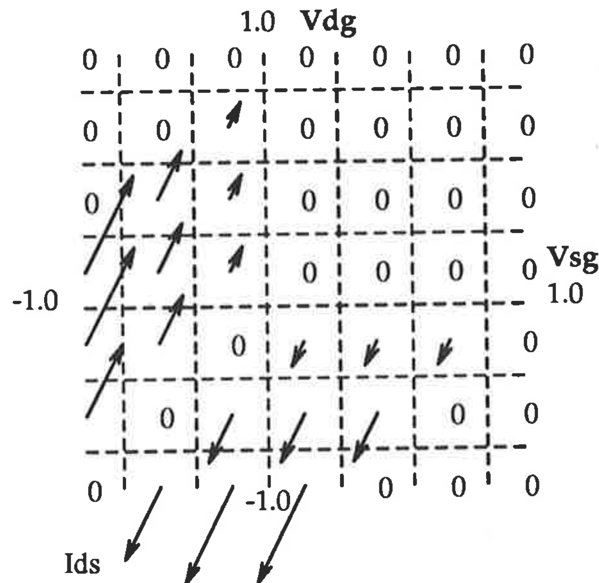


Figure 3.29: Current sampling

with a series of absolute tests of V_{sg} and V_{dg}, but with sampled devices the quickest method is to test whether the V_{sg} and V_{dg} boundaries of the region the device is currently operating in have been crossed.[4] With regional devices the customary test for a significant change in I_{ds} can reasonably be reduced to an equality test— note how many adjacent regions have equal characteristic I_{ds} in the arrays for devices of a (fictitious, scaled voltage) CMOS process shown in figures 3.30 and 3.31.

These I_{ds} arrays have a secondary use in suggesting good choices of region geometry for explicit regional devices. They are generated (at semi-arbitrary degrees of sampling) by a simple companion utility.

Although the arrays have 256 regions, they contain only 29 distinct values

[4]Methods involving division of voltages by the grid spacing require less storage but are slower.

0.0	0.0	0.0	0.0	0.0	0.0	0.0	21.2u	21.2u	20.7u	19.0u	16.1u	11.9u	6.59u	0.0
0.0	0.0	0.0	0.0	0.0	0.0	14.6u	14.6u	14.6u	14.1u	12.4u	9.50u	5.36u	0.0	−6.59u
0.0	0.0	0.0	0.0	0.0	9.22u	9.22u	9.22u	9.22u	8.74u	7.05u	4.14u	0.0	−5.36u	−11.9u
0.0	0.0	0.0	0.0	5.08u	5.08u	5.08u	5.08u	5.08u	4.60u	2.91u	0.0	−4.14u	−9.50u	−16.1u
0.0	0.0	0.0	2.16u	2.16u	2.16u	2.16u	2.16u	2.16u	1.69u	0.0	−2.91u	−7.05u	−12.4u	−19.0u
0.0	0.0	474n	474n	474n	474n	474n	474n	474n	0.0	−1.69u	−4.60u	−8.74u	−14.1u	−20.7u
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	−474n	−2.16u	−5.08u	−9.22u	−14.6u	−21.2u
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	−474n	−2.16u	−5.08u	−9.22u	−14.6u	−21.2u
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	−474n	−2.16u	−5.08u	−9.22u	−14.6u	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	−474n	−2.16u	−5.08u	−9.22u	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	−474n	−2.16u	−5.08u	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	−474n	−2.16u	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	−474n	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Figure 3.30: Sampled pMOS device current array

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	1.38u	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	6.31u	1.38u	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	14.8u	6.31u	1.38u	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	26.9u	14.8u	6.31u	1.38u	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	42.5u	26.9u	14.8u	6.31u	1.38u	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
61.7u	42.5u	26.9u	14.8u	6.31u	1.38u	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
61.7u	42.5u	26.9u	14.8u	6.31u	1.38u	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
60.4u	41.1u	25.5u	13.4u	4.93u	0.0	−1.38u	−1.38u	−1.38u	−1.38u	−1.38u	−1.38u	−1.38u	0.0	0.0
55.4u	36.2u	20.6u	8.50u	0.0	−4.93u	−6.31u	−6.31u	−6.31u	−6.31u	−6.31u	−6.31u	0.0	0.0	0.0
46.9u	27.7u	12.1u	0.0	−8.50u	−13.4u	−14.8u	−14.8u	−14.8u	−14.8u	−14.8u	0.0	0.0	0.0	0.0
34.9u	15.6u	0.0	−12.1u	−20.6u	−25.5u	−26.9u	−26.9u	−26.9u	−26.9u	0.0	0.0	0.0	0.0	0.0
19.2u	0.0	−15.6u	−27.7u	−36.2u	−41.1u	−42.5u	−42.5u	−42.5u	0.0	0.0	0.0	0.0	0.0	0.0
0.0	−19.2u	−34.9u	−46.9u	−55.4u	−60.4u	−61.7u	−61.7u	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Figure 3.31: Sampled nMOS device current array

of I_{ds}. Performance of 256 region sampled devices is still quite good with respect to six region explicit devices, and the smaller steps between regions reduces the potential for oscillation— but not entirely as it is once again possible to oscillate between regions of opposite current across the line $V_{dg} = V_{sg}$. Therefore sampled devices may also need certain regions to be deformed to accommodate an explicit $I_{ds} = 0$ diagonal region as shown in figure 3.32. This is of course unnecessary if sampling is sufficiently fine that the $V_{dg} = V_{sg}$ diagonal is "insulated" on at least one side by $I_{ds} = 0$ regions.

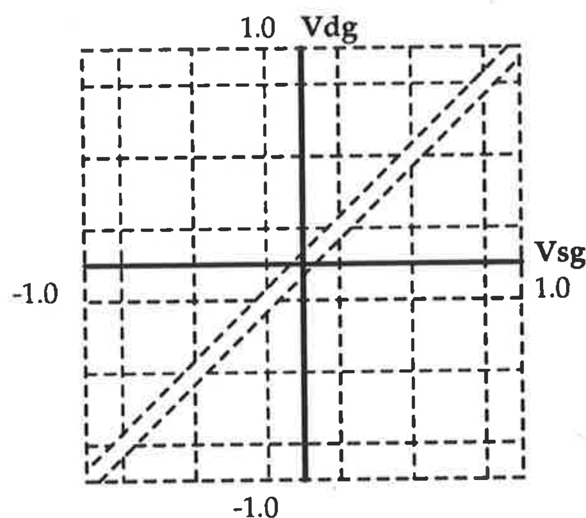


Figure 3.32: Corrected sampled device region geometry

More extreme sampling methods are possible— for example for technologies where I_{ds} varies rapidly in particular places one could contemplate a secondary sampling of the relevant regions— thus the I_{ds} array becomes a quadtree.

To conclude consideration of devices that use regions of constant I_{ds}, here is the regional device algorithm—

Algorithm 3.5 *Regional Device*

On awakening at time t_k

Determine region

If region has changed then

  If $I_{ds,new} \neq I_{ds,old}$ *then*

    Send $\triangleleft \Delta I = \triangleright$ *messages to source and drain*

  Update region

Examine dV_{sg}/dt, dV_{dg}/dt *and if approaching a region boundary then*

  Calculate t_{next}

Else $t_{next} = never$

Schedule wakeup at t_{next}

3.4.5 Analog Devices

By way of contrast Loge includes quasi-analog device models. These resemble evaluated devices in calculation method, but rather than relying on $\triangleleft V = \triangleright$ messages they follow algorithm 3.6.

Algorithm 3.6 *Analog Device*

On awakening at time t_k

Calculate I_{ds} *(using detailed equation)*

Calculate $\Delta I_{ds} = I_{ds,new} - I_{ds,old}$

If $\mid \Delta I_{ds} \mid > I_\epsilon$ *then*

  Send $\triangleleft \Delta I = \triangleright$ *messages to source and drain*

Find $\Delta V_{max} = \max dV_g/dt, dV_d/dt, dV_s/dt$

$$t_{next} = \begin{cases} \text{never} & (\Delta V_{max} = 0) \\ t_i + t_{step} \times f(\Delta V_{max}) & \text{otherwise} \end{cases}$$

Schedule wakeup at t_{next}

This is essentially a "parallel"-analog mode, where instead of a characteristic single central time-step mechanism, each device controls its own local time step, where the minimum time step is t_{step} and $f(\Delta V_{max})$ is a integer valued function that returns a number loosely inversely proportional to ΔV_{max}.

3.4.6 Example

First, device classes must be defined with MAKE-DEVICEDEF—

```
(defconstant *std-explicit-pmos* (make-devicedef
    :explicit   ; subtype
    :pmos       ; ''technology''
    1.0         ; base width
    #e50p       ; hysteresis delay
    0.010       ; Vdiag
    0.014       ; V01
    0.4         ; V12
    0.01        ; Vovershoot
    '(#e0.0u #e15.0u #e7.50u #e0.0u #e-7.50u #e-16.0u)
                ; Ids
))
```

A simple form to switch between device subtypes is—

```
(set-device-subtype [
  :ANALOG | :EVALUATED | :EXPLICIT | :SAMPLED
  ])
```

Note: This form binds standard symbols **ENN** and **ENP** to the appropriate *<device-defn>*. With these primitives in place, a device-level model of a CMOS inverter can be defined as—

```
(defmodule CMOS-inverter (I O &aux de dp)
  (build
    (de enn () '(1 ,I ,O ,vss))
    (dp enp () '(1 ,I ,O ,vdd))
    )
  (init)
  (structural-module-only)
  (done)
)
```

Notes: Standard symbols VDD and VSS are assumed to be bound to appropriate constant nodes. The parameters to a device instantiation are a width, and three nodes in the order gate, drain, source.

3.4.7 Summary

Device classes vary considerably depending on the basic device subtype they define, however device instances are identical, (save that non-regional models do not use the device region field). Note that despite the considerable variety in Loge node and device subtypes, they may all be reliably intermixed, with the sole proviso that evaluated devices will need to be connected to nodes that broadcast $\triangleleft V\!=\triangleright$ messages sufficiently often. *Implementation Note:* The flexibility this provides is at some cost in code complexity— care must be taken to prevent messaging loops between interconnected objects (the usual

technique is to temporarily mark a message sender inactive, so that "ricochet" messages are damped).

Parent Module
Fanout
Class *(Device)*
Local Time
Event List
I_{ds}
Device width
(Device region)

Figure 3.33: Device instance

<table>
<tr><td>:ANALOG</td><td>:EVALUATED</td><td>:EXPLICIT</td><td>:SAMPLED</td></tr>
<tr><td colspan="4">Technology</td></tr>
<tr><td colspan="4">Hysteretic delay</td></tr>
<tr><td colspan="2">β</td><td>Region $k_{r,x,j}$</td><td>Region V_{sg},V_{dg}</td></tr>
<tr><td colspan="2">V_{th}</td><td colspan="2">Region I_{ds}</td></tr>
<tr><td>t_{step}</td><td></td><td colspan="2">Overshoot</td></tr>
</table>

Figure 3.34: Device classes

3.5 Simulation

This section provides further examples of Loge modules and features, by way of introducing a lightweight verification strategy.

3.5.1 Simulation control

Control is transferred from the HDL interpreter to the simulator proper by the `SIMULATE` function—

`(simulate` *<module-reference>* *<time>*`)`

—which implements algorithm 3.7—

Algorithm 3.7 **`simulate`**
Run connection type checks
Set $t = 0$, enable event queue
Schedule final event if any
Execute all init code
Run event driven simulation
Execute all done code
Clean up
Return <module-reference>

A simulation may be interrupted, its parameters modified, and continued at will. It is also possible to run open ended simulations where the duration is not specified. A simulation may be concluded at any time by calling `(terminate-simulation)`.

3.5.2 Tools

Experience with Loge has lead to a simple methodology for module development, simulation and testing summarized in figure 3.35. Whenever a module M is developed, one or more test bed modules (M-tester) are created, along with an HDL wrapper which initiates simulation of an M-tester. The tester

module must obviously contain a submodule of type M, which will require nodes to connect to its ports, which in turn require *source* and *sink* modules to provide test I/O and satisfy the node connectivity rules. Note that a tester module has no ports.

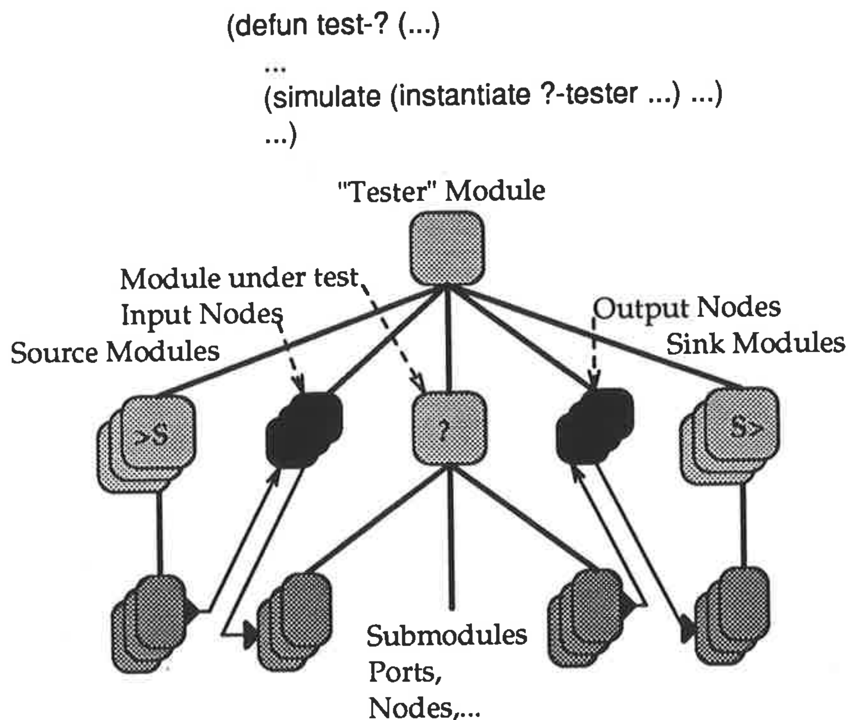


Figure 3.35: Loge methodology

Source

Some source-type modules have already been presented in section 3.4.3. The main sources in the Loge toolkit are a constant source, a counter (see page 119), and the general source—

```
(defmodule source (n0 0 period pattern &aux q)
```

```
  (build
    (O O-port nO O))
  (init
    (ignore-ports O)
    (setq q (apply #'make-queue pattern))
    (awaken-after 0))
  (run
    (let ((x (dequeue q)))
      (setf !O x)
      (enqueue q x))
    (awaken-after period))
  (done))

(defun clock-pattern (n)
  ; Return source pattern for N-phase clock
  ; e.g. '(0 1 0 2) for N=2
  ...)
(defun count-pattern (from to &optional (by 1))
  ; Return source pattern for counter that
  ; counts from FROM to TO by steps BY
  ...)
```

other patterns

Usage:

```
(defmodule M (... input-vectors ...)
  (build ...
    (cl source () (list ; 2 phase non-overlap 10ns
        2 (nodes cn) #e10n (clock-pattern 2)))
```

```
(co source () (list ; count down 15->0 and repeat
    4 (nodes xn) #e40n (count-pattern 15 0 -1)))
(so source () (list ; input vectors
    32 (nodes input-bus) #e40n input-vectors))
...)
```

Notes: By choice of `pattern` a general source can act as a constant, counter, clock, etc source (albeit inefficiently in some cases). The general source uses a hard coded Lisp extension, a *queue* type which allows swift enqueue and dequeue operations— access to source code again proves useful! This example also introduces the convenient shorthand whereby `(port-write` *<port>* *<value>*`)` can be written in the abbreviated form— `(setf !`*<port>* *<value>*`)` and `(port-read` *<port>*`)` becomes `!`*<port>*.

Sink

The dual of the `source` is the `sink`. The main types of sink are a null sink (provided merely as an alternate way of satisfying connection rules), a clocked (sampled) sink, and the general asynchronous sink—

```
(defmodule sink (nI I state
    &optional (func #'sink-print-fn))
  (build
    (I I-port nI I))
  (init
    (watch-ports I)
    (setq state (funcall func state I)))
  (run
    (setq state (funcall func state I)))
```

```
  (done
    (setq state (funcall func state I)))
)

(defun sink-print-fn (name port)
  (format t "At ~A: ~A = ~S~%"
    (timeprint) name (port-read port))
  name)
(defun analog-sink-print-fn (name port)
  (format t "At ~A: ~A = ~S~%"
    (timeprint) name (port-analog-read port))
  name)
```

Usage:

```
  (build ...
    (op nodes 16) ; output bus
    (si sink () '(16 ,(nodes no) "Out"
        #'analog-sink-print-fn))
    ...)
```

Notes: The obvious application for the general sink is to provide a traceprint facility for monitoring system nodes.

Output

A general sink is alas a somewhat clumsy output primitive. Apart from interpreter overhead, it is somewhat awkward to connect and disconnect sinks at arbitrary points in the module hierarchy, especially when the simulation

is running. A better solution to the overall problem of observing the simulation is to yet again modify the port instances to include a *probe* flag, which when set causes the port to emit trace output on receipt of ◁*V*=▷ messages, which are decoded and converted to output on a standard bitmapped display by an independent output daemon **xplot**. The probing primitive is (probe-ports <*port...*>), which will often be found in module init code with its relatives like WATCH-PORTS. Alternately an explicit **probe** module may be used—

```
(defmodule probe (nI probe)
  (build
    (probe I-port nI probe))
  (init
    (ignore-ports probe)
    (probe-ports probe))
  (run)
  (done)
)
```

3.5.3 Verification

Verification is a large topic beyond the scope of this thesis, nevertheless this section presents an example of light-weight verification tools realized with Loge.

Versional Blocks

In [Lathrop+85] the term ***versional blocks*** was introduced in the description of a verification technique using multiple modules which provide implemen-

tations of the same function. Each versional module receives the same input and their output is compared[5] with the intent of showing that the blocks are functionally identical or not as the case may be.

In Loge, every module has the potential to act as its own versional block, as functional and structural mode operation provide distinct implementations of what is intended to be the same function— figure 3.36. This hierarchical verification should be a fundamental step in the development of a multi-mode Loge module.

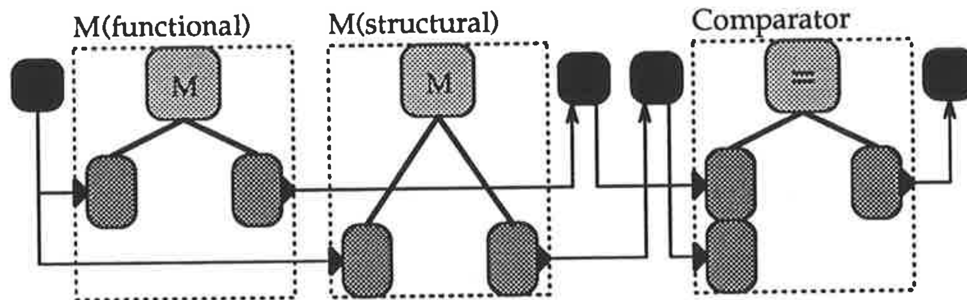


Figure 3.36: Loge hierarchical comparison

Analog comparator

To compare versional blocks, an N bit analog comparator is required—

```
(defmodule comparator (n I0 I1 O)
  (build
    (I0 I-port n I0)
    (I1 I-port n I1)
    (O  O-port n O))
```

[5]One "trusted" output may be connected to the rest of the system.

```
  (init
    (watch-ports I0 I1)
    (analog-watch-ports I0 I1)
    (ignore-ports O)
    (awaken-after 0))
  (run
    (loop for j from 0 below n doing
      ; V = 1/2 if no difference
      (setf (port-bit-analog O j)
        (/ (- (port-bit-analog I1 j)
              (port-bit-analog I0 j) -1.0) 2.0))))
  (done))
```

Notes: `(setf (port-bit-analog...) ...)` operates directly on the attached node by sending a ◁$V\leftarrow$▷ message. `ANALOG-WATCH-PORTS` and consequent receipt of ◁$dV/dt=$▷ messages allows the comparator to output correct values at the point where the current into input nodes changes. A more accurate version would also modify the slope of the comparators output node voltages with the `PORT-BIT-CURRENT` function.

"Figure of Merit"

A central property of any verification strategy is the amount of information required as input from the user, and similarly the amount of information output by the verifier which the user must scan to detect problems. Either category can easily become burdensome, perhaps unavoidably. For example, the comparator of the previous section could be improved by adding another input which defines when the input data is valid— during invalid periods the

comparison is ignored, thus any comparator output fluctuation is more likely to indicate a legitimate problem. This refinement increases the accuracy of the verification method, at the expense of greater user input.

Some method of summarizing verification information is required. In a large system with many comparator-based tests in progress the user may become overwhelmed. A direct heuristic is to integrate the comparator outages over a complete simulation, using this value to generate a "figure of merit" for a module instance. If many modules are present, the user may then home in quickly on the worst problems by detailed examination of modules in order of lowest merit. A module to calculate figures of merit is the FOM-sink—

```
(defmodule FOM-sink (nI I name &aux fom tim)
  (build
    (I I-port nI I))
  (init
    (analog-watch-ports I)
    (watch-ports I)
    (setq fom 0.0 tim 0))
  (FOM-run)
  (done
    (FOM-run)
    (format t "Figure of Merit for module ~A is ~F~%"
      name (- 1.0 (/ fom n (run-time))))))

(defun FOM-run ()
  (let ((x 0.0))
    (loop for j from 0 below n doing
      (setq x (+ x (* 2 (abs (- 0.5
```

```
                (port-bit-analog I j)))))))
    (setq
      fom (+ fom (* x (- (simulation-time) tim)))
      tim (simulation-time)))
```

Notes: `FOM-sink` finally provides an example of use of the done code.

Verification sink

Figure of merit verification can be simplified slightly with the following structural module.

```
(defmodule verification-sink (name
    mof ; functional mode module
    mos ; structural mode module
    nI fI sI &aux nfm cmp fom)
  (build
    (nfm nodes nI)
    (cmp comparator () (list nI fI sI nfm))
    (fom FOM-sink () (list nI nfm name)))
  (init (setf
      (module-mode mof) :functional
      (module-mode mos) :structural
      (module-mode cmp) :functional
      (module-mode fom) :functional
      ))
  (structural-module-only))
  (done))
```

Notes: Unfortunately it is difficult in general to avoid requiring independent instantiation of the two module under comparison— although it would be more elegant to include them in the `verification-sink` this would require some awkward argument passing.

Example

At last, all the submodules necessary for to give an example of the test and verification methodology have been defined. The module under test will be the following two input exclusive OR gate—

```
(defmodule xor (I O &aux de dp no)
  (build
    (n node ())
    (de enn 3 '(
        (1 ,(node I 0) ,no ,vss)
        (1 ,(node I 1) ,O ,no)
        (1 ,no ,(node I 1) ,O)))
    (dp enp 3 '(
        (1 ,(node I 0) ,no ,vdd)
        (1 ,(node I 1) ,O ,(node I 0))
        (1 ,(node I 0) ,(node I 1) ,O)))
    (I I-port 2 I)
    (O O-port 1 O)
    )
  (init
    (watch-ports I)
    (awaken-after 0))
  (run
```

```
    (let ((in !I))
      (if in (setf !O (if (or (= 1 in) (= 2 in)) 1 0)))))
  (done)
)
```

—for which the tester module is—

```
(defmodule xor-tester (&aux pow nin nou xra sou tst ver)
  (build
    (pow power ())
    (nin node     2)
    (nou node     2)
    (sou source   () '(2 ,nin #e2n
        (0 1 2 3 0 2 1 3 2 0 3 1)))
    (tst xor      2 '(
        (,nin ,(node nou 0))
        (,nin ,(node nou 1))))
    (ver verification-sink () '("Xor"
        ,(first tst) ,(second tst) 1
        (,(node nou 0)) (,(node nou 1))))
    (xra probe    () '(4 (,@nin ,@nou)))
    )
  (setf (module-mode (the-module)) :structural)
  (structural-module-only)
  (done)
)
```

Note the double instantiation of **xor**, sharing input from a general source and feeding output through two nodes to a verification sink. With the following commands—

```
% loge xor-test
Loading <xor-test>...
loge> (set-device-subtype :explicit)
:EXPLICIT
loge> (defvar x-t (instantiate xor-tester))
X-T
loge> (simulate x-t #e26n)
Figure of Merit for module Xor is 0.9136909
NIL
```

— a simulation is run, with the FOM-sink reporting 91% agreement between the functional and device-level models. Given that the quick but inaccurate explicit device subtype was used this figure suggests good correspondence between the models. This is confirmed by inspection of a trace of the input and output voltages (figure 3.37), which reveals that the 9% disagreement occurs in some sizable glitches synchronized with input changes, much as one would expect.

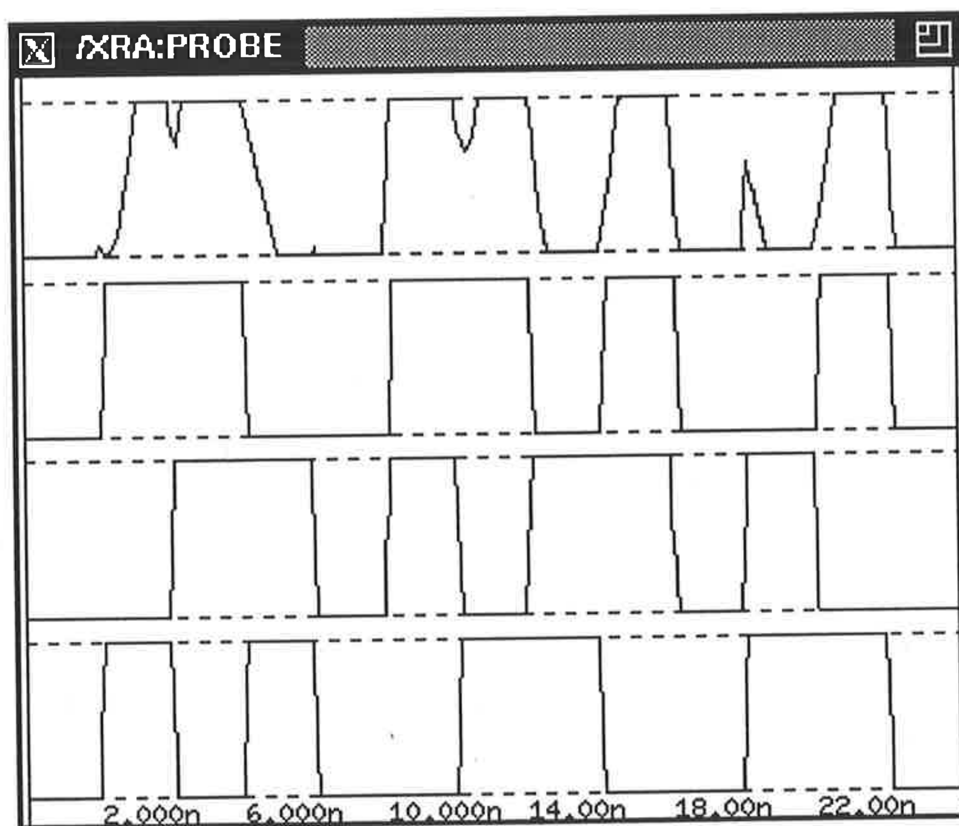


Figure 3.37: Xor tester simulation

Chapter 4

Results

4.1 Introduction

This chapter is divided between performance tests, a detailed investigative functional simulation case study, and an illustration of adaptation of Loge to an alternate device technology, in this case GaAs.

4.2 General Performance Tests

Loge definitions of some typical or theoretically interesting circuits have been developed, and their behaviour under simulation is presented in this section. Wherever possible, generic modules have been specified where the module definition accepts a *problem size* parameter N. This allows comparisons of mode performance over wide ranges of number of simulation objects with a minimum qualitative variation in the composition of the system under test (simulated time is held constant). Wherever possible, tests were run on a Sparcstation 1+ with twenty four megabytes of memory and as little other load as possible.

Since the general emphasis in Loge is more toward simulation turnaround time than accuracy this is reflected in the data presented in this chapter. Correctness of simulation is checked either by visual inspection, automatically by comparison with a reference module, or in selected (simple) cases by comparison with SPICE output.

Some simple performance improvements are possible over those quoted here. Firstly, the version of Loge used is itself highly instrumented for debugging and statistics generation— this overhead adds at least 10% to turnaround. Another few percent is lost in an unnecessarily inefficient output interface, and yet more in places where the overhead of object oriented procedure calls is unwarranted. Overall, it must be admitted that the code is not of production quality.

4.2.1 Ring Oscillator

Inevitably some digital simulation tasks are difficult or impossible to some simulation modes, for example it is inherently difficult (and misguided, but not quite impossible!) to write a functional model for a transmission gate— detailed analog behaviour is the casualty of abstraction. A ring oscillator provides a good stress test to illustrate the general nature of the idiosyncratic modelling inaccuracies of each modes. In figure 4.1 a ring of seven inverters is simulated in (reading from top to bottom in decreasing order of abstraction) functional, explicit device, sampled device, evaluated device and analog device modes. All modes correctly produce the expected oscillatory pattern, but there are some instructive differences.

- Predictably there is considerable similarity between the waveforms produced by analog and evaluated devices, with the more conservative analog algorithm being more accurate. The evaluated devices operate on

twenty distinct voltages, which results in an only very slightly jagged output, masked here by the pixel granularity of the display format.

- The waveform produced by the sampled devices is similar, but the effect of the regional approximation can be seen in a more piecewise waveform. The sampled devices used in these tests are the 256/29 region versions of figures 3.30 and 3.31, which are sampled from the same equations and parameters used by the analog and evaluated models— the intention is to maintain as good a correspondence as possible between devices in each mode. The general behaviour of other granularities of sampling is that, much as one would expect, there is a rough proportionality of accuracy and number of samples— although this rule of thumb fails when there are few distinct currents.

  Note that the period of oscillation is shorter than that of the analog devices. Due to the relatively few regions of distinct current, these sampled devices tend to switch on more quickly, resulting in steeper transitions. In the case of the ring oscillator this difference accumulates at each inverter stage, multiplying the individually small errors by the number of stages, producing a noticeable change in period. This effect naturally implies that precise timing information should not be taken from simulations of sampled devices, especially in this type of asynchronous system where the error can accumulate (as distinct from conventional clocked digital circuits).

- The explicit device parameters have been deliberately ill chosen, with the result that the output tends to ring at the rails. Currents for the explicit model were arbitrarily chosen at the peak and half peak values used by the sampled model— thus the total current driven through the explicit devices is relatively high, giving a yet shorter period of oscilla-

tion. Choice of explicit model region currents is somewhat arbitrary—a superior scheme was presented in [Ruan+88].

- The functional mode (after an initial bout of transient confusion) settles down to oscillate at the predictable $T \approx 2NR_{port}C_{node}$.

A rough measure of the relative turnaround of the submodes can be gleaned from event counts for the various device types (table 4.1).

Analog	Evaluated	Sampled	Explicit	Modules
15150	3632	2047	3340	97

Table 4.1: Ring oscillator event counts

This is a very rough measure as it ignores node events which will be significantly more frequent in the evaluated device submode. Naturally, much better performance is available from explicit devices if more care is taken with the device parameters. The event count for modules includes that of the probing sink.

4.2.2 Shift Register

As implied in section 2.4.3 the CMOS shift register cell of figure 2.12 is relatively easy to simulate as there is little scope for bidirectionality related problems. Therefore a shift register built of N such cells is a minimally difficult yet variable size simulation problem. Results are given in table 4.2, and turnaround summarized in figure 4.2. The table shows total node and device events, counts of the number of times an event was preempted, and the real turnaround time from the start of execution of init code to the

completion of done code. The number of simulation objects is $\approx 6N$, and simulated time 256ns in all cases.

Allowing for slight variations in response due to system load, simulation turnaround time, preempt, device and (where applicable) node event counts scale near linearly in N— apart from the occasional artifact where an unusual combination of event or preempt arrival times causes large enough variations in the scheduler performance to be noticeable. The surprise of this test is the consistent superiority of sampled devices to explicit devices as N increases, despite this time taking care to suppress explicit device oscillation.

4.2.3 Adder

An elegant design for a CMOS exclusive OR gate is built around a transmission gate (see page 317 in [Weste+85] and section 3.5.3). Without care, this design may cause difficulties to switch-level simulators [Svensson+88], and as such is a interesting test for Loge. The raw gate itself will be examined in section 4.2.4, but its effects are felt from within a Loge N bit adder definition, built from N half adder cells each containing an exclusive OR gate, four transmission gates and five inverters— for a total of $\approx 25N$ simulation objects.

Encouragingly, Loge has no difficulty correctly simulating this system, and once again sampled devices provide superior performance— see figure 4.3 (tabular form omitted in this case). One begins to suspect that sampled devices are particularly well suited to highly directional systems.

For this test a functional model of the adder cell was also developed but as turnaround times were $\approx N/4$ seconds these do not appear on the graph. Shown in figure 4.4 is an eight bit sampled device simulation of a number of additions, with the effects of carry propagation showing clearly.

N	Events			T(s)	Events			T(s)
	Node	Device	Preempt		Node	Device	Preempt	
	Analog				Evaluated			
2	144	19079	23190	22	2265	3959	5703	7
3	145	29314	36892	33	2836	6123	8736	10
4	144	39055	47487	43	3360	8181	11607	13
5	145	48351	57723	53	3868	10183	14588	16
6	145	57328	69550	63	4321	12050	17197	19
7	145	65910	79520	73	4749	13834	19657	22
8	145	74073	93278	83	5137	15478	21919	24
9	144	81701	101044	93	5504	17077	24083	27
10	144	89029	104731	100	5822	18536	26020	27
11	146	95819	119409	107	6129	19986	28315	29
12	144	102327	124421	146	6387	21287	29838	30
	Sampled				Explicit			
2	219	2167	2580	2	174	2443	3547	2
3	252	3384	3955	3	209	3854	5545	3
4	287	4605	5320	4	236	5456	7617	5
5	309	5702	6522	5	259	6723	9323	6
6	337	6789	7690	5	277	7935	10960	7
7	373	7889	8866	7	292	9139	12607	8
8	406	8860	9806	7	322	10261	14125	9
9	422	9729	10728	8	337	11287	15453	10
10	433	10530	11489	8	352	12536	17074	12
11	449	11275	12249	9	359	13012	17600	12
12	468	12022	12878	10	373	13714	18558	13

Table 4.2: Shift register simulation results

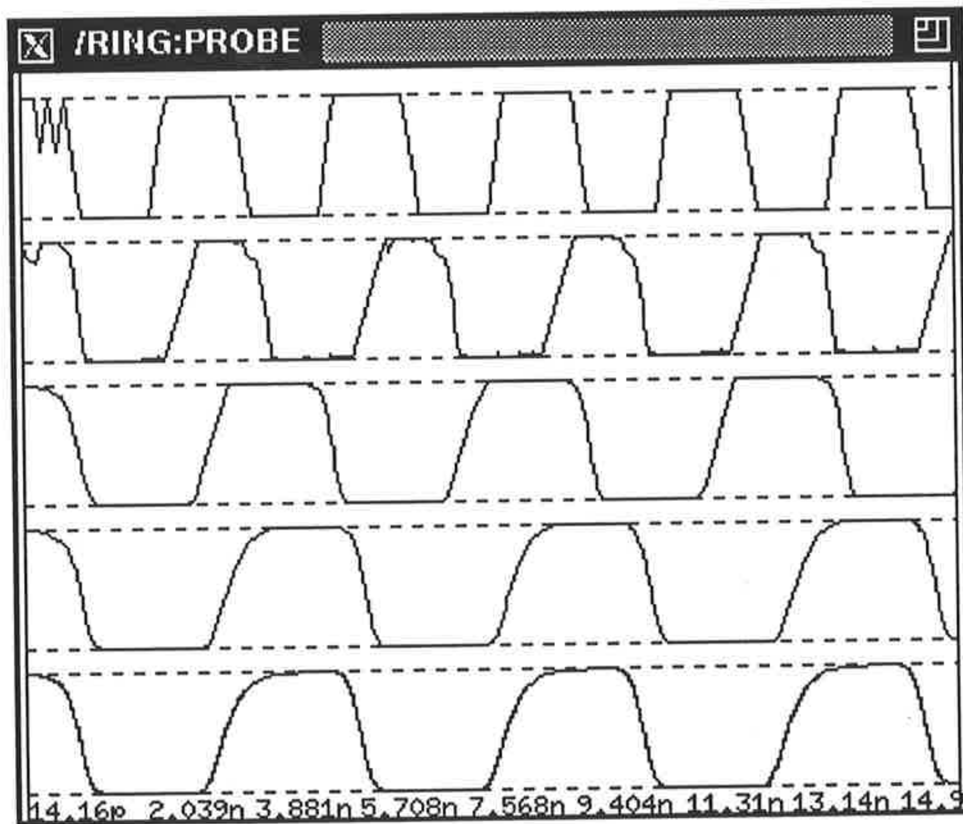


Figure 4.1: Ring oscillator in functional, explicit, sampled, evaluated and analog modes

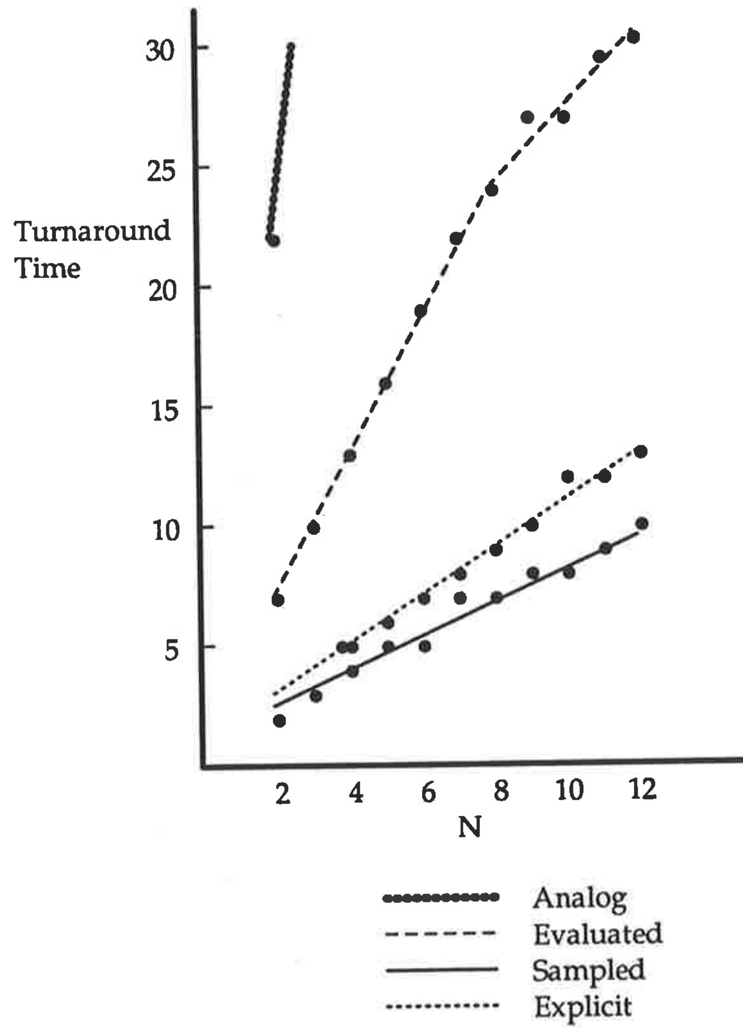


Figure 4.2: Shift register turnaround

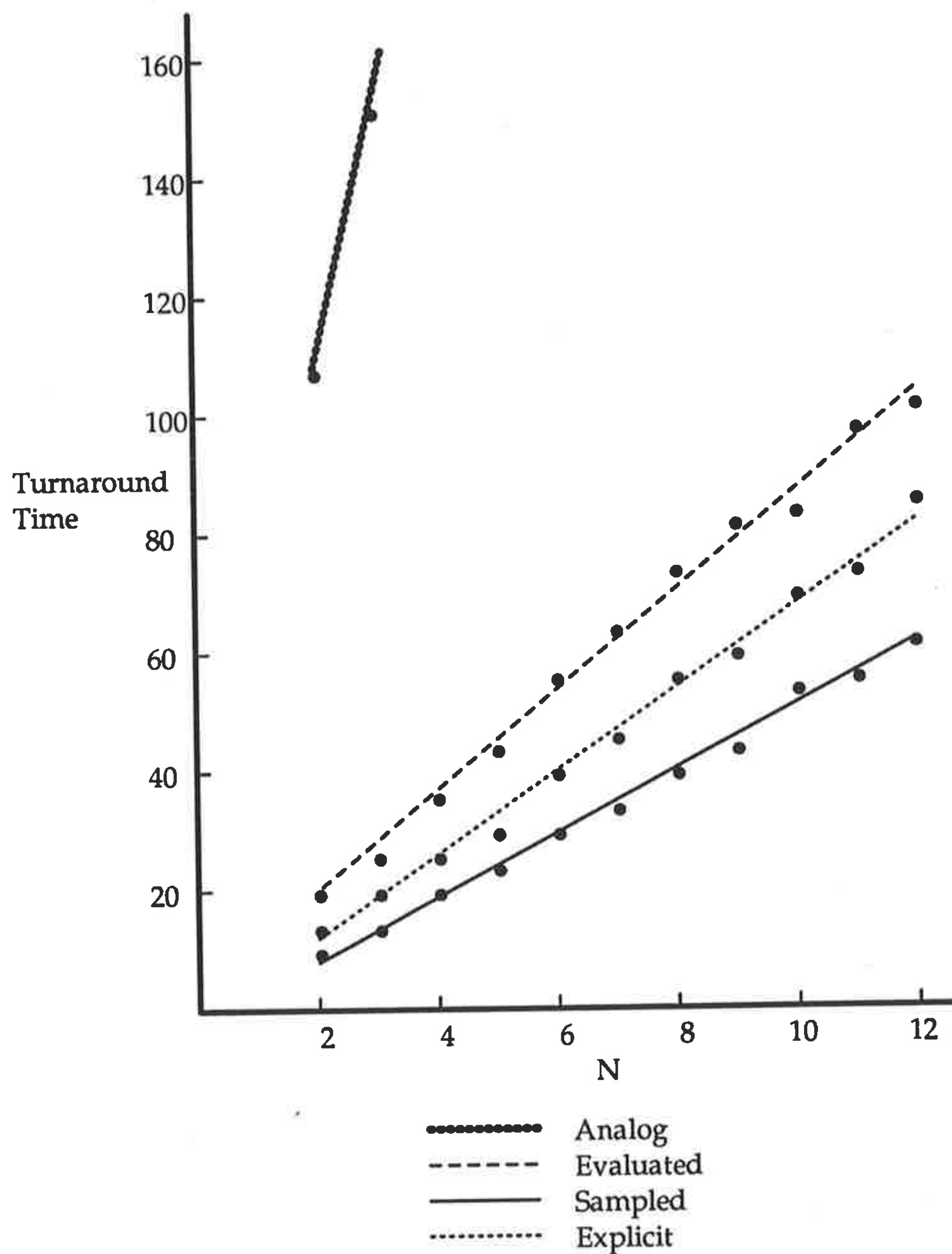


Figure 4.3: Adder turnaround

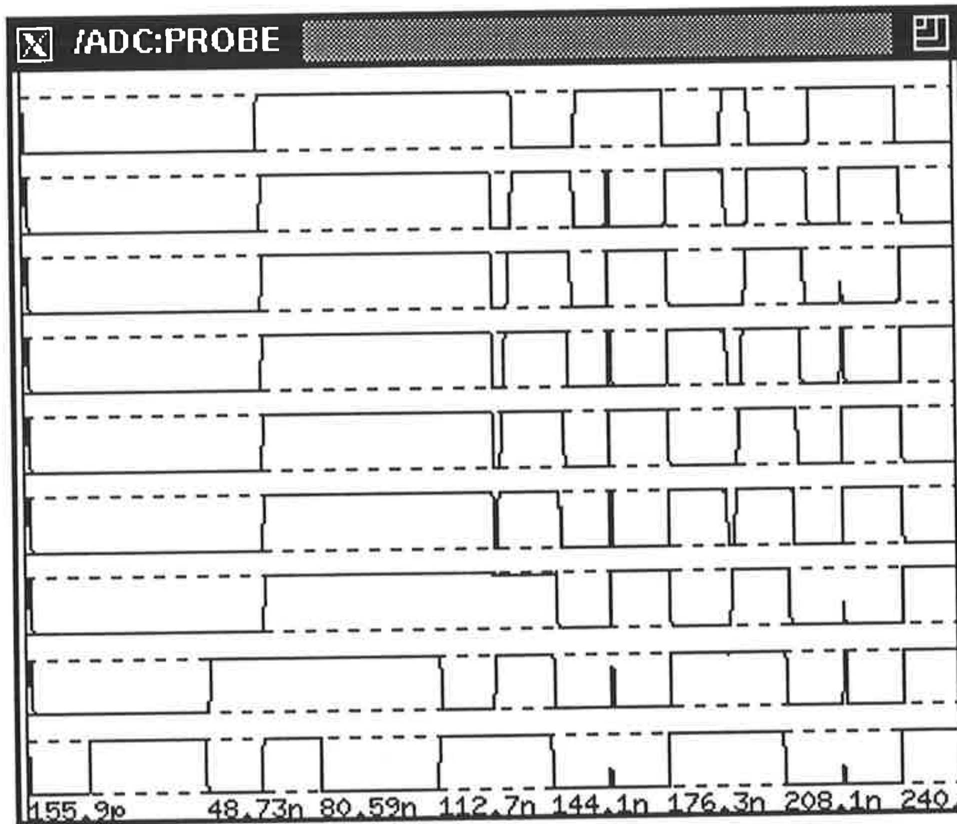


Figure 4.4: Sampled device eight bit adder simulation

4.2.4 Binary Trees

All tests so far have involved problems where the number of simulation objects is linear in N. A harder simulation task is that of a binary tree of two input logic gates with results rising up through the tree to a final value at the root. The Loge definition of the variable sized binary tree structure used for these tests is perhaps interesting—

```
(defmodule tree-2-N (N gate I O &aux no mo)
  (let* ((n-inputs (ash 1 n)) (n-modules (1- n-inputs))
        (n-internal (ash n-modules -1)))
    (build
      (no node (- n-inputs 2))
      (mo gate n-modules (module-vector j ; heap
          at 0 (
            (nodes no 0 2)
            O)
          over 1 n-internal (
            (nodes no (ash j 1) 2)
            (node no (1- j)))
          over n-internal n-modules (
            (nodes I (ash (- j n-internal) 1) 2)
            (node no (1- j)))
          ))))
   (setf (module-mode (the-module)) :structural)
   (structural-module-only)
   ())
```

Note that the type of module to be instantiated within the tree is supplied as the parameter `gate`— NAND, NOR and (transmission gate) exclusive OR gates were used. While no explicit feedback is yet present, the NAND and NOR gates include configurations prone to bucket brigade oscillation.

Also present in this test are purely functional models of the gates, and a functional model for a complete variable sized tree for each of the subject gates. The full tree models are used as reference models for comparison with the output of device based trees— the integrated differences are given under the heading *Err%* in the tables of results. This streamlining of result generation, although it removes the necessity to inspect all output waveforms (one need only scan for unusually large integrated errors once the reference model is correct), introduces a complicating factor to the measurement of turnaround— simulation is slowed by the calculations of the integrator module. The integrator must operate every time one of its input voltages changes slope[1]— thus the induced performance penalty is proportional to the accuracy of the output waveform, disproportionately penalizing analog and evaluated modes. Some measure of this penalty can be grasped with reference to the event counts for modules, which is dominated by those specific to the integrator.

Tabulated results follow in tables 4.3 through 4.5, and turnaround summarized in figure 4.5 through 4.7. The number of simulation objects for the device-based versions is $\approx 8 \times 2^N$. *Err%* grows steadily with N as the extra delays in the device based simulation result in an output waveform that diverges further and further from the reference functional model— the main interest of *Err%* apart from detection of errors of functionality is in comparison with the values produced by the accurate analog mode.

[1]Integrator overhead has been reduced by recoding in C++.

NAND N	2	3	4	5	6	7	8
Functional							
Node	115	226	441	878	1747	3492	6977
Module	105	142	214	368	667	1269	2467
Preempt	296	484	863	1638	3174	6257	12412
Err%	3.41	2.79	2.89	2.96	3.06	3.13	3.23
T	10	14	25	46	90	186	343
Explicit							
Node	110	210	408	798	1527	3057	6085
Device	2025	3973	7887	15409	30061	60286	119478
Module	251	136	103	95	88	98	110
Preempt	5981	11766	22602	44582	88850	143447	351763
Err%	2.59	2.98	2.82	3.09	3.29	3.39	3.36
T	21	22	34	63	132	295	624
Sampled							
Node	154	300	573	1125	2221	4421	8827
Device	1954	4405	8978	17965	36102	72291	144772
Module	204	192	171	160	162	171	165
Preempt	2311	4610	8947	17510	34800	69575	138960
Err%	2.90	3.02	3.47	3.67	3.85	4.05	4.23
T	18	24	38	67	132	247	493
Evaluated							
Node	868	1650	3171	6268	12415	19842	49270
Device	2511	5511	11132	22428	44965	73023	179226
Module	502	476	414	381	386	362	382
Preempt	4030	6548	12516	24130	46622	93263	184886
Err%	2.61	4.23	4.94	5.08	5.26	5.48	5.74
T	62	99	177	332	709	1581	3701
Analog							
Node	78	134	248	476	923	1829	3648
Device	16418	36913	70335	141775	284559	568231	1137316
Module	894	922	311	298	323	703	337
Preempt	20649	43810	80473	159076	323404	647841	1295938
Err%	2.91	3.05	3.36	3.45	3.80	3.85	3.94
T	73	103	121	232	462	965	1869

Table 4.3: NAND tree simulation results

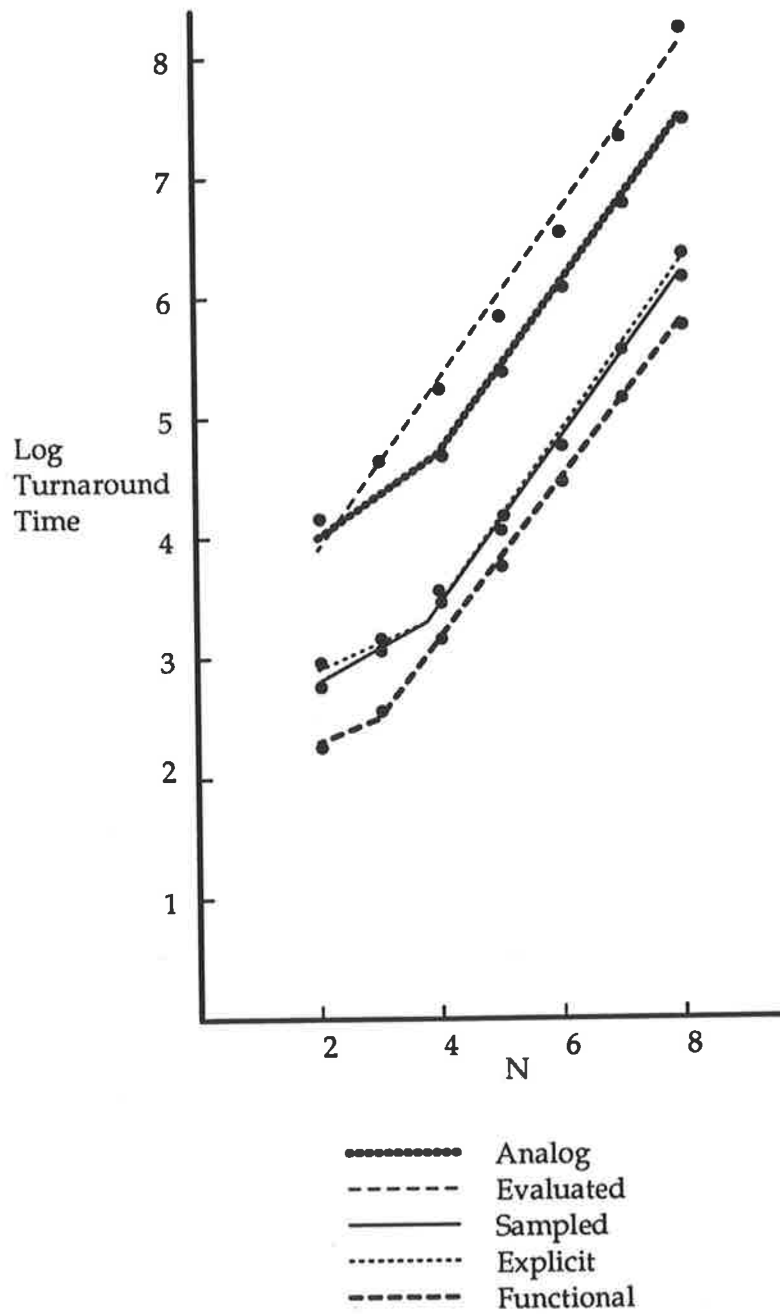


Figure 4.5: NAND tree log turnaround

NOR N	2	3	4	5	6	7	8
Functional							
Node	127	218	417	822	1627	3244	6473
Module	118	135	203	349	632	1202	2336
Preempt	329	465	824	1559	3015	5938	11773
Err%	3.45	2.75	2.83	2.88	2.96	3.01	3.09
T	9	13	24	45	88	180	334
Explicit							
Node	143	254	479	960	1879	3781	7489
Device	2175	4363	8271	16277	32147	64086	127884
Module	302	189	100	149	104	141	106
Preempt	3482	5885	10553	20591	40407	80236	160029
Err%	2.21	2.90	2.64	2.88	2.81	3.06	3.11
T	23	24	33	66	125	274	518
Sampled							
Node	194	327	612	1178	2324	4593	9103
Device	2638	5321	10484	20763	41264	82316	163703
Module	264	191	129	130	130	134	134
Preempt	3118	5484	10316	20190	39946	79391	157829
Err%	3.82	3.24	3.33	3.49	3.50	3.77	3.91
T	21	24	35	65	128	264	502
Evaluated							
Node	1016	1767	3338	6512	12779	25804	11694
Device	3018	5888	11479	22590	44556	88850	42769
Module	582	410	381	376	346	377	120
Preempt	7360	12532	23751	46342	90772	180426	86909
Err%	4.38	4.48	4.82	5.06	5.15	5.48	6.02
T	65	96	169	334	690	1798	757
Analog							
Node	122	240	448	858	1745	3452	6892
Device	17952	34115	66952	132928	263272	526060	1049254
Module	1304	779	495	460	535	445	493
Preempt	23979	41379	79516	155201	306314	607102	1209213
Err%	3.38	2.43	3.39	3.70	3.78	4.10	4.16
T	97	93	129	233	454	896	1782

Table 4.4: NOR tree simulation results

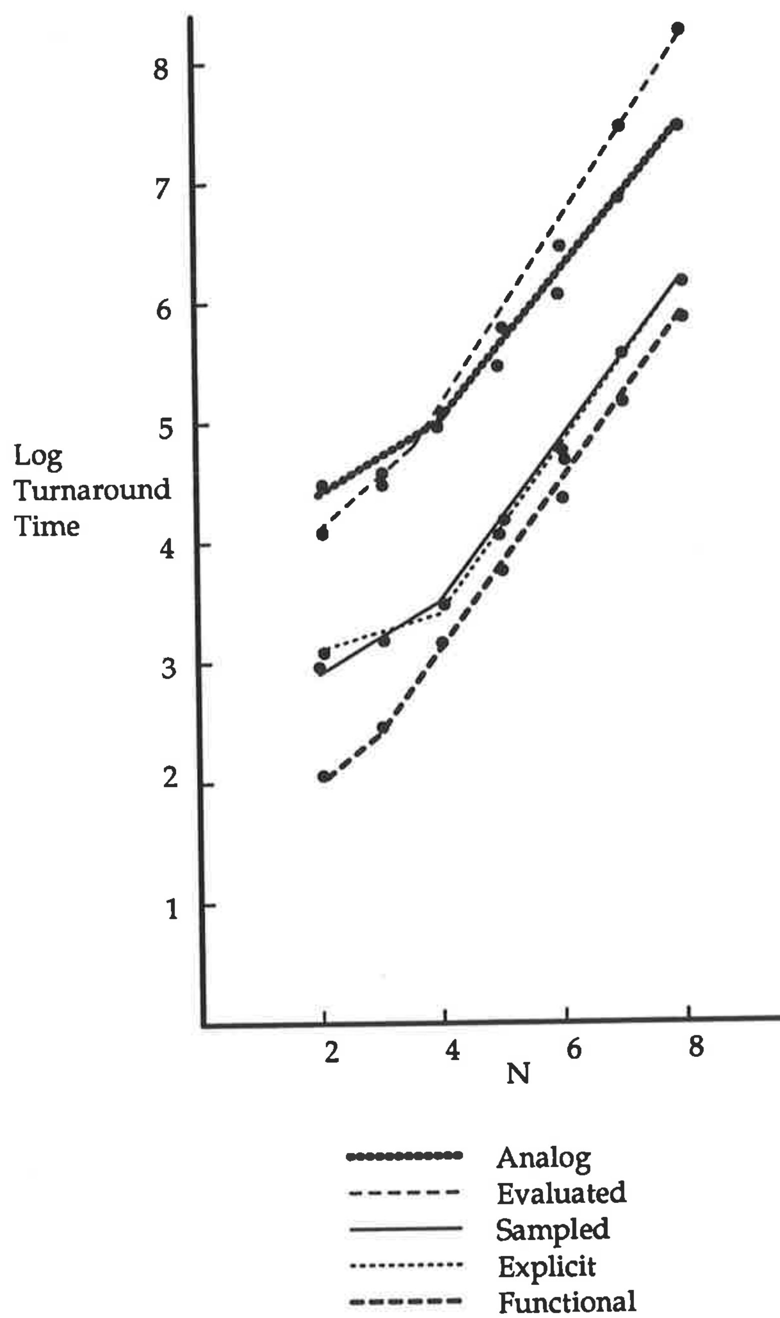


Figure 4.6: NOR tree log turnaround

Xor							
N	2	3	4	5	6	7	8
Functional							
Node	108	188	340	636	1220	2380	4692
Module	94	124	180	288	500	920	1756
Preempt	274	428	728	1316	2480	4796	9416
Err%	1.62	1.69	1.77	1.84	1.92	2.00	2.08
T	4	4	5	7	10	18	32
Explicit							
Node	171	323	612	1188	2271	4474	8789
Device	3483	8563	16403	31850	59463	109701	207516
Module	337	395	366	381	409	367	372
Preempt	5760	12643	23023	43853	81150	148836	281963
Err%	2.93	3.28	4.03	3.72	4.84	3.71	6.94
T	22	31	37	57	92	156	281
Sampled							
Node	191	774	1635	2988	6839	13657	27142
Device	4540	17436	36294	72241	138659	248409	447076
Module	281	858	823	1450	1431	1374	653
Preempt	4795	19073	40004	80068	151256	265947	472793
Err%	2.74	6.15	6.24	8.22	7.98	8.70	10.1
T	19	59	71	134	203	299	473
Evaluated							
Node	566	1961	3848	7500	15482	29301	58192
Device	3081	12296	26410	54330	124520	237463	479170
Module	430	654	722	942	1657	1595	2867
Preempt	6667	23300	47466	95258	211760	400166	805145
Err%	1.75	2.84	3.68	4.97	6.15	7.68	9.39
T	27	47	71	122	263	437	867
Analog							
Node	682	7977	17392	32836	67620	111426	250315
Device	36052	129808	259327	520444	1.06e6	1.91e6	4.05e6
Module	1971	8187	9921	7491	6239	6186	5861
Preempt	44163	187654	378625	738501	1.48e6	2.56e6	5.44e6
Err%	1.45	2.02	2.75	3.92	5.32	6.98	8.91
T	152	643	932	1063	1660	2754	5005

Table 4.5: Exclusive OR tree simulation results

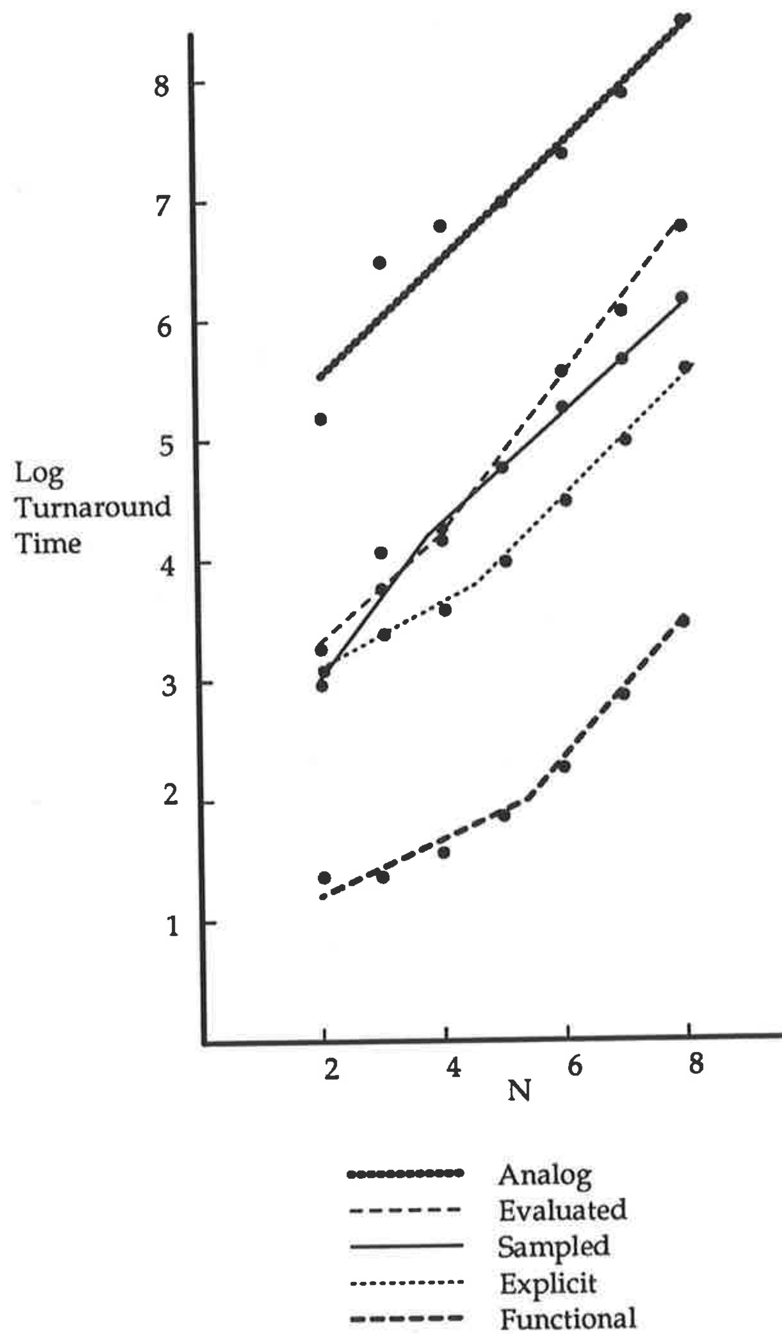


Figure 4.7: Exclusive OR tree log turnaround

After the first few points, in most cases the relevant event counts and turnaround times approximately double with each increment of N as expected. The main exception appears to be that for NAND and NOR trees, evaluated mode performs relatively poorly with a slightly parabolic increase in values of log turnaround time.

Performance of the NAND and NOR trees shows considerable quantitative similarity, which was also predictable. What is interesting is that the gap between sampled and explicit mode has narrowed somewhat, and that the evaluated mode is now significantly slower than analog mode. A limited investigation of this unexpected change showed that some analog nodes were oscillating between rail voltages and their nearest neighbouring discrete voltage— the iterations whereby $\Delta V \rightarrow 0$; $V_i = V_{rail} \pm \Delta V$ were requiring a significant number of steps. The rail voltage was always that connected to two series devices in the subject gates— one is forced to conclude that this problem is characteristic of the evaluated device implementation used, and that it had not previously been noticed until this test— where the high proportion of vulnerable devices revealed it.

The exclusive OR gate simulations return to the pattern of previous results. There is a simplification in the functional model of this gate and that of the NAND and NOR cases— thus the improved functional model performance is not unexpected.

4.2.5 Barrel Shifter

A barrel shifter is a worthwhile test since it consists completely of cascaded bidirectional devices— the CMOS version is composed purely of transmission gates. The Loge model of the barrel shifter structure is pleasantly concise—

```
(defmodule barrel-shifter-N (n L I R &aux tga)
```

```
(build
  (tga tgate '(,n ,n) (module-array x y
      over (0 n) (0 n) (
        (node L (+ y x)) (nodes I (+ x x) 2) (node R y))
      )))
(setf (module-mode (the-module)) :structural)
(structural-module-only)
())
```

Turnaround is graphed in figure 4.8— the number of simulation objects is $\approx 3N^2+5N$. Explicit devices are the clear winner here— unlike the previous examples containing transmission gates this time there are no associated inverters to impart some specific direction of information flow.

4.2.6 Memory

The preceding simulations have all been rather small— of the order of thousands of simulation objects at the maximum. This is of course not at odds with the underlying philosophy to Loge that only a few small modules should be simultaneously simulated in low level modes. Nevertheless some larger examples are called for.

Another point in which the examples presented so far are atypical is that many exhibit unusually high levels of circuit activity. In contrast, the circuit activity of an N bit address by M bit word memory (using a six device static cell ([Weste+85], page 353)), can never greatly exceed $100/2^N\%$ for sufficiently large N and M. Note that the memory cell is dependent on feedback for correct operation.

This time, the figure 4.9 shows turnaround plotted against total number of

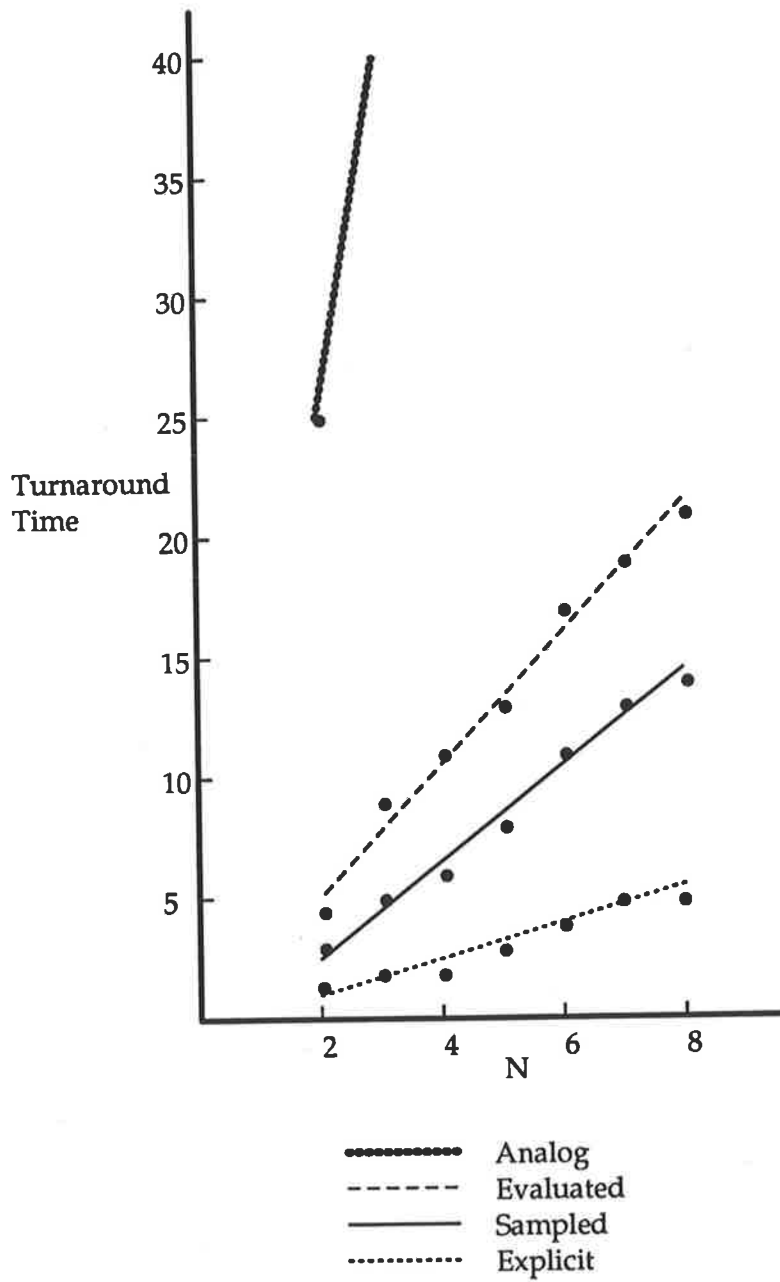


Figure 4.8: Barrel shifter turnaround

simulation objects, for various sizes of memories, and for three cases extracted from fixed physical layout — a data-path, ALU and block of registers, a large state machine incorporating a PLA and some assorted gates, and a multiplier. Reasonable proportionality between number of objects and turnaround is seen, once again with some variations in mode performance on different test systems. These simulations were performed on a larger machine (the 256×8 element memory reached a peak size of 12 Megabytes and the next test began to thrash).

4.2.7 Simulator performance

A measure of simulator performance in the form of events serviced per unit time (E/T) and events plus preemptions per unit time ($(E+P)/T$) appears in table 4.6. Only the smaller systems are shown, as they are homogeneous— each exemplifies a significantly different type of circuit (except for the adder and exclusive OR tree), whereas in the larger, more heterogeneous systems the efficiency converges towards the average.

The order of magnitude variation across this small sample of test circuits is a telling illustration of the awkward conditions a general purpose simulator must accommodate. Given such variation, one must be wary of the trap of tuning simulator performance for a limited set of input configurations.

One might expect that number of events per unit time be proportional to the level of accuracy. This appears to hold true across the various modes, except for evaluated mode, which gives good accuracy from a relatively low rate of events. This is true in part because only in evaluated mode are node events frequent and computationally of the same order of difficulty as device events. Thus there is scope for further optimization of evaluated mode performance, and it is clearly the most event-efficient mode.

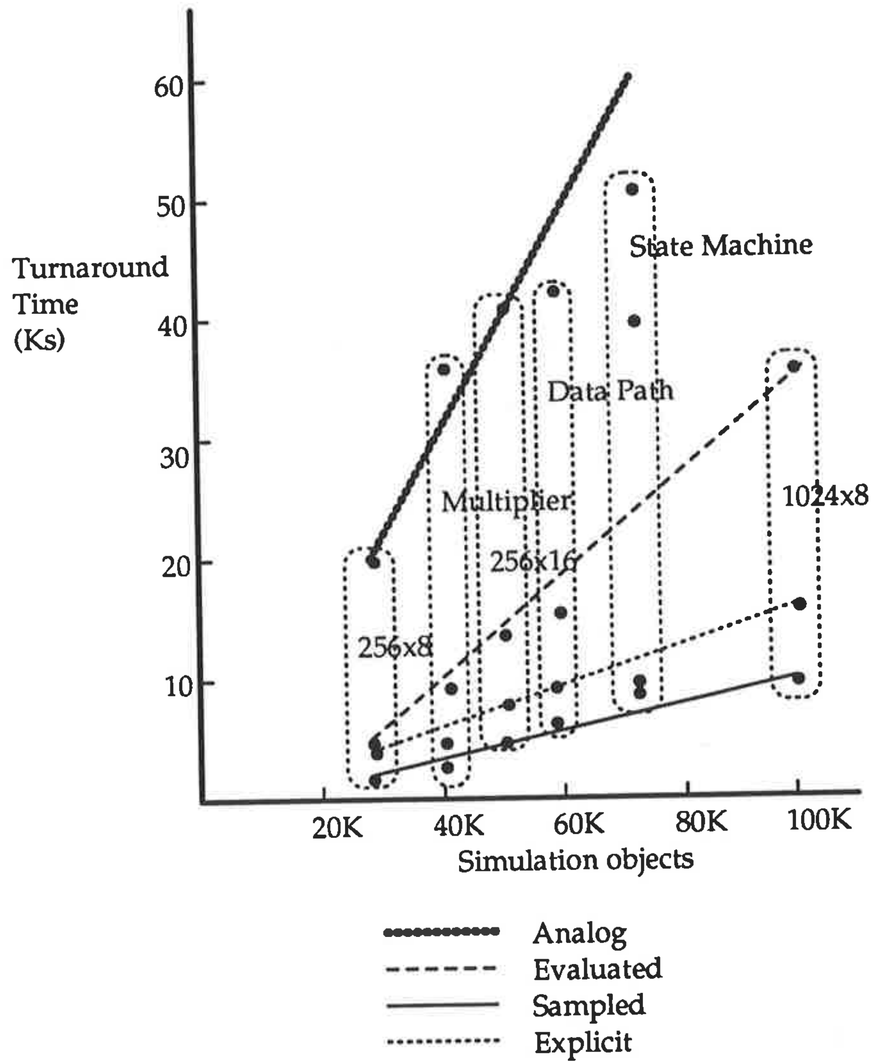


Figure 4.9: Turnaround of larger systems

4.2.8 Summary

These performance tests fail to show a sustained superiority of any one mode over all inputs. While this is disappointing, the provision of the different modes is thus an advantage of Loge.

The analog mode is quite robust and reliable, as expected from this more conservative algorithm. The waveforms it produces diverge only slightly from those of SPICE— usually within 2%.

Apart from a marginal implementation problem, the evaluated mode provides good accuracy and reasonable turnaround— with the benefit that the

Case	E/T	$(E+P)/T$
Each test		
Shift Register	973	2170
Adder	890	1880
Nand-Tree	303	657
Nor-Tree	188	406
Xor-Tree	787	1785
Barrel	1350	3910
Each mode		
Analog	745	1670
Evaluated	127	319
Sampled	595	1160
Explicit	393	905
Functional	35	81
Average	469	1050

Table 4.6: Events and preempts per unit time

speed/accuracy tradeoff is now easily variable. Evaluated mode is qualitatively different from other modes in that the nodes take a more active role— this is an advantage in terms of being able to guarantee a certain degree of precision, and a disadvantage in that it somewhat artificially shifts responsibility away from the devices, which are the seat of modelling complexity.

Sampled mode is less variable than evaluated mode, as the process of generating sampling arrays requires more steps than changing the *Voltages* set. Nevertheless, once a good sampling has been chosen this mode is frequently the fastest means of studying the basic functionality of a system.

Explicit modes are only variable in parameters, like analog modes. They are however possibly the most aggressive method of device modelling. The results are somewhat mixed— abstraction related problems such as the bucket brigade oscillatory effect are most noticeable with explicit devices. The topology of the regions chosen (and hence the computational load of region transition tests), is the most important influence on explicit mode performance— the simplicity of the rectangles of sampled mode are difficult to improve upon. Nevertheless, sometimes explicit mode triumphs.

Functional mode interacts cleanly with the other modes, but it can not be meaningfully compared to the hybrid modes— they are too fundamentally dissimilar. More typical tests of the functional mode appear in the case study of section 4.4.

4.3 GaAs technology

Gallium Arsenide devices while generally similar to MOS devices, are significantly more complicated to design with, process and model. This section illustrates inclusion of GaAs depletion and enhancement mode metal semiconductor field-effect transistors in Loge. The abstract nature of Loge device submodes makes this a perhaps surprisingly straightforward task.

4.3.1 MESFET current models

In very general terms, the MESFET I_{ds} equation [Eshraghian88] can be expressed in the form—

$$I_{ds} = \begin{cases} 0 & V_{gs} - V_{thr} \leq 0 \\ \beta((V_{gs} - V_{thr})^k & 0 < V_{ds} < V_{gs} - V_{thr} \\ \quad +\lambda(V_{gs} - V_{thr})^m V_{ds}) \tanh(\alpha V_{ds}) & \\ \beta((V_{gs} - V_{thr})^l & 0 < V_{gs} - V_{thr} < V_{ds} \\ \quad +\lambda(V_{gs} - V_{thr})^m V_{ds}) \tanh(\alpha V_{ds}) & \end{cases} \tag{4.1}$$

Continual evaluation of these equations in their full generality is prohibitive. This is recognized in the literature, where simplifications such as $k = l$ [Goyal+87] and $k = m = 2$ [Statz+87] reduce the above system to—

$$I_{ds} = \begin{cases} 0 & V_{gs} - V_{thr} \leq 0 \\ \beta(V_{gs} - V_{thr})^2(1 + \lambda V_{ds}) \tanh(\alpha V_{ds}) & V_{gs} - V_{thr} > 0 \end{cases} \tag{4.2}$$

A further computational simplification is to replace $\tanh(\alpha V_{ds})$ with $1 - (1 - \alpha V_{ds}/n)^n$; $n = 2$ or 3 [Statz+87]. The final computation cost for an active device is thus seven multiplications, an addition, a subtraction, and the hyperbolic tangent (approximated as two subtractions and two or three

multiplications), which is rather costly compared to the equivalent MOS I_{ds} equation. Indeed it suggests that the performance of Loge analog and evaluated mode MESFET simulations based on this equation will be at least three times slower than the equivalent MOS cases.

Such complexity is fortunately irrelevant to the sampled submode, as all I_{ds} evaluations are done in advance— this freedom from modelling complexity is one of its significant advantages. With the parameters (subsequently scaled where necessary) $\beta_d = 213uA/V^2$, $V_{thr,d} = -0.545V$, $\lambda = 0.03$, $\alpha = 1.5$, $\beta_e = 362.8uA/V^2$, $V_{thr,e} = 0.219V$, sampling equation 4.2 results in the arrays of figures 4.10 and 4.11.

0.0	0.0	0.0	0.0	0.0	0.0	0.0	26.0u	3.07u	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	71.3u	25.9u	3.04u	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	139u	70.9u	25.6u	2.98u	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	229u	138u	70.2u	25.0u	2.81u	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	341u	228u	137u	68.6u	23.6u	2.40u	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	476u	339u	225u	134u	64.8u	20.2u	1.48u	0.0	0.0	0.0	0.0	0.0	0.0
0.0	633u	473u	336u	220u	126u	55.2u	12.4u	0.0	−1.48u	−2.40u	−2.81u	−2.98u	−3.04u	−3.07u
813u	629u	468u	328u	208u	108u	34.1u	0.0	−12.4u	−20.2u	−23.6u	−25.0u	−25.6u	−25.9u	−26.0u
808u	623u	458u	310u	177u	66.5u	0.0	−34.1u	−55.2u	−64.8u	−68.6u	−70.2u	−70.9u	−71.3u	0.0
799u	609u	433u	264u	109u	0.0	−66.5u	−108u	−126u	−134u	−137u	−138u	−139u	0.0	0.0
782u	575u	369u	163u	0.0	−109u	−177u	−208u	−220u	−225u	−228u	−229u	0.0	0.0	0.0
738u	490u	228u	0.0	−163u	−264u	−310u	−328u	−336u	−339u	−341u	0.0	0.0	0.0	0.0
629u	303u	0.0	−228u	−369u	−433u	−458u	−468u	−473u	−476u	0.0	0.0	0.0	0.0	0.0
389u	0.0	−303u	−490u	−575u	−609u	−623u	−629u	−633u	0.0	0.0	0.0	0.0	0.0	0.0
0.0	−389u	−629u	−738u	−782u	−799u	−808u	−813u	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Figure 4.10: Sampled DMESFET current array

The distribution of currents in the EMESFET array is sufficiently similar to that of the n-channel enhancement mode MOSFET that the same type of six region explicit model may be used. The DMESFET can also be modelled given a slight shift of boundaries (figure 4.12).

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	2.85u	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	36.6u	2.83u	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	108u	36.4u	2.80u	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	218u	108u	36.0u	2.74u	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	366u	217u	107u	35.2u	2.59u	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	553u	364u	215u	104u	33.3u	2.21u	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
777u	549u	360u	210u	98.5u	28.4u	1.36u	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
772u	543u	352u	198u	84.0u	17.5u	0.0	−1.36u	−2.21u	−2.59u	−2.74u	−2.80u	−2.83u	−2.85u	0.0
764u	531u	333u	169u	51.9u	0.0	−17.5u	−28.4u	−33.3u	−35.2u	−36.0u	−36.4u	−36.6u	0.0	0.0
747u	502u	284u	104u	0.0	−51.9u	−84.0u	−98.5u	−104u	−107u	−108u	−108u	0.0	0.0	0.0
706u	428u	175u	0.0	−104u	−169u	−198u	−210u	−215u	−217u	−218u	0.0	0.0	0.0	0.0
602u	264u	0.0	−175u	−284u	−333u	−352u	−360u	−364u	−366u	0.0	0.0	0.0	0.0	0.0
371u	0.0	−264u	−428u	−502u	−531u	−543u	−549u	−553u	0.0	0.0	0.0	0.0	0.0	0.0
0.0	−371u	−602u	−706u	−747u	−764u	−772u	−777u	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Figure 4.11: Sampled EMESFET current array

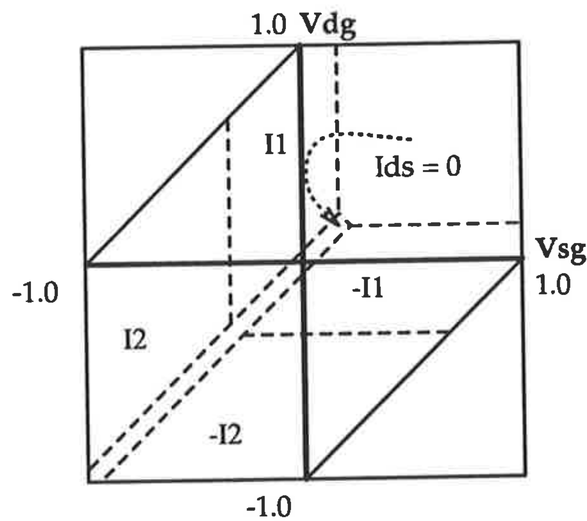


Figure 4.12: Explicit six region DMESFET

4.3.2 Simulations

Depletion and enhancement mode MESFETS are the devices used in the direct coupled FET logic (DCFL) class of GaAs logic circuits. This design idiom is very similar to nMOS, with EMESFETs and DMESFETs replacing enhancement mode and depletion mode MOSFETS— a DCFL inverter is shown in figure 4.13.

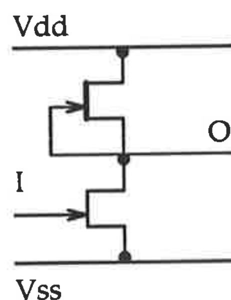


Figure 4.13: DCFL inverter

As in the case of nMOS, the depletion mode device acts as a resistive load. Once again, this means that whenever a hybrid mode pull down device switches on and pulls the output low, as soon as it switches off the output will drift high again causing the pull down to switch on, etc — the bucket brigade effect. This can be seen in the "thick" low values in figure 4.14, which is a simulation of seven DCFL inverters configured as a ring oscillator, in functional, explicit, sampled, evaluated and analog modes (listing from top to bottom). Sampled mode is especially badly affected, with secondary ripples appearing.

In this case, clamped nodes (section 3.4.4) may be used to improve matters considerably— figure 4.15 shows the same simulation with the sampled and explicit mode EMESFETs connected to nodes that appear clamped to

a minimum voltage just above that at which the devices would enter their diagonal $I_{ds} = 0$ region. This modification reduces the number of device events for these modes by an order of magnitude.

In the simulation of figure 4.15, evaluated devices did not share the benefit of connection to clamped nodes, and their performance was degraded almost to the level of analog devices. This effect is reproduced in simulation of a DCFL D flip-flop primitive (figure 4.16, input D is sampled whenever input Clk is low).

Even over a short (10ns simulated time) simulation the dramatic change in evaluated device performance is clear (table 4.7)—

Submode	Events			T(s)
	Node	Device	Preempt	
Analog	134	17048	15293	39
Evaluated	3338	11646	18700	34
Sampled	103	1122	1258	2
Explicit	62	349	467	1

Table 4.7: DCFL flip-flop event counts

Evaluated device performance is also of course worsened by the relatively complex MESFET equations. A simulation appears in figure 4.17, the waveforms from top to bottom being Q, Clk and D. An equivalent nMOS version is between 3.5 to 4.5 times slower in evaluated and analog mode.

Sampled mode however shows gratifyingly similar performance between closely equivalent DCFL and CMOS circuits. Simulation of a tree of DCFL NOR gates (instantiated exactly as in section 4.2.4) produces the results shown in table 4.8—

DCFL NOR							
Sampled							
Node	107	288	631	1304	2631	5272	10534
Device	1169	2428	4860	9655	19242	38457	76835
Module	203	152	111	107	104	109	112
Preempt	1844	3278	6123	11943	23541	46925	93480
Err%	1.69	1.68	2.56	2.85	2.90	2.95	3.00
T	18	22	34	61	124	224	484

Table 4.8: Sampled mode DCFL NOR gate results

While there are major qualitative differences in event distribution, in all but one case, turnaround is within 5% of CMOS NOR tree performance. On reflection, this is unsurprising— the pull down devices act almost identically, while the event traffic generated by a DMESFET is likely to be equivalent to that of the two pMOS device pull up chain acting as a bucket brigade. On the other hand for a NAND gate tree the GaAs version is at least twice as slow, as this time the DMESFET replaces two parallel pull up devices which are not prone to bucket brigade behaviour. The error grows steadily, and event counts roughly double with successive N as expected. Figure 4.18 shows a simulation of the three level tree, with an input period of 1ns.

Similar behaviour is seen in explicit mode, albeit with a greater tendency to spurious oscillation— indeed the six region explicit model appears to be too gross an approximation to be used consistently in the presence of load devices— more robust results are available if the six region model is converted into an eight region model (characterized by three rather than two distinct non-zero currents). As with MOS, absolute fastest turnaround goes to either

explicit or sampled mode depending on circuit configuration.

4.3.3 Summary

Existing Loge device models can be relatively easily adapted to handle MESFETs. Unsurprisingly however, in cases where I_{ds} must be evaluated the greater complexity of GaAs device modelling equations with respect to MOS imposes performance penalties. With MESFETs, simple circuits of the DCFL logic family can be successfully modelled, however hybrid mode algorithms will perform poorly unless the tendency of the depletion load devices to induce semi-permanent bucket brigade oscillation is suppressed. The not entirely satisfactory clamped node technique is one such solution.

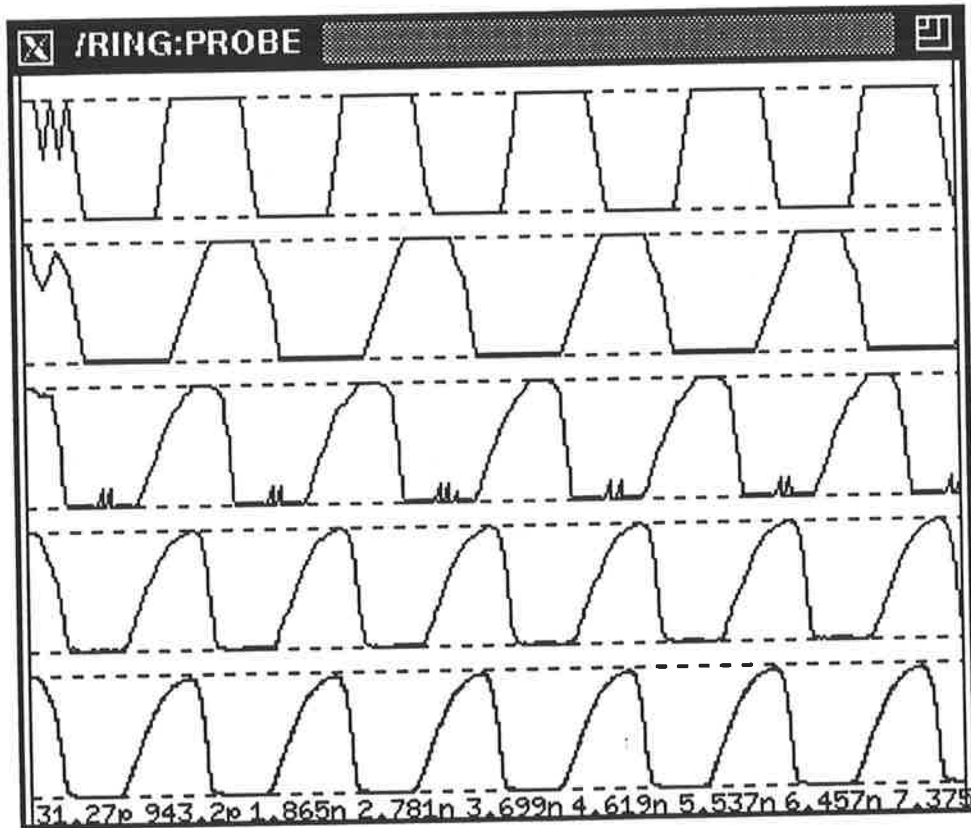


Figure 4.14: DCFL inverter simulation

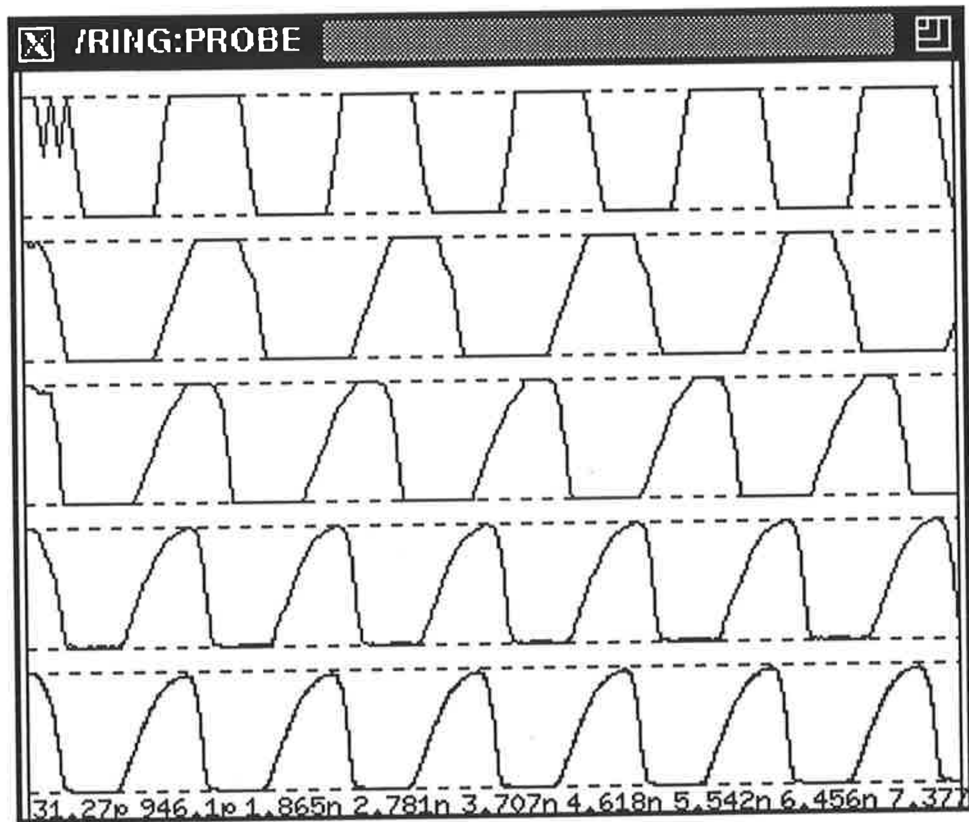


Figure 4.15: DCFL inverter simulation with clamped nodes

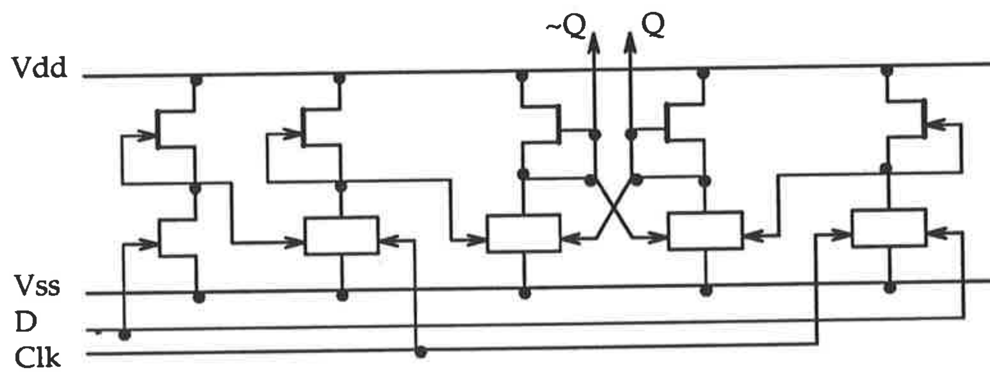


Figure 4.16: DCFL flip-flop

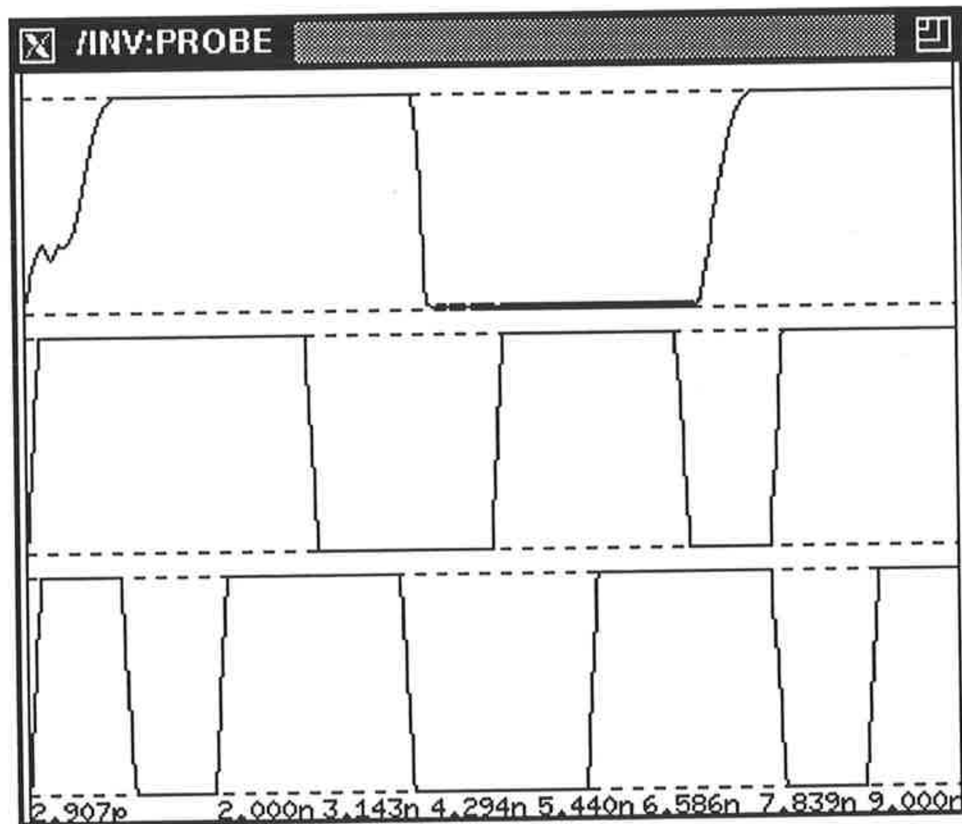


Figure 4.17: DCFL D flip-flop simulation, with evaluated devices

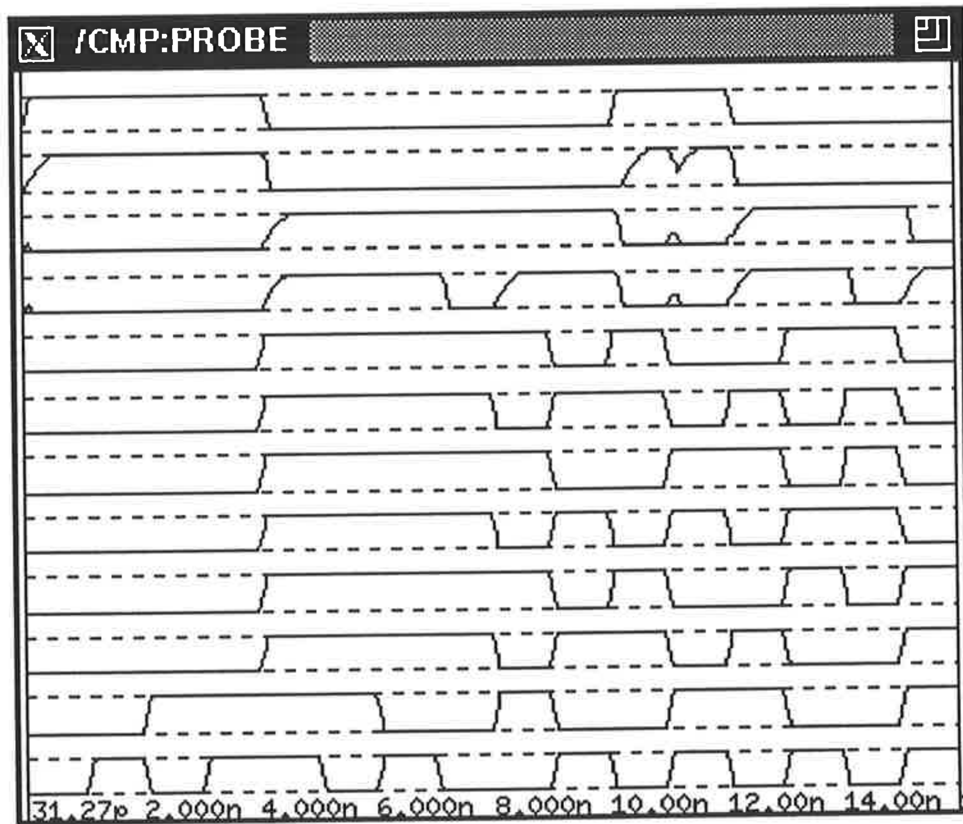


Figure 4.18: DCFL NOR tree simulation

4.4 Design of a Combinator Engine

4.4.1 Introduction

This section is a case study of simulation of a non-trivial architecture from an otherwise unrelated research project. The intention is to illustrate flexibility in composition and operation, and the investigative capability provided by the functional mode of a highly general simulator such as Loge.

The following matter falls into two main divisions— an introduction to the architecture to be simulated, and details of the simulation itself. In both cases, some detail/space tradeoffs have been made in the interest of brevity.

4.4.2 Functional Programming and Combinators

Current research in computer programming and languages is particularly active in the area of functional programming. Pure functional programming languages have many advantages [Backus78], notably with regard to their suitability for parallel execution. Unfortunately these advantages are often difficult to realize with conventional sequential processors. Therefore, some investigation of special purpose architectures for functional programming appears worthwhile.

A notable implementation technique for functional languages is the use of *combinators* [Turner79]. Expressions without side effects or assignments may be transformed to a lambda calculus form, and thence to combinator form by application of a set of simple transforms—

$$\begin{array}{lll} \lambda x.\ e_1\ e_2 & \Rightarrow & \mathcal{S}\ (\lambda x.\ e_1)\ (\lambda x.\ e_2) & \\ \lambda x.\ c & \Rightarrow & \mathcal{K}\ c & c \neq x \\ \lambda x.\ x & \Rightarrow & \mathcal{I} & \end{array}$$

Combinator compilation is illustrated in section 4.4.3 which contains a excerpted trace of the compilation of a simple applicative Lisp expression. In addition, there are some simplifications that can be made to some common patterns of combinators— giving rise to *optimization* rules. Note that some optimizations generate new higher-order combinators.

$$
\begin{array}{lcl}
\mathcal{S}\ (\mathcal{K}\ \mathrm{p})\ (\mathcal{K}\ \mathrm{q}) & \Rightarrow & \mathcal{K}\ \mathrm{p}\ \mathrm{q} \\
\mathcal{S}\ (\mathcal{K}\ \mathrm{p})\ \mathcal{I} & \Rightarrow & \mathrm{p} \\
\mathcal{S}\ (\mathcal{K}\ \mathrm{p})\ \mathrm{q} & \Rightarrow & \mathcal{B}\ \mathrm{p}\ \mathrm{q} \\
\mathcal{S}\ \mathrm{p}\ (\mathcal{K}\ \mathrm{q}) & \Rightarrow & \mathcal{C}\ \mathrm{p}\ \mathrm{q} \\
\mathcal{S}\ (\mathcal{B}\ \mathrm{p}\ \mathrm{q})\ \mathrm{r} & \Rightarrow & \mathcal{S}'\ \mathrm{p}\ \mathrm{q}\ \mathrm{r} \\
\mathcal{S}\ (\mathcal{K}\ \mathrm{p})\ (\mathcal{B}\ \mathrm{q}\ \mathrm{r}) & \Rightarrow & \mathcal{B}*\ \mathrm{p}\ \mathrm{q}\ \mathrm{r} \\
\mathcal{S}\ (\mathcal{B}\ \mathrm{p}\ \mathrm{q})\ (\mathcal{K}\ \mathrm{r}) & \Rightarrow & \mathcal{C}'\ \mathrm{p}\ \mathrm{q}\ \mathrm{r}
\end{array}
$$

Various sets of combinators have been proposed [Peyton Jones87]— the pair $\mathcal{S}$ and $\mathcal{K}$ is the theoretically minimum useful set.[2] A combinator expression is "executed" or *reduced* by applying the transforms—

$$
\begin{array}{lcl}
\mathcal{S}\ \mathrm{f}\ \mathrm{g}\ \mathrm{x} & \Rightarrow & \mathrm{f}\ \mathrm{x}\ (\mathrm{g}\ \mathrm{x}) \\
\mathcal{K}\ \mathrm{c}\ \mathrm{x} & \Rightarrow & \mathrm{c} \\
\mathcal{I}\ \mathrm{x} & \Rightarrow & \mathrm{x} \\
\mathcal{B}\ \mathrm{f}\ \mathrm{g}\ \mathrm{x} & \Rightarrow & \mathrm{f}\ (\mathrm{g}\ \mathrm{x}) \\
\mathcal{C}\ \mathrm{f}\ \mathrm{g}\ \mathrm{x} & \Rightarrow & \mathrm{f}\ \mathrm{x}\ \mathrm{g} \\
\mathcal{S}'\ \mathrm{c}\ \mathrm{f}\ \mathrm{g}\ \mathrm{x} & \Rightarrow & \mathrm{c}\ (\mathrm{f}\ \mathrm{x})\ (\mathrm{g}\ \mathrm{x}) \\
\mathcal{B}*\ \mathrm{c}\ \mathrm{f}\ \mathrm{g}\ \mathrm{x} & \Rightarrow & \mathrm{c}\ (\mathrm{f}\ (\mathrm{g}\ \mathrm{x})) \\
\mathcal{C}'\ \mathrm{c}\ \mathrm{f}\ \mathrm{g}\ \mathrm{x} & \Rightarrow & \mathrm{c}\ (\mathrm{f}\ \mathrm{x})\ \mathrm{g}
\end{array}
$$

A vital property of combinator expressions is that their final reduced form is independent of the order in which individual combinators are processed (from the Church-Rosser theorem). This property makes combinators

[2]The identity combinator $\mathcal{I}$ can be expressed in terms of $\mathcal{S}$ and $\mathcal{K}$.

particularly attractive, as it allows almost arbitrary distribution of a reduction of a combinator expression amongst multiple parallel processors. These processors need merely be implementations of the preceding reduction rules, and "fire" the rules *opportunistically.*

Pseudo-Combinators

Extra *pseudo-combinators* are introduced to support programming idioms of higher level than can be simply provided by raw combinators. For example—

$If <test> $Pro <then> $Pro <else> ⇒ *<then>* or *<else>*

—where *$If* is a conditional operator pseudo-combinator, and *$Pro* another pseudo-combinator which suppresses reduction of the following expression (supporting lazy evaluation). More pseudo-combinators for list processing and arithmetic appear in tables 4.9 to 4.12.

A distinguishing feature of pseudo-combinators is that they have *type-specific* firing conditions. At the very least they are constrained not to operate on expressions of true combinators— *$If* can not be fired until *<test>* is reduced to a boolean value.

4.4.3 Combinator Compilation Example

Consider the Lisp definition—

```
(defun fact (x) (if (<= x 1) 1 (* x (fact (- x 1)))))
```

This expression declares the symbol `fact` to represent a function taking one parameter `x`. `fact` computes x factorial (assuming x to be a positive integer) by recursive self-application. As the syntax of Lisp is already a rough

approximation to lambda calculus, the computational part of this definition may be transcribed as—

$\lambda X.$ (*\$If* (\$≤ X 1) (*\$Pro* 1) (*\$Pro* (\$* X (FACT (\$– X 1)))))

Now by repeated application of the compilation and optimization rules (partially shown in figure 4.19) this is converted to a combinator expression—

```
(S (S (S
        (K $If) (S $≤ (K 1)))
      (K ($Pro 1)))
   (S (K $Pro) (S $*
        (S (K FACT) (S $– (K 1))))))
```

4.4.4 Basic Architecture

A straightforward hardware implementation of a combinator engine has been proposed in [Dickinson89], and is shown in figure 4.20.

The system consists of a large number of combinator processors interspersed with buffers, connected along a unidirectional data *stream*. Data flows down the stream subject to handshaking signals— a processor may need to locally block the stream, causing upstream sites to retain their data until the blockage is cleared. There is a special *null* token which has no meaning as data, and may be overwritten arbitrarily. At regular intervals all stream elements emit a token— nulls are to be written if no other useful information is available. These nulls give the stream a limited amount of elasticity— for when a processor blocks and there are nulls in the stream buffer immediately upstream, the effect of the blockage is localized until the space taken by the nulls is filled (the stream buffers are essentially FIFOs).

From	Rule	To
$\lambda X.$ ($If ($≤ X 1) ($Pro 1) ($Pro ($* X (FACT ($− X 1)))))	$\xrightarrow{S}$	$\mathcal{S}$ ($\lambda X.$ ($If ($≤ X 1) ($Pro 1))) ($\lambda X.$ (($Pro ($* X (FACT ($− X 1))))))
$\lambda X.$ ($If ($≤ X 1) ($Pro 1))	$\xrightarrow{S}$	$\mathcal{S}$ ($\lambda X.$ ($If ($≤ X 1))) ($\lambda X.$ (($Pro 1)))
$\lambda X.$ ($If ($≤ X 1))	$\xrightarrow{S}$	$\mathcal{S}$ ($\lambda X.$ ($If)) ($\lambda X.$ (($≤ X 1)))
$\lambda X.$ $If	$\xrightarrow{K}$	$\mathcal{K}$ $If
$\lambda X.$ ($≤ X 1)	$\xrightarrow{S}$	$\mathcal{S}$ ($\lambda X.$ ($≤ X)) ($\lambda X.$ (1))
$\lambda X.$ ($≤ X)	$\xrightarrow{S}$	$\mathcal{S}$ ($\lambda X.$ ($≤)) ($\lambda X.$ (X))
$\lambda X.$ $≤	$\xrightarrow{K}$	$\mathcal{K}$ $≤
$\lambda X.$ X	$\xrightarrow{I}$	$\mathcal{I}$
($\mathcal{S}$ ($\mathcal{K}$ $≤) $\mathcal{I}$)	$\xrightarrow{=}$	$≤
$\lambda X.$ 1	$\xrightarrow{K}$	$\mathcal{K}$ 1
$\lambda X.$ ($Pro 1)	$\xrightarrow{S}$	$\mathcal{S}$ ($\lambda X.$ ($Pro)) ($\lambda X.$ (1))
$\lambda X.$ $Pro	$\xrightarrow{K}$	$\mathcal{K}$ $Pro
$\lambda X.$ 1	$\xrightarrow{K}$	$\mathcal{K}$ 1
($\mathcal{S}$ ($\mathcal{K}$ $Pro) ($\mathcal{K}$ 1))	$\xrightarrow{=}$	($\mathcal{K}$ ($Pro 1))
$\lambda X.$ ($Pro ($* X (FACT ($− X 1))))	$\xrightarrow{S}$	$\mathcal{S}$ ($\lambda X.$ ($Pro)) ($\lambda X.$ (($* X (FACT ($− X 1)))))
⋮	⋮	⋮

Note: The lines marked $\xrightarrow{=}$ denote optimizations.

Figure 4.19: Compilation of `fact`

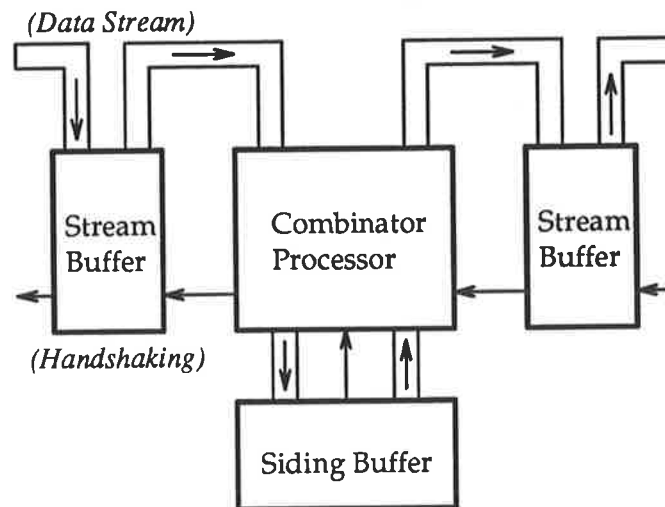


Figure 4.20: Combinator Engine Architecture

Additionally, each processor has a secondary buffer known as its *siding* for storing temporary values.

This architecture calls for a minimalist design philosophy. The utility of the combinator processor derives not from any inherent sophistication but from using a large number of them— generally speaking, the combinator processor should be as small and uncomplicated as possible.

4.4.5 Tokens

Given the combinator processor operates on a stream of tokens, what then are these tokens? The useful tokens can be divided into several categories—

True Combinators: $\mathcal{S}$, $\mathcal{K}$ are mandatory. $\mathcal{I}$ is a trivial addition. The other optimizing combinators ($\mathcal{B}$, $\mathcal{C}$, $\mathcal{S}'$, $\mathcal{B}*$ and $\mathcal{C}'$) are optional.

Pseudo-Combinators: Constructs such as *$If* (section 4.4.2).

Symbolic Constants: Tokens representing constant objects— such as the boolean false (*NIL*) and boolean true (*T*) values, and potentially others.

Functions: Function tokens are effectively a combination of constant and pseudo-combinator. A function token stands for a constant combinator expression— this replacement occurs subject to conditions similar to those for firing pseudo-combinators. In section 4.4.3, *FACT* is an example of a function token.

Numbers: Numbers are fundamentally useful.

Cons: A token to allow data to be structured into lists. (Existence of *$CONS* suggests some more pseudo-combinators to do car and cdr ("first" and "rest") type operations.)

Meta-Tokens: Very special tokens that have no real significance as operators or data, but are necessary to the operation of the system, perhaps to act as helpers to evaluation. For example the elasticity of the stream requires the existence of the null token *$NULL*.

Note: This document shows true combinators in a calligraphic font, pseudo-combinators in mixed case italic with a prepended $, constants and functions in upper case italic, and everything else in upper case italic with a prepended $ except numeric tokens which have a prepended #.

Given the above set of tokens, tables 4.9 through 4.12 show an extended set of true and pseudo-combinators. The pseudo-combinators are chosen to support compilation of a restricted Lisp-like language with no side effects, and some list processing and numeric capability. Each combinator has its firing preconditions— an associated minimum argument count ($Argc_{min}$) and types for a number of following arguments. The designation *Terminal* is the

least specific type restriction— it implies an argument which is incapable of being reduced further, and thus does not consist of combinators of any description, (tokens of Constant, Number or Cons type are acceptable).

LHS	$Argc_{min}$	Argument Types	RHS
$\mathcal{S}$ f g x	3	Dont care	f x (g x)
$\mathcal{K}$ c x	2	Dont care	c
$\mathcal{I}$ x	1	Dont care	x
$\mathcal{B}$ f g x	3	Dont care	f (g x)
$\mathcal{C}$ f g x	3	Dont care	f x g
$\mathcal{S}'$ c f g x	4	Dont care	c (f x) (g x)
$\mathcal{B}*$ c f g x	4	Dont care	c (f (g x))
$\mathcal{C}'$ c f g x	4	Dont care	c (f x) g

Table 4.9: True Combinators

LHS	$Argc_{min}$	Argument Types		RHS
$If c t e	3	Terminal	Dont care	{ t, e }
$Not x	1	Terminal		{*NIL*, *T*}
$Symbolp x	1	Terminal		{*NIL*, *T*}
$Numberp x	1	Terminal		{*NIL*, *T*}
$Consp x	1	Terminal		{*NIL*, *T*}
$Car x	1	Cons		(car x)
$Cdr x	1	Cons		(cdr x)
$Eq a b	2	Terminal	Terminal	{*NIL*, *T*}

Table 4.10: Pseudo-Combinators

LHS	$Argc_{min}$	Argument Types		RHS
$= n1 n2	2	Number	Number	$\{NIL, T\}$
$≠ n1 n2	2	Number	Number	$\{NIL, T\}$
$> n1 n2	2	Number	Number	$\{NIL, T\}$
$< n1 n2	2	Number	Number	$\{NIL, T\}$
$≥ n1 n2	2	Number	Number	$\{NIL, T\}$
$≤ n1 n2	2	Number	Number	$\{NIL, T\}$
$+ n1 n2	2	Number	Number	$n1 + n2$
$− n1 n2	2	Number	Number	$n1 - n2$
$* n1 n2	2	Number	Number	$n1 \times n2$
$/ n1 n2	2	Number	Number	$n1/n2$
$*Mod* n1 n2	2	Number	Number	$n1 \bmod n2$
$*1+* n	1	Number		$n + 1$
$*1-* n	1	Number		$n - 1$

Table 4.11: Arithmetic Pseudo-Combinators

LHS	$Argc_{min}$	Argument Types		RHS
$*LogNot* n	1	Number		$\bar{n}$
$*LogAnd* n1 n2	2	Number	Number	$n1 \text{and} n2$
$*LogIor* n1 n2	2	Number	Number	$n1 \text{or} n2$
$*LogXor* n1 n2	2	Number	Number	$n1 \text{xor} n2$

Table 4.12: Logical Pseudo-Combinators

4.4.6 Argument Counting

The use of (and) to convey grouping information is inefficient as a stream-based processor can not determine whether to fire (for example) an $\mathcal{S}$ until three complete arguments have been read (and presumably stored for processing). A superior scheme is for the number of arguments available to each token to be embedded in the token, allowing firing decisions to be made before any arguments need be read. The transformation is a simple "flattening" of the list structure—

$(\mathcal{S}\ (\mathcal{S}\ (\mathcal{S}$

$\qquad\qquad (\mathcal{K}\ \$\mathit{If})\ (\mathcal{S}\ \$\leq\ (\mathcal{K}\ 1)))$

$\qquad\quad (\mathcal{K}\ (\$\mathit{Pro}\ 1)))$

$\quad (\mathcal{S}\ (\mathcal{K}\ \$\mathit{Pro})\ (\mathcal{S}\ \$*$

$\qquad\qquad (\mathcal{S}\ (\mathcal{K}\ \mathrm{FACT})\ (\mathcal{S}\ \$-\ (\mathcal{K}\ 1))))))$

—becomes—

$\mathcal{S}{:}2\ \mathcal{S}{:}2\ \mathcal{S}{:}2$

$\qquad\qquad \mathcal{K}{:}1\ \$\mathit{If}{:}0\ \mathcal{S}{:}2\ \$\leq{:}0\ \mathcal{K}{:}1\ 1{:}0$

$\qquad\quad \mathcal{K}{:}1\ \$\mathit{Pro}{:}1\ 1{:}0$

$\quad \mathcal{S}{:}2\ \mathcal{K}{:}1\ \$\mathit{Pro}{:}0\ \mathcal{S}{:}2\ \$*{:}0$

$\qquad\qquad \mathcal{S}{:}2\ \mathcal{K}{:}1\ \mathrm{FACT}{:}0\ \mathcal{S}{:}2\ \$-{:}0\ \mathcal{K}{:}1\ 1{:}0$

In this form a processor can recognize argument boundaries with the following algorithm—

Algorithm 4.1 *Argument counting*
(given current token is the combinator or at end of argument)

$n = 1$
while $(n > 0)$
  Get next token
  $n = n - 1+$ *token argument count*
(current token is now the end of the next argument)

—which may be repeated as necessary to read multiple arguments.

4.4.7 Combinator Execution Example

Following this paragraph is a trace of the execution of a combinator expression interpreter as it evaluates combinators corresponding to the lisp expression `(fact 2)`. It contains two instances of a replacement of the function token *FACT*.

```
FACT:1 #2:0
S:3 S:2 S:2 K:1 $If:0 S:2 $≤:0 K:1 #1:0 K:1 $Pro:1 #1:0 S:2 K:1 $Pro:0 S:2 $*:0 S:2 K:1 FACT:0 S:2 $−:0 K:1 #1:0 #2:0
S:4 S:2 K:1 $If:0 S:2 $≤:0 K:1 #1:0 K:1 $Pro:1 #1:0 #2:0 S:3 K:1 $Pro:0 S:2 $*:0 S:2 K:1 FACT:0 S:2 $−:0 K:1 #1:0 #2:0
S:5 K:1 $If:0 S:2 $≤:0 K:1 #1:0 #2:0 K:2 $Pro:1 #1:0 #2:0 K:3 $Pro:0 #2:0 S:3 $*:0 S:2 K:1 FACT:0 S:2 $−:0 K:1 #1:0 #2:0
K:5 $If:0 #2:0 S:3 $≤:0 K:1 #1:0 #2:0 $Pro:1 #1:0 $Pro:1 $*:2 #2:0 S:3 K:1 FACT:0 S:2 $−:0 K:1 #1:0 #2:0
$If:3 $≤:2 #2:0 K:2 #1:0 #2:0 $Pro:1 #1:0 $Pro:1 $*:2 #2:0 S:3 K:1 FACT:0 S:2 $−:0 K:1 #1:0 #2:0
$If:3 $≤:2 #2:0 #1:0 $Pro:1 #1:0 $Pro:1 $*:2 #2:0 S:3 K:1 FACT:0 S:2 $−:0 K:1 #1:0 #2:0
$If:3 NIL:0 $Pro:1 #1:0 $Pro:1 $*:2 #2:0 S:3 K:1 FACT:0 S:2 $−:0 K:1 #1:0 #2:0
$*:2 #2:0 S:3 K:1 FACT:0 S:2 $−:0 K:1 #1:0 #2:0
$*:2 #2:0 K:3 FACT:0 #2:0 S:3 $−:0 K:1 #1:0 #2:0
$*:2 #2:0 FACT:1 $−:2 #2:0 K:2 #1:0 #2:0
$*:2 #2:0 FACT:1 $−:2 #2:0 #1:0
$*:2 #2:0 FACT:1 #1:0
```

$*:2 #2:0 S:3 S:2 S:2 K:1 $If:0 S:2 $≤:0 K:1 #1:0 K:1 $Pro:1 #1:0 S:2 K:1 $Pro:0 S:2 $*:0 S:2 K:1 FACT:0 S:2 $−:0 K:1 #1:0 #1:0

$*:2 #2:0 S:4 S:2 K:1 $If:0 S:2 $≤:0 K:1 #1:0 K:1 $Pro:1 #1:0 #1:0 S:3 K:1 $Pro:0 S:2 $*:0 S:2 K:1 FACT:0 S:2 $−:0 K:1 #1:0 #1:0

$*:2 #2:0 S:5 K:1 $If:0 S:2 $≤:0 K:1 #1:0 #1:0 K:2 $Pro:1 #1:0 #1:0 K:3 $Pro:0 #1:0 S:3 $*:0 S:2 K:1 FACT:0 S:2 $−:0 K:1 #1:0 #1:0

$*:2 #2:0 K:5 $If:0 #1:0 S:3 $≤:0 K:1 #1:0 #1:0 $Pro:1 #1:0 $Pro:1 $*:2 #1:0 S:3 K:1 FACT:0 S:2 $−:0 K:1 #1:0 #1:0

$*:2 #2:0 $If:3 $≤:2 #1:0 K:2 #1:0 #1:0 $Pro:1 #1:0 $Pro:1 $*:2 #1:0 S:3 K:1 FACT:0 S:2 $−:0 K:1 #1:0 #1:0

$*:2 #2:0 $If:3 $≤:2 #1:0 #1:0 $Pro:1 #1:0 $Pro:1 $*:2 #1:0 S:3 K:1 FACT:0 S:2 $−:0 K:1 #1:0 #1:0

$*:2 #2:0 $If:3 T:0 $Pro:1 #1:0 $Pro:1 $*:2 #1:0 S:3 K:1 FACT:0 S:2 $−:0 K:1 #1:0 #1:0

$*:2 #2:0 #1:0

#2:0

4.4.8 Stream

Given the token types in section 4.4.5, enough information is present to design the stream. As the stream must carry data and argument counts, and a likely execution bottleneck is the test for combinator readiness, it is probably unwise to use a complex bus encoding or transfer data and argument count sequentially. Similar reasoning applies to type information, as it is also implicated in the readiness test. Thus the obvious design for the stream bus is three parallel fields for argument count, raw data and tag (Type) bits.

With regard to sizings, tradition and general usefulness requires that the data be at least eight bits wide, with thirty-two a sensible value if serious arithmetic capacity is required— eight bits is chosen here for simplicity. There are seven basic token types,[3] thus the type part requires at least three

[3] Some freedom is desirable here as it may prove efficient to combine some types (for example, constants and functions), or add extra encodings (for example, represent $NULL as a distinct type so as to make it more easily recognizable by the buffers).

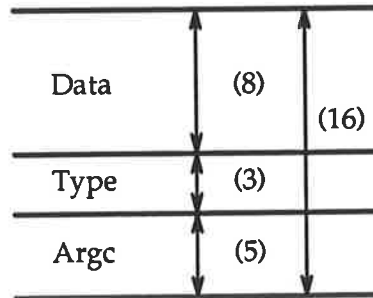


Figure 4.21: Stream

bits. The width of the argument count part defines an upper bound on the number of arguments a combinator or function may have. The choice of a five bit field gives a reasonably generous maximum argument count of thirty-one.

4.4.9 Control

Although this architecture is well suited to the use of asynchronous logic [Sutherland89], we assume the use of a conventional N-phase non-overlapping clock. The operation of combinator processors may depend on whether the downstream module is able to accept data, thus some care is necessary in assigning phases. Buffers do not have this sensitivity, thus the following order of *logical phases* is practical—

```
...Buffer-Write-Control
   Processor-Read-Control
      Processor-Write-Control
         Buffer-Read-Control
            Buffer-Write-Data
               Processor-Read-Data
                  Processor-Write-Data
```

```
Buffer-Read-Data...
```

It is interesting to note that while data flows downstream, control flows upstream!

For control purposes, no distinction is made between stream and siding buffers, or between any different type of processor attached to the stream. All stream elements are either buffers or processors, and a buffer must always be surrounded by processors, and vice versa.

4.4.10 Combinator Processor Design

Design of the combinator processor requires some analysis of the implementation requirements for execution of each combinator—

- $\mathcal{K}$ and $\mathcal{I}$ must be able to pass a complete argument from input to output. This requires arithmetic accumulation of the argument count field of successive tokens using algorithm 4.1.
- $\mathcal{K}$ must be able to nullify a complete argument by writing *$NULL* to the stream until the input argument is exhausted.
- $\mathcal{S}$ must be able to duplicate an argument, as it produces two copies of x.
- $\mathcal{S}$ must be able to interchange successive arguments, as g becomes interchanged with one x.
- An *$If* requires the processor to test its first argument, and either ignore the whole *$If* expression if the firing condition fails, or replace it with the *<then>* or *<else>* argument.

- *$Pro* requires that the processor pass both its arguments and the *$Pro* itself unevaluated. *$CONS* is similar.
- *$Car* and *$Cdr* require a similar test-and-replace operation to *$If*.
- *$Not*, *$Symbolp*, *$Numberp* and *$Consp* are also test-and-replace type operations, but the replacement token is always either T or *NIL*. Comparison operations are similar but are type specific in two arguments.
- The arithmetic and logical operations are mostly two argument type specific functions which return a number to the stream. Calculating the number may require significant circuitry.
- Function application is conditional on tests of potentially many arguments.

Good practice in processor design suggests that the choice of which combinators to directly implement should be driven by their relative frequency of occurrence in typical programs. Without extensive testing, a rough indication of these frequencies can be found by examining compiled combinator expressions produced for a number of simple quasi-Lisp expressions (some of which appear in section 4.4.16) used in the development and testing of the combinator compiler, interpreter and Loge model of the combinator engine—the results of which are summarized in table 4.13.

The most striking feature of this simple measurement is that true combinators account for more than three-quarters of all tokens, even though all instances of $\mathcal{I}$ were optimized away.[4] Other classes of operations were within a factor of two of each other, except for function application which is noticeably less frequent.

[4]Only $\mathcal{S}$, $\mathcal{K}$ and $\mathcal{I}$ are generated by the complier used for this test.

Operation	f %
Application	3
Arithmetic	7
List Processing	11
$If, *$Pro*	13
$\mathcal{I}$	0
$\mathcal{K}$	30
$\mathcal{S}$	36

Table 4.13: Rough Combinator Frequencies

Given the combination of relative infrequency and large functional requirements of arithmetic combinators and function application, these operations will be split off into separate or external processors. This leaves the combinator processor to execute only combinators that are at most type specific in the first argument— which can be expedited if a processor has access to the type bits of the leading token of its first argument. (Similarly, an arithmetic processor will require two token type lookahead— figure 4.22.)

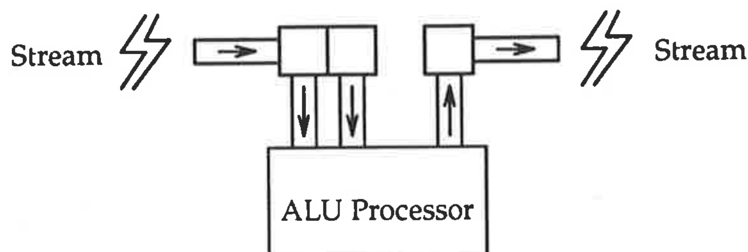


Figure 4.22: Arithmetic processor

To implement the other processor requirements listed above, some *microinstructions* are defined—

Wait Pass tokens from the stream input to stream output, checking for combinators that are ready to fire.

Pass Pass a complete argument from stream input to stream output.

Blot Nullify a complete argument— emit *$NULL* while reading from the stream input.

Copy Copy an argument from stream input to both the stream output and to the siding buffer.

Hold Read an argument from stream input into to the siding buffer only, while writing *$NULL* to the stream output.

Dump Empty an argument out of the siding buffer to the stream output, while blocking stream input.

Skip Nullify a single token only.

—with which the combinators may be broadly defined as microprograms. For example one might naively consider $\mathcal{S}$ to be—

```
pass   ; pass f
hold   ; save g in siding
copy   ; duplicate x
dump   ; emit saved g
dump   ; emit copy of x
```

Grouping causes a minor complication— the architectural implication of the parentheses in $f\ x\ (g\ x)$ is that the argument count of the first token of g

must be incremented to allow it to *capture* the following x. This refinement actually simplifies the $\mathcal{S}$ program to only require one Dump microinstruction. Similarly the argument counts for the f of an $\mathcal{S}$ and the c of a $\mathcal{K}$ need modification, as these *inherit* any other arguments of the firing combinator. As this process increases argument count fields, it often causes new combinators to become ready to fire— one may visualize this as combinators arising out of a morass of parentheses, thus the effect will be called *raising.*

\$If potentially takes either of two paths of execution, which is decided by testing its first argument for equality with *NIL*. This suggests an optimization (adopted from Common Lisp), that *NIL* be of a distinct type, allowing the branch choice to be decided as part of the firing condition analysis. An implementation that allows the alternate paths of execution is to cause the basic *\$If* to metamorphose into either $\$If_{true}$ or $\$If_{false}$, with the resulting microprograms—

$\$If_{true}$		$\$If_{false}$	
blot	; forget *<test>*	blot	; forget *<test>*
skip	; strip *\$Pro*	blot	; omit *<then>*
pass	; enable *<then>*	skip	; strip *\$Pro*
blot	; omit *<else>*	pass	; enable *<else>*

—and similarly for the predicates and list processing combinators.

4.4.11 Combinator Processor: A Loge Functional Description

The preceding sections have clarified the design requirements sufficiently to allow the development of a Loge HDL description of a simple combinator processor implementation.

Conventions Some common conventions are adopted in all the following descriptions of modules which connect to the stream. Firstly, in all cases stream modules will have a stream input port called `StreamI`, and a stream output port called `StreamO`. Similarly, blocking information will be transferred through ports named `StreamBI` and `StreamBO`. As an additional discipline, the blocking ports are accessed solely through the access predicate `getblock` and update function `setblock`.

All stream modules will also have an input port `Phase`, which will be set to successive stream phases as defined in section 4.4.9. This is the sole port on any stream module which need be watched for events— all other inputs are to be sampled at the start of the appropriate phase.

Some inconsistencies in syntax may appear— these are due to layout editing and omissions in the interest of brevity. However, since a secondary goal of this section is to show the conciseness of representation available in Loge models it should be noted that at least in the case of the Combinator-Processor model, the only excisions are large blocks of comments.

Combinator-Processor: Declaration

```
(defmodule combinator-processor (
    StreamI StreamO StreamBI StreamBO
    SidingI SidingO SidingBI SidingBO Phase
    &aux dataIn dataOut blockage
    proc-type proc-state proc-argc proc-raise)
```

Notes: Declare the stream and siding ports, plus variables local to this model.

Combinator-Processor: Build

```
(build
```

```
    (StreamI  I-port stream-width StreamI)
    (StreamO  O-port stream-width StreamO)
    (StreamBI I-port 1            StreamBI)
    (StreamBO O-port 1            StreamBO)
    (SidingI  I-port stream-width SidingI)
    (SidingO  O-port stream-width SidingO)
    (SidingBI I-port 1            SidingBI)
    (SidingBO O-port 1            SidingBO)
    (Phase    I-port phase-width  Phase))
```

Notes: Build the stream and siding interface. Note the siding ports follow similar conventions to the stream ports.

Combinator-Processor: Init

```
(init
    (ignore-ports StreamI StreamO StreamBI StreamBO
      SidingI SidingO SidingBI SidingBO)
    (watch-ports Phase)
    (setq
      proc-state wait_u
      proc-type nil
      proc-argc 1
      proc-raise 0
      dataIn stream-null
      dataOut stream-null
      blockage (cons nil nil)))
```

Notes: Establish watch/ignore strategy, and initialize all the following internal variables—

`proc-state`	The current microinstruction
`proc-type`	The combinator currently being fired (irrelevant when waiting)
`proc-argc`	The current argument count (when waiting this may be any non-zero value)
`proc-raise`	The amount that the next argument must be raised
`dataIn`	The most recently read token, used for type lookahead
`dataOut`	The token under consideration for output
`blockage`	Scratch variable to contain blocking inputs

Combinator-Processor: Run (Control)

```
(cond
    ((on Phase B-Write-Control)
      ; If out of arguments, change state.
      (if (<= proc-argc 0) (step-combinator)))
    ((on Phase P-Read-Control)
      (setq blockage (cons (getblock StreamBI)
          (getblock SidingBI))))
    ((on Phase P-Write-Control) (cond
        ((= proc-state wait_u)
          (setq blockage (car blockage))
          (setblock StreamBO blockage)
          (setblock SidingBO t))
        ((= proc-state pass_u)
          (setq blockage (car blockage))
          (setblock StreamBO blockage)
          (setblock SidingBO t))
        ((= proc-state blot_u)
          (setq blockage nil)
          (setblock StreamBO nil)
          (setblock SidingBO t))
```

```
((= proc-state copy_u)
  (setq blockage (or
      (car blockage) (cdr blockage)))
  (setblock StreamBO blockage)
  (setblock SidingBO t))
((= proc-state hold_u)
  (setq blockage (cdr blockage))
  (setblock StreamBO blockage)
  (setblock SidingBO t))
((= proc-state dump_u)
  (setq blockage (car blockage))
  (setblock StreamBO t)
  (setblock SidingBO blockage))
((= proc-state skip_u)
  (setq blockage (car blockage))
  (setblock StreamBO blockage)
  (setblock SidingBO t))))
```

Notes: When the processor is awakened, operation depends on the system Phase. During the Read-Control phase a simple sampling of the blocking inputs is made. Write-Control is more complex, with the writing of the upstream buffer and local siding buffer block control outputs depending on the current microinstruction and the input blocking values. Note that with this version of the combinator processor model, once the siding is full, the assertion of its blocking output will prevent further operation of microinstructions that write to the siding. The `blockage` variable will be true if external conditions will prevent useful work this cycle.

As an optimization, the processor changes internal state in response to exhaustion of an argument list during the buffer Write-Control phase— this could be done at other times, however this phase is appropriate as the processor is otherwise inactive at this point. The details of the state change are handled by the function `step-combinator`.

Combinator-Processor: Run (Data)

```
((on Phase P-Read-Data)
    (if blockage
      (setq dataOut stream-null)
      ;else
      (cond
        ((= proc-state wait_u)
          (setf dataOut dataIn dataIn !StreamI))
        ((= proc-state pass_u)
          (setf dataOut dataIn dataIn !StreamI))
        ((= proc-state blot_u)
          (setf dataOut dataIn dataIn !StreamI))
        ((= proc-state copy_u)
          (setf dataOut dataIn dataIn !StreamI))
        ((= proc-state hold_u)
          (setf dataOut dataIn dataIn !StreamI))
        ((= proc-state dump_u)
          (setf dataOut !SidingI))
        ((= proc-state skip_u)
          (setf dataOut dataIn dataIn !StreamI)))
      (setq dataOut (examine-data dataOut))))
  ((on Phase P-Write-Data)
    (cond
      ((= proc-state wait_u) (setf
          !StreamO dataOut !SidingO stream-null))
      ((= proc-state pass_u) (setf
          !StreamO dataOut !SidingO stream-null))
      ((= proc-state blot_u) (setf
          !StreamO dataOut !SidingO stream-null))
      ((= proc-state copy_u) (setf
          !StreamO dataOut !SidingO dataOut))
      ((= proc-state hold_u) (setf
          !StreamO stream-null !SidingO dataOut))
      ((= proc-state dump_u) (setf
          !StreamO dataOut !SidingO stream-null))
      ((= proc-state skip_u) (setf
          !StreamO stream-null !SidingO stream-null))))
  (t nil)))
```

Notes: During the processor Read-Data phase, the appropriate port (if any) for the current microinstruction is read into **dataIn** while the previous value of **dataIn** is processed (in function **examine-data**) and transferred to **dataOut**. The dump state is exceptional in taking input from the siding rather than the stream. Then in Write-Data, **dataOut** is written to the the appropriate port.

Combinator-Processor: `examine-data`

```
(defun examine-data (data)
  ; Run the processor on new data, return "output" value
  (multiple-value-bind (arg argt argc)
    (unpack-stream-data data)
    (if (not (= argt $nul_t))
      (cond
        ((= proc-state wait_u)
          ; Waiting for a ready combinator...
          (if (and
              (or (= $com_t argt) (= $sym_t argt))
              (db-fireable-p arg argc (list dataIn)))
            ; ...got one!
            (setq data (load-combinator arg argc data
                (stream-type-part dataIn))))
          ) ;else just pass the data on
        ((= proc-state skip_u)
          ; Skip operates on one token only
          (setq data stream-null
            proc-raise 0
            proc-argc 0))
        ((= proc-state blot_u)
          ; Blot always destroys data
          (setq data stream-null
            proc-raise 0
            proc-argc (+ proc-argc argc -1)))
        (t ; (standard behaviour)
```

```
          (setq data (pack-stream-data arg argt
              (+ proc-raise argc))
            proc-raise 0
            proc-argc (+ proc-argc argc -1))
          ))))
  data)
```

Notes: Always provided the data is not null, the usual action of an active processor is to return data with a count field increased by the current value of `proc-raise` (which is then zeroed), and to continue the usual argument counting operation. If however the processor is waiting it must test the data to see if it is a combinator which is ready to fire— this case is handled by the function `load-combinator`. Another variation is that when skipping the data is nullified, and `proc-argc` is zeroed to force an immediate change of state. Similarly in a blot instruction the data is nullified, no raising is performed, but normal argument counting continues.

Combinator-Processor: `load-combinator`

```
(defun load-combinator (arg argc data auxt)
  ; The processor is to fire combinator ARG
  ; Return the data to be passed—
  ;   usually null, or the result of a predicate.
  ; Set proc-type, proc-state, proc-argc and proc-raise.
  ; Use type lookahead in AUXT as required.
  (setq proc-type arg)
  (cond
    ((= proc-type $S) (setq
        proc-state pass_u proc-argc 1
        proc-raise (1- argc)
        data stream-null))
    ((= proc-type $K) (setq
        proc-state pass_u proc-argc 1
        proc-raise (- argc 2)
        data stream-null))
```

```
  ((= proc-type $I) (setq
      proc-state pass_u proc-argc 1
      proc-raise (1- argc)
      data stream-null))
  ((= proc-type $pro)
    (setq ; Pass $pro
      proc-state pass_u proc-argc 1 proc-raise 0))
  ((= proc-type $cons)
    (setq ; Pass $cons
      proc-state pass_u proc-argc 2 proc-raise 0))
  ((= proc-type $not) (setq
      proc-state blot_u proc-argc 1 proc-raise 0
      data (if (= auxt $nil_t) stream-t stream-nil)))
  ((= proc-type $consp) (setq
      proc-state blot_u proc-argc 1 proc-raise 0
      data (if (= auxt $cns_t) stream-t stream-nil)))
  ((= proc-type $symbolp) (setq
      proc-state blot_u proc-argc 1 proc-raise 0
      data (if (= auxt $sym_t) stream-t stream-nil)))
  ((= proc-type $numberp) (setq
      proc-state blot_u proc-argc 1 proc-raise 0
      data (if (= auxt $num_t) stream-t stream-nil)))
  ((= proc-type $car) (setq
      proc-state skip_u proc-argc 1 proc-raise 0
      data stream-null))
  ((= proc-type $cdr) (setq
      proc-state skip_u proc-argc 1 proc-raise 0
      data stream-null))
  ((= proc-type $if)
    (if (= auxt $nil_t)
      (setq proc-type $nil
        proc-state blot_u proc-argc 2 proc-raise 0
        data stream-null)
      (setq
        proc-state blot_u proc-argc 1 proc-raise 0
        data stream-null)))
  (t (setq proc-argc 1 proc-raise 0)))
data)
```

Notes: This function initializes the processor for a firing of a particular combinator. The data to output this cycle is returned— it is often null, except in the case of predicate functions which may be immediately evaluated. Unrecognized functions are simply ignored by failing to emerge from the wait microinstruction.

Combinator-Processor: `step-combinator`

```
(defun combinator-done ()
  (setq
    proc-state wait_u proc-argc 1
    proc-raise 0 proc-type 0))

(ifndefun step-combinator ()
  ; The processor has used up an argument— change state.
  (cond
    ((= proc-type $S) (cond
        ((= proc-state pass_u)
          (setq proc-state hold_u proc-argc 1
            proc-raise 1))
        ((= proc-state hold_u)
          (setq proc-state copy_u proc-argc 1
            proc-raise 0))
        ((= proc-state copy_u)
          ; one dump only— the siding has been raised
          (setq proc-state dump_u proc-argc 1
            proc-raise 0))
        ((= proc-state dump_u)
          (combinator-done))
        (t nil)))
    ((= proc-type $K) (cond
        ((= proc-state pass_u)
          (setq proc-state blot_u proc-argc 1
            proc-raise 0))
        ((= proc-state blot_u)
          (combinator-done))
        (t nil)))
```

```
((= proc-type $I) (cond
    ((= proc-state pass_u)
      (combinator-done))
    (t nil)))
((= proc-type $pro) (cond
    ((= proc-state pass_u)
      (combinator-done))
    (t nil)))
((= proc-type $if)
  ; $if nil → $nil, but $if t → $if, then $t
  (setq proc-type $t proc-state skip_u proc-argc 1
    proc-raise 0))
((= proc-type $nil) (cond
    ((= proc-state blot_u)
      (setq proc-state skip_u proc-argc 1
        proc-raise 0))
    ((= proc-state skip_u)
      (setq proc-state pass_u proc-argc 1
        proc-raise 0))
    ((= proc-state pass_u)
      (combinator-done))
    (t nil)))
((= proc-type $t) (cond
    ((= proc-state skip_u)
      (setq proc-state pass_u proc-argc 1
        proc-raise 0))
    ((= proc-state pass_u)
      (setq proc-state blot_u proc-argc 1
        proc-raise 0))
    ((= proc-state blot_u)
      (combinator-done))
    (t nil)))
((member proc-type
    (list $not $consp $symbolp $numberp)) (cond
    ((= proc-state blot_u)
      (combinator-done))
    (t nil)))
((= proc-type $cons) (cond
```

```
        ((= proc-state pass_u)
          (combinator-done))
        (t nil)))
    ((= proc-type $car) (cond
        ((= proc-state skip_u)
          (setq proc-state pass_u proc-argc 1
            proc-raise 0))
        ((= proc-state pass_u)
          (setq proc-state blot_u proc-argc 1
            proc-raise 0))
        ((= proc-state blot_u)
          (combinator-done))
        (t nil)))
    ((= proc-type $cdr) (cond
        ((= proc-state skip_u)
          (setq proc-state blot_u proc-argc 1
            proc-raise 0))
        ((= proc-state blot_u)
          (setq proc-state pass_u proc-argc 1
            proc-raise 0))
        ((= proc-state pass_u)
          (combinator-done))
        (t nil)))
    (t
      (error "Step: Type ~S not supported~%" proc-type))))
```

Notes: This function handles transitions between microinstructions depending solely on the currently executing combinator type and current state (a slight infelicity in the handling of *$If* makes this possible). At each transition `proc-argc` is set to the number of arguments to be processed in this state. `proc-raise` is also set, usually to zero.

4.4.12 Combinator Processor: Hardware Partitioning

Derivation of the Combinator-Processor model required a non-trivial amount of thought, refinement and testing. However this effort yields benefits such as—

- Identifying errors, optimizations and generally troublesome areas.
- Building confidence in the practicality of the architecture.
- Clarification of the design.
- A starting point for further specification.

As an illustration of the last point, given the preceding functional description, it is possible to directly transcribe a hardware partitioning, using simple heuristics such as dedicating a register to each processor local variable. After a modicum of consideration, the structure of figure 4.23 is reached.

Registers `S`, `T`, `C`, `R`, `I` correspond directly to local variables— `proc-state`, `proc-type`, `proc-argc`, `proc-raise` and `dataIn`. Variable `dataOut` is distributed over the `O*` registers.

The block labelled `Control` contains the basic state machine. It is more highly interconnected with other blocks of the processor than it is possible to show clearly. The `Fire?` block contains the complicated expression for combinator readiness and provides a qualified data path from register `I` to register `T`.

Two arithmetic units are shown. The upper one is simply an adder which adds the contents of `R` to the argument count field of a stream token. `R` may need to be loaded with constant values {-2, -1, 0, 1}, and also to accumulate the result of an addition (to allow arguments of $\mathcal{S}$ and $\mathcal{K}$ to be raised).

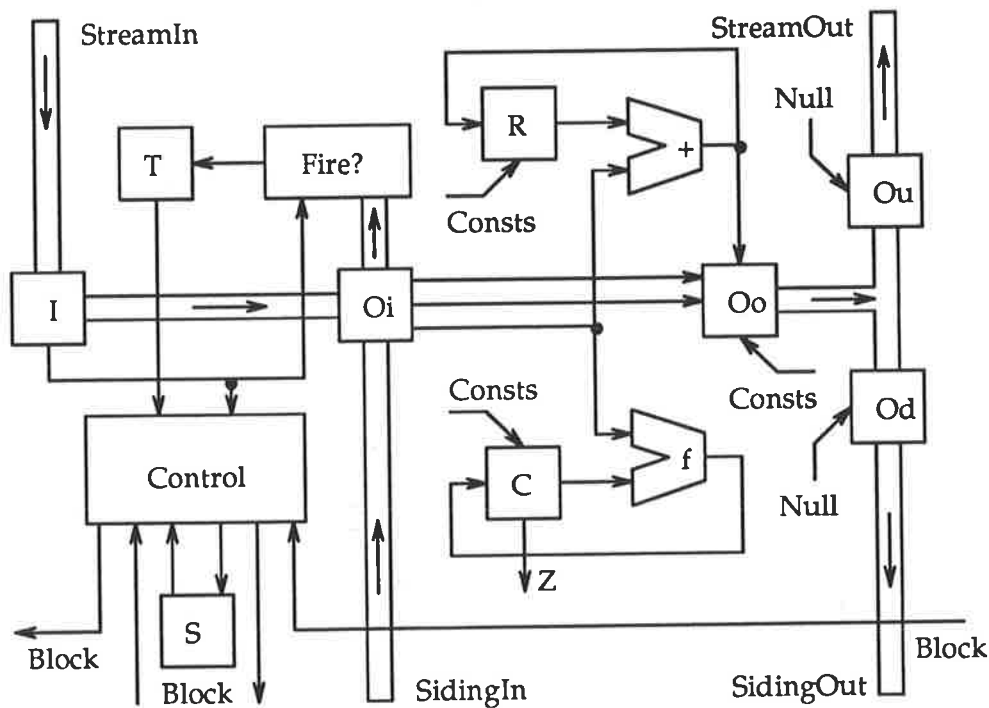


Figure 4.23: Combinator Processor Design

The lower arithmetic unit is always used to accumulate argument counts. Its precise function is add-with-decrement and accumulate. It has a zero detection output `Z`.

Register `Oi` may accept data from `I` or direct from the data siding. `Oo` usually accepts data from `Oi` and the raised argument count, however it may also be set to certain stream constants, such as *$NULL*, *T* and *NIL* (when blotting or firing a predicate). Data from `Oo` is passed to independently nullifiable registers `Ou` and `Od` connected to the stream and siding outputs respectively.

4.4.13 Other stream elements

The ALU processor is a straightforward stream element that accumulates three tokens. If the leading token is a ready arithmetic or logical combinator and the second and third tokens are numbers, the combinator is fired and the result output to the stream. Otherwise, the tokens are simply passed through the processor. Note that this processor does not respect *$Pro* tokens, thus protected arithmetic expressions may be unexpectedly executed. Its functional description is similar (but shorter) than that of the combinator processor and is omitted. Also omitted is the specification of the buffers— these are multiple registers configured as a FIFO which ignore *$NULL*, and are thus relatively trivial.

4.4.14 Stream Group

Given buffers, ALU and combinator processor definitions, the next level of hierarchy is a stream of N processors, interspersed with buffers, and concluding with an ALU processor. Omitted is an intermediate stage in which modules Processor-Group and ALU-Group are defined— these consist of a buffer preceding a processor, with a siding buffer for a combinator processor.

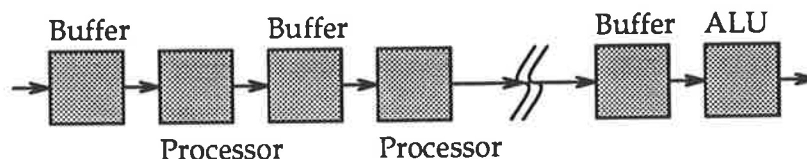


Figure 4.24: Stream Group

```
(defmodule stream-group (n
    StreamI StreamO StreamBI StreamBO Phase
```

```
    &aux nbl nst pgr agr xxx)
  (build
    (StreamI  I-port stream-width StreamI)
    (StreamO  O-port stream-width StreamO)
    (StreamBI I-port 1            StreamBI)
    (StreamBO O-port 1            StreamBO)
    (Phase    I-port phase-width  Phase)
    (nbl      node '(,n 1))
    (nst      node '(,n ,stream-width))
    (pgr processor-group n (module-vector i
        at 0 (
          (nodes StreamI)
          (nth 0      nst)
          (nth 0      nbl)
          (nodes StreamBO)
          (nodes Phase))
        over 1 n (
          (nth (1- i) nst)
          (nth i      nst)
          (nth i      nbl)
          (nth (1- i) nbl)
          (nodes Phase))))
    (agr      alu-group () (list
        (nth (1- n) nst)
        (nodes StreamO)
        (nodes StreamBI)
        (nth (1- n) nbl)
        (nodes Phase)))
    )
  (setf (module-mode (the-module)) :structural)
  (structural-module-only)
  ()
)
```

4.4.15 Top Level

The top level tester module instantiates a Stream-Group of a specified size, which is connected in a loop through a Stream-Circle module (figure 4.25). A Stream-Circle is a combination of stream source and sink— given an initial combinator expression at initialization, this is written to the input of the Stream-Group, while any output is accumulated into a result expression. Once the Stream-Group has completely processed the input expression (detected by using an "end of stream" meta-token), the input and output expressions are compared— if they are identical then there is no further possibility of reduction and the simulation is terminated, otherwise a new *stream cycle* is begun, using the previous result as the new input to the Stream-Group. The Stream-Circle also scans expressions for functions which are ready to fire, replacing them with their combinator expression where possible.

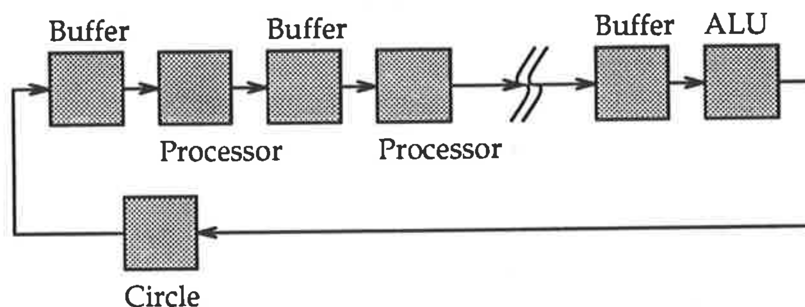


Figure 4.25: Stream-Group, with Stream-Circle

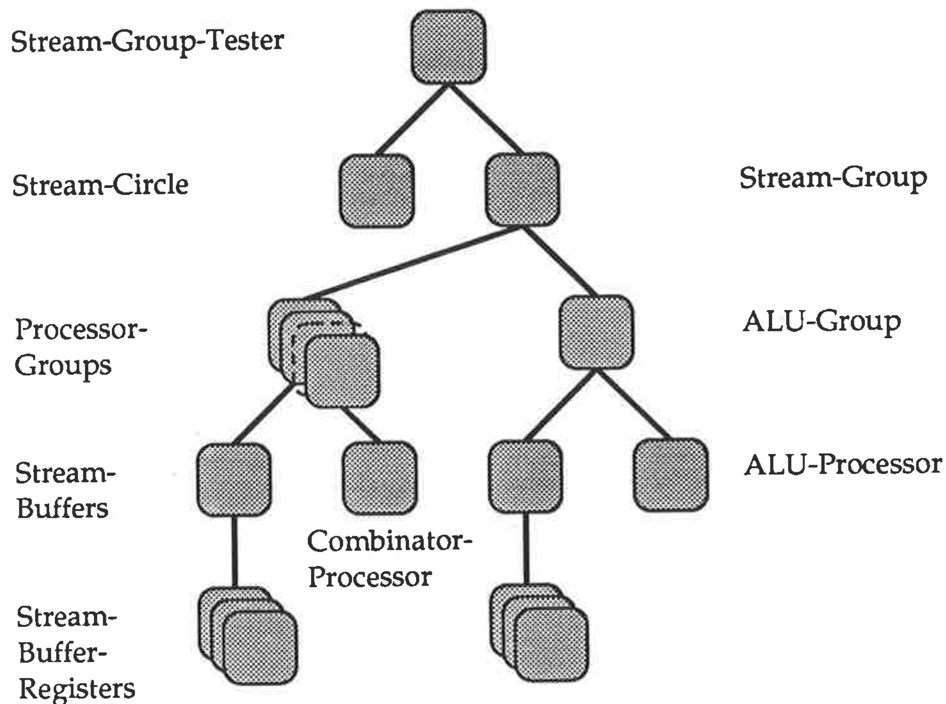


Figure 4.26: Complete Stream-Group-Tester module partitioning

4.4.16 Simulations

Four processor simulation of `(fact 2)`

The **`test-stream-group`** function produces the same final result as the interpreter of section 4.4.7. Not surprisingly however, the intermediate stages are somewhat different. To closely monitor an N processor simulation, stream monitoring modules are connected to the output of each stream processor. The gist of these **`Stream-Sink`** modules is simply to sample the stream during the buffer's **`Read`** phase and output a triple of (*<clock tick>* *<node index>* *<stream value>*)— this data can then be post-processed with a trivial filter, producing the following trace—

Four Processor Stream-Group `(fact 2)`

Input	1	2	3	4	ALU
$\mathcal{S}$:3					
$\mathcal{S}$:2					
$\mathcal{S}$:2					
$\mathcal{K}$:1	$\mathcal{S}$:4				
$\$\mathit{If}$:0	$\mathcal{S}$:2				
$\mathcal{S}$:2	$\mathcal{K}$:1				
$\$\leq$:0	$\$\mathit{If}$:0	$\mathcal{S}$:5			
$\mathcal{K}$:1	$\mathcal{S}$:2	$\mathcal{K}$:1			
#1:0	$\$\leq$:0	$\$\mathit{If}$:0			
$\mathcal{K}$:1	$\mathcal{K}$:1	$\mathcal{S}$:2	$\mathcal{K}$:5		
$\$\mathit{Pro}$:1	#1:0	$\$\leq$:0	$\$\mathit{If}$:0		
#1:0	$\mathcal{K}$:1	$\mathcal{K}$:1			
$\mathcal{S}$:2	$\$\mathit{Pro}$:1	#1:0		$\$\mathit{If}$:3	
$\mathcal{K}$:1	#1:0				
$\$\mathit{Pro}$:0					
$\mathcal{S}$:2					$\$\mathit{If}$:3
$\$*$:0					
$\mathcal{S}$:2					
$\mathcal{K}$:1					
FACT:0					
$\mathcal{S}$:2					
$\$-$:0					
$\mathcal{K}$:1					
#1:0					
#2:0					
	#2:0				
	$\mathcal{S}$:3				
	$\mathcal{K}$:1	#2:0			
	$\$\mathit{Pro}$:0	$\mathcal{K}$:2			
	$\mathcal{S}$:2	$\$\mathit{Pro}$:1	#2:0		
	$\$*$:0	#1:0	$\mathcal{S}$:3		
	$\mathcal{S}$:2	#2:0	$\$\leq$:0		
	$\mathcal{K}$:1		$\mathcal{K}$:1		
	FACT:0	$\mathcal{K}$:3	#1:0	$\$\leq$:2	
	$\mathcal{S}$:2	$\$\mathit{Pro}$:0	#2:0		
	$\$-$:0				
	$\mathcal{K}$:1		$\$\mathit{Pro}$:1	#2:0	$\$\leq$:2
	#1:0		#1:0	$\mathcal{K}$:2	
	#2:0			#1:0	
				#2:0	#2:0

	$\$Pro{:}1$		$\mathcal{K}{:}2$
		$\$Pro{:}1$	$\#1{:}0$
		$\#1{:}0$	$\#2{:}0$
		$\$Pro{:}1$	
$\#2{:}0$			$\$Pro{:}1$
$\mathcal{S}{:}3$			$\#1{:}0$
$\$*{:}0$			$\$Pro{:}1$
$\mathcal{S}{:}2$			
$\mathcal{K}{:}1$	$\$*{:}2$		
FACT:0			
$\mathcal{S}{:}2$		$\$*{:}2$	
$\$-{:}0$			
$\mathcal{K}{:}1$			
$\#1{:}0$			$\$*{:}2$
$\#2{:}0$			

$\#2{:}0$		
$\mathcal{S}{:}3$		
$\mathcal{K}{:}1$	$\#2{:}0$	
FACT:0	$\mathcal{S}{:}3$	
$\mathcal{S}{:}2$	$\mathcal{K}{:}1$	
$\$-{:}0$	FACT:0	$\#2{:}0$
$\mathcal{K}{:}1$	$\mathcal{S}{:}2$	$\mathcal{S}{:}3$
$\#1{:}0$	$\$-{:}0$	$\mathcal{K}{:}1$
$\#2{:}0$	$\mathcal{K}{:}1$	FACT:0
	$\#1{:}0$	$\mathcal{S}{:}2$
	$\#2{:}0$	$\$-{:}0$
		$\mathcal{K}{:}1$
		$\#1{:}0$
		$\#2{:}0$

$\$If{:}3$					
$\$\leq{:}2$					
$\#2{:}0$	$\$If{:}3$				
$\mathcal{K}{:}2$	$\$\leq{:}2$				
$\#1{:}0$	$\#2{:}0$	$\$If{:}3$			
$\#2{:}0$		$\$\leq{:}2$			
$\$Pro{:}1$	$\#1{:}0$	$\#2{:}0$	$\$If{:}3$		
$\#1{:}0$			$\$\leq{:}2$		
$\$Pro{:}1$	$\$Pro{:}1$	$\#1{:}0$	$\#2{:}0$	$\$If{:}3$	
$\$*{:}2$	$\#1{:}0$			$\$\leq{:}2$	
$\#2{:}0$	$\$Pro{:}1$	$\$Pro{:}1$	$\#1{:}0$	$\#2{:}0$	
$\mathcal{S}{:}3$	$\$*{:}2$	$\#1{:}0$			$\$If{:}3$

K:1	#2:0	$Pro:1	$Pro:1	#1:0	$≤:2
FACT:0	S:3	$*:2	#1:0		#2:0
S:2	K:1	#2:0	$Pro:1	$Pro:1	
$−:0	FACT:0	S:3	$*:2	#1:0	#1:0
K:1	S:2	K:1	#2:0	$Pro:1	
#1:0	$−:0	FACT:0	S:3	$*:2	$Pro:1
#2:0	K:1	S:2	K:1	#2:0	#1:0
	#1:0	$−:0	FACT:0	S:3	$Pro:1
	#2:0	K:1	S:2	K:1	$*:2
		#1:0	$−:0	FACT:0	#2:0
		#2:0	K:1	S:2	S:3
			#1:0	$−:0	K:1
			#2:0	K:1	FACT:0
				#1:0	S:2
				#2:0	$−:0
					K:1
					#1:0
					#2:0

$If:3					
$≤:2					
#2:0	$If:3				
#1:0	$≤:2				
$Pro:1	#2:0	$If:3			
#1:0	#1:0	$≤:2			
$Pro:1	$Pro:1	#2:0	$If:3		
$*:2	#1:0	#1:0	$≤:2		
#2:0	$Pro:1	$Pro:1	#2:0	$If:3	
S:3	$*:2	#1:0	#1:0	$≤:2	
K:1	#2:0	$Pro:1	$Pro:1	#2:0	
FACT:0	S:3	$*:2	#1:0	#1:0	$If:3
S:2	K:1	#2:0	$Pro:1	$Pro:1	NIL:0
$−:0	FACT:0	S:3	$*:2	#1:0	
K:1	S:2	K:1	#2:0	$Pro:1	
#1:0	$−:0	FACT:0	S:3	$*:2	$Pro:1
#2:0	K:1	S:2	K:1	#2:0	#1:0
	#1:0	$−:0	FACT:0	S:3	$Pro:1
	#2:0	K:1	S:2	K:1	$*:2
		#1:0	$−:0	FACT:0	#2:0
		#2:0	K:1	S:2	S:3
			#1:0	$−:0	K:1
			#2:0	K:1	FACT:0
				#1:0	S:2

				#2:0	$−:0
					𝒦:1
					#1:0
					#2:0

$*If*:3					
NIL:0					
$*Pro*:1					
#1:0					
$*Pro*:1					
$∗:2					
#2:0					
𝒮:3	$∗:2				
𝒦:1	#2:0				
FACT:0	𝒮:3	$∗:2			
𝒮:2	𝒦:1	#2:0			
$−:0	FACT:0		$∗:2		
𝒦:1	𝒮:2	𝒦:3	#2:0		
#1:0	$−:0	FACT:0		$∗:2	
#2:0	𝒦:1			#2:0	
	#1:0		FACT:1		
	#2:0				$∗:2
				FACT:1	#2:0
		#2:0			
		𝒮:3			
		$−:0			FACT:1
		𝒦:1			
		#1:0	$−:2		
		#2:0			
				$−:2	
			#2:0		
			𝒦:2		
			#1:0	#2:0	$−:2
			#2:0		
				#1:0	
					#2:0
					#1:0

$∗:2	
#2:0	
FACT:1	$∗:2
$−:2	#2:0

#2:0	FACT:1	$∗:2			
#1:0	$−:2	#2:0			
	#2:0	FACT:1	$∗:2		
	#1:0	$−:2	#2:0		
		#2:0	FACT:1	$∗:2	
		#1:0	$−:2	#2:0	
			#2:0	FACT:1	
			#1:0	$−:2	$∗:2
				#2:0	#2:0
				#1:0	FACT:1
					#1:0

$∗:2					
#2:0					
S:3	$∗:2				
S:2	#2:0				
S:2		$∗:2			
K:1	S:4	#2:0			
$*If*:0	S:2		$∗:2		
S:2	K:1		#2:0		
$≤:0	$*If*:0	S:5		$∗:2	
K:1	S:2	K:1		#2:0	
#1:0	$≤:0	$*If*:0			
K:1	K:1	S:2	K:5		$∗:2
$*Pro*:1	#1:0	$≤:0	$*If*:0		#2:0
#1:0	K:1	K:1			
S:2	$*Pro*:1	#1:0		$*If*:3	
K:1	#1:0				
$*Pro*:0					
S:2					$*If*:3
$∗:0					
S:2					
K:1					
FACT:0					
S:2					
$−:0					
K:1					
#1:0					
#1:0					

	#1:0	
	S:3	
	K:1	#1:0

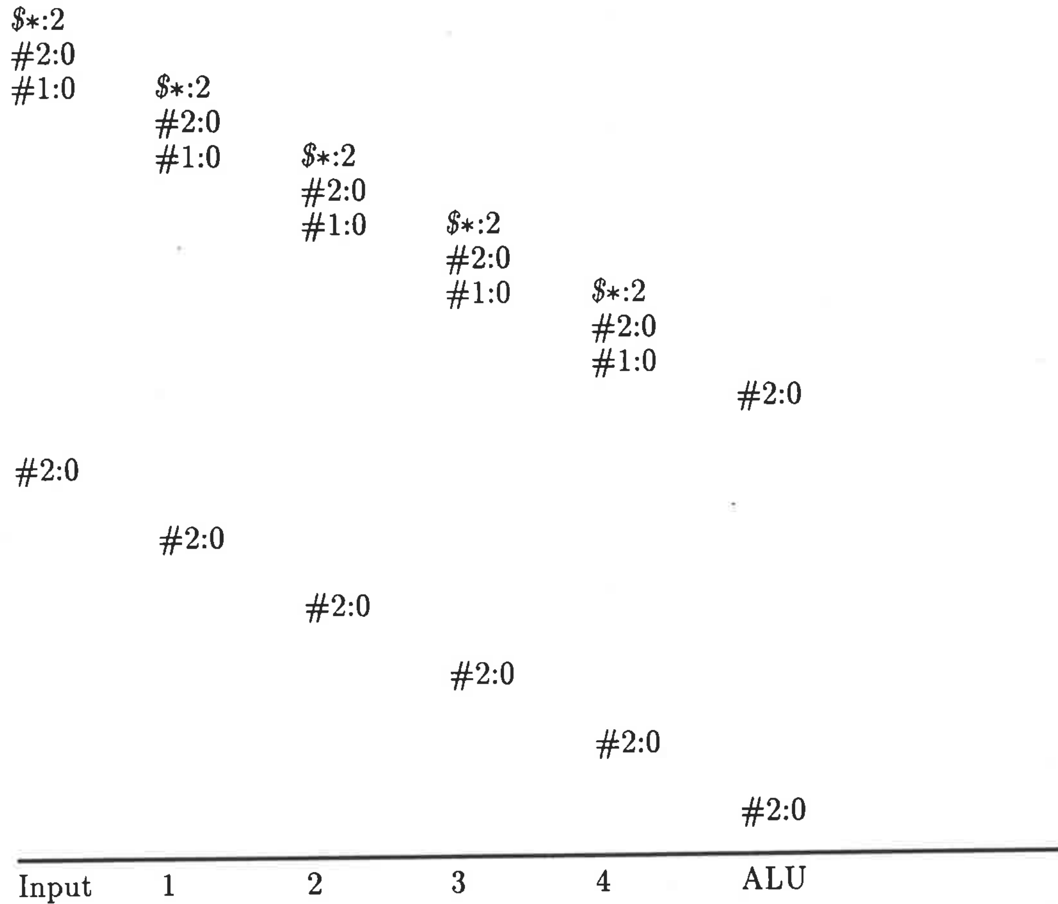
$\$Pro$:0	$\mathcal{K}$:2			
$\mathcal{S}$:2	$\$Pro$:1	#1:0		
$\$*$:0	#1:0	$\mathcal{S}$:3		
$\mathcal{S}$:2	#1:0	$\$\leq$:0		
$\mathcal{K}$:1		$\mathcal{K}$:1		
FACT:0	$\mathcal{K}$:3	#1:0	$\$\leq$:2	
$\mathcal{S}$:2	$\$Pro$:0	#1:0		
$\$-$:0				
$\mathcal{K}$:1		$\$Pro$:1	#1:0	$\$\leq$:2
#1:0		#1:0	$\mathcal{K}$:2	
#1:0			#1:0	
			#1:0	#1:0
		$\$Pro$:1		$\mathcal{K}$:2
			$\$Pro$:1	#1:0
			#1:0	#1:0
			$\$Pro$:1	
	#1:0			$\$Pro$:1
	$\mathcal{S}$:3			#1:0
	$\$*$:0			$\$Pro$:1
	$\mathcal{S}$:2			
	$\mathcal{K}$:1	$\$*$:2		
	FACT:0			
	$\mathcal{S}$:2		$\$*$:2	
	$\$-$:0			
	$\mathcal{K}$:1			
	#1:0			$\$*$:2
	#1:0			
		#1:0		
		$\mathcal{S}$:3		
		$\mathcal{K}$:1	#1:0	
		FACT:0	$\mathcal{S}$:3	
		$\mathcal{S}$:2	$\mathcal{K}$:1	
		$\$-$:0	FACT:0	#1:0
		$\mathcal{K}$:1	$\mathcal{S}$:2	$\mathcal{S}$:3
		#1:0	$\$-$:0	$\mathcal{K}$:1
		#1:0	$\mathcal{K}$:1	FACT:0
			#1:0	$\mathcal{S}$:2
			#1:0	$\$-$:0
				$\mathcal{K}$:1
				#1:0
				#1:0

```
$*:2
#2:0
$If:3     $*:2
$≤:2      #2:0
#1:0      $If:3     $*:2
K:2       $≤:2      #2:0
#1:0      #1:0      $If:3     $*:2
#1:0                $≤:2      #2:0
$Pro:1    #1:0      #1:0      $If:3     $*:2
#1:0                          $≤:2      #2:0
$Pro:1    $Pro:1    #1:0      #1:0      $If:3
$*:2      #1:0                          $≤:2      $*:2
#1:0      $Pro:1    $Pro:1    #1:0      #1:0      #2:0
S:3       $*:2      #1:0                          $If:3
K:1       #1:0      $Pro:1    $Pro:1    #1:0      $≤:2
FACT:0    S:3       $*:2      #1:0                #1:0
S:2       K:1       #1:0      $Pro:1    $Pro:1
$−:0      FACT:0    S:3       $*:2      #1:0      #1:0
K:1       S:2       K:1       #1:0      $Pro:1
#1:0      $−:0      FACT:0    S:3       $*:2      $Pro:1
#1:0      K:1       S:2       K:1       #1:0      #1:0
          #1:0      $−:0      FACT:0    S:3       $Pro:1
          #1:0      K:1       S:2       K:1       $*:2
                    #1:0      $−:0      FACT:0    #1:0
                    #1:0      K:1       S:2       S:3
                              #1:0      $−:0      K:1
                              #1:0      K:1       FACT:0
                                        #1:0      S:2
                                        #1:0      $−:0
                                                  K:1
                                                  #1:0
                                                  #1:0
```

```
$*:2
#2:0
$If:3     $*:2
$≤:2      #2:0
#1:0      $If:3     $*:2
#1:0      $≤:2      #2:0
$Pro:1    #1:0      $If:3     $*:2
#1:0      #1:0      $≤:2      #2:0
$Pro:1    $Pro:1    #1:0      $If:3     $*:2
$*:2      #1:0      #1:0      $≤:2      #2:0
```

#1:0	$\$\mathit{Pro}:1$	$\$\mathit{Pro}:1$	#1:0	$\$\mathit{If}:3$	
$\mathcal{S}:3$	$\$*:2$	#1:0	#1:0	$\$\leq:2$	$\$*:2$
$\mathcal{K}:1$	#1:0	$\$\mathit{Pro}:1$	$\$\mathit{Pro}:1$	#1:0	#2:0
FACT:0	$\mathcal{S}:3$	$\$*:2$	#1:0	#1:0	$\$\mathit{If}:3$
$\mathcal{S}:2$	$\mathcal{K}:1$	#1:0	$\$\mathit{Pro}:1$	$\$\mathit{Pro}:1$	$T:0$
$\$-:0$	FACT:0	$\mathcal{S}:3$	$\$*:2$	#1:0	
$\mathcal{K}:1$	$\mathcal{S}:2$	$\mathcal{K}:1$	#1:0	$\$\mathit{Pro}:1$	
#1:0	$\$-:0$	FACT:0	$\mathcal{S}:3$	$\$*:2$	$\$\mathit{Pro}:1$
#1:0	$\mathcal{K}:1$	$\mathcal{S}:2$	$\mathcal{K}:1$	#1:0	#1:0
	#1:0	$\$-:0$	FACT:0	$\mathcal{S}:3$	$\$\mathit{Pro}:1$
	#1:0	$\mathcal{K}:1$	$\mathcal{S}:2$	$\mathcal{K}:1$	$\$*:2$
		#1:0	$\$-:0$	FACT:0	#1:0
		#1:0	$\mathcal{K}:1$	$\mathcal{S}:2$	$\mathcal{S}:3$
			#1:0	$\$-:0$	$\mathcal{K}:1$
			#1:0	$\mathcal{K}:1$	FACT:0
				#1:0	$\mathcal{S}:2$
				#1:0	$\$-:0$
					$\mathcal{K}:1$
					#1:0
					#1:0

$\$*:2$					
#2:0					
$\$\mathit{If}:3$	$\$*:2$				
$T:0$	#2:0				
$\$\mathit{Pro}:1$		$\$*:2$			
#1:0		#2:0			
$\$\mathit{Pro}:1$			$\$*:2$		
$\$*:2$	#1:0		#2:0		
#1:0				$\$*:2$	
$\mathcal{S}:3$		#1:0		#2:0	
$\mathcal{K}:1$					
FACT:0			#1:0		$\$*:2$
$\mathcal{S}:2$					#2:0
$\$-:0$				#1:0	
$\mathcal{K}:1$					
#1:0					
#1:0					#1:0



It can now be seen that there is some failure to fire otherwise ready combinators due to the injection of *$NULL* between arguments, causing argument type checks to fail. This effect also interacts adversely with the provision of only a single ALU to cause whole stream cycles where only one combinator is fired (in the ALU), suggesting that the processors be redesigned to wait for all typed arguments to be non-null. Exposure of this type of previously overlooked problem is the fundamental reason for functional simulation.

Relative performance with compiled models

It is undeniable that the overhead imposed on a simulator by an embedded Lisp interpreter is considerable. Table 4.14 shows the performance of a Stream-Group as successive stages of it are rewritten in C++.

C++ Modules	T(s)
None	3788
Stream-N-Buffer	2060
and Stream-Sink	1765
and Combinator-Processor	691
and ALU-Processor	633
and 90% of Stream-Circle	531

Table 4.14: Functional mode lisp v C++ performance

In all cases the system under test was an eight processor Stream-Group, simulating `(fact 8)`. Turnaround includes the build time in this instance.

Miscellaneous quasi-Lisp simulations

Table 4.15 shows some performance parameters extracted from simulations of an eight processor Stream-Group, using various simple source quasi-Lisp inputs. All parameters are collected by the Stream-Circle module, which is ideally placed to monitor the system performance.

The parameters are—

Cyc: The number of stream cycles needed to fully reduce the input to the correct answer.

Clo: The number of stream clock ticks from start to finish.

Input Lisp	Cyc	Clo	Para
`(reduce - '(3 1) 0)`	15	1783	2.889
`(sum '(1 1))`	16	1820	2.844
`(map numberp '(t 1 (0 1)))`	10	1644	3.151
`(member 1 '(t 1 (0 1)))`	21	6716	3.184
`(append '(1) '(2))`	8	617	2.316
`(reverse '(2 1))`	17	1414	2.143
`(fact 2)`	10	386	1.085
`(gcd 2 4)`	8	406	2.811

Table 4.15: Quasi-Lisp performance

Para: The average parallelism throughout the simulation— this is measured by accumulating the number of processors doing useful work (that is, not in the Wait microinstruction, or handling *$Pro*) each clock tick, and dividing the total by the number of ticks.

Interestingly, `(fact 2)` appears to be atypically sequential. These results add weight to the conclusions of section 4.4.16 that the performance of the factorial function could be significantly improved.

For completeness, here are the definitions of the functions used above. Note that `NULL` compiles to *$Not*, and that `EQL` is a macro that combines *$Not*, *$=*, and *$Eq*.

```
(defun append (x y)
  ; append two lists
  (if (null x) y
    (cons (car x) (append (cdr x) y))))

(defun member (x l)
```

```
  ; is x a member of l?
  (if (null l) nil
    (if (eql x (car l)) t
      (member x (cdr l)))))

(defun reverse (x)
  ; reverse a list
  (if (null x) nil
    (append (reverse (cdr x)) (cons (car x) nil))))

(defun map (f x)
  ; map the function unary f over the list x
  (if (null x) nil
    (cons (apply f (car x)) (map f (cdr x)))))

(defun reduce (f x i)
  ; apply the binary function f to the list x such that
  ; the result is: (f x1 (f x2 ... (f xn i)))
  (if (null x) i
    (apply f (car x) (reduce f (cdr x) i))))

(defun sum (x)
  ; sum the elements of list x
  (reduce + x 0))

(defun gcd (a b)
  (if (= 0 b) a (gcd b (rem a b))))

(defun fact (x)
  (if (<= x 1) 1 (* x (fact (- x 1)))))
```

Chapter 5

Conclusions

There are many difficult problems in VLSI systems simulation. No single system can be expected to solve all equally effectively. The simulator described in this thesis is in many ways the dual of SPICE in that it abandons the focused rigour of precise analog device modelling in favour of a wider range of higher abstractions.

In the battle with complexity, abstraction is the last weapon left after all other techniques are overwhelmed. Provision of a variation in level of abstraction at arbitrary points in the module hierarchy is the characteristic feature of mixed-mode simulators. The improved turnaround available from the most aggressive abstractions is considerable, but this benefit carries with it a correspondingly heavy burden of responsibility for modelling accuracy. Full verification of the equivalence of pairs of models is as yet an open problem.

The simulator, Loge, has been developed without reference to any specific methodological policy. Flexibility is a key attribute, with the intention of providing a rich, non-dictatorial modelling capability. This goal leads to the choice of interpreted Lisp as the basis of the functional mode hard-

ware description language, and to the provision for highly generic modules—generic not only in simple numeric parameters but also in shape, composition and behaviour of submodules, as illustrated by example throughout the later chapters.

While functional models are highly flexible in construction, within every module instance tree all connections are strictly required to follow port typing rules. Similarly, while the Lisp level definitions of a system may roam throughout memory, actual instances are strictly placed in a compact sequential block. These issues illustrate how the demands of high performance simulation oppose those of modelling freedom. Nevertheless, Loge succeeds in providing a firm framework for lower level modes simultaneously with a functional mode with which all manner of synchronous, asynchronous or self-timed systems may be modelled. Additionally, the ability to insert arbitrary instrumentation into a functional model should not be underrated.

Pure digital simulation is easily corrupted by analog phenomena, which can only be absolutely avoided by restricting the range of systems capable of being simulated. Loge operates internally on analog quantities, while turning a primarily digital face to the world. Continuous calculation at regular time intervals is avoided in favour of purely event-driven algorithms.

Given internal analog modelling, hybrid device models are a natural route to swift simulation turnaround without too great a loss of accuracy. While sometimes prone to unnecessary oscillation, explicit, sampled and evaluated device submodes each have useful roles to play in simulation of the wide variety of design idioms and technologies encountered at the device-level. Particularly interesting is the good turnaround achievable with sampled devices in comparison with the more aggressively simplified explicit devices—it seems that the six region explicit devices are sometimes represent too great an approximation. Another key feature of the hybrid modes is the ability

to control the turnaround/accuracy tradeoff— responsive systems are often preferred by experienced users.

Perhaps the most pleasing feature of Loge is that the interfacing between all modes is clean, efficient, transparent to the user, and free from arbitrary restrictions. This goal was the first design criterion chosen, and is therefore a basic test of the success of Loge.

Bibliography

[Abelson+85] Abelson, A., Sussman, G., and Sussman, J., 1985. *Structure and Interpretation of Computer Programs.* McGraw-Hill, 1985.

[Ackland+81] Ackland, B. and Weste, N., 1981. Functional Verification in an Interactive Symbolic IC Design Environment. In *Proceedings of the 2nd Caltech Conference on VLSI*, pages 285–298. California Institute of Technology, 1981.

[Ackland+85] Ackland, B., Ahuja, S., Lindstrom, T., and Romero, D., 1985. CEMU — A Concurrent Timing Simulator. In *IEEE International Conference on Computer-Aided Design ICCAD-85* [IEE85], pages 122–124.

[Ackland+86] Ackland, B., Ahuja, S., DeBenedictis, E., London, T., Lucco, S., and Romero, D., 1986. MOS Timing Simulation on a Message Based Multiprocessor. In *IEEE International Conference on Computer-Aided Design ICCAD-86* [IEE86], pages 446–450.

[Ackland+89] Ackland, B. and Clark, R., 1989. Event-EMU: An Event Driven Timing Simulator for MOS VLSI Circuits. In *IEEE International Conference on Computer-Aided Design ICCAD-89* [IEE89], pages 80–83.

[ACM79] ACM/IEEE. *16th Design Automation Conference*, 1979.

[ACM80] ACM/IEEE. *17th Design Automation Conference*, 1980.

[ACM81] ACM/IEEE. *18th Design Automation Conference*, 1981.

[ACM82] ACM/IEEE. *19th Design Automation Conference*, 1982.

[ACM83] ACM/IEEE. *20th Design Automation Conference*, 1983.

[ACM84] ACM/IEEE. *21st Design Automation Conference*, 1984.

[ACM85] ACM/IEEE. *22nd Design Automation Conference*, 1985.

[ACM86] ACM/IEEE. *23rd Design Automation Conference*, 1986.

[ACM87] ACM/IEEE. *24th Design Automation Conference*, 1987.

[ACM88] ACM/IEEE. *25th Design Automation Conference*, 1988.

[ACM89] ACM/IEEE. *26th Design Automation Conference*, 1989.

[ACM90] ACM/IEEE. *27th Design Automation Conference*, 1990.

[Adler86] Adler, D., 1986. SIMMOS: A Multiple Delay Switch-Level Simulator. In *Proceedings of the 23rd Design Automation Conference* [ACM86], pages 159–163.

[Adler88] Adler, D., 1988. A Dynamically-Directed Switch Model for MOS Logic Simulation. In *Proceedings of the 25th Design Automation Conference* [ACM88], pages 506–511.

[Agrawal+80] Agrawal, V., Bose, A., Kozak, P., Nham, H., and Pacas-Skewes, E., 1980. A Mixed-Mode Simulator. In *Proceedings of the 17th Design Automation Conference* [ACM80], pages 618–625.

[Agrawal82] Agrawal, V., 1982. Synchronous Path Analysis in MOS Circuit Simulators. In *Proceedings of the 19th Design Automation Conference* [ACM82], pages 629–635.

[Agrawal+88] Agrawal, P., Robinson, S., and Szymanski, T., 1988. Automatic Modeling of Switch-level Networks Using Partial Orders. In *IEEE International Conference on Computer-Aided Design ICCAD-88* [IEE88], pages 350–353.

[Almeida+84] Almeida, C. and Lanca, M., 1984. An Eight-Value Modelling Technique for Logic Simulation of Transmission Gates and Buses. *IEE Electronic Design Automation.*, pages 23–27, 1984.

[Appel88] Appel, A., 1988. Simulating Digital Circuits with One Bit Per Wire. *IEEE Transactions on Computer-Aided Design*, pages 987–993, Sep 1988.

[Armstrong+81] Armstrong, J. and Devlin, D., 1981. GSP: A Logic Simulator for LSI. In *Proceedings of the 18th Design Automation Conference* [ACM81], pages 518–524.

[Armstrong84] Armstrong, J., 1984. Chip Level Modelling of LSI Devices. *IEEE Transactions on Computer-Aided Design*, pages 288–297, Oct 1984.

[Arnout+78] Arnout, G. and DeMan, H., 1978. The Use of Threshold Functions and Boolean-Controlled Network Elements for Macromodelling of LSI Circuits. *IEEE Journal of Solid-State Circuits*, pages 326–332, Jun 1978.

[Aylor+86] Aylor, J., Waxman, R., and Scarratt, C., 1986. VHDL— Feature Description and Analysis. *IEEE Design and Test*, pages 17–27, Apr 1986.

[Backus78] Backus, J., 1978. Can Programming be Liberated From the Von Neumann Style? A Functional Style and its Algebra of Programs. *Communications of the ACM*, pages 613–641, Aug 1978.

[Barros+83] Barros, J. and Johnson, B., 1983. Equivalence of the Arbiter, the Synchroniser, the Latch and the Inertial Delay. *IEEE Transactions on Computer-Aided Design*, pages 603–614, Jul 1983.

[Barzilai+88] Barzilai, Z., Beece, D., Huisman, L., Iyengar, V., and Silberman, G., 1988. SLS— A Fast Switch-Level Simulator. *IEEE Transactions on Computer-Aided Design*, pages 839–849, Aug 1988.

[Bauer+88] Bauer, R., Fang, J., Ng, A., and Brayton, R., 1988. XPSim: A MOS VLSI Simulator. In *IEEE International Conference on Computer-Aided Design ICCAD-88* [IEE88], pages 66–69.

[Beatty+88] Beatty, D. and Bryant, R., 1988. Fast Incremental Circuit Analysis Using Extended Hierarchy. In *Proceedings of the 25th Design Automation Conference* [ACM88], pages 495–500.

[Beckett86] Beckett, W., 1986. MOS Circuit Models in Network C. In *Proceedings of the 23rd Design Automation Conference* [ACM86], pages 171–178.

[Bening79] Bening, L., 1979. Developments in Computer Simulation of Gate Level Physical Logic. In *Proceedings of the 16th Design Automation Conference* [ACM79], pages 561–567.

[Bening+82] Bening, L., Lane, T., and Smith, J., 1982. Developments in Logic Network Path Delay Analysis. In *Proceedings of the 19th Design Automation Conference* [ACM82], pages 605–615.

[Blaauw+89] Blaauw, D., Saab, D., Mueller-Thuns, R., Abraham, J., and Rahmeh, J., 1989. Automatic Generation of Bahavioral Models From Switch-Level Descriptions. In *Proceedings of the 26th Design Automation Conference* [ACM89], pages 179–184.

[Booch90] Booch, G., 1990. *Object Oriented Design with Applications.* Benjamin/Cummings, 1990.

[Borrione+83] Borrione, D., Humbert, M., and LeFaou, C., 1983. Hierarchical Mixed Mode Simulation Mechanisms in the Cascade Project. In *Proceedings of the IFIP TC 10/WG 10.5 International Conference on Very Large Scale Integration* [Int83], pages 119–129.

[Breuer72] Breuer, M., 1972. A Note on Three-Valued Logic Simulation. *IEEE Transactions on Computers*, pages 399–402, Apr 1972.

[Breuer+81] Breuer, M. and Parker, A., 1981. Digital System Simulation: Current Status and Future Trends. In *Proceedings of the 18th Design Automation Conference* [ACM81], pages 269–285.

[Brocco+88] Brocco, L., McCormick, S., and Allen, J., 1988. Macromodeling CMOS Circuits for Timing Simulation. *IEEE Transactions on Computer-Aided Design*, pages 1237–1249, Dec 1988.

[Brown+83] Brown, H., Tong, C., and Foyster, G., 1983. Palladio: An Exploratory Environment for Circuit Design. *IEEE Computer*, pages 41–56, Dec 1983.

[Bryant81a] Bryant, R., 1981. MOSSIM: A Switch Level Simulator for MOS LSI. In *Proceedings of the 18th Design Automation Conference* [ACM81], pages 786–790.

[Bryant81b] Bryant, R., 1981. A Switch Level Model of MOS Logic Circuits. In *Proceedings of the IFIP TC 10/WG 10.5 International Conference on Very Large Scale Integration*, pages 329–340. International Federation for Information Processing, 1981.

[Bryant83] Bryant, R., 1983. Race Detection in MOS Circuits by Ternary Simulation. In *Proceedings of the IFIP TC 10/WG 10.5 International Conference on Very Large Scale Integration* [Int83], pages 85–95.

[Bryant84] Bryant, R., 1984. A Switch-Level Model and Simulator for MOS Digital Systems. *IEEE Transactions on Computers*, pages 160–177, Feb 1984.

[Bryant87a] Bryant, R., 1987. A Survey of Switch Level Algorithms. *IEEE Design and Test*, pages 26–40, Aug 1987.

[Bryant+87b] Bryant, R., Beatty, D., Brace, K., Cho, K., and Sheffler, T., 1987. COSMOS: A Compiled Simulator for MOS Circuits. In *Proceedings of the 24th Design Automation Conference* [ACM87], pages 9–16.

[Bryant88] Bryant, R., 1988. Data Parallel Switch-Level Simulation. In *IEEE International Conference on Computer-Aided Design ICCAD-88* [IEE88], pages 354–357.

[Bryant90] Bryant, R., 1990. Symbolic Simulation— Techniques and Applications. In *Proceedings of the 27th Design Automation Conference* [ACM90], pages 517–521.

[Brzozowski+79] Brzozowski, J. and Yoeli, M., 1979. On a Ternary Model of Gate Networks. *IEEE Transactions on Computers*, pages 178–184, Mar 1979.

[Cal83] California Institute of Technology. *3rd Caltech Conference on VLSI*, 1983.

[Canright86] Canright, R., 1986. Simulating and Controlling the Effects of Transmission Line Impedance Mismatches. In *Proceedings of the 23rd Design Automation Conference* [ACM86], pages 778–785.

[Cerny+85] Cerny, E. and Gecsei, J., 1985. Simulation of MOS Circuits by Decision Diagrams. *IEEE Transactions on Computer-Aided Design*, pages 685–693, Oct 1985.

[Chadla+88] Chadla, R., Visweswariah, C., and Chen, C., 1988. M^3 - A Multi-Level Mixed-Mode Mixed D/A Simulator. In *IEEE International Conference on Computer-Aided Design ICCAD-88* [IEE88], pages 258–261.

[Chamberlain+86] Chamberlain, R. and Franklin, M., 1986. Collecting Data About Logic Simulation. *IEEE Transactions on Computer-Aided Design*, pages 405–411, Jul 1986.

[Chandra+89] Chandra, S. and Patel, J., 1989. Accurate Logic Simulation in the Presence of Unknowns. In *IEEE International Conference on Computer-Aided Design ICCAD-89* [IEE89], pages 34–37.

[Chang+87] Chang, H. and Abraham, J., 1987. The Complexity of Accurate Logic Simulation. In *IEEE International Conference on Computer-Aided Design ICCAD-87*. IEEE, 1987.

[Chawla+75] Chawla, B., Gummel, H., and Kozak, P., 1975. MOTIS- an MOS Timing Simulator. *IEEE Transactions on Circuits and Systems*, pages 901–910, Dec 1975.

[Chen+83] Chen, M. and Mead, C., 1983. A Hierarchical Simulator Based on Formal Semantics. In *Proceedings of the 3rd Caltech Conference on VLSI* [Cal83], pages 207–223.

[Chen+84] Chen, C., Lo, C., Nham, H., and Subramaniam, P., 1984. The Second Generation MOTIS Mixed Mode Simulator. In *Proceedings of the 21st Design Automation Conference* [ACM84], pages 10–17.

[Cherry88] Cherry, J., 1988. Pearl: A CMOS Timing Analyzer. In *Proceedings of the 25th Design Automation Conference* [ACM88], pages 148–153.

[Choi+88] Choi, K., Hwang, S., and Blank, T., 1988. Incremental-in-Time Algorithm for Digital Simulation. In *Proceedings of the 25th Design Automation Conference* [ACM88], pages 501–505.

[Chu+87] Chu, C. and Horowitz, M., 1987. Charge-Sharing Models for Switch-Level Simulation. *IEEE Transactions on Computer-Aided Design*, pages 1053–1060, Nov 1987.

[Clarke82] Clarke, R., 1982. AUSMPC 5/82 Designer Documentation. Technical report, CSIRO Division of Computing Research VLSI Program, Aug 1982.

[Cohn+89] Cohn, R., Gross, T., Lam, M., and Tseng, P., 1989. Architecture and Compiler Tradeoffs for a Long Instruction Word Microprocessor. In *Proceedings of the Third SIGARCH/SIGPLAN Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 1–14, 1989.

[Cox+88] Cox, P., Burch, R., and Yang, P., 1988. A Dormant Subcircuit Model for Maximizing Iteration Latency. In *IEEE International Conference on Computer-Aided Design ICCAD-88* [IEE88], pages 438–441.

[da Cruz87] da Cruz, F., 1987. *Kermit, A File Transfer Protocol.* Digital Press, 1987.

[d'Abreu+84] d'Abreu, M., Cheong, K., and Flanagan, C., 1984. Oracle - A Simulator for Bipolar and MOS IC Design. In *Proceedings of the 21st Design Automation Conference* [ACM84], pages 343–349.

[d'Abreu85] d'Abreu, M., 1985. Gate-Level Simulation. *IEEE Design and Test*, pages 63–71, Dec 1985.

[Dagenais+86] Dagenais, M. and Rumin, N., 1986. Circuit-Level Timing Analysis and Design Verification of High-Performance MOS Computer Circuits. In *IEEE International Conference on Computer-Aided Design ICCAD-86* [IEE86], pages 356–359.

[Dahl+70] Dahl, O.-J., Myrhaug, B., and Nygaard, K., 1970. *SIMULA Common Base Language.* Norwegian Computing Center, 1970.

[de Geus84] de Geus, A., 1984. SPECS: Simulation Program for Electronic Circuits and Systems. *IEEE International Symposium on Circuits and Systems*, pages 534–537, May 1984.

[De Micheli+83] De Micheli, G., Newton, A., and Sangiovanni-Vincentelli, A., 1983. Symmetric Displacement Algorithms for the Timing Analysis of Large Scale Circuits. *IEEE Transactions on Computer-Aided Design*, pages 167–179, Jul 1983.

[Des Marias+82] Des Marias, P., Shew, E., and Wilcox, P., 1982. A Functional Level Modelling Language for Digital Simulation. In *Proceedings of the 19th Design Automation Conference* [ACM82], pages 315–320.

[Deutsch+84] Deutsch, J. and Newton, A., 1984. A Multiprocessor Implementation of Relaxation Based Electrical Circuit Simulation. In *Proceedings of the 21st Design Automation Conference* [ACM84], pages 350–357.

[Dewilde+85] Dewilde, P., van Genderen, A., and de Graaf, A., 1985. Switch Level Timing Simulation. In *IEEE International Conference on Computer-Aided Design ICCAD-85* [IEE85], pages 182–184.

[Dickinson88] Dickinson, A., 1988. *Complexity Management and Modelling of VLSI Systems.* PhD thesis, University of Adelaide, 1988.

[Dickinson89] Dickinson, A., 1989. Cytoarchitectures: Machines in Genes. *AT&T Bell Labs Technical Memorandum, No.11356-890512-13*, May 1989.

[Doshi+84] Doshi, M., Sullivan, R., and Schuler, D., 1984. Themis Logic Simulator - A Mixed Mode Multilevel, Hierarchical Interactive Digital Circuit Simulator. In *Proceedings of the 21st Design Automation Conference* [ACM84], pages 24–31.

[Dumlugöl+83] Dumlugöl, D., DeMan, H., Stevens, P., and Schrooten, G., 1983. Local Relaxation Algorithms for Event Driven Simulation of MOS Networks Including Assignable Delay Modelling. *IEEE Transactions on Computer-Aided Design*, pages 193–202, Jul 1983.

[Dumlugöl+87] Dumlugöl, D., Odent, P., Cockx, J., and De Man, H., 1987. Switch-Electrical Segmented Waveform Relaxation for Digital MOS VLSI and Its Acceleration on Parallel Computers. *IEEE Transactions on Computer-Aided Design*, pages 992–1005, Nov 1987.

[Elder+84] Elder, W., Zenewicz, P., and Alvarodiaz, R., 1984. An Interactive System for VLSI Chip Physical Design. *IBM Journal of Research and Development*, pages 524–535, Sep 1984.

[Engl+82] Engl, W., Laur, R., and Dirks, H., 1982. MEDUSA — A Simulator for Modular Circuits. *IEEE Transactions on Computer-Aided Design*, pages 85–93, Apr 1982.

[Engl+83] Engl, W., Dirks, H., and Meinerzhagen, B., 1983. Device Modelling. *Proceedings of the IEEE*, pages 10–33, Jan 1983.

[Erdman+89] Erdman, D. and Rose, D., 1989. A Newton Waveform Relaxation Algorithm for Circuit Simulation. In *IEEE International Conference on Computer-Aided Design ICCAD-89* [IEE89], pages 404–407.

[Eshraghian+85] Eshraghian, K., Bryant, R., Dickinson, A., Fensom, D., Franzon, P., Pope, M., Rockliff, J., and Zyner, G., 1985. The Transform and Filter Brick: A New Architecture for Signal Processing. In *Proceedings of the IFIP TC 10/WG 10.5 International Conference on Very Large Scale Integration*, pages 135–144. International Federation for Information Processing, 1985.

[Eshraghian88] Eshraghian, K., 1988. Fundamentals of Ultra High Speed Systems: Gallium Arsenide VLSI Technology, 1988.

[Etiemble+84] Etiemble, D., Adeline, V., Duyet, N., and Ballegaer, J., 1984. Micro-Computer Oriented Alogorithms for Delay Estimation in MOS Gates. In *Proceedings of*

the 21st Design Automation Conference [ACM84], pages 358–364.

[Fan+77] Fan, S., Hsueh, M., Newton, A., and Pederson, D., 1977. MOTIS-C A New Circuit Simulator for MOS LSI Circuits. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 700–703, 1977.

[Flake+83] Flake, P., Moorby, P., and Musgrave, G., 1983. An Algebra for Logic Signal Strength. In *Proceedings of the 20th Design Automation Conference* [ACM83], pages 615–618.

[Foyster86] Foyster, G., 1986. VLSI Functional Verification from Specification to Test. In *Proceedings of the Australian and Pacific Area Region Microelectronics Conference* [IRE86], pages 287–293.

[Frank86] Frank, E., 1986. Exploring Parallelism in a Switch-Level Simulation Machine. In *Proceedings of the 23rd Design Automation Conference* [ACM86], pages 20–26.

[Goyal+87] Goyal, R. and Scheinberg, N., May 1987. GaAs MESFET Model for Precision Analog IC Design. *VLSI Systems Design*, pages 52–55, May 1987. GaAs.

[Hansen88] Hansen, C., 1988. Hardware Logic Simulation by Compilation. In *Proceedings of the 25th Design Automation Conference* [ACM88], pages 712–715.

[Hayes82] Hayes, J., 1982. A Unified Switching Theory with Applications to VLSI Design. *Proceedings of the IEEE*, pages 1140–1151, Oct 1982.

[Hayes86a] Hayes, J., 1986. Digital Simulation with Multiple Logic Values. *IEEE Transactions on Computer-Aided Design*, pages 274–283, Apr 1986.

[Hayes86b] Hayes, J., 1986. Pseudo-Boolean Logic Circuits. *IEEE Transactions on Computers*, pages 602–612, Jul 1986.

[Hennion+85] Hennion, B. and Senn, P., 1985. A New Algorithm for Third Generation Circuit Simulators: The One-Step Relaxation Method. In *Proceedings of the 22nd Design Automation Conference* [ACM85], pages 137–143.

[Heydemann+88] Heydemann, M. and Dure, D., 1988. The Logic Automaton Approach to Accurate and Efficient Gate and Functional Level Simulation. In *IEEE International Conference on Computer-Aided Design ICCAD-88* [IEE88], pages 250–253.

[Hill+79] Hill, D. and van Cleemput, W., 1979. Sable - A Tool for Generating Structured Multilevel Simulations. In *Proceedings of the 16th Design Automation Conference* [ACM79], pages 272–279.

[Hirakawa+82] Hirakawa, K., Shiraki, N., and Muraoka, M., 1982. Logic Simulation for LSI. In *Proceedings of the 19th Design Automation Conference* [ACM82], pages 755–761.

[Hirchhorn+81] Hirchhorn, S., Hommel, M., and Bures, C., 1981. Functional Level Simulation in FANSIM3 — Algorithms, Data Structures and Results. In *Proceedings of the 18th Design Automation Conference* [ACM81], pages 248–255.

[Hitchcock82] Hitchcock, R., 1982. Timing Verification and Timing Analysis Program. In *Proceedings of the 19th Design Automation Conference* [ACM82], pages 594–604.

[Hodgson84] Hodgson, S., 1984. A Multilevel Mixed Mode Simulator for Hierarchical Design Verification. *IEE Electronic Design Automation*, pages 107–110, 1984.

[Holt+81] Holt, D. and Hutchings, D., 1981. A MOS/LSI Oriented Logic Simulator. In *Proceedings of the 18th Design Automation Conference* [ACM81], pages 280–287.

[Hwang+86] Hwang, S., Kim, Y., and Newton, A., 1986. An Accurate Delay Modelling Technique for Switch-Level Timing Verification. In *Proceedings of the 23rd Design Automation Conference* [ACM86], pages 227–233.

[IEE83] IEEE. *International Conference on Computer Design*, 1983.

[IEE85] IEEE. *International Conference on Computer-Aided Design*, 1985.

[IEE86] IEEE. *International Conference on Computer-Aided Design*, 1986.

[IEE88] IEEE. *International Conference on Computer-Aided Design*, 1988.

[IEE89] IEEE. *International Conference on Computer-Aided Design*, 1989.

[Ins88] Institute of Electrical and Electronics Engineers, Inc. *IEEE Standard VHDL Language Reference Manual*, 1988.

[Int83] International Federation for Information Processing. *VLSI 83*, 1983.

[Int87] Integrated Silicon Design, Pty Ltd. *Phase 1 VLSI Design Suite User Manual*, 1987.

[IRE86] IREE and I.E.Aust. *Microelectronics 86*, 1986.

[Jea+79] Jea, Y. and Szygenda, S., 1979. Mappings and Algorithms for Gate Modelling in a Digital Simulation Environment. *IEEE Transactions on Circuits and Systems*, pages 304–315, May 1979.

[Johnson+78] Johnson, S. and Seeley, D., 1978?. *A Tour Through the Portable C Compiler.* Department of Computer Science, University of Utah, 1978?

[Jouppi83a] Jouppi, N., 1983. Timing Analysis for nMOS VLSI. In *Proceedings of the 20th Design Automation Conference* [ACM83], pages 411–418.

[Jouppi83b] Jouppi, N., 1983. TV: An nMOS Timing Analyser. In *Proceedings of the 3rd Caltech Conference on VLSI* [Cal83], pages 71–85.

[Jouppi87] Jouppi, N., 1987. Derivation of Signal Flow Direction in MOS VLSI. *IEEE Transactions on Computer-Aided Design*, pages 480–490, May 1987.

[Kao+88] Kao, R., Alverson, B., Horowitz, M., and Stark, D., 1988. Bisim: A Simulator for Custom ECL Circuits. In *IEEE International Conference on Computer-Aided Design ICCAD-88* [IEE88], pages 62–65.

[Katzenelson+86] Katzenelson, J. and Weitz, E., 1986. VLSI Simulation and Data Abstractions. *IEEE Transactions on Computer-Aided Design*, pages 371–378, Jul 1986.

[Kim+89] Kim, Y., Hwang, S., and Newton, A., 1989. Electrical-Logic Simulation and Its Applications. *IEEE Transactions on Computer-Aided Design*, pages 8–22, Jan 1989.

[Kravitz+89] Kravitz, S., Bryant, R., and Rutenbar, R., 1989. Massively Parallel Switch-Level Simulation: A Feasability Study. In *Proceedings of the 26th Design Automation Conference* [ACM89], pages 91–97.

[Lathrop+85] Lathrop, R. and Kirk, R., 1985. An Extensible Object-Oriented Mixed-Mode Functional Simulation System. In *Proceedings of the 22nd Design Automation Conference* [ACM85], pages 630–636.

[LeBlanc+85] LeBlanc, T., Cockerill, T., Ledak, P., Hsieh, H., and Ruehli, A., 1985. Recent Advances in Waveform Relaxation Based Circuit Analysis. In *IEEE International Conference on Computer-Aided Design ICCAD-85* [IEE85], pages 594–596.

[Leinwand81] Leinwand, S., 1981. Process Oriented Logic Simulation. In *Proceedings of the 18th Design Automation Conference* [ACM81], pages 511–517.

[Lelarasmee+82a] Lelarasmee, E., Ruehli, A., and Sangiovanni-Vincentelli, A., 1982. A Waveform Relaxation Method for Time Domain Analysis of Large Scale Integrated Circuits. *IEEE Transactions on Circuits and Systems*, pages 131–145, Jul 1982.

[Lelarasmee+82b] Lelarasmee, E. and Sangiovanni-Vincentelli, A., 1982. Relax: A New Circuit Simulator for Large Scale MOS ICs. In *Proceedings of the 19th Design Automation Conference* [ACM82], pages 682–687.

[Lewis89] Lewis, D., 1989. Hierarchical Compiled Event-Driven Logic Simulation. In *IEEE International Conference on Computer-Aided Design ICCAD-89* [IEE89], pages 498–501.

[Lieberherr83] Lieberherr, K., 1983. Multilevel Simulation for VLSI. In *Proceedings of the International Conference on Computer Design* [IEE83], pages 441–444.

[Lin+84] Lin, T. and Mead, C., 1984. Signal Delay in General RC Networks. *IEEE Transactions on Computer-Aided Design*, pages 331–349, Oct 1984.

[Lin+86] Lin, T. and Mead, C., 1986. A Hierarchical Timing Simulation Model. *IEEE Transactions on Computer-Aided Design*, pages 188–197, Jan 1986.

[Lipsett+86] Lipsett, R., Marschner, E., and Shahdad, M., 1986. VHDL — The Language. *IEEE Design and Test*, pages 28–41, Apr 1986.

[Lo+83] Lo, C., Nham, H., and Bose, A., 1983. A Data Structure for MOS Circuits. In *Proceedings of the 20th Design Automation Conference* [ACM83], pages 619–624.

[Luckham+86] Luckham, D., Stanculescu, A., Huh, Y., and Ghosh, S., 1986. The Semantics of Timing Constructs in Hardware Description Languages. In *IEEE International Conference on Computer-Aided Design ICCAD-86* [IEE86], pages 10–14.

[Maissel+82] Maissel, L. and Ostapko, D., 1982. Interactive Design Language: A Unified Approach to Hardware Simulation, Synthesis and Documentation. In *Proceedings of the 19th Design Automation Conference* [ACM82], pages 193–201.

[Martin+88] Martin, D. and Rumin, N., 1988. Delay Computation in Switch-Level Models of Non-Treelike MOS Circuits. In *IEEE International Conference on Computer-Aided Design ICCAD-88* [IEE88], pages 358–361.

[Matson85] Matson, M., 1985. Macromodelling of Digital MOS VLSI Circuits. In *Proceedings of the 22nd Design Automation Conference* [ACM85], pages 144–151.

[Mayaram+88] Mayaram, K. and Pederson, D., 1988. CODECS: A Mixed-level Device and Circuit Simulator. In *IEEE International Conference on Computer-Aided Design ICCAD-88* [IEE88], pages 112–115.

[McDermott82] McDermott, R., 1982. Transmission Gate Modelling in an Existing 3-valued simulator. In *Proceedings of the 19th Design Automation Conference* [ACM82], pages 678–681.

[McFarland86] McFarland, M., 1986. Using Bottom Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioural Descriptions. In *Proceedings of*

the 23rd Design Automation Conference [ACM86], pages 474–480.

[McWilliams80] McWilliams, T., 1980. Verification of Timing Constraints on Large Digital Systems. In *Proceedings of the 17th Design Automation Conference* [ACM80], pages 139–147.

[Mead+80] Mead, C. and Conway, L., 1980. *Introduction to VLSI Systems.* Addison Wesley, 1980.

[Miyoshi+85] Miyoshi, M., Kazama, Y., Tada, O., Nagura, Y., and Amano, N., 1985. Speed Up Techniques of Logic Simulation. In *Proceedings of the 22nd Design Automation Conference* [ACM85], pages 812–815.

[Mokari-Bolhassan+85] Mokari-Bolhassan, M., Smart, D., and Trick, T., 1985. A New Robust Relaxation Technique for VLSI Circuit Simulation. In *IEEE International Conference on Computer-Aided Design ICCAD-85* [IEE85], pages 26–28.

[Murphy+85] Murphy, B., Kleckner, J., and Tam, K., 1985. STA: A Mixed-Mode Timing Analyser. In *IEEE International Conference on Computer-Aided Design ICCAD-85* [IEE85], pages 176–178.

[Murphy90] Murphy, E., 1990. Design Tools. *IEEE Spectrum*, pages 34–36, Feb 1990.

[Nagel75] Nagel, L., 1975. SPICE2: A Computer Program to Simulate Semiconductor Circuits. ERL Memo, University of California, Berkeley, May 1975.

[Nash+80] Nash, D., Russell, K., Silverman, P., and Thiel, M., 1980. Functional Level Simulation at Raytheon. In *Proceedings of the 17th Design Automation Conference* [ACM80], pages 634–641.

[Nash+86] Nash, J. and Saunders, L., 1986. VHDL Critique. *IEEE Design and Test*, pages 54–65, Apr 1986.

[Newton79] Newton, A., 1979. Techniques for the Simulation of Large-Scale Integrated Circuits. *IEEE Transactions on Circuits and Systems*, pages 741–749, Sep 1979.

[Newton81] Newton, A., 1981. Timing, Logic and Mixed-Mode Simulation for Large MOS Integrated Circuits. *NATO Advanced Study Institute on Computer Aided Design for VLSI Cirucits*, pages 175–239, 1981.

[Newton+84] Newton, A. and Sangiovanni-Vincentelli, A., 1984. Relaxation Based Electrical Simulation. *IEEE Transactions on Computer-Aided Design*, pages 308–331, Oct 1984.

[Ng+81] Ng, P., Glauert, W., and Kirk, R., 1981. A Timing Verification System Based on Extracted MOS/VLSI Circuit Parameters. In *Proceedings of the 18th Design Automation Conference* [ACM81], pages 288–292.

[Nguyen+89] Nguyen, T., Feldmann, P., Director, S., and Rohrer, R., 1989. SPECS Simulation Validation with Efficient Transient Sensitivity Computation. In *IEEE International Conference on Computer-Aided Design ICCAD-89* [IEE89], pages 252–255.

[Nham+80] Nham, H. and Bose, A., 1980. A Multiple Delay Simulator for MOS LSI Circuits. In *Proceedings of the 17th Design Automation Conference* [ACM80], pages 610–617.

[Nomura+82] Nomura, M., Sato, S., Takano, N., Aoyama, T., and Yamada, A., 1982. Timing Verification System Based on Delay Time Hierarchical Nature. In *Proceedings of the 19th Design Automation Conference* [ACM82], pages 622–628.

[Odryna+86] Odryna, P., Nazareth, K., and Christensen, C., 1986. A Workstation-Based Mixed Mode Circuit Simulator. In *Proceedings of the 23rd Design Automation Conference* [ACM86], pages 186–192.

[Okazaki+83] Okazaki, K., Moriya, T., and Yahara, T., 1983. A Multiple Media Delay Simulator for MOS LSI Circuits. In *Proceedings of the 20th Design Automation Conference* [ACM83], pages 279–285.

[Ousterhout83] Ousterhout, J., 1983. Crystal: A Timing Analyser for nMOS VLSI. In *Proceedings of the 3rd Caltech Conference on VLSI* [Cal83], pages 57–69.

[Ousterhout85] Ousterhout, J., 1985. A Switch-Level Timing Verifier for Digital MOS VLSI. *IEEE Transactions on Computer-Aided Design*, pages 336–349, Jul 1985.

[Overhauser+88] Overhauser, D. and Hajj, I., 1988. A Tabular Macromodelling Approach to Fast Timing Simulation Including Parasitics. In *IEEE International Conference on Computer-Aided Design ICCAD-88* [IEE88], pages 70–73.

[Overhauser+89] Overhauser, D., Hajj, I., and Hsu, Y., 1989. Automatic Mixed-Mode Timing Simulation. In *IEEE International Conference on Computer-Aided Design ICCAD-89* [IEE89], pages 84–87.

[Penfield+81] Penfield, P. and Rubinstein, J., 1981. Signal Delay in RC Tree Networds. In *Proceedings of the 18th Design Automation Conference* [ACM81], pages 613–617.

[Peyton Jones87] Peyton Jones, S., 1987. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[Pilotny+82] Pilotny, R. and Borrione, D., 1982. The CONLAN Project: Status and Future Plans. In *Proceedings of the 19th Design Automation Conference* [ACM82], pages 202–212.

[Rabbat+79] Rabbat, N., Sangiovanni-Vincentelli, A., and Hsieh, H., 1979. A Multilevel Newton Algorithm with Macromodelling and Latency for the Analysis of Large-Scale Nonlinear Circuits in the Time Domain. *IEEE Transactions on Circuits and Systems*, pages 733–741, Sep 1979.

[Raeth+81] Raeth, P., Acken, J., Lamont, G., and Borky, J., 1981. Functional Modelling for Logic Simulation. In *Proceedings of the 18th Design Automation Conference* [ACM81], pages 791–795.

[Ramachandran83] Ramachandran, V., 1983. An Improved Switch-level Simulator for MOS Circuits. In *Proceedings of the 20th Design Automation Conference* [ACM83], pages 293–299.

[Rathmell86] Rathmell, J., 1986. The Abstraction of Input Rise/Fall Times in Analysing IC Performance. In *Proceedings of the Australian and Pacific Area Region Microelectronics Conference* [IRE86], pages 169–175.

[Ruan+85] Ruan, G. and Vlach, J., 1985. Current Limited Switch-Level Timing Simulator for MOS Logic Networks. In *IEEE International Conference on Computer-Aided Design ICCAD-85* [IEE85].

[Ruan+88] Ruan, G., Vlach, J., and Barby, J., 1988. Current-Limited Switch-Level Timing Simulator for MOS Logic Networks. *IEEE Transactions on Computer-Aided Design*, pages 659–667, Jun 1988.

[Rubinstein+83] Rubinstein, J., Penfield, P., and Horowitz, M., 1983. Signal Delay in RC Tree Networks. *IEEE Transactions on Computer-Aided Design*, pages 202–210, Jul 1983.

[Ruehli81] Ruehli, A., 1981. Survey of Analysis, Simulation and Modelling for Large Scale Logic Circuits. In *Proceedings of the 18th Design Automation Conference* [ACM81], pages 124–129.

[Saab+88] Saab, D., Mueller-Thuns, R., Blaauw, D., Abraham, J., and Rahmeh, J., 1988. CHAMP: Concurrent Hierarchical and Multilevel Program for Simulation of VLSI Circuits. In *IEEE International Conference on*

Computer-Aided Design ICCAD-88 [IEE88], pages 246–249.

[Sakai+82] Sakai, T., Tscuchida, Y., Yasuura, H., Oi, Y., Ono, Y., Kano, H., Kimura, S., and Yajima, S., 1982. An Interactive Simulation System for Structured Logic Design — ISS. In *Proceedings of the 19th Design Automation Conference* [ACM82], pages 747–754.

[Sakallah85a] Sakallah, K., 1985. Polynomial Terminal Equivalent Circuits as Dormant Models in Event Driven Circuit Simulation. In *IEEE International Conference on Computer-Aided Design ICCAD-85* [IEE85], pages 179–181.

[Sakallah+85b] Sakallah, K. and Director, S., 1985. SAMSON2: An Event Driven VLSI Circuit Simulator. *IEEE Transactions on Computer-Aided Design*, pages 668–684, Oct 1985.

[Sakuma+83] Sakuma, H., Fujinami, Y., and Kurobe, T., 1983. Nelsim: A Hierarchical VLSI Design Verification System. In *Proceedings of the IFIP TC 10/WG 10.5 International Conference on Very Large Scale Integration* [Int83], pages 109–118.

[Sakurai88] Sakurai, T., 1988. CMOS Inverter Delay and Other Formulas Using α-Power Law MOS Model. In *IEEE International Conference on Computer-Aided Design ICCAD-88* [IEE88], pages 74–77.

[Salz+89] Salz, A. and Horowitz, M., 1989. IRSIM: An incremental MOS Switch-Level Simulator. In *Proceedings of the 26th Design Automation Conference* [ACM89], pages 173–178.

[Sangster+83] Sangster, A. and Monahan, J., 1983. Aquarius: Logic Simulator on an Engineering Workstation. In *Proceedings of the 20th Design Automation Conference* [ACM83], pages 93–99.

[Sasaki+80] Sasaki, T., Yamada, A., Kato, S., Nakazawa, T., Tomita, K., and Nomizu, N., 1980. MIXS: A Mixed Level Simulator for Large Digital System Logic Verification. In *Proceedings of the 17th Design Automation Conference* [ACM80], pages 626–633.

[Saviz+88] Saviz, P. and Wing, O., 1988. PYRAMID - A Hierarchical Waveform Relaxation-Based Circuit Simulation Program. In *IEEE International Conference on Computer-Aided Design ICCAD-88* [IEE88], pages 442–445.

[Schaefer85] Schaefer, T., 1985. A Transistor-Level Logic-With-Timing Simulator for MOS Circuits. In *Proceedings of the 22nd Design Automation Conference* [ACM85], pages 762–765.

[Scheifler+86] Scheifler, R. and Gettys, J., 1986. The X Window System. *ACM Transactions on Graphics*, pages 79–109, Apr 1986.

[Schomburgk84] Schomburgk, G., 1984. A TicToc Description of the Complete TFB Chip. Technical report, University of Adelaide, Department of Electrical and Electronic Engineering, Oct 1984.

[Sedgewick88] Sedgewick, R., 1988. *Algorithms*, chapter 11. Addison-Wesley, 1988.

[Seshu+62] Seshu, S. and Freeman, D., 1962. The Diagnosis of Asynchronous Sequential Switching Systems. *IRE Transactions on Electronic Computers*, pages 459–465, Aug 1962.

[Shahdad86] Shahdad, M., 1986. An Overview of VHDL Language and Technology. In *Proceedings of the 23rd Design Automation Conference* [ACM86], pages 320–326.

[Sherwood81] Sherwood, W., 1981. A MOS Modelling Technique for 4-State True-Valued Hierarchical Logic Simulation. In *Proceedings of the 18th Design Automation Conference* [ACM81], pages 775–785.

[Sheu+88] Sheu, B., Hsu, W., and Ko, P., 1988. A MOS Transistor Charge Model for VLSI Design. *IEEE Transactions on Computer-Aided Design*, pages 520–527, Apr 1988.

[Smith86] Smith, R., 1986. Fundamentals of Parallel Logic Simulation. In *Proceedings of the 23rd Design Automation Conference* [ACM86], pages 2–12.

[Smith+87] Smith, S., Mercer, M., and Bishop, B., 1987. Demand Driven Simulation: BACKSIM. In *Proceedings of the 24th Design Automation Conference* [ACM87], pages 181–187.

[Soulé+88] Soulé, L. and Blank, T., 1988. Parallel Logic Simulation on General Purpose Machines. In *Proceedings of the 25th Design Automation Conference* [ACM88], pages 166–171.

[Stallman90] Stallman, R., 1990. *Using and Porting GNU CC.* Free Software Foundation, 1990.

[Statz+87] Statz, H., Newman, P., Smith, I., Pucel, R., and Haus, H., February 1987. GaAs FET Device and Circuit Simulation in SPICE. *IEEE Transactions on Electron Devices*, pages 160–169, February 1987. modelling.

[Stevens+83] Stevens, P. and Arnout, G., 1983. BIMOS, an MOS Oriented Multi-Level Logic Simulator. In *Proceedings of the 20th Design Automation Conference* [ACM83], pages 100–106.

[Stroustrup86] Stroustrup, B., 1986. *The C++ Programming Language.* Addison-Wesley, 1986.

[Subramanian+90] Subramanian, K. and Zergham, M., 1990. Distributed and Parallel Demand Driven Logic Simulation. In *Proceedings of the 27th Design Automation Conference* [ACM90], pages 485–490.

[Sundblad+87] Sundblad, R. and Svensson, C., 1987. Full Dynamic Switch-Level Simulation of CMOS Circuits. *IEEE Transactions on Computer-Aided Design*, pages 282–288, Mar 1987.

[Sussman+80] Sussman, G. and Steele, G., 1980. CONSTRAINTS — A Language for Expressing Almost-Hierarchical Descriptions. *Artificial Intelligence*, pages 115–135, Aug 1980.

[Sutherland89] Sutherland, I., 1989. Micropipelines. *Communications of the ACM*, pages 720–738, Jun 1989.

[Svensson+88] Svensson, C. and Tjärnström, R., 1988. Switch-Level Simulation and the Pass Transistor Exor Gate. *IEEE Transactions on Computer-Aided Design*, pages 994–997, Sep 1988.

[Szygenda72] Szygenda, S., 1972. TEGAS-2 — Anatomy of a General Purpose Test Generation and Simulation System for Digital Logic. In *Proceedings of the 9th Design Automation Workshop*, pages 116–127. ACM/IEEE, 1972.

[Szygenda+73] Szygenda, S. and Lekkos, A., 1973. Integrated Techniques for Functional and Gate-Level Digital Logic Simulation. In *Proceedings of the 10th Design Automation Conference*, pages 159–172. ACM/IEEE, 1973.

[Terman83] Terman, C., 1983. RSIM — A Logic Level Timing Simulator. In *Proceedings of the International Conference on Computer Design* [IEE83], pages 437–440.

[Thompson+80] Thompson, E., Karger, P., Read, W., Ross, D., Smith, J., and von Blucher, R., 1980. The Incorporation of Functional Level Element Routines into an Existing Digital Simulation System. In *Proceedings of the 17th Design Automation Conference* [ACM80], pages 394–401.

[Tsao+85] Tsao, D. and Chen, C., 1985. A "Fast Timing" Simulation for Digital MOS Circuits. In *IEEE International Conference on Computer-Aided Design ICCAD-85* [IEE85], pages 185–187.

[Turner79] Turner, D., 1979. A New Implementation Technique for Applicative Languages. *Software Practice and Experience*, pages 31–49, Sep 1979.

[Ulrich80] Ulrich, E., 1980. Table Lookup Techniques for Fast and Flexible Digital Logic Simulation. In *Proceedings of the 17th Design Automation Conference* [ACM80], pages 560–563.

[Vidigal+86] Vidigal, L., Nassif, S., and Director, S., 1986. CINNAMON: Coupled Integration and Nodal Analysis of MOS Networks. In *Proceedings of the 23rd Design Automation Conference* [ACM86], pages 179–185.

[Visweswariah+89] Visweswariah, C. and Rohrer, R., 1989. Piecewise Approximate Circuit Simulation. In *IEEE International Conference on Computer-Aided Design ICCAD-89* [IEE89], pages 248–251.

[VLS85] Logic Simulator Survey, Mar 1985.

[Vogel85] Vogel, R., 1985. Analytical MOSFET Models with Easily Extracted Parameters. *IEEE Transactions on Computer-Aided Design*, pages 127–134, Apr 1985.

[Walker88] Walker, M., 1988. Mixed-Mode Simulation Requires Both Accuracy and Efficiency. *Computer Design*, pages 58–65, Feb 1988.

[Wallace+88] Wallace, D. and Séquin, C., 1988. ATV: An Abstract Timing Verifier. In *Proceedings of the 25th Design Automation Conference* [ACM88], pages 154–159.

[Wang+87] Wang, L., Hoover, N., Porter, E., and Zasio, J., 1987. SSIM: A Software Levelized Compiled-Code Simulator. In *Proceedings of the 24th Design Automation Conference* [ACM87], pages 2–8.

[Weinreb+81] Weinreb, D. and Moon, D., 1981. *Lisp Machine Manual.* Symbolics Inc, 1981.

[Weste+85] Weste, N. and Eshraghian, K., 1985. *Principles of CMOS VLSI Design: A Systems Perspective.* Addison-Wesley, 1985.

[White+85] White, J., Saleh, R., Sangiovanni-Vincentelli, A., and Newton, A., 1985. Accelerating Relaxation Algorithms for Circuit Simulation using Waveform Newton Iterative Size Refinement and Parallel Techniques. In *IEEE International Conference on Computer-Aided Design ICCAD-85* [IEE85], pages 5–7.

[White+87] White, J. and Sangiovanni-Vincentelli, A., 1987. *Relaxation Techniques for the Simulation of VLSI Circuits.* Kluwer Academic Publishers, 1987.

[Wolf89] Wolf, W., 1989. A Practical Comparison of Two Object-Oriented Languages. *IEEE Software*, pages 61–69, Sep 1989.

[Wong+86] Wong, K., Franklin, M., Chamberlain, R., and B, S., 1986. Statistics on Logic Simulation. In *Proceedings of the 23rd Design Automation Conference* [ACM86], pages 13–19.

[Yang85] Yang, P., 1985. An Efficient Ordering Algorithm in the Modified Nodal Approach for VLSI Circuit Simulations. In *IEEE International Conference on Computer-Aided Design ICCAD-85* [IEE85], pages 237–239.

[Yourdon+75] Yourdon, E. and Constantine, L., 1975. *Structured Design.* Yourdon Inc, 1975.

[Zwolinski+84] Zwolinski, M. and Nichols, K., 1984. The Design of a Hierarchical Circuit Level Simulator. *IEE Electronic Design Automation*, pages 9–12, 1984.