An Interactive Digital MOS Timing Simulator with an APL User Interface

by

Roland Schneider

A thesis
presented to the University of Manitoba
in partial fulfillment of the
requirements for the degree of
Master of Science
in
Electrical Engineering

Winnipeg, Manitoba, 1985

December 3, 1985

ISBN Ø-315-33859-8

AN INTERACTIVE DIGITAL MOS TIMING SIMULATOR WITH AN APL USER INTERFACE

BY

ROLAND SCHNEIDER

A thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

December 3, 1985

Abstract

The design of high-speed digital MOS integrated circuits requires software simulation tools which can provide accurate timing information from a description of the actual chip layout. Experience with switch, timing, and circuit simulators has demonstrated that circuit simulators are too slow for large designs while switch and timing level simulators are not accurate or reliable enough to do more than verify the logical operation of a circuit; there is a the need for a simulator with capabilities somewhere in-between the timing and circuit levels of simulation.

The simulator presented in this thesis uses the iterated timing analysis (ITA) method to solve the set of non-linear differential equations resulting from the evaluation of realistic MOSFET model equations. The simulation is accomplished by what is essentially a first-order circuit analysis. Because ITA is a relaxation method, it is possible to efficiently exploit the latency and sparse interconnections inherent in large digital circuits, making the simulator 50 to 70 times faster than SPICE.

The user interactively controls the simulator through extensions to the APL programming language. The unique combination of a fast, accurate, simulator with the powerful interactive vector and matrix manipulation capabilities of APL gives the IC designer a flexible tool for the design and verification of digital MOS integrated circuits.

Acknowledgements

I would like to thank my advisor, Prof. H.C. Card for his assistance and encouragement throughout this project.

I would also like to thank my brother, Christian Schneider, for his patience in endless discussions concerning this work and Peter Hortensius and Karl Mann for their efforts in providing the test circuits used to debug the simulator.

Financial support from the Natural Sciences and Engineering Research Council of Canada and equipment loans from the Canadian Microelectronics Corporation are gratefully acknowledged.

Table of Contents

List of Figures

Chapter 1
Introduction

1.1. The Need for Simulation

It has long been recognized that computer simulation of integrated circuits is a necessary part of the design process. Computer simulation is inexpensive when compared to the cost of fabricating a chip and frequently gives more insight into problems with a design than testing the actual chip does. For this reason, there has been an enormous volume of simulator research in recent years. The research has led in two directions: simulators which are accurate and simulators which are fast. There is usually a tradeoff between the two and a balance must be struck according to the particular function the simulator is to perform in the design process.

With the continuous increase in integration density and the corresponding growth in chip complexity, the demands placed on simulators have grown and have frequently made old simulation tools useless. Even at the University of Manitoba, we have been unable to perform accurate timing verification on any but the smallest of our designs. The simulator described in this thesis was developed to allow University of Manitoba designers to perform better simulations than were previously possible.

1.2. History of the Project

Integrated circuit design at the University of Manitoba started in earnest in mid 1983 with the successful installation of the *Electric* [19] integrated circuit design program. It was immediately recognized that there was a deficiency in our simulation tools. Up to this point, all IC simulation in our group had been performed with SPICE [14] but this was impractical for the larger circuits which could easily be designed with *Electric*. The first practical simulation was performed with a version of MIT's *ESIM* [1] NMOS switch level simulator which we modified for use with the

CMOS circuits which we were designing. With the receipt of a Metheus IC design workstation from the Canadian Microelectronics Corporation, a timing simulator was added to our tools. For various reasons, to be discussed later, none of these simulators fit our need for reasonably accurate timing verification of large circuits. A project was undertaken to fill this need. The resulting simulator has been dubbed APLSIM because it is a combination of an APL user interface and a fast circuit simulator.

1.3. Goals

At the start of the simulator project, a number of goals were identified. The primary objective was to create a simulator which filled a specific niche in the simulation tools at our disposal. With the huge volume of North American research in the simulation field, there was little hope of creating anything truly novel so the primary goal was to develop a simulation tool which would be useful in the context of digital interated circuit design at the University of Manitoba.

The goals we set out for ourselves were to create a simulator which was interactive, powerful enough to simulate large circuits, and which was capable of providing reasonably accurate timing information. Each of these goals will be discussed in more detail.

1.3.1. Interactive Simulation

There are two basic approaches to the design of a user interface for a simulator: batch and interactive. SPICE is a typical batch program. Inputs and outputs are specified on a series of 'input cards' and the program is run. It reads-in the cards, checks for errors, and hopefully performs the desired simulation. The user then looks at the output to see if the circuit performed as expected or to ascertain some information about critical timing or voltage levels. Preprocessors (programs which run before SPICE) and postprocessors (programs which run after) can be used to make the input simpler and the output more readable but the basic simulation remains a batch operation. This is not to say that SPICE could not be rewritten to be interactive; we are simply using it as an example of an existing batch simulator.

The other approach to designing a user interface is to make the program interactive. This means that the user receives feedback from the simulator during the simulation process. He can set inputs, observe outputs, and control the simulation process as it proceeds. There are several reasons why the interactive approach is, in most cases, more desirable than the batch approach.

The first and most obvious reason is that interactive simulation allows the designer to debug a circuit in much the same way he would if he were dealing with an actual printed circuit board and test equipment. Inputs can be set, outputs observed, and problems traced. Of prime importance is the user's ability to react to the outputs the program generates instead of having to anticipate the operation of the circuit when specifying the sequence of inputs to be used. Of course, debugging a simulated circuit is easier than debugging a real one because time can be stopped to take observations, and problems in the circuit can be traced much more quickly and easily. Depending on the design of the simulator, it may even be possible to correct errors in the circuit interactively and continue the simulation.

Closely coupled with the simplified debugging of the circuit is the fact that an interactive simulator, or any interactive program for that matter, is much easier to use than its batch counterpart. If the user makes an error in input syntax, the program informs him immediately. He does not have to submit the simulation job to batch and wait for the program to tell him, for example, that he made a silly mistake on the seventh line of input to the program and that processing cannot continue.

The final benefit of an interactive simulator is that it helps the designer to understand the circuit he is dealing with. It may seem that the designer should understand what he has created but MOS transistor circuits, especially CMOS circuits, can be quite odd. Examples are the CMOS exclusive-or gate and the CMOS adder which will be presented later. With an interactive simulator, it is easy to trace the operation of the circuit and thereby gain a good understanding of its operation.

1.3.2. Speed

It is almost redundant to say that a simulator should be as fast as possible. Time spent waiting for results from a program is almost always unproductive and frequently quite aggravating. If a simulator is interactive, reasonable performance becomes even more important. If an interactive simulator must be left to run overnight before it produces results, most of the benefits of interactive operation are lost.

With the advances in IC technology, even Canadian university chips can now have on the order of 10 000 transistors and programs like SPICE become useless except for simulating very small portions of a design. Even reasonable functional subblocks of a realistic IC network are too large for SPICE to handle. Also, the process of partitioning a circuit into manageable parts is difficult, tedious, and error prone.

1.3.3. Accuracy

To date, designers in our laboratory have generally limited simulation to verification of logic using switch level simulators. While the success rate for chips which have been fabricated is near 100%, there is a concern that the lack of timing verification makes it impossible to design chips to operate at high clock frequencies. An example is a recently designed Viterbi decoder chip. [10] Only a few minutes of simulation with a preliminary version of APLSIM confirmed suspicions that one particular logic structure, employed to save space, is probably the factor limiting the clock frequency. We expect that redesign of that portion alone might allow a doubling of the clock frequency. Therefore, one of the goals in designing this simulator was to make it accurate enough to give useful timing information.

There was no attempt to create a simulator which could compete with SPICE for accuracy. Since the main interest at the University of Manitoba is in digital circuits, no attempt was made to be able to simulate, say, an operational amplifier. It was also recognized that the only technology which would be used by the University of Manitoba in the near future would be MOS, especially CMOS. The resulting goal, then, was to build a simulator which was well suited to simulating static and dynamic digital CMOS full-custom[1] integrated circuits.

1.3.4. Educational Value

The last goal was to learn something about simulation. Using programs written by other people at other universities or in industry is instructive, but cannot compare to building a simulator from scratch. Another benefit of designing an in-house simulator is that it can be enhanced as needed and any bugs which surface can be corrected. Since the interrest was in simulation and not in numerical analysis or device physics, the theory behind the numerical methods and transistor models employed has not been explored in any great depth.

The numerical method, called iterated timing analysis, was developed at the University of California, Berkeley, and is well documented in the literature. It will be analysed only in an intuitive and largely superficial way. Detailed analysis is better left to people whose specialty is numerical analysis.

[1] Full-custom refers to the integrated design method where the designer creates each chip "from scratch" or starts with a few simple building blocks. This is contrasted with methods such as gate-array where the designer specifies only the interconnection of standardized gates.

APLSIM uses the well-known Sichmann-Hodges MOS transistor models [14]. These came from SPICE. (model level 1) The model equations are not discussed but the simplifications employed in the less accurate APLSIM model levels are mentioned in Appendix A.

The APLSIM program is currently considered to be operational but not yet fully debugged. With actual use by IC designers at the University, more will be learned about all aspects of simulation and the program will be modified and expanded accordingly.

1.4. Simulators in Use at the University of Manitoba

There are a number of simulators currently used by our group. All have different capabilities and are used in different phases of the design process. The simulators can be broken down into four classifications: gate level, switch level, timing, and circuit simulators.

1.4.1. Gate Level Simulators

Gate level simulators generally simulate functional blocks like gates and, in some cases, higher level constructs like flip-flops, adders, etc. Traditionally, they are used to simulate printed circuit board systems, although some attempt has been made to extend their use to custom IC design. There are several problems with using gate level simulators in designing integrated circuits.

The first is that gate level simulators usually employ fixed delays for each logic block. For example, the input to output delay of a *NAND* gate might be $5ns$ with some effect from output loading taken into account. The problem with this approach is that the designer must know what the delays are, so he must use other simulators to find them first because there are no data books for custom circuits. If a cell library is employed, there may be some hope of assigning delays but there are circuits which have delays which are not easy to characterize. In CMOS especially, circuits can use transmission gates, resulting in long chains of nodes which all depend on each other. The CMOS full adder (figure 3) is a case in point. If all the inputs are zero, the outputs are connected to the inputs so the inputs must supply the current to charge or discharge the output capacitances. If one of the inputs is logic one, one of the outputs will be connected to V_{DD} or V_{SS} instead. This kind of behavior is almost impossible to characterize on the gate level.

Another problem with gate level simulators is that they cannot usually handle bi-directional circuit elements properly. CMOS transmission gates are a common feature in digital designs and must be simulated properly, preferably without the designer having to employ special tricks.

The last problem with gate level simulators is that they cannot be used to verify the result of full-custom manual IC layout. Errors in layout can change the function of a circuit and no gate level simulator can check this. Also, the size of transistors and the length of interconnecting wires have major effects on circuit performance. These are factors which can only be determined from the actual layout, not from gate level descriptions.

1.4.2. Switch Level Simulators

Switch level simulators, like MIT's ESIM, model transistors as ideal switches. The simulators deal with binary logic levels of differing strengths. The problem is that transistors are not ideal switches and nodes can have an infinite number of intermediate voltages. When ESIM was received by our group, it worked only for NMOS circuits. It was converted for CMOS by changing the tables and subroutines which controlled the effects of the combinations of different logic values through active transistors. It was immediately discovered that this approach would not work for CMOS. In NMOS, the pullup transistors had a nice stabilizing effect on the relaxation iterations of the simulator. In CMOS, most circuits would cause the simulator to oscillate forever. This problem was overcome by completely changing the internal operation of the simulator so that logic levels were computed by tracing a path through active transistors to either V_{DD} or V_{SS}.[2] This made the simulator quite useful but some circuits have been encountered where even this simple method becomes unstable. The other problem with switch level simulation is that it does not provide any timing information.

1.4.3. Timing Simulators

Timing simulators like Hg[3] [13] are fast and can generate output waveforms based on the capacitances and resistances in a network. However, they have trouble with the same CMOS circuits that switch level simulators cannot handle. The problem is with certain types of feedback paths which are incorrectly analysed and create

[2] Dynamic circuits could no longer be evaluated, of course.

[3] Hg (Mercury) is also known as RNL. It is available on the Metheus IC design workstation.

incorrect results or cause the simulator to 'hang'. These are not esoteric circuits from some obscure text, but rather common blocks such as a CMOS adder, (figure 3) an exclusive-or gate, and a master-slave D-flip-flop. (figure 5)

The other problem with timing simulators like Hg is that the user must employ what are essentially 'fudge factors' in calibrating the simulator. Static and dynamic resistances are used to control the logic levels and the rise and fall times of nodes. This is an unrealistic idealization of the operation of MOS circuits.

1.4.4. Circuit Simulators

SPICE is a widely used circuit simulator. It is quite accurate and will work for any circuit but it is enormously slow and clumsy to deal with. It requires large amounts of memory even for small circuits and is unusable for large ones. In spite of having been around for many years, SPICE still has problems. Most notably, the time step control mechanism frequently runs into trouble although there are some versions of the program where this seems to have been fixed.

The basic problem is that SPICE is unnecessarily accurate. Even when using the least accurate MOS transistor model, level 1, factors are taken into account which have little bearing on digital circuits: for instance, reverse leakage current through diodes into the substrate. These can be important in analog designs, but digital circuits are inherently robust and small leakage currents and voltage dependent capacitances are seldom critical. There is also a concern that the complicated SPICE MOSFET model parameters (especially level 2) are hard to obtain from actual measurements and may vary between chips or processing runs. The old 'garbage in garbage out' adage may apply here.

1.5. Other Simulators

The volume of research in the simulator field makes it impossible to present more than a cursory review of the integrated circuit simulators which have been developed at other universities and in industry. The simulators which will be mentioned are those whose features were either emulated or avoided during the design of APLSIM.

1.5.1. SPLICE Mixed Mode Simulator

SPLICE is a mixed mode simulator, meaning it incorporates several different simulation methods which are coordinated to allow them to simultaneously simulate different parts of a single network. By combining functional, switch, timing, and circuit level simulation, a mixed mode simulator gives good performance in non-critical parts of a circuit while still providing accurate timing and voltage information where desired. The iterated timing analysis technique [17] developed at the University of California, Berkeley, and used in APLSIM, evolved from the timing simulation part of SPLICE. Iterated timing analysis has been implemented in the newer versions of SPLICE.

1.5.2. MOSSIM Switch Level Simulator

MOSSIM [2] is one of the first switch level simulators to be developed. It models transistors as ideal switches in a way similar to ESIM. MOSSIM employs a rather complicated technique for partioning and ordering the network for simulation. Since it was developed for NMOS circuits, it is hard to say whether is would work for CMOS or not. As mentioned previously, the NMOS pullup transistors tend to have a stabilizing influence on switch level simulators. Literature describing MOSSIM, and other switch level simulators, convinced us that a more realistic network model must be applied if accurate results are expected.

1.5.3. SHIELD Multilevel Simulator

SHIELD [4] is a mixed mode simulator developed at the Hughes Aircraft Company. The program is currently about 100000 lines long and implements mixed-mode simulation using a set of fairly autonomous sub-simulators coordinated by a *global multiplexer*. Since it was not practical to write a program anywhere near 100000 lines long for this project, it was decided that only a simple mixed mode facility would be provided in APLSIM for the time being.

1.6. Notes and Observations about IC Simulation

There are a few general conclusions which can be drawn from experience with other simulators:

1) Gate level simulation, while generally very fast, is not appropriate for the later stages of full custom integrated circuit design. Therefore nothing more will be said about gate level simulators.

2) It is desirable to model the behavior of a circuit with methods which mimic reality instead of using gross idealizations. This means that digital MOS transistor circuits must be treated as the analog creatures they really are. Simplifications like assuming that the transistors are perfect switches or calculating delays from RC networks will invariably result in problems. Circuits should be modeled by non-linear differential equations.

3) A simulator must be relatively fast for large digital circuits. This means, among other things, that the latency property of large digital designs must be exploited. Latency refers to the observation that because only a small fraction of the total function of a chip is utilized simultaneously, fewer than 20% of the nodes in a typical 500 transistor circuit are changing at any one time. Simulator performance can therefore be improved by at least a factor of five by exploiting this latency efficiently and evaluating only those portions of the network which are changing.

4) A simulator should use 'real' transistor parameters, not fudge factors. Transistor beta[4] and threshold voltage are in widespread use and are available for virtually any fabrication facility. Using parameters like these allows calibration on the basis of simple DC and capacitance measurements. Since the simulation is based on these simple parameters, the user is guaranteed reasonable results so long as the parameters are accurate. Using fudge factors in the manner of the Hg timing simulator always leaves some doubt about the results when simulating a circuit different from the one the simulator was calibrated with.

1.7. The Development of APLSIM

The rest of this thesis will discuss the theoretical basis for, and the practical implementation of, the APLSIM simulator. Chapter 2 deals with simulation in a general way starting with networks and the differential equations used to model their behavior. Circuit, timing, and switch level simulators are discussed along with some of the problems we have encountered when using them. Chapter 3 introduces *iterated timing analysis*, an accurate relaxation method for solving the non-linear differential equations encountered in MOS circuit simulation, developed at the University of California, Berkeley.

[4] The beta of a MOS transistor is defined by $\beta = \mu C_{ox}$ where μ is the majority carrier mobility and C_{ox} is the gate oxide capacitance.

The development of the APLSIM simulator is discussed in chapters 4 and 5. Chapter 4 describes how the network is represented in the program and how this relates to the efficient implementation of iterated timing analysis. The details of the actual implementation of the numerical method, and the additions needed to make it work for real circuits, are discussed. Chapter 5 presents the APLSIM simulator's somewhat unusual user interface, which is embedded in the APL programming language. The general requirements of a simulator user interface are explored, followed by a description of how the simulator and the APL interpreter were combined to meet those needs. Chapter 6 presents the results of comparisons between APLSIM and SPICE, followed by a list of planned enhancements to the simulator.

Appendix A presents the transistor model equations and their derivatives as used by the APLSIM simulator. Appendices B and C are, respectively, the APLSIM user guide and a short tutorial on the aspects of the APL language which are necessary for basic simulator operation. These two manuals will give the interrested reader a better understanding of the operation of the simulator from the user's point of view and show how neatly a digital simulator user interface can be mapped into the APL language.

Chapter 2
The Fundamentals of Simulation

2.1. Networks

Before a network can be simulated, it must be described to the simulation program. Since our interest is in the simulation of digital MOS integrated circuits, the discussion here will focus on that type of network. An MOS transistor network can be described as a set of nodes interconnected by active and passive elements such as transistors, resistors, and capacitors. Note that inductors are not necessary in the evaluation of digital MOS circuits since inductance effects in integrated circuits are small when compared to capacitance and resistance effects. The nodes are, in general, the wires on the actual integrated circuit, and the node capacitance is actually the distributed capacitance of the wires.

Transmission line effects are not taken into account and all capacitance is assumed to be lumped at the nodes. While this is obviously a simplification, transmission line effects are second order, at least in a reasonably designed chip operating at frequencies typical of today's MOS technology.

The network description can be generated in a number of ways. For simple circuits, it can be created manually. More usually, it is generated automatically from a physical description of the integrated circuit mask layers. There are programs which can extract information about network interconnection, transistors, and capacitances from physical descriptions such as CIF. [12]

2.2. Network State

The potentials of all the nodes in the network at a particular time point t_0 uniquely determine the state of the network. If the internal node potentials are considered to be a vector $\mathbf{v}$ and the inputs to the network a vector $\mathbf{u}$, we have

$$S(t_0) = P(\mathbf{v}(t_0), \mathbf{u}(t_0)) \tag{1}$$

where $S(t_0)$ is the current logical state of the network and P is a function mapping the set of internal and input node potentials into the set of all possible network states. In other words, the current state of the network is completely determined by the potentials of its internal nodes and its external inputs. This is hardly a surprising result since there are no inductors in the circuit. The process of simulation involves finding the new network node potentials $\mathbf{v}(t_{k+1})$ at time t_{k+1} from the current potentials $\mathbf{v}(t_k)$ and the new set of inputs $\mathbf{u}(t_{k+1})$. In digital simulation, we are primarily interested in the new network state $S(t_{k+1})$, allowing certain simplifications to be made.

2.3. Equation Formulation

Since we are dealing with a restricted class of circuits, namely digital MOS networks for integrated circuits, we will make a few simplifications before formulating the equations which describe the behavior of the circuit. The first simplification is that all capacitances are assumed to be linear and time invariant. It is well known that MOS capacitors are not linear but the saturated nature of most digital circuits means that any error introduced by this assumption will be limited to the periods when nodes are in transition from one logic level to the other and choosing an average capacitance over the range of the power supply voltage will give sufficiently accurate estimations of the time required for the logic transition.

The other simplifying assumption we shall make is that there is a non-zero capacitance between each node in the circuit and the reference, or ground, node. We shall also assume that there are no floating (ungrounded) capacitors in the circuit. That the first assumption is valid is obvious: every wire in an integrated circuit has some capacitance to ground. The second assumption is not necessarily a universal one since some designs, particularly 'bootstrap' circuits, use floating capacitors to generate voltages outside the power supply range. However, with the exception of 'bootstrap' circuits, capacitances between nodes in a circuit can be approximated by adjusting their capacitance to ground. Circuit extraction programs currently available to us do not handle inter-node capacitances so having the ability to simulate them would not be of much use.

It should be noted at this point that while the current version of the APLSIM simulator makes these assumptions, there is nothing inherent in the numerical method employed which requires them. It will be possible to modify the simulator in the future should non-linear or floating capacitors become important. The non-zero capacitance requirement has already been relaxed in certain cases as will be seen later. Using the assumptions above, we can write a set of differential equations describing the response of the network.

$$C_i \dot{v}_i(t) = f_i(\mathbf{v}(t), \mathbf{u}(t)) \qquad i = 1, 2, \cdots N \tag{2}$$

where

C_i is the capacitance of node i with respect to ground.

$\dot{v}_i(t)$ is the time derivative of the potential at node i.

$f_i(\cdot)$ is the net current flowing into node i.

We can write an equation similar to (2) for all the non-input nodes in the network. Note that $f_i(\cdot)$, the current charging the node capacitance, C_i, is a function of $\mathbf{v}(t)$ and $\mathbf{u}(t)$, the voltages of the nodes in the network. Actually, the current is only a function of a subset of the node and input voltages, a fact which will become important later.

A more general form of (2) which allows floating and non-linear capacitors can be found in [17]. Since APLSIM currently deals only with linear grounded capacitors, (2) will suffice for this discussion.

Equation (2) can be written for each node in the network at any time t. Since $f_i(t)$ is the sum of the branch currents flowing through MOS transistors, we have a set of non-linear differential equations. In general, there will be N equations where N is the number of internal or non-input nodes in the network.

It is the method used to solve these equations which differentiates between conventional circuit simulators like SPICE, timing simulators like Hg (RNL) and iterated timing analysis based simulators like APLSIM and the newer versions of SPLICE.

2.4. Circuit Simulators

In a circuit simulator like SPICE, an integration method such as the trapezoidal rule is used to discretize the differential equations and generate a system of simultaneous non-linear difference equations. These equations are linearized by the Newton-Raphson method and solved by LU decomposition or Gaussian Elimination. The Newton-Raphson/Linear equation solution sequence is repeated until the Newton-Raphson method converges and we have the solution to the difference equations for the current time point. The time step to the next time point is computed and the process is repeated.

One of the problems is that the coefficient matrix for the linear equations is very large, usually $N \times N$. This wastes both computer time solving the equations and memory storing them. The matrix is actually very sparse, typically only about 2% filled in. There are two reasons for this: 1) the interconnections in a large circuit, particularly a digital one, tend to be quite localized, that is, the transistors are in relatively small, tightly interconnected groups and; 2) even when two nodes are connected through the channel of a transistor, there will be no interaction unless the transistor is on which is, on average, only 50% of the time. If the transistor is off, there will be no coupling between the equations of the two nodes.

All this assumes that the myriad of second order effects which occur in real circuits and are modeled by SPICE can be neglected. This is possible in the type of digital simulation we are aiming for. When these effects are important, a circuit simulator like SPICE should be used.

2.5. Timing Simulators

The idea behind a timing simulator is to give reasonably accurate timing information about a circuit, but to do so much more quickly and using less memory than a circuit simulator, thereby allowing large IC designs to be simulated. One method, used by programs like the old version of SPLICE, is to discretize the differential equations using a method like the trapezoidal rule and then make a single pass through the network with a Newton-Raphson algorithm to approximate the new node potentials, taking into account the node capacitances, the time step, and the conductances of the pertinent transistors. See [17] for a more rigorous description.

Effectively, timing simulators solve for new node voltages at each time point by making one pass through the network and calculating new potentials from the old ones, the node capacitances, and the calculated branch currents. The problem is that this approach does not actually solve the non-linear difference equations resulting from the application of the trapezoidal rule. Because each node is evaluated only once per time step, using a time step which is too large can result in the generation of incorrect results, including non-existent oscillations and can sometimes cause the simulation to 'blow up'.

Another problem with timing simulators is that they can generate what is known as *timing error*. Consider a series of inverters:

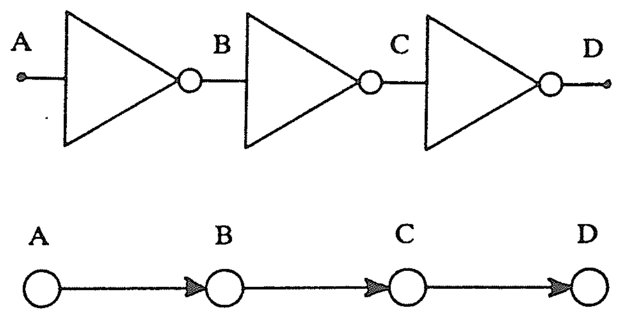

Figure 1. *A series connection of inverters and its signal flow graph*[5]

If the potential at node A changes and new node values are computed in the order D, C, B, an error of one time step will be made on each pass through the network because information from the left is only propagated to the right one node per time step. If the nodes are evaluated from left to right, the direction in which the signal flows, no timing error will occur. In general, this order of evaluation is easy to achieve, as we shall see later. The problem is that if a circuit contains feedback loops, there will always be a one-time-step timing error in going around the loop.

[5] A signal flow graph is a directed graph showing the signal propagation paths in a network. There is an edge from each transistor gate node to the corresponding source and drain nodes and one edge in each direction between the source and drain.

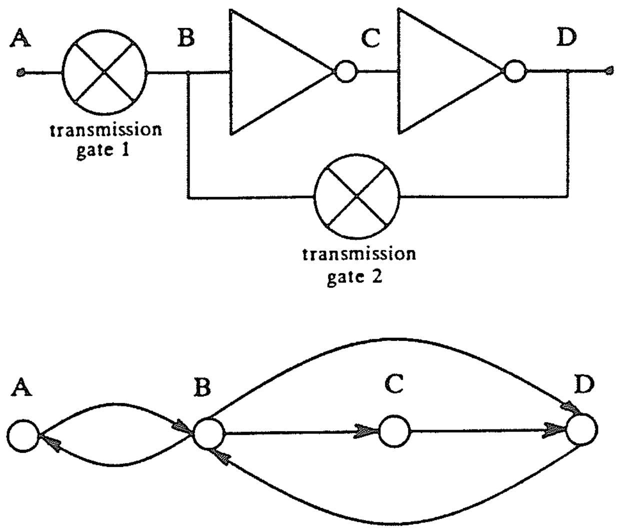

Figure 2. *Two inverters with feedback: A D-Latch and its signal flow graph*

Consider the D-Latch in figure 2. A perturbation at node B, perhaps caused by a glitch in the circuitry controlling transmission gate 1, would propagate through the circuit and be fed back to B. In a timing simulator, node B would only be evaluated once per time step, resulting in a one-time-step error for any signal being fed back around the loop. This would certainly give a false output waveform and could even cause the simulator to mistakenly determine that the latch changed state when it should not have or visa versa.

As another example, consider the 20 transistor implementation of a CMOS full adder from Mavor [11] shown in figure 3 and the associated signal flow graph in figure 4.

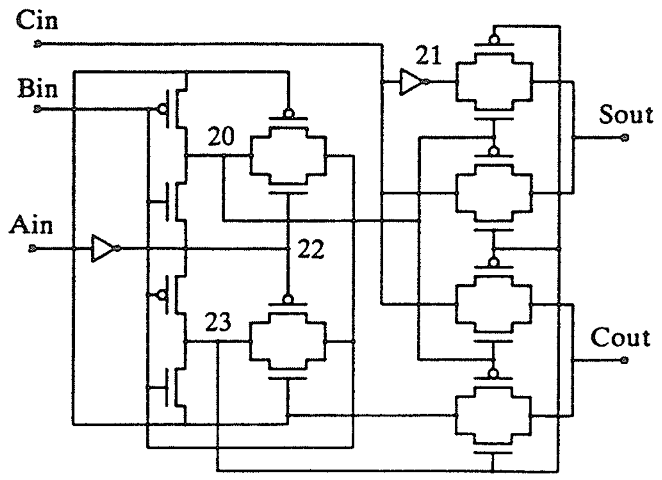

Figure 3. *A CMOS one bit full adder employing transmission gates*[6]

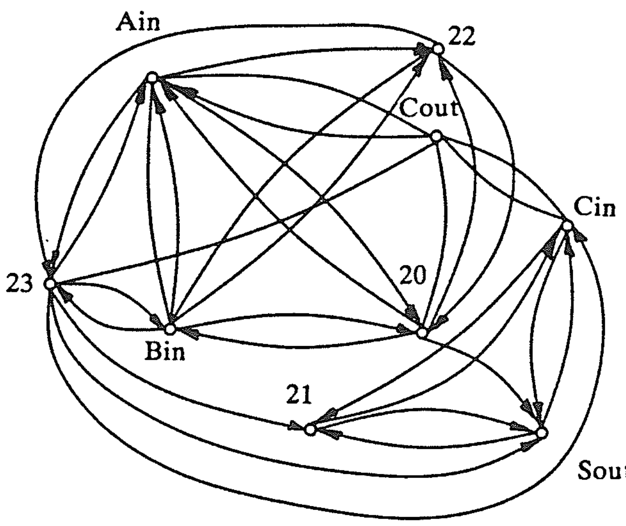

Figure 4. *Signal flow graph for the CMOS 1-bit adder*

From the simulator's point of view, this circuit is full of feedback paths. Actually, of course, this is a combinational circuit and contains no feedback at all. What fools the simulator is that a particular output can, under certain conditions, come from more than one source at a time. All the sources agree, so the circuit works,

[6] This schematic was prepared by Peter Hortensius.

but a simulator which analyses the circuit one node at a time will become confused.

Another circuit which has proven troublesome for both timing and switch level simulators is the D-flip-flop in figure 5. This circuit comes from the National Semiconductor CMOS data book [15], page 1-28.

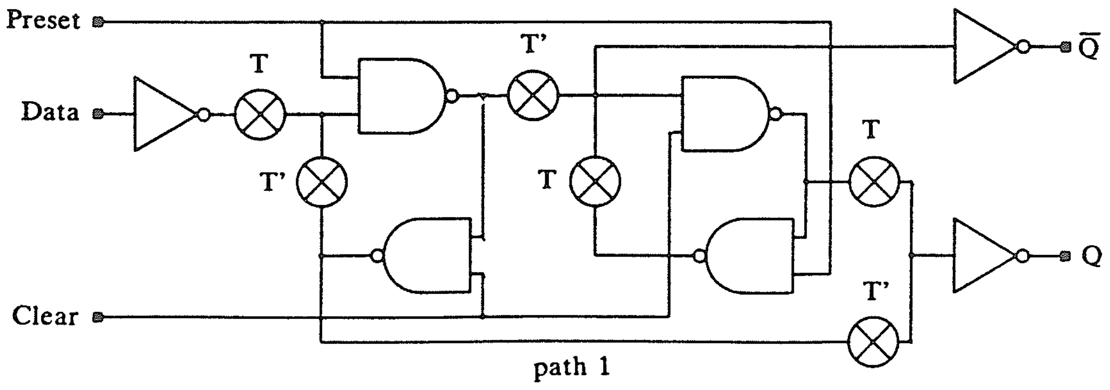

Figure 5. *The troublesome D-flip-flop*

The trouble again stems from what appear to be multiple feedback paths. In actual fact, path 1, labeled in figure 5, is a feed *forward* path. It is used to generate the Q output while the input clock is high so that the PRESET and CLEAR inputs will act on Q and $\overline{Q}$ simultaneously.

The principal cause of these problems is that timing simulators do not actually solve the difference equations which model the circuit. Only an approximation is made. The *simultaneous* nature of events in the network is ignored. There is, in general, no way to fix this problem. While the feedback in figure 2 can probably be taken into account, there would always be other circuits like the full adder in figure 3 and the flip-flop in figure 5 which could not be covered. The basic problem is that real signals in real circuits do not propagate in the sequential manner modeled by timing simulators.

The next chapter presents a method known as iterated timing analysis which is much faster than circuit simulation (SPICE) and more reliable and accurate than timing simulation (Hg). In iterated timing analysis, the difference equations are completely solved at each time point using a relaxation method.

Chapter 3
Iterated Timing Analysis

3.1. Introduction

The simulation method known as iterated timing analysis overcomes the difficulties associated with timing simulators while still operating much more quickly than circuit simulators like SPICE. We do not present a rigorous mathematical treatment of the method here. The reader is referred to [17] for more details.

The principle behind iterated timing analysis (ITA) is that the system of simultaneous non-linear difference equations produced by applying an integration formula to (2) must be solved completely, as in circuit simulation, but it can be solved using a relaxation method, as in timing simulation. Instead of making only one pass through the network per time step, the iterations continue until convergence is achieved and all the equations are satisfied. Because all the equations are solved simultaneously, no timing error results and the method is, at least in theory, every bit as accurate as that used in circuit simulation.

3.2. Gauss-Seidel and Gauss-Jacobi

Gauss-Seidel and Gauss-Jacobi are two well-known relaxation methods for solving a system of simultaneous linear equations. [7], p. 214 A sufficient condition which guarantees convergence is that the coefficient matrix be diagonally dominant. In other words, if we have a system of linear equations

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

and the absolute value of the coefficient of x_i in equation i, namely $|A_{ii}|$, is larger than the sum of the absolute values of the other coefficients in equation i for all equations $i = 1 \cdots N$, the Gauss-Seidel and Gauss-Jacobi methods will converge to the solution independent of the initial guess.

The Gauss-Seidel method proceeds as follows:

```
while not converged
    for each equation $i$
        solve $A_{i1}x_1^{k+1} + A_{i2}x_2^{k+1} + \cdots + A_{ii}x_i^{k+1} + A_{ii+1}x_{i+1}^k + \cdots + A_{iN}x_N^k = b_i$
            for $x_i^{k+1}$ while holding all the other $x_j$, $j \neq i$, constant.
```

Notice that in the Gauss-Seidel method, the value obtained for x_i^{k+1} is computed from $x_1^{k+1}, x_2^{k+1}, \cdots x_{i-1}^{k+1}$ from this iteration and $x_{i+1}^k, x_{i+2}^k, \cdots x_N^k$ from the previous iteration. A moment of thought will reveal that if the coefficient matrix, A, is lower triangular, only a single iteration of the method is needed to find the solution. The Gauss-Jacobi method is similar, except that only results from the previous iteration are used in finding x_i^{k+1}. Therefore, there is no benefit from having a lower triangular coefficient matrix but it becomes possible to implement the method on a parallel computing architecture.

3.3. Gauss-Seidel-Newton

Since the difference equations to be solved are not linear, it is necessary to modify the Gauss-Seidel algorithm by using a single iteration of the Newton-Raphson method to estimate the value of x_i from equation i, instead of solving for x_i directly as in the linear case. Notice that the Newton-Raphson step only solves for a single variable, x_i, at a time, and is therefore very easy to implement. It is not necessary to go through the computationally demanding process of finding a Jacobian.

The Gauss-Seidel-Newton method proceeds as follows. Assume we have a system of non-linear equations $g_i(\mathbf{x}) = 0$, $i = 1 \cdots N$.

```
while not converged
    for each equation $g_i$
        solve $g_i(x_1^{k+1}, x_2^{k+1}, \ldots x_i^{k+1}, x_{i+1}^k, \cdots, x_N^k) = 0$
            for $x_i^{k+1}$ using one iteration of the Newton-Raphson method
```

Intuitively, the convergence criterion of 'diagonal dominance' should still hold, although the meaning is a little hard to fathom. More precisely, the Jacobian of the equations should be diagonally dominant. Notice again that, as in the linear case, if x_i depends only on x_j, $j < i$ (the equivalent of a lower triangular coefficient matrix) the iterative method will converge to the solution more rapidly than would otherwise be the case.

3.4. Iterated Timing Analysis

We start by recalling equation (2):

$$C_i \dot{v}_i(t) = f_i(\mathbf{v}(t), \mathbf{u}(t)) \tag{3}$$

which describes the behavior of the potential $v_i(t)$ at node i as a function of the node capacitance C_i and the net input current $f_i(\mathbf{v}(t), \mathbf{u}(t))$, where $\mathbf{v}(t)$ is the vector of potentials of all non-input nodes in the network, and $\mathbf{u}(t)$ is the vector of all externally-specified input voltages. If the equations are discretized with the backward Euler formula, we get

$$g_i(\mathbf{v}, \mathbf{u}) \;=\; C_i[v_i(t+h) - v_i(t)] - hf_i(\mathbf{v}(t+h), \mathbf{u}(t+h)) = 0 \tag{4}$$

where h is the time step and $\mathbf{v}$ and $\mathbf{u}$ are assumed to be functions of time. This set of simultaneous difference equations can now be solved using the Gauss-Seidel-Newton method, assuming the derivatives $\dfrac{\partial f_i(\mathbf{v}(t+h), \mathbf{u}(t+h))}{\partial v_i}$ are known.

We will take one last look at the convergence criteria for this method. Intuitively, the larger the value of C_i, the more likely the $C_i v_i(t+h)$ term of equation (4) is likely to be dominant. A smaller time step, h, also makes the term more dominant be decreasing the importance of the branch currents, $f_i(\cdot)$. Large terms in $f_i(\cdot)$ are generated by tight feedback loops. A smaller time step is required in such cases. [17] states that it has been proven that the Jacobian can always be made diagonally dominant by making h sufficiently small.

3.5. Implementation

In the APLSIM program, ITA uses the trapezoidal rule to perform the discretization of the differential equations (2). We get

$$g_i(\mathbf{v}, \mathbf{u}) \;=\; C_i[v_i(t+h) - v_i(t)] - h\left[\frac{f_i(\mathbf{v}(t), \mathbf{u}(t)) + f_i(\mathbf{v}(t+h), \mathbf{u}(t+h))}{2}\right] \tag{5}$$

and

$$\frac{\partial g_i(\mathbf{v}, \mathbf{u})}{\partial v_i} \;=\; C_i - \frac{h}{2}\left[\frac{\partial f_i(\mathbf{v}(t+h), \mathbf{u}(t+h))}{\partial v_i}\right] \tag{6}$$

because the currents and voltages from the previous time step are not functions of the present voltage.

The Newton-Raphson iteration is expressed as

$$v_i^{k+1} = v_i^k - \frac{g(\mathbf{v}, \mathbf{u})}{\partial g_i(\mathbf{v}, \mathbf{u})/\partial v_i} \tag{7}$$

Eventually, when we are near convergence, $g_i(\cdot) \approx 0$ for all i and the difference equations will be satisfied.

The function $f_i(\cdot)$ is the sum of all the branch currents flowing into node i. Each branch current flows through a circuit element, that is, a MOS transistor or a linear resistor. If the element is a linear resistor, the current flowing from node j into node i is a function of v_i and v_j and the resistance of the element, R_{ij}, and can be found using Ohm's law.

$$I_{ji} = \frac{v_j - v_i}{R_{ij}} \tag{8}$$

and the derivative of the current with respect to v_i is

$$\frac{\partial I_{ji}}{\partial v_i} = -\frac{1}{R_{ij}} \tag{9}$$

If the element is a transistor, a non-linear MOSFET transistor model equation must be evaluated. In the APLSIM program, the Sichmann-Hodges equations are used. These were chosen because they are relatively simple, and therefore quick to evaluate, but still provide adequate accuracy. SPICE model 1 also uses the Sichmann-Hodges equations. The current flowing from node j into node i is a function of many variables but only three, the gate, source, and drain voltages change during circuit operation. We obtain the following expression:

$$I_{ji} = I_{ds}(v_{gate}, v_i, v_j) \tag{10}$$

and the derivative of the current with respect to v_i is given by

$$\frac{\partial I_{ji}}{\partial v_i} = \frac{\partial I_{ds}(v_{gate}, v_i, v_j)}{\partial v_i} \tag{11}$$

where v_{gate} is the potential of the node which is connected to the gate of the transistor whose drain and source are nodes i and j. The values I_{ji} and $\frac{\partial I_{ji}}{\partial v_i}$ are both computed by the model equations in appendix A. (10) and (11) (and (8) and (9)) are the current and its derivative for one transistor (or resistor) connected to node i. In order to find the total current and its derivative, we sum I_{ji} over all the nodes j connected to i through active transistors or resistors.

$$f_i(\mathbf{v}, \mathbf{u}) = \sum_j I_{ji} \tag{12}$$

and

$$\frac{\partial f_i \mathbf{u}, \mathbf{v})}{\partial v_i} = \sum_j \frac{\partial I_{ji}}{\partial v_i} \tag{13}$$

Equations (12) and (13) can now be substituted into (5) and (6) so we have everything we need for the Newton-Raphson iteration, (7). In the actual program, it is not necessary to compute the $f_i(\mathbf{v}(t), \mathbf{u}(t))$ term of (5) as it can be stored at the previous time point when it was computed as the $f_i(\mathbf{v}(t+h), \mathbf{u}(t+h))$ term. This 'last current' value is stored with every node in the network. We are now ready to present the simulation algorithm.

```
while not at end time
    while not converged
        for each network node $i$
            {
            compute $f_i = \sum_j I_{ji}$
            compute $\frac{\partial f_i}{\partial v_i} = \sum_j \frac{\partial I_{ji}}{\partial v_i}$
            compute $g_i(\cdot)$ from $f_i$ using (5)
            compute $\frac{\partial g_i(\cdot)}{\partial v_i}$ from $\frac{\partial f_i}{\partial v_i}$ using (6)
            set $v_i$ to the new value computed using the Newton-Raphson step (7)
            }
```

3.6. Selective Trace

One of the weaknesses of circuit simulators pointed out previously is that they always solve N equations in N unknowns, even when a large number of the equations are uncoupled because of the nature of the signal flow graph. In addition, a large fraction of the equations remains unchanged over a number of time steps because of the latency inherent in digital circuits. Various sparse matrix and network partitioning techniques are used to try to circumvent these problems, but because of the marginal effectiveness of these schemes and the overhead involved in their implementation, performance improvements have only been on the order of a factor of four or so. [17]

One of the great advantages of ITA is that an algorithm known as *selective trace*[7], which avoids these problems, can be implemented with minimal computational cost. Selective trace gets its name from the fact that it *selectively traces* the propagation of signals through the network. It proceeds as follows:

> Assume that the circuit is in a steady DC state, that is, nothing is changing. One of the inputs is then altered by the user. The simulator looks at the portion of the network connected to that input node and schedules any nodes which may be affected by the input change for evaluation. The following nodes are scheduled:
>
> 1) source and drain nodes of transistors for which the input node is the gate
> 2) source nodes of transistors for which the input node is the drain
> 3) drain nodes of transistors for which the input node is the source
> 4) nodes connected to the input through linear resistors

Note that if 'input' is replaced by 'node which changed', this same algorithm will work anywhere in the network, whether the change was due to an input or to calculation by the simulator itself. This technique is in many ways similar to that used in event driven gate and switch level simulators and also in timing simulators. The main difference is that, when a node is scheduled for evaluation in ITA, it will be evaluated repeatedly in sequence with all the other affected nodes until its value does not change significantly from one iteratiion to the next. (it converges) The simulator time will not be advanced until all the node voltages have converged.

The selective trace algorithm has two beneficial effects on the simulation. The first, mentioned previously, is that only nodes which are changing will be evaluated. The effects of changes are *propagated* through the network, but only to those nodes which are potentially affected. If a node is scheduled for processing but does not actually change, perhaps because it was already at the right potential, it will be evaluated only once and then dropped from the evaluation list because it has converged. By avoiding the unnecessary evaluation of nodes which are unaffected by changes, selective trace greatly speeds up simulation.

[7] *Selective trace* is a common method of exploiting latency in a digital circuit. It is very similar to the method used in *event driven* switch and gate level simulators.

The second, somewhat more subtle, effect of selective trace is that it naturally orders the evaluation of nodes in the direction of signal propagation. In a switch or timing simulator, this is crucial to operation, but it is useful in ITA too. Recall that the Gauss-Seidel and Gauss-Seidel-Newton methods converge more quickly when the coefficient matrix is lower triangular. Each equation is a function only of variables which were evaluated in previous steps of the current iteration. The closer the matrix is to lower triangular, the faster the method will converge. Now consider the chain of inverters:

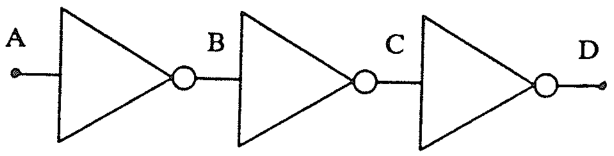

Figure 6. *A chain of inverters*

If node A is an input and is changed, node B will be scheduled for evaluation but when B is evaluated, C will be affected and will also be scheduled, followed by D in the same manner. The order of evaluation is now B, C, D, the ideal order for the Gauss-Seidel-Newton method. It is easy to see that this will work in any network, no matter how complicated, as long as there are no feedback paths.

While switch and timing simulators run into trouble with feedback, ITA simply converges more slowly since the equations can no longer be ordered in a lower triangular way. However, *it still converges*, and it converges to a solution which is as accurate as desired. Therefore feedback paths do not cause ITA any serious problems.

Chapter 4
The Simulator

4.1. Introduction

There are two parts to the APLSIM program: the simulator and the user interface. The implementation of the simulator will be discussed in this chapter. The next chapter describes the interface and how it connects to the simulator.

4.2. Real Simulation

The previous chapters described simulation in general and iterated timing analysis in particular. While the numerical method does form the heart of a simulation program, it is important to realize that the numerical process, with the exception of the device models, represents only about 250 lines of code out of a total of over 10 000. Only 2½% of the APLSIM program is actually used to implement the ITA algorithm. Of the rest, 25% is used directly in simulation, 60% for the user interface, and 15% for connecting the two parts.

The rest of this chapter will be devoted to the 25% of the program which surrounds ITA and makes it work. This consists of modifications to the simulation loop, network management, and time management.

4.3. Network Description

The network is described to the simulator in the form of a list of n and p-channel transistors with a specified length and width. The capacitance of each node is specified separately. Each node has either a name or a number; those which are important to the outside world, like inputs and outputs, are usually named. The rest are numbered. The user specifies the names by 'exporting ports' in the Electric design tool [19]. All other nodes in the circuit are assigned arbitrary numbers by the circuit extraction program.

When APLSIM reads-in the network description from a disk file, or the user types it on his terminal, the simulator builds a data structure which is in many ways similar to the signal flow graphs presented earlier.[8] The basic data structure came from the *ESIM* switch level simulator written by Chris Terman at MIT. Blocks of information, called *structures* in the 'C' programming language, are stored for each transistor and each node in the network. They are connected by several linked lists. The two types of structures, named **Node** and **Trans**, are used as follows:

1) Each node (**Node** structure) has storage locations for the potential of the node, its capacitance, some information from previous time steps, and a number of links and flags for housekeeping and representing the network.

2) Each transistor (**Trans** structure) has information about its size, the transistor β, and the model to use in simulating it. It also has linking information for representing the network. A linear resistor is stored as a transistor with a dummy node for its gate and different model information.

3) Each **Node** has a single pointer to each of: a transistor whose gate it is, a transistor whose drain it is, and a transistor whose source it is.

4) Each **Trans** has pointers to the nodes which form its gate, its drain, and its source. It also has pointers to other transistors whose gates, drains, and sources share the same node.

[8] Storing the information in a simple list would be much too inefficient since it would require searching the list repeatedly.

Consider the circuit:

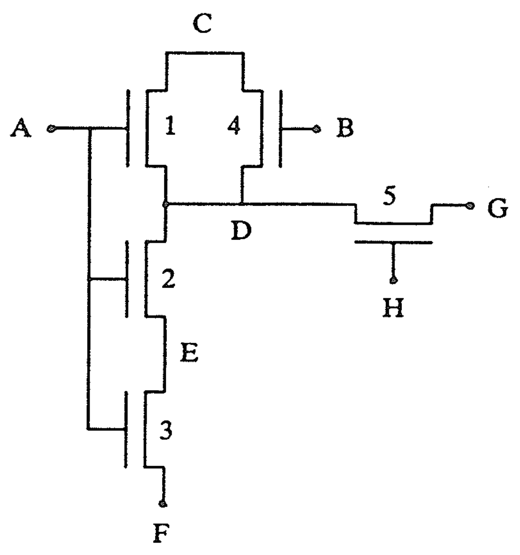

Figure 7. *An arbitrary circuit*

The following data structure will be generated:

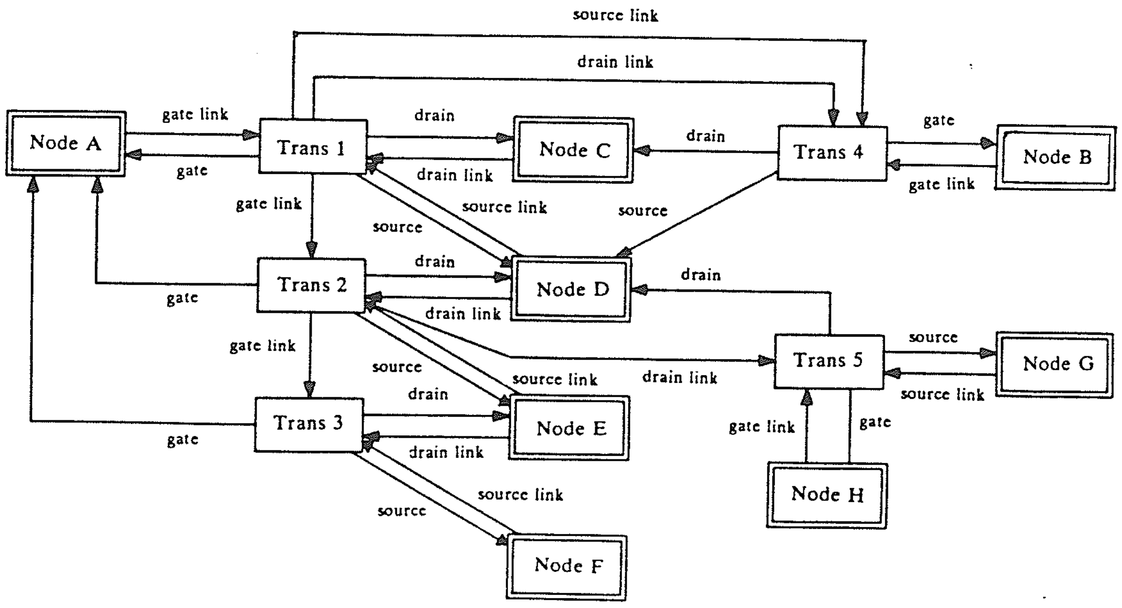

Figure 8. *Data structure corresponding to the circuit in figure 7*

It is easy to see that this data structure is ideal for implementing the selective trace algorithm described in the previous chapter. Suppose that a change occurs on node A. We follow the gate link from node A to **Trans** 1 and queue the drain and source nodes for evaluation. We then follow the gate link down to **Trans** 2 and **Trans** 3 and do the same thing. Nodes C, D, E, and F will now be evaluated. If node D changes, its source and drain links are followed and lead to, among other places, transistor 5. By looking at its gate node, H, we can determine if it is on or not. If it is, its source link is followed to node G and G is added to the list of nodes to be evaluated.

Notice that the signal propagation path was followed from A to C, D, E, and F and finally to G, resulting in the best order of evaluation for the Gauss-Seidel-Newton method used in ITA. Also notice that no searching of any kind was performed; the links were simply followed until the chain ended.

4.4. Variable Time Step

The APLSIM simulator employs a variable time step algorithm. The method currently in use is one of the least satisfactory parts of the simulator and will be modified in the near future.

When all the node equations have converged and the simulator is ready to proceed to the next time point, it visits all the nodes which were evaluated at the current time point and calculates a new time step, h, from the node potential and time information stored at the nodes.

The calculation is based on an estimate of the local truncation error (LTE) resulting from the application of the trapezoidal integration formula to the node currents to obtain the new potentials. For this we need the second time derivative of the potential at each node. The usual discrete differentiation formula cannot be used because the time steps between the points are variable. The solution is to fit a parabola to the previous three time/potential points, and to find the second derivative at the third (last) point.

Let v_0, v_1, and v_2 be the node potentials at t_0, t_1, and t_2 respectively. t_2 is the current time point. We have

$$\begin{pmatrix} t_0^2 & t_0 & 1 \\ t_1^2 & t_1 & 1 \\ t_2^2 & t_2 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} v_0 \\ v_1 \\ v_2 \end{pmatrix} \tag{14}$$

and have to solve for a, b, and c so that we can find $\left.\frac{d^2v}{dt^2}\right|_{t_2}$ from $v = at^2 + bt + c$.

Since we only need the second derivative and

$$\frac{d^2v}{dt^2} = 2a \tag{15}$$

we need only determine a, which is done most efficiently by evaluating

$$a = \frac{v_0(t_1-t_2) - t_0(v_1-v_2) + v_1t_2 - v_2t_1}{t_0^2(t_1-t_2) - t_0(t_1^2-t_1t_2^2)} \tag{16}$$

From [7], p. 319, we have

$$LTE = -\frac{1}{12}\left.\frac{d^2v}{dt^2}\right|_{t_3} h^3 \tag{17}$$

where h is the time step.

We actually want to find h given the desired LTE so we solve for h:

$$h = \left(\frac{12LTE}{d^2v/dt^2}\right)^{\frac{1}{3}} \tag{18}$$

Because this time step control mechanism does not always select the optimum step size, and to make the simulator faster, the time step is limited between user-specified upper and lower bounds, and is only allowed to grow by at most a factor of two each time step. Putting a lower bound on the time step is not as unreasonable as it sounds: while the user-specified local truncation error will not be achieved in some cases, the lower bound serves to set the maximum resolution of the simulation. Since we are dealing with digital circuits, resolution much beyond the performance of the technology being employed is not necessary.

4.5. Continued Evaluation

When an input changes and causes a number of nodes to be placed on the evaluation list, the nodes stay on the list until their equations converge. When all the equations have converged, time is advanced by the time step. The question is, which nodes should be evaluated at the new time point? When the node equations converge at a time step, it means only that their difference equations have been satisfied, not that the the nodes have settled to new steady state values. If a node has not settled, it must be evaluated again at the next time point.

When a node equation converges, the simulator knows the new value of the node potential and also the actual net input current to the node at the present time point. The net current is compared to a small constant value to determine if the node should be processed at the next time point or not. The principle behind this approach is that a node with a very small net input current will not change value and does not need to be evaluated. By using input current instead of the change in potential, problems when charging large capacitances will be avoided. The user can adjust the net current criterion to affect the tradeoff between simulation speed and accuracy.

4.6. DC Analysis

One of the assumptions made when ITA was developed in the previous chapter was that each node in the network had a non-zero capacitance to ground. While this assumption is obviously always true because of the physical nature of MOS integrated circuits, the capacitance of a node can be so small compared to others in the circuit that it might as well be zero. For example, node Q in the two-input *NAND* gate in figure 9 has a very small capacitance because it is generally a short piece of diffusion, and has no connection to the gates of any transistors.

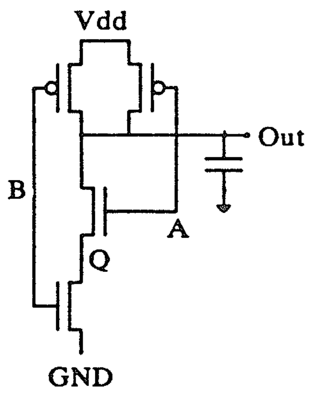

Figure 9. *NAND gate with small capacitance on internal node Q*

Since the stability of ITA requires either large capacitances or small time steps, it seems that a very small time step would have to be used in this case to guarantee that the Gauss-Seidel-Newton iterations will converge. In fact, unless the capacitance is *very* small, the numerical method will still converge, but it will do so quite slowly, requiring many iterations per time step, and the results could be inaccurate due to roundoff error.

If the capacitance of node Q is small compared to the output capacitance of the *NAND* gate, it will have very little effect on the performance of the circuit. For example, if node Q is at zero volts with both A and B at logical zero and A is changed to logical one, a small transient droop will be expected in the output until the capacitor at Q charges. However, the droop will be insignificant if the capacitance at Q is small compared to the output capacitance. For this reason, APLSIM computes the ratio of node capacitance to time step, and uses DC analysis if it falls below a certain user-adjustable threshold. In DC analysis, a binary search technique is used to find the node potential necessary for a zero net input current to the node. The justification for this is that it is impossible for there to be any net current flowing into or out of a node if it has a zero (or very small) capacitance. The binary search technique should be replaced by a Newton-Raphson algorithm.

4.7. The Action Queue

While APLSIM is an interactive program, and the user could stop simulation frequently to set inputs and give the program other commands, it is more convenient in some cases if a series of inputs can be specified to occur at some time in the future. To allow this, APLSIM implements an *action queue*. This is not the same as the event queue in a gate or switch level simulator, which is a list of nodes to be evaluated; rather it is a series of actions to be performed at specified times in the future. At each time point, the simulator examines the action queue to determine if there are any inputs to change, etc. Actions which are no longer in effect are discarded. For example, if the user specifies a piecewise linear function of input voltages for a particular node, a series of 'piecewise linear segments' is placed on the action queue. Each action describes a single segment of the piecewise linear input. As the simulator time passes the segments, they are discarded.

Controlling the scheduling of time dependent actions in this way has two benefits. It allows easy expansion of the delayed actions which the simulator can perform and it allows the time step to be adjusted to the beginnings of actions in an efficient manner. Adjusting the time step is important because a time point should occur exactly at the beginning of each action on the queue. This is accomplished by checking the time for the next action on the queue and decreasing the time step if the current simulator time plus the time step calculated from the LTE would exceed the time of the next action.

We are now ready to present the complete simulation algorithm. Initialization is not shown; it is assumed we are in the middle of a series of time steps.

The Simulation Algorithm

```
while not at end time
    {
    check action queue for maximum time step and maximum next simulator time
    adjust time step downward if necessary
    advance simulator time by the time step
    perform the actions on the action queue up to the simulator time
    remove old evaluation list and create one for next time step (empty)

    while current evaluation list is not empty
        for each node $j$ on the current evaluation list
            {
            if $\frac{C_i}{h} > constant$
                {            << transient analysis >>
                compute $f_i = \sum_j I_{ji}$
                compute $\frac{\partial f_i}{\partial v_i} = \sum_j I_{ji}$
                estimate $v_i^{k+1}$ using a single Newton-Raphson iteration
                }
            else             << DC analysis >>
                find $v_i^{k+1}$ so that $\sum_j I_{ji} = 0$

            if $|v_i^k - v_i^{k+1}| > \epsilon$
                add fanout nodes to current evaluation list
            else
                {            << converged >>
                remove node $i$ from the current evaluation list
                if $|f_i| > minimum$
                    add node $i$ to evaluation list for next time step
                }
            }

    calculate new time step from LTE of all nodes which were evaluated
    ensure that the new time step falls between a user-specified
        minimum and maximum and that it is no greater that double
        the previous time step.
    }
```

This algorithm forms the heart of the simulation loop. Various details have been omitted for clarity. The next chapter describes how the simulator is connected to the APL user interface.

Chapter 5
The User Interface

5.1. Introduction

The user interface of the APLSIM simulator is based on the APL programming language. APL can best be described as a matrix-oriented mathematical language designed for desk calculator-like interactive use. The language is interpreted rather than compiled. The interpreter was written at the University of California and distributed in source code form with Berkeley UNIX. The simulator is controlled through a number of extensions and modifications to the language.

5.2. User Interface Design

The first step in designing a user interface is to try to predict all the types of operations a user is likely to want to perform, to order them according to frequency of use, and to try to come up with a reasonable means of implementing them. Two overriding concerns are flexibility and consistency. Flexibility allows the user to perform any operation he requires. Consistency makes it easier to learn to use the program because knowing how a particular function is used is a large step towards knowing how all the other related functions are used. It is frequently just as difficult to decide how a particular operation should look to the user as it is to implement the function.

The set of operations a user needs to perform during simulation are:

1) The user must be able to set inputs to the network. In its simplest form, setting an input is simply setting a node to a specified potential. Digital circuits seldom have arbitrary input voltages so the user should be able to specify a logic value which is automatically translated into the appropriate voltage.

2) It should be possible to specify a series of input potentials in the form of a piecewise linear function. In a digital circuit, it is more likely that a series of

logic ones and zeros is desired so the user should be able to specify a vector of logic values, the frequency with which they are applied, and the transition times.

3) The results of simulation appear at the outputs of the network and the user must be able to examine these. It should also be possible to plot the output potentials against time to obtain a graphical representation of circuit performance.

4) Digital circuits frequently have a number of parallel inputs or outputs which represent a binary number. It should be possible to combine a set of these into a 'bus' which can be represented by a single decimal or hexadecimal number.

5) The operation of the simulator must be controlled. There has to be a way to start simulation and specify the period of time which is to be simulated. The user must also be able to adjust the parameters controlling the simulator's accuracy/performance tradeoff.

6) The simulator should be able to automatically examine the network state under certain conditions and perform a user-specified action. This feature allows the simulation of partially designed circuits by substituting functional blocks for the undesigned portion.

5.3. Why a language?

There are two approaches to designing an interactive user interface for a simulator like APLSIM. One option is to give the user a rich set of commands for controlling the simulator. Each command performs a specific simulator function like setting an input potential, running simulation, manipulating a data bus, etc. The difficulty is that it is hard to anticipate all the needs the user may have. Also, the set of commands will be large and could easily be inconsistent, making the program hard to learn. In general, a user interface designed this way tends to be less flexible than it should be.

The other approach, used in APLSIM, is to 'embed' the simulator commands in a computer language. The simulator commands then become part of the language and can be used from inside programs written by the user. The features of the simulator and the language should be available to the user in an interactive way. The set of commands needed in this type of user interface is much smaller than in the first approach. The reason for this is that the user can easily create his own specialized commands by combining a small program he writes with the simulator commands embedded in the language. This gives the user much more flexibility because he can tailor the operation of the simulator to his specific needs. The smaller set of commands also makes it easier to learn to use the advanced features of the simulator.

There are a few rules which have to be followed when embedding the simulator commands in a computer language:

1) The language should be, as much as possible, a general programming language. This means that *any* program could be written in the language. That is not to say that the language must be *well suited* to general programming: it should be general for flexibility.

2) It is a good idea to use an existing computer language and extend it or at least base the new language on an existing one. The advantage is that 'reinventing the wheel' is avoided because the syntax, functions, and operating characteristics of the language have already been developed and evaluated. Also, if a common language is used, some IC designers may already be familiar with it and be able to learn to use the simulator much more quickly.

3) The language must be interactive. The normal cycle of compile, link, and execute is not acceptable for an interactive simulator. The capabilities of the simulator and, if possible, the language should be available to the user on an immediate line by line basis. The simulator commands embedded in the language can be used to control the simulator and the language can be used as a kind of desk calculator. The most common interactive language is BASIC. If a statement in BASIC is typed without a line number, it is executed immediately instead of being added to the program. Any language can be made interactive with certain modifications, but some are better suited to it than others.

4) The language should have constructs which fit well with the application, in this case simulation. For digital simulation, vectors of logical values are one type of object which frequently has to be manipulated.

5.4. APL + SIMulation = APLSIM

The programming language known as APL was chosen as the basis for the user interface of the APLSIM simulator. It was chosen for a number of reasons:

1) APL is a well known language (at least in computer science) and is adored by many of its users because it can perform some remarkably powerful operations with exceptionally short programs. Most people find it fun to play with, indicating that it is a very interactive language.

2) APL is an interpreted language and an interpreter, written in 'C', was available. An interpreted language is a language which is executed by running a program in another language, in this case 'C', which performs the actions specified by the user. The *commands* to the 'C' program form the APL language. An interpreted language is no different than a compiled language from the user's point of view except that it does not have to be compiled and therefore executes immediately, making it ideal in an interactive environment. Also, because it uses an interpreted language as its user interface, the simulator and interpreter can be moved to any computer with a 'C' compiler with little modification.

   The APL interpreter came from the University of California on a Berkeley UNIX distribution tape. Even though the interpreter required modification and numerous bugs had to be fixed, using an existing program saved a lot of effort.

3) APL makes a good desk calculator since expressions can be evaluated and the results printed simply by typing them in. This is useful while simulating since it allows the designer to calculate rise and fall times, delays, and other simulation related values.

4) APL has built-in functions for manipulating vectors and matrices. Creating, transposing, rotating, and assigning matrices are all single functions in APL. There are also functions for generating random numbers, converting between bases, and many general manipulation operations. These can be used interactively to generate input vectors for the network and to manipulate outputs so they can be analysed more easily.

5) APL is well suited to writing the kind of short programs typically used to control simulation. There is no overhead, variables do not have to be declared and user defined functions are executed simply by typing their names.

There are some problems with APL as well. For one, it is not well suited to writing long, complicated programs. The control structures (if statements, while statements, etc.) are very primitive, although somewhat amusing. Solutions to this problem are being investigated. Another problem is that APL uses a character set all its own. It therefore requires a terminal capable of displaying the APL character set. This is not a problem on the Sun workstation on which the simulator was developed, but makes it difficult to use normal ASCII terminals to run the simulator.

Overall, APL was a reasonable choice as the interface language for the APLSIM simulator. If a new language had been invented, it would have been better in some respects but worse in others. It is likely that a new language would have had fewer powerful matrix manipulation functions but would have had better control structures.

5.5. Controlling the Simulator with APL

When the commands for controlling a program like a simulator are embedded into a language, two things are done: new functions related to the simulator are added to the language and the existing capabilities of the language are extended to meet the new demands of the simulator. When the interface is based on an existing computer language, it is a good idea to keep any new functions as consistent with the existing ones as possible. If an existing function is extended, consistent operation must also be maintained.

The APLSIM simulator adds only one new function to APL and extends one other one. It also adds a number of 'system commands' which are not really part of the language but are needed to perform some of the less common operations. The small number of modifications to the APL language does not indicate that the interaction between the simulator and the APL interpreter is weak; quite the contrary is true. The control of the simulator is so embedded in the very heart of APL that most operations can be performed with the existing APL functions.

5.5.1. Assigning and Examining Node Potentials

The most fundamental user operations in simulation are assigning values to inputs and examining outputs. In APLSIM, this capability is provided by making all the nodes in the network being simulated into APL variables of the same name, at least from the user's point of view.[9] An input is set to a new value with the normal

[9] Actually, the values in the APL variables are copied in and out whenever they are referenced but this is transparent to the user and almost transparent to the interpreter.

APL assignment function, the left arrow. (←) Similarly, the value of a node is read by simply referencing the corresponding variable name. This allows node potentials to be compared, stored, manipulated, and displayed using the normal APL functions. When a new value is assigned to an input node, the simulator generates a piecewise linear segment to simulate the rise time an actual signal would have. If the value being assigned is either zero or one, the simulator substitutes predefined logic voltages. This makes it easy to assign logic values to nodes. If the user really wants to specify a one volt input, he can assign 1.00000001 instead.

The assignment function can actually do a lot more than just set a node voltage to a single value. In APL, it is possible to use a single assignment function to assign an entire vector or matrix. In APLSIM, assigning a vector (1-d array) to a node will cause the simulator to generate a series of piecewise linear segments, and to put them on the action queue as described in the previous chapter. The segments correspond to a stream of bits with a specified rise time applied at a specified clock frequency. More will be said about other variations on the assignment function later.

5.5.2. Controlling Simulation

We have seen how inputs and outputs can be manipulated by the use of APL variables. Many of the other simulator functions can be controlled in much the same way. For example, the APL variable TIME is the current simulator time. Reading it gives the simulated real time in nanoseconds. Assigning a new value to it causes the simulator to evaluate the network until it reaches the specified time. Other variables are used to control the convergence criteria of the simulator, and thus the accuracy/performance tradeoff. The values assigned to these variables are automatically copied to the proper 'C' variables so the simulator can use them.

5.5.3. The Characteristics Function

The one new function which was added to APL adds a characteristic to a data value so that it will be given special treatment by the assignment function. The idea of these characteristics is quite foreign to APL so this function is used only when assigning values to simulator related variables. The effects of the characteristics depends on what variable the data is assigned to. For example, if data with the FORCE flag set is assigned to a node variable, the node will immediately be forced to the specified value with zero rise time. If a value with the FORCE flag set is assigned to TIME, the simulator time will be set without running the simulation. The APLSIM

User's Guide in appendix B describes the characteristics function in detail.

5.5.4. Bus Variables

If a variable is designated as a *bus variable* using a system command as described below, it will become a vector whose elements are synonyms for other variables, usually simulator nodes. Using this mechanism, a set of related nodes can be treated as a single entity. For example, if we are dealing with an output bus, APL base conversion functions can be used to display the decimal equivalent of a set of bits on a bus. Assigning a vector or matrix to a bus variable is like assigning a single value or a vector to each of the individual nodes comprising the bus. Once again, APL functions can be used to generate the vector or matrix from user input. For example, a string of decimal numbers could be converted into a matrix of ones and zeros which could then be assigned to a bus variable.

5.5.5. System Commands

The APL language has a number of so-called system commands which are used to read files, edit functions, log off, and so on. They are usually infrequently used commands or commands whose syntax does not fit well with the rest of APL. They consist of a single right) followed by a command, followed by one or more arguments. Here the commands are reasonable English instead of special symbols. APLSIM adds more system commands for reading in networks, setting up bus variables, and so on. System commands cannot occur in APL functions so they are usually typed interactively on the keyboard. As an APLSIM extension, they can also be read from a command file.

5.6. Mixed Mode Simulation

A mechanism has been implemented in APLSIM which allows a form of mixed mode simulation. It is not true mixed mode simulation since APLSIM actually has only one evaluation technique: iterated timing analysis. APLSIM allows the user to assign character strings with the EXEC characteristic set to simulator nodes. The character strings must contain valid APL commands, usually ones which execute user-defined functions. A string will be executed as an APL command whenever the node it is assigned to changes from one logic state to the other. APL functions can be written which replace portions of the circuit or represent the world outside the chip. For example, a 'RAM' function could be written which examines certain network nodes (the RAM address lines and control signals) and then assigns data to a bus variable (the RAM data lines) from an APL array representing the RAM (read operation) or puts information from the bus variable into the array (write operation).

This mechanism is currently still quite primitive and will be expanded to allow other triggers to execute APL strings. Abilities which are needed include delayed execution through use of the action queue and more flexible capabilities for triggering on node potential changes.

5.7. Workspaces

In APL, the user works in a *workspace* which contains all his variables and functions. This workspace can be saved and later recalled. With the addition of the simulator, the network and all related information is added to the workspace. The user can thus run the simulator to get the network into a certain state and then save the workspace so that he can retrieve it later. This means that a long network initialization sequence need only be performed once and saved.[10]

5.8. Conclusions about the User Interface

There is a lot more to the user interface than appears in this chapter. The interrested reader is referred to appendix B, which describes how to use APLSIM, and appendix C which gives a short tutorial on the features of APL necessary for basic simulator operation.

By embedding the commands for controlling the simulator in the APL programming language, the APLSIM simulator has been made more versatile and more powerful. The user has all the features of APL at his disposal at all times while simulating. The use of a programming language as the user interface has also provided a powerful and consistent mechanism for implementing a form of mixed mode simulation.

[10] Note: this capability is not currently operational

Chapter 6
Performance Evaluation and Future Work

6.1. Simulator Performance

Since APLSIM is still a new piece of software, its performance and accuracy have not yet been fully evaluated. The defacto standard reference for simulators in the same class as APLSIM has traditionally been SPICE. Generally, SPICE is considered to be about as accurate as is necessary so a simulator is considered to perform well if it gives results similar to SPICE but does so in a much shorter time.

6.1.1. Problems in Comparisons with SPICE

It is well known that computer benchmarks can be quite treacherous. It is possible to make any computer hardware or software look good with a judicious choice of test cases; the same is true of simulators. In a comparison of APLSIM and SPICE, the first concern is that the transistor models used be similar, so the level 1 SPICE MOSFET model and the level 0 APLSIM model were used. The SPICE model takes non-linear capacitance and substrate diodes into account: APLSIM ignores both. The second concern was that the simulators should give results of approximately the same accuracy. It is very difficult to do this through the examination of internal simulator time steps as the numerical methods used have significantly different properties. Instead, plots of the results of the two programs were compared and found to be sufficiently similar. The concerns about models and accuracy are, of course, intended to avoid overrating APLSIM. There is no doubt that SPICE is more accurate, but it is questionable whether the accuracy is needed in digital simulation.

APLSIM has been used to simulate many circuits as part of the development and debugging process but a three bit adder was chosen as the basis of performance comparisons with SPICE. It is suspected that the circuit underrates APLSIM because an uncharacteristicly high[11] percentage of the network nodes were active in this test and

[11] In a typical digital circuit, about 20% of the nodes are changing at any one time.

the circuit is full of feedback paths. The selective trace and equation ordering techniques APLSIM employs are thus rendered considerably less effective than they would be for a larger, less tightly interconnected circuit.

The following sequence of numbers was added: (0, 0), (0, 1), (0, 3), (0, 7), (1, 7), (3, 7), (7, 7). The simulators gave approximately the same result, as shown below. According to this benchmark, APLSIM is between 50 and 70 times as fast as SPICE. This translates to a run time of about 45 seconds for APLSIM as opposed to over one-half hour for SPICE. It is suspected that the speedup factor would be even greater for a larger, more realistic circuit. Further benchmarks will be performed in the future.

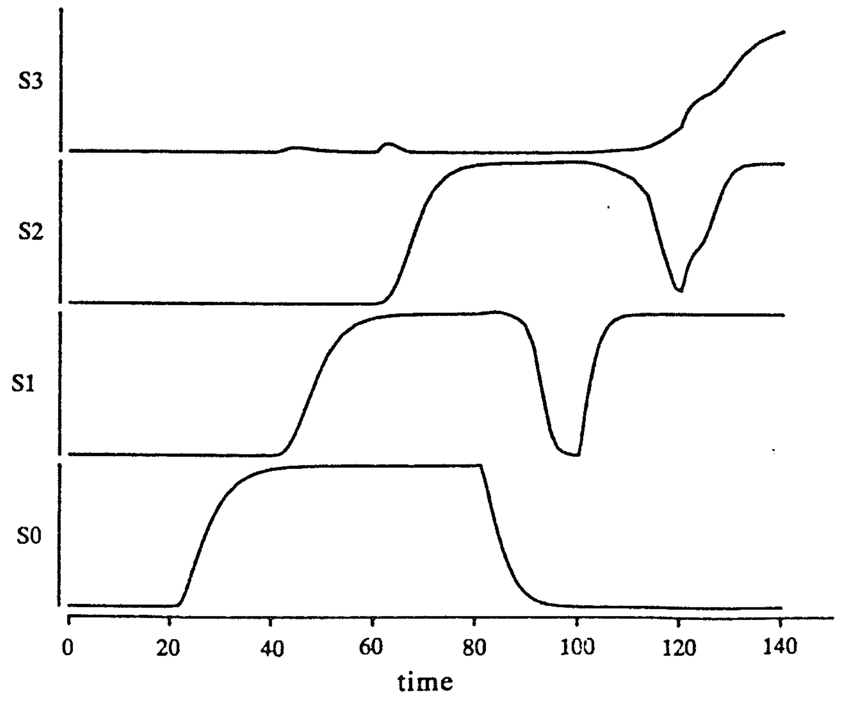

Figure 10. *SPICE simulation of a three bit adder (30 minutes CPU time)*

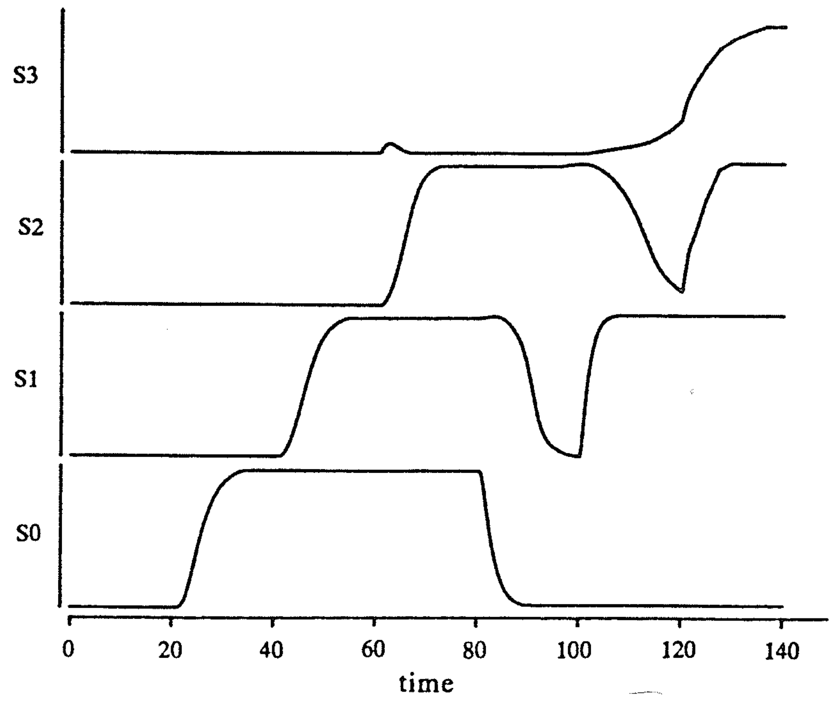

Figure 11. *APLSIM simulation of a three bit adder (¾ minutes CPU time)*

6.2. Future Work

The APLSIM simulator is the type of program which can never be pronounced *complete.* This is not to say that it is not useful in its present form but there are always additions and refinements which make the program better. In the process of creating and testing APLSIM and writing its user manual and this report, a number of possible enhancements to the program came to light. These and other improvements to the program will be implemented in the near future.

6.2.1. Static and Dynamic Equation Ordering

As was mentioned previously, the selective trace algorithm implemented in APLSIM automatically orders the nodes for efficient evaluation by the Gauss-Seidel-Newton method. However, the order can get jumbled on successive time steps without input changes. The program will be modified to preserve the evaluation order. Also, a further optimization may be possible by ordering the network while it is being read in. If a signal path branches, this ordering will determine which of the branches will be evaluated first. It is hard to say how much effect these changes will have on simulator performance.

6.2.2. Time Step Control

As was mentioned previously, the time step control mechanism does not currently work as well as it should. A better method of determining the next time step must be found. Also needed is a method of dynamically partitioning the circuit so that different time steps can be used in different blocks of the circuit. In some cases this could speed up simulation substantially.

6.2.3. Breakpoints and APL Function Execution

Chapter 5 described a mechanism by which APL functions can replace portions of a network and simulate their operation algorithmically. This feature must be enhanced to provide a more sophisticated method of triggering the execution of functions than is currently available. The user should be able to define a general logical condition which triggers execution, making the facility much more flexible. The ability to use an APL function to stop the simulator (breakpoint) would help in debugging a circuit because simulation could be stopped the instant a node reaches a specified value, making it easier to trace problems. The breakpoint feature can be implemented as another APL variable which will cause simulation to stop when it is set to a non-zero value.

6.2.4. Hierachal Node Identification

Nodes are identified by a name or number in the current simulator. They should actually be identified in a hierarchal way. For example, if a node is the output of a NAND gate in a particular J-K flip-flop, it should be identified as such. Tracing through the network should also be done in a hierarchal way. This would make it easier to examine the network being simulated. The problem is that simulation is an inherently *flat* process and circuit extraction programs do not extract hierarchy information.

6.2.5. Graphing

Instead of graphing outputs by writing node potentials into a file and running a plotting program, the user should be able to assign a node to a special variable and see its history graphed immediately. For large circuits it will be necessary to specify the nodes which are of interrest ahead of time so that the simulator does not have to retain information for nodes which are never going to be graphed. This form of graphing will add to the interactive nature of the program.

6.2.6. Improvements to APL

As mentioned previously, APL provides only primitive control structures. It should be relatively easy to incorporate modern if-then-else and while-do constructs into the interpreter. While these will not be strictly consistent with the rest of APL, they should make it significantly easier to write APL programs which model an unimplemented portion of a network or perform simulator control functions like breakpointing.

6.2.7. Mixed Mode Simulation

In a true mixed mode simulator, it is possible to simulate different portions of the network using different techniques. For instance, on part could be simulated with a functional level simulator, another with a switch level simulator, and still another with a circuit simulator. The simulators all communicate with each other so signals passing from one block to another are handled properly. Mixed mode simulators are typically quite difficult to write so this is a long-range goal.

6.2.8. Table Lookup Techniques

Since a large portion of the simulator CPU time is spent evaluating the device models, table lookup techniques could be used to improve performance by about a factor of two. Because of the way the device models are implemented in APLSIM, it will be possible to use the less accurate but faster table lookup models in one portion of a circuit while simulating another portion with the normal models. The potential for instability in the ITA numerical method due to the finite resolution of the lookup tables must be taken into account when table lookup is implemented.

Chapter 7
Conclusions

Experience with gate, switch, timing, and circuit level simulators has demonstrated a need for a timing verification tool which can provide accuracy approaching that of a circuit simulator while retaining the speed of a timing simulator. It was found that the one-pass techniques utilized by switch and timing simulators were not capable of providing the desired accuracy. Circuit simulators, on the other hand, use matrix methods to solve what are usually very sparse systems, resulting in unnecessarily slow operation.

A search of relevant literature revealed a method known as iterated timing analysis. (ITA) ITA allows efficient exploitation of the latency and sparse connectivity of digital MOS networks. Because ITA handles latency, simulator performance depends only on the number of nodes which are changing at any one time: it is independent of the size of the network. This allows the simulation of large integrated circuit designs very much more quickly than is possible with a circuit simulator like SPICE and more accurately than is possible with a timing simulator.

It was found that because of the saturated operation of digital logic circuits, certain compromises in simulation accuracy could be tolerated, resulting in faster simulation. For instance, leakage currents and the non-linearity of MOS capacitances could be ignored in most cases without seriously sacrificing the accuracy of the simulation results.

It was decided that a good user interface is a vital part of a simulator. A set of required features was determined:

1) The user interface should be interactive, making the simulator easier to use and more flexible than a batch program like SPICE.

2) The simulator commands should be embedded in a programming language. APL was chosen because of its powerful vector and matrix manipulation capabilities.

3) The simulator commands and programming language should be coupled in such a way that that programs in the language can control the simulator enough to allow a primitive form of mixed mode simulation.

Experience with the current version of the APLSIM simulator has shown that the combination of a fast simulator and an APL-based user interface has created a useful simulation tool for digital integrated circuit design, filling the gap between existing timing and circuit level simulators. Planned improvements to the numerical method and the user interface, including the addition of a true mixed mode simulation capability, will allow APLSIM to replace all the other simulators we currently use for most of our design work.

Appendix A: Transistor Models

The transistor models used in APLSIM are derived from the SPICE model 1 mosfet equations. APLSIM level 0 corresponds to SPICE level 1 and APLSIM levels 1 and 2 are simplifications of level 0. There is little use in having more accurate models because of the use of linear capacitor models. While the simulation algorithum used in APLSIM does not preclude better models for both transistors and capacitors, the simulation would be slowed down considerably which defeats the primary purpose of this simulator.

The following is the level 0 model. In level 1, λ is assumed to be zero. In level 2, both λ and γ are assumed to be zero.

forward region:

$V_{ds} > 0$:

$V_{te} = V_{t_0} + \gamma(\sqrt{\Phi - V_{bs}} - \sqrt{\Phi})$

$$I_{ds} = \begin{cases} 0 & V_{gs} < V_{te} \\ \beta(V_{gs}-V_{te})^2\,(1+\lambda V_{ds}) & V_{gs} < V_{ds}+V_{te} \\ \beta V_{ds}\left\{2(V_{gs}-V_{te})-V_{ds}\right\}(1+\lambda V_{ds}) & V_{gs} > V_{ds}+V_{te} \end{cases}$$

$$\frac{dI_{ds}}{dV_d} = \begin{cases} 0 & V_{gs} < V_{te} \\ \beta\lambda(V_{gs}-V_{te})^2 & V_{gs} < V_{ds}+V_{te} \\ \beta\Big[\left\{2(V_{gs}-V_{te})-V_{ds}\right\}(1+\lambda V_{ds}) + V_{ds}\Big\{\lambda\left\{2(V_{gs}-V_{te})-V_{ds}\right\} \\ \qquad - (1+\lambda V_{ds})\Big\}\Big] & V_{gs} > V_{ds}+V_{te} \end{cases}$$

reverse region:

$V_{ds} < 0$

$V_{te} = V_{t_0} + \gamma(\sqrt{\Phi - V_{bd}} - \sqrt{\Phi})$

$$\frac{dV_{te}}{dV_d} = \frac{\gamma}{2\sqrt{\Phi - V_{bd}}}$$

$$I_{ds} = \begin{cases} 0 & V_{gd} < V_{te} \\ -\beta(V_{gd} - V_{te})^2\,(1-\lambda V_{ds}) & V_{gd} < -V_{ds} + V_{te} \\ \beta V_{ds}\left\{2(V_{gd} - V_{te}) + V_{ds}\right\}(1-\lambda V_{ds}) & V_{gd} > -V_{ds} + V_{te} \end{cases}$$

$$\frac{dI_{ds}}{dV_d} = \begin{cases} 0 & V_{gd} < V_{te} \\ \beta\left[2(V_{gd} - V_{te})\left(1 + \frac{dV_{te}}{dV_d}\right)(1-\lambda V_{ds}) + \lambda(V_{gd} - V_{te})^2\right] & V_{gd} < -V_{ds} + V_{te} \\ \beta\left[\left\{2(V_{gd} - V_{te}) + V_{ds}\right\}(1-\lambda V_{ds}) - V_{ds}\left(1 + 2\frac{dV_{te}}{dV_d}\right)(1-\lambda V_{ds}) \right. & V_{gd} > -V_{ds} + V_{te} \\ \qquad \left. - \lambda V_{ds}\left\{2(V_{gd} - V_{te}) + V_{ds}\right\}\right] & \end{cases}$$

Appendix B
APLSIM User's Guide

APLSIM
User's Guide

Roland Schneider
Department of Electrical Engineering
University of Manitoba
Winnipeg, Manitoba
Canada

1. Introduction

APLSIM is a circuit simulator designed to provide reasonable waveform accuracy combined with acceptable performance for large digital circuits. While APLSIM does not have the accuracy of SPICE, it runs 50 to 70 times faster, allowing it to simulate large circuits where SPICE is useless. Unlike SPICE, it is an interactive program so the user can set inputs and look at the results as they are produced.

The user interface for APLSIM is based on an APL interpreter which was obtained from the University of California. The APL language has been extended to allow control of the simulator but all the normal features of APL are still available. The use of APL as the interface requires the user to have some limited knowledge of the APL language to run the simulator. It also requires a terminal capable of displaying the APL character set. The accompanying document "An APL Primer" discusses some of the simpler features of APL necessary to use the simulator. The rest of this document assumes the reader is somewhat familiar with APL.

2. Why APL?

Anyone even remotely familiar with APL will realize that it is not in any way a "user friendly" language. The question arises, why was APL chosen as the interface for this simulator? There are several reasons:

1) The user interface for a simulator can be written by creating a large set of specialized commands to control the simulator. Invariably, however, the user will want to perform some operation which the program author has not thought of. Including a general purpose language as part of the simulator makes it possible for the user to write a small program to perform virtually any operation.

2) If the simulator includes a language, there are two approaches: There can be two 'modes' of operation, one with normal simulator commands and one using the language, or the simulator can be controlled exclusively through the language. The latter approach was chosen for APLSIM because it is more self-consistent. Simple operations like setting and reading node potentials are still simple so basic simulator operation is not made more difficult.

3) The 'C' source code for an APL interpreter was available. The interpreter consists of about 6000 lines of code and although it had to be adapted to run on the Sun workstation and a number of bugs had to be fixed, using an existing program saved a lot of time and allowed a significantly more sophisticated interface to be used than would have otherwise been possible.

4) APL can deal with vectors and matrices in a unified way and provides powerful matrix manipulation functions. Since a sequence of inputs to a circuit can be considered vectors and a set of vectors can be considered a matrix, the ability to handle these structures was deemed to be important.

5) APL makes a good desk calculator. In immediate mode (no function being executed) expressions can be evaluated and the results printed merely by typing them in. Many results can be evaluated simultaneously by using vectors and matrices. The user therefore has a desk calculator built into the simulator.

3. Notation

The following conventions are used throughout this document:

italics	indicate the item is to be replaced with the proper value
bold	indicates the item must be typed exactly as it appears
{brackets}	indicate optional items
UPPERCASE	indicates APL variables or functions

Some simulator inputs and outputs are automatically scaled to convenient values:

time values are in nanoseconds	(1×10^{-9}s)
capacitor values are in picofarads	(1×10^{-12}F)

4. Interface to APL

The user talks to the simulator almost exclusively through APL. Several functions have been added to APL to control the simulator. Most conform to normal APL syntax and can be used on normal APL data.

The basic interface between APL and the simulator is achieved by making the names of network nodes into APL variables. Reading the value of a node variable gives the potential of the node at the current time. Assigning a value to a node variable sets the potential of the node in most cases. There are also a number of other variables, some of which can be both read and set and others which can only be read. Any variable name not used by the simulator can be used as a general APL variable.

The effect of assigning values to nodes and certain other APL variables depends on the characteristics of the value being assigned. The characteristics of a value are set by the diamond extension function. The sequence *flag⋄expression* where *flag* is one of the predefined APLSIM flags and *expression* is any valid APL expression sets the characteristics of the result of the expression according to *flag*. Most APL functions discard the flag value so this function should be used immediately to the right of the operator assigning the value to a node.

Assigning to APLSIM Variables			
APL variable	**flag**	**data**	**result**
node	none	vector	A piecewise linear sequence is created for the specified node. The rise and fall times depend on the variable RISE. The rate of application of the entries depends on the variable CLOCK. If the vector contains only 1's and 0's, the values LOGIC1 and LOGIC0 are substituted. If the data is a scalar quantity, it will be treated as a one-element vector and the piecewise linear input will start at the current time.
	DELAY	vector	The result is the same as when no flag is specified except that the first element of the vector is the time in nanoseconds to delay before the vector starts.
	FORCE	scalar	The specified value is assigned to the node immediately with zero rise time. Note that the logic conversion described above still applies.
	FLOAT	scalar	The same as FORCE except that the node is not flagged as an input so it can change during simulation. This is useful for initializing a node which must be allowed to change or releasing a node which was previously an input.
	CAP	scalar	The capacitance of the node is set instead of its potential. The value is in picofarads.
	EXEC	string	The string is executed as an APL command whenever the node voltage changes logical value as determined by the variables THRESH0 and THRESH1.
TIME	none	scalar	The simulator is run to the specified time. Note that time cannot be run backwards.
	SETTLE	scalar	The simulator runs until the network has settled. The value specified is irrelevant.
	FORCE	scalar	The current time is set to the specified value without running the simulator. UNTESTED
LOGIC0	none	scalar	The logic zero input voltage is set.
LOGIC1	none	scalar	The logic one input voltage is set.
THRESH0	none	scalar	The logic zero threshold is set.
THRESH1	none	scalar	The logic one threshold is set.
CLOCK	none	scalar	The clock period is set.
RISE	none	scalar	The rise/fall time of inputs is set.
LTE	none	scalar	The local truncation error is set.
MINDT	none	scalar	The minimum time step is set.
MAXDT	none	scalar	The maximum time step is set.
MAXPWL	none	scalar	The maximum time step during piecewise linear is set.
DCLIM	none	scalar	The value of C/dt below which DC analysis is used is set.
CONVERGE	none	scalar	The convergence delta is set.
UNQUEUE	none	scalar	The minimum change for continued processing is set.

5. Getting Started

5.1. Running the Simulator

On the Sun, the simulator is executed by typing **aplsim**. A red-bordered window will appear somewhere on the screen with the word APLSIM on the header line. Use the mouse to move the arrow cursor into the window. A different character set is used inside the window. What would normally be lowercase characters come out as SMALL UPPERCASE while the uppercase characters come out as special symbols. The new keyboard layout is shown at the end of the accompanying document "An APL Primer".

5.2. Loading the Models

Before the network description can be loaded, the simulator must know the names and parameters of the transistor models to use. These, along with other initialization commands, are normally in a file named **setup**. Use the command)FILE SETUP to load them.

5.3. Getting Off

To end a session with the simulator, type)OFF. If desired, the)SAVE *filename* command can be used to save the current status of the simulator and all the functions and variables on disk for later retrieval by the)LOAD *filename* command. If the command)CONTINUE is used to terminate a session, the current status is automatically saved and will be automatically reloaded when the simulator is run the next time. **NOTE: saving does not currently work.**

5.4. Generating a Netlist

The first step in simulating a circuit is to describe the network to the simulator. The description of the network should be placed in a file although it is possible to create it interactively from inside the simulator. Each line of the file contains a description of a transistor, a resistor, or a capacitor. The descriptions take the following form:

Element	Input Line
transistor	**trans** *model gate source drain {length width}*
resistor	**resist** *term1 term2 value*
capacitor	**cap** *node value*

The program **cvtsim** converts a SPICE input deck into an APLSIM netlist description. It is designed specifically to process the output of the **phlex** circuit extractor on the Metheus workstation. Note that APLSIM does not currently allow floating capacitors; all capacitors must have one terminal connected to ground.

Any node names which consist only of numbers are automatically renamed to the letter *N* followed by the original number. This is done so that they can be referenced as normal APL variables. Node names may have no special characters in them. The node names must be in lowercase although in the APL character set they will appear as SMALL UPPERCASE.

6. Running the Simulation

6.1. Specifying Inputs

Inputs are specified by assigning a value to the APL variable corresponding to the network node. All nodes are considered floating when the simulator starts so it is important to assign a value to all the input nodes before the network is initialized. This is especially true if the LOGIC0 voltage is not 0.00V or if an input is the drain or source of a transistor or the terminal of a resistor. Inputs are considered perfect voltage sources and will hold the node to the specified voltage no matter what the current is.

When a value is assigned to a node with the standard APL assignment INPUT← VALUE, the input will have a LOGIC0 to LOGIC1 rise (or fall) time of RISE nanoseconds. If VALUE is a vector, the elements will be fed to the input with a period determined by CLOCK. If the DELAY flag is specified, the first element of the vector will be taken to be the delay before the vector is applied. Note that assigning a vector generates a series of piecewise linear entries for the specified node and that this is done immediately when the assignment statement is executed. Changing RISE or CLOCK after the assignment will have no effect.

If all the values in a vector, with the exception of the delay, are 1's and 0's, the values of LOGIC1 and LOGIC0 are used instead. This makes it easy to set logical values. To actually enter a zero or one volts, use something like 1.000001.

The FORCE flag will immediately force the input to the specified value. The FLOAT flag will immediately set the input to the specified value but will not mark the node as an input so that it can change during simulation. The FLOAT flag is also used to release a node which was previously an input. For example, to float node NODE without changing its value, type NODE← FLOAT⍎NODE. Note that this will also cancel any pending piecewise linear input for NODE. The conversion of 0's and 1's into LOGIC0 and LOGIC1 applies for FORCE and FLOAT as well.

The last way to set a node is with the simulator escape SET. The command)SIM 'SET *node value*' will immediately set the node voltage to the floating point constant *value*. No conversion of 1's or 0's takes place. The command is provided primarily to allow initialization of certain nodes such as VDD and GND in the network file read by the)NET command. The **cvtsim** program sets VDD to 5.0 volts and GND to 0.0 volts.

6.2. Advancing Time

The simulator is run by assigning a new value to the TIME variable. The simulator will run until the specified time is reached. The construct TIME← TIME+*value* will advance the time by *value*. Note that the command could be put into an APL function called, say, GO.

```
GO DELTA
TIME← TIME+DELTA
```

and the user would only have to type GO *value* to simulate for *value* nanoseconds.

As an alternative, especially useful for initialization, the SETTLE flag can be used to run the simulation until the network is no longer changing significantly. The command TIME← SETTLE⍎0 will do this. The 0 is irrelevant and could be any value. In the future, a non-zero value will indicate a maximum time.

7. Simulator System Commands

APL system commands are those which start with a) followed by the command. These commands cannot occur in functions, although they can be read with the)FILE command.

)FILE The argument is the filename to be read. The file can contain any valid APL command, including other system commands. The files do not currently nest, so another)FILE command can only appear at the end of a file.

)SIM The argument is a string (enclosed in single quotes: uppercase K) which is a command for processing directly by the simulator. The commands are ones which do not map well into the APL syntax or are infrequently used. The first word in quotes is a keyword which determines what the command is. The rest of the string serves as an argument to the command. The following escape commands are currently supported:

GRAPH The keyword is followed by a filename and a series of node names. The values of the specified nodes will be written to the file at each time step so that they can be plotted by the **gplot** program.

TRANS The keyword is followed by a model name, the gate node, source node, drain node, and optionally the channel length and width in microns. A transistor with the specified characteristics is inserted into the network. If any of the terminal nodes do not exist, they will be created. The gate capacitance specified in the model, multiplied by the gate area, will be added to the capacitance of the gate node.

CAP The keyword is followed by a node and a capacitance value in picofarads. The capacitance of the specified node will be set. When a transistor is added to the network, the gate capacitance is added to the capacitance of the node connected to the gate. It is therefore important to ensure that the node capacitances are specified before the transistors when the network is input. Otherwise the node capacitances will override the effect of the gates. This assumes, of course, that the calculated node capacitances do not include the effect of the gates.

MODEL The keyword is followed by a series of parameter names and their corresponding values, all separated by spaces. Some of the parameters are mandatory, others are optional and have a default value. The following parameters are currently defined:

Model Parameters

name	default	function
NAME	---	Set the model name. The model is later referred to by this name.
LEVEL	1	Set the model level. 0, 1, and 2 are currently defined. 0 is the most accurate, 2 is the fastest.
POLARITY	---	The polarity is either N or P indicating an N or P channel transistor.
BETA	2×10^{-5}	The beta of the transistor is set. This may or may not be the same as the beta in SPICE because SPICE calculates an effective channel length while APLSIM uses the channel dimensions specified.

Model Parameters		
name	default	function
VTO	1.0	The transistor threshold voltage is set. Note that the threshold voltage is always considered positive, for both P and N channel devices.
VB	0.0 or 5.0	The substrate bias voltage for the transistor is set. Strictly speaking, the substrate bias should be found from a node voltage, but tying the substrate to anything but V_{DD} or *GND* or some other fixed supply voltage is unusual in a digital circuit. The default is 0.00V for an N channel transistor and 5.00V for a P channel transistor.
GAMMA	1.0	This is the same as the gamma in SPICE. It controls the effect of V_{BS} on the threshold voltage. This only applies for model levels 0 and 1.
PHI	0.6	This is the same as the phi in SPICE. It is the offset for the effect of V_{BS} on the threshold voltage. This only applies for model levels 0 and 1.
LAMBDA	0.01	This is the same as the lambda in SPICE. It controls the additional effect of V_{DS} on I_{DS}. This only applies for model level 0.
CAP	400×10^{-18}	The gate capacitance per square micron of gate area is set. When a transistor is added to the network, the gate capacitance is added to the capacitance of the node connected to the gate.

QUEUE The names of all nodes on the evaluation queue are printed. This can be used to determine which nodes have not yet settled at any time during simulation. The SETTLE flag on assignments to TIME will cause simulation to proceed until the evaluation queue is empty.

INIT This command places all the nodes in the network on the evaluation queue. It is a good idea to issue the INIT command after the network has been loaded to ensure that all nodes will be processed and be consistent with the rest of the network. The **cvtsim** program automatically puts an INIT command at the end of the network description.

SET The keyword is followed by a node name and a value. The potential of the node is set to the value specified. This is similar to a node assignment with the FORCE flag except that no conversion of 1's or 0's is performed. The **cvtsim** program automatically SETs node VDD to 5.00V and node GND to 0.00V.

BUS The keyword is followed by an APL variable name and a list of other APL variables or simulator nodes. The effect is that the first APL variable name becomes a vector whose elements are bound to the specified variables and nodes. The elements will change when the nodes or variables change and assigning a vector to the bus variable will set all the variables or nodes. The only restriction is that it is not currently possible to assign to a subscript of the bus variable. This command is intended to create a "bus" consisting of several inputs or outputs and to be able to treat them as one object. APL functions can be used to generate inputs for the bus or to convert results from a bus into decimal, hexadecimal, etc. Assigning a matrix to a bus is like assigning vectors to all its elements. The first dimension of the matrix must be the same as the shape (as in ρ) of the bus.

)NET The argument is the name of a file containing only simulator escape commands. Typically these are network specification commands like TRANS and CAP but can be any other valid escape command as detailed above. Only the part of the command which is normally in quotes is put in this file, the)SIM part is left off.

)NODES This is much like the APL)VARS command except that it displays only variables which are simulator nodes. The)VARS command displays only real APL variables.

)FLAGS This command displays the simulator flags like FORCE, SETTLE, etc. It is not possible to set the value of simulator flags through APL.

)CONSTS The names of the simulator constants are displayed. They include TIME, LTE, and so on. These are all settable by the user but the effect of assignment depends on the variable and the flags set on the data.

)WSSIZE The maximum amount of memory the program has required since startup is printed.)ERASEing variables will free space but will not change the workspace size.

)CPU The cpu time used since the last)CPU command is printed.

)TRACE The keyword is followed by the name of a node. The network is traced forwards (gates controlled) and backwards (source and drain connections) from the specified node. This allows the user to walk through the network to find the source of erroneous results. The output has one line for each transistor as follows:

```
CALCULATED FROM:
*model1   gate1   source1   node     ratio1   GATE = potential1
 model2   gate2   node      drain2   ratio2   GATE = potential2
   .        .       .         .        .        .
   .        .       .         .        .        .
   .        .       .         .        .        .
CONTROLS:
*model3   node    source3   drain3   ratio3
 model4   node    source4   drain4   ratio4
   .        .       .         .        .
   .        .       .         .        .
   .        .       .         .        .
```

An astrisk in the first column indicates that the transistor is on, or at least not cut off. The name of the model is the same as was specified when the network was created. The gate, source and drain node names are printed. The next column is the width to length ratio of the transistor. On the backward trace, the gate potential is also printed.

8. Simulator Constants

The APL variables described previously which control the operation of the simulator are referred to as the simulator constants. Their names can be displayed with the)CONSTS command. A more detailed description of each of the variables follows.

TIME	Assigning a value to TIME will cause the simulator to run up to the specified time. TIME is always in nanoseconds. Reading the value of TIME gives the current simulator time whether the simulator is running or not. If the SETTLE flag is specified in a value assigned to TIME, the simulator will run until no node changes significantly from one time step to the next. This is especially useful during initialization when the network is supposed to settle completely before simulation is started. TIME can then be assigned a nice even number so that it is easy to determine when simulation started if the outputs are being graphed. It is also possible to assign a value to TIME without running the simulation. The FORCE flag allows TIME to be set to any value. (UNTESTED)
LOGIC0 LOGIC1	These variables contain the voltages to be used as logic 0 and 1 inputs. If a vector containing only 0's and 1's is assigned to a node, the values in these variables will be substituted. This makes it easy to assign logical values. This feature is provided because it is rare to assign intermediate voltages in digital circuits.
THRESH0 THRESH1	The variables contain the logic 0 and 1 threshold voltages. If an APL string is assigned to a node with the EXEC flag, the string will be executed whenever the node changes logical value. If the node is a logic 0 and its potential rises above THRESH1 or the node is logic 1 and its potential falls below THRESH0, the string will be executed.
CLOCK	The CLOCK variable determines the rate at which vector inputs to nodes will be applied. It is the period (in nanoseconds) between the beginnings of the ramps for logic changes in the vector. The vector is converted to a piecewise linear input stream for the node as soon as it is assigned. Therefore the value of CLOCK only applies before the assignment; changing it later has no effect.
RISE	The RISE variable determines the rise and fall times of input vectors. The value is in nanoseconds. As with CLOCK, the value only applies at the time of assignment, not afterwards.
LTE	The local truncation error is used to calculate the time step for the simulator. Values between 1×10^{-10} and 1×10^{-12} give reasonable results. The larger the value of LTE, the larger the time step. Large time steps result in lower accuracy but faster simulation.
MINDT MAXDT MAXPWL	These are the minimum time step, the maximum time step and the maximum time step during execution of a piecewise linear segment. These variables all default to reasonable values at startup.

9. Mixed Mode Simulation

This simulator has only one numerical method so it cannot really be called a mixed mode simulator. It does, however, have the ability to trigger the execution of APL expressions when certain events occur. The APL function executed can set new inputs for nodes, etc. This makes it possible, for example, to write an APL function which simulates the operation of a RAM. The circuit being simulated generates outputs to the RAM and expects to get inputs back, for example, data whose address has been specified. Using an APL function to replace the large amount of circuitry involved in a RAM speeds up simulation at the cost of accuracy.

Another application of this execution trigger feature is to replace parts of the circuit which have not yet been designed with an APL function which implements the algorithm of the undesigned circuit. Similarly, if an entire chip has been designed, the off-chip world can be replaced by APL functions.

10. Some Useful APL Functions

Here are a few useful APL functions which aid in simulation:

This function converts the input vector POT to a vector of 1's and 0's. This is useful for converting the output of nodes to logical values for use by other APL functions.

```
LOGIC POT
POT≥2.5
```

This function generates a BITS long vector of 1's and 0's from the decimal number NUM. If NUM is a vector, a matrix will be generated. The matrix is suitable for assignment to a bus variable of shape BITS.

```
BITS BIN NUM
(BITSρ2)⊤NUM
```

This function converts the vector of 1's and 0's VEC into a decimal number.

```
DEC VEC
((ρVEC)ρ2)⊥VEC
```

This function runs the simulator for DELTA nanoseconds.

```
GO DELTA
TIME← TIME+DELTA
```

These functions can be combined. For instance, the LOGIC and DEC functions can be called like this:

```
DEC LOGIC bus
```

where *bus* is a bus variable. This will give the decimal equivalent of the binary number on the bus. Of course, this statement itself could be put into a function.

11. A Sample Session

Here is an example of a series of commands which could be used to simulate the operation of a single bit full adder. The inputs are AIN, BIN, and CIN. The outputs are SOUT and COUT. The object of the run is to cycle the adder as fast as it can go and graph the outputs. To achieve this, a function, SGO has been written. It is executed every time there is a logic change on SOUT. The input sequence is a gray code. Comments are shown in italics to the right of the statements.

The following function is used:

```
LEN SGO CODE                        The function is named SGO
→ (COUNT> ((2*LEN)-1)/END           The count is checked for end condition
BINARY← (LENρ2) ⊤ CODE[COUNT+1]     A binary number is generated
AIN← BINARY[1]                      The nodes are assigned from the binary number
BIN← BINARY[2]
CIN← BINARY[3]
END: COUNT← COUNT+1                 The count is incremented
```

Here is the sequence of commands:

```
)FILE SETUP                         The file setup is read. It contains the models
)NET ADD                            The network description in the file add is read
AIN← 0                              The A input is set to logic 0
BIN← 0                              The B input is set to logic 0
CIN← 0                              The C input is set to logic 0
TIME← 100                           The simulator is run to 100ns for initialization
)READ SGO                           The function in the file sgo is read
COUNT← 2                            The APL variable COUNT is set to 2
GRAY← 0 1 3 2 6 7 5 4               The gray code sequence is specified
SOUT← EXEC  '3 SGO GRAY'            The string to execute the function is assigned
)SIM 'GRAPH RESULT AIN BIN CIN SOUT COUT'
                                    The graphing file is set to result
TIME← 280                           The simulator is run to 280ns
)OFF                                The session is terminated
```

The command)SIM 'BUS INP AIN BIN CIN' could have been used to make INP a bus variable consisting of the three adder inputs. The three separate statements assigning values in the SGO function could then have been replaced by the single statement INP← BINARY.

12. Transistor Model Equations

The transistor models used in APLSIM are derived from the SPICE model 1 MOSFET equations. APLSIM level 0 corresponds to SPICE level 1 and APLSIM levels 1 and 2 are simplifications of level 0. There is little use in having more accurate models because of the use of linear capacitor models. While the simulation algorithum used in APLSIM does not preclude better models for both transistors and capacitors, the simulation would be slowed down considerably which defeats the primary purpose of this simulator.

The following is the level 0 model. In level 1, λ is assumed to be zero. In level 2, both λ and γ are assumed to be zero.

forward region:

$V_{ds} > 0$:

$V_{te} = V_{t_0} + \gamma(\sqrt{\Phi - V_{bs}} - \sqrt{\Phi})$

$$I_{ds} = \begin{cases} 0 & V_{gs} < V_{te} \\ \beta(V_{gs}-V_{te})^2\,(1+\lambda V_{ds}) & V_{gs} < V_{ds}+V_{te} \\ \beta V_{ds}\left\{2(V_{gs}-V_{te})-V_{ds}\right\}(1+\lambda V_{ds}) & V_{gs} > V_{ds}+V_{te} \end{cases}$$

$$\frac{dI_{ds}}{dV_d} = \begin{cases} 0 & V_{gs} < V_{te} \\ \beta\lambda(V_{gs}-V_{te})^2 & V_{gs} < V_{ds}+V_{te} \\ \beta\Big[\left\{2(V_{gs}-V_{te})-V_{ds}\right\}(1+\lambda V_{ds}) + V_{ds}\Big\{\lambda\left\{2(V_{gs}-V_{te})-V_{ds}\right\} - (1+\lambda V_{ds})\Big\}\Big] & V_{gs} > V_{ds}+V_{te} \end{cases}$$

reverse region:

$V_{ds} < 0$

$V_{te} = V_{t_0} + \gamma(\sqrt{\Phi - V_{bd}} - \sqrt{\Phi})$

$\dfrac{dV_{te}}{dV_d} = \dfrac{\gamma}{2\sqrt{\Phi - V_{bd}}}$

$$I_{ds} = \begin{cases} 0 & V_{gd} < V_{te} \\ -\beta(V_{gd} - V_{te})^2\,(1 - \lambda V_{ds}) & V_{gd} < -V_{ds} + V_{te} \\ \beta V_{ds}\left\{2(V_{gd} - V_{te}) + V_{ds}\right\}(1 - \lambda V_{ds}) & V_{gd} > -V_{ds} + V_{te} \end{cases}$$

$$\frac{dI_{ds}}{dV_d} = \begin{cases} 0 & V_{gd} < V_{te} \\ \beta\left[2(V_{gd} - V_{te})\left(1 + \dfrac{dV_{te}}{dV_d}\right)(1 - \lambda V_{ds}) + \lambda(V_{gd} - V_{te})^2\right] & V_{gd} < -V_{ds} + V_{te} \\ \beta\left[\left\{2(V_{gd} - V_{te}) + V_{ds}\right\}(1 - \lambda V_{ds}) - V_{ds}\left(1 + 2\dfrac{dV_{te}}{dV_d}\right)(1 - \lambda V_{ds}) \right. & V_{gd} > -V_{ds} + V_{te} \\ \qquad \left. - \lambda V_{ds}\left\{2(V_{gd} - V_{te}) + V_{ds}\right\}\right] & \end{cases}$$

13. The gplot Plotting Program

The **gplot** program is designed specifically to plot the output of the APLSIM simulator. It can also be used to plot SPICE output. There are two modes of display: timing and voltage. In timing mode, the waveforms are displayed one above the other. In voltage mode, the waveforms are superimposed but plotted in different colors. Voltage mode is the default; timing mode is selected with the **-t** option. The program will automatically determine whether its input is from APLSIM or from SPICE. For SPICE, only transient analysis can be plotted.

gplot {-t} {-p} *filename* {*node1 node2 node3* ...}

options:

-t	plot in timing mode
-p	indicate simulator time steps

If no node names are specified, all nodes in the file will be plotted. If names are specified, the nodes will be plotted in the specified order, starting at the bottom of the graph. (applies for timing mode only)

14. The cvtsim SPICE to APLSIM Conversion Program

The **cvtsim** program converts a SPICE input deck, specifically one produced by the **phlex** circuit extractor on the Metheus workstation, into a file suitable for reading with the)NET command. In UNIX, it is what is called a *filter* – it reads from standard input and writes to standard output. To run it, type:

cvtsim < *inputfile* > *outputfile*

In some cases the program will report syntax errors. These are usually caused by node names with underscores in them. These will be removed by the **phlex** circuit extractor, leaving a space. Use an editor to remove them.

Appendix C
An APL Primer

An APL Primer

Roland Schneider
Department of Electrical Engineering
University of Manitoba
Winnipeg, Manitoba
Canada

1. Introduction

APL is a mathematical programming language originally developed by IBM for mainframe computers. It's main feature is the ability to deal with matrices in a very efficient manner. It has a large set of built-in functions, most of which can deal with matrices of arbitrary dimension. APL is an interpreted language and is therefore not very fast. It is generally a good idea to avoid loops, and to use the matrix handling functions to perform the desired operation in parallel. Of course, the interpreter, written in 'C', still performs the operation for each element of the matrices, but it does this relatively quickly.

The APL interpreter described here came on a Berkeley UNIX distribution tape. It was converted to run under Sun 4.2 UNIX. At the same time, a number of bugs were fixed and some functions were enhanced to bring them closer to standard APL. The language is compatible with standard APL in most ways. There are a few functions missing, a few which are not as general as they should be, and a few with names different from those in the IBM manuals.

The APLSIM simulator talks to the user through the APL interpreter. This guide describes the basic APL functions, APL terminology, and a few of the more advanced commands which are useful for simulation. The user should be forewarned that APL cannot be considered a *user friendly* language in any sense of the word. It is powerful, but it uses many strange symbols and has a unique way of evaluating expressions which will frighten and frustrate any new user. However, once it is mastered, APL can be made to do some remarkable things with very short programs.

2. Terminology

APL has its own set of buzz-words for describing operations and the data they deal with. These should be mastered or this document will appear as formidable as the language itself.

function	In APL, a function is what would be called an operator in most other languages. The APL functions are mostly single characters; some are Greek letters, some are mathematical symbols, and some were made up by the inventors of the language. Most functions can take multi-dimensional matrices as their arguments although there are some restrictions. A user defined function is used in the same way a built-in function is, but it has a name instead of a symbol.
operator	An APL operator does something most other languages can't even dream of: it applies a function, as defined above, between selected elements of matrices. This is a very powerful capability but requires a fairly good understanding of APL to use.
nonadic	A function is called nonadic if it takes no arguments. There are no built-in nonadic functions; only user defined functions can fit into this category.
monadic	A function is called monadic if it takes one argument. The argument is on the right side of the function name. Remember that an APL function is what is normally considered an operator. Most other programming languages have a few monadic operators, for example, unary minus. APL has many more, some of which will be described later.
dyadic	A dyadic function has two operands: one on the left and one on the right. The functional operation is usually performed between the two. There are some built-in functions which can be either monadic or dyadic. The two versions are usually related in some way. Context determines whether the monadic or dyadic version will be used.
shape	The shape of an APL variable refers to its size in its various dimensions. For example, a 2×5×7 matrix has a shape of 2 5 7.

rank	The rank of an APL variable is the number of dimensions it has. In the example above, the rank is 3.
scalar	A scalar is a variable or constant which has rank 0. That is, it is a single number. Note that it is possible to have a 1×1×1 · · · matrix which also consists of only one number. This is not a scalar.
vector	Because they are used so frequently, rank one matrices are given the name vector. This is consistent with normal mathematical terminology. In APL, it is possible to specify vector constants.
matrix	The term matrix usually refers to variables with a rank greater than one.

3. Input Syntax and Order of Evaluation

The input syntax for APL is relatively straightforward, although the symbols are sometimes quite weird. APL is an uppercase only language, although the characters typed are actually lowercase. The shifted versions are the special APL characters. The keyboard is all messed up: the normal characters and numbers are where they should be but all the special characters have been rearranged. See the table at the end of this document for the APL keyboard layout.

Variables	All APL variables must consist only of letters and numbers as almost all the special symbols are reserved for functions. A variable name can be any length and should start with a letter. In this document, APL variable names will be printed as SMALL UPPERCASE letters.
Constants	Constants can be either scalar or vector. A scalar constant is entered as a single number. A vector constant is entered as a series of numbers separated by spaces. Character string constants are enclosed in quotes. Note that a character string is actually a vector of single characters.
Expressions	An expression is any combination of variables, constants, and operators. Expressions are evaluated *from right to left* with no operator precedence. Parentheses can be used to force a particular order of operation if needed. The result of an expression always falls out the left side. This is true even when the assignment operator is used. If the last (left hand) operation is not an assignment and there is no user-defined function executing, the result of an expression will be printed. This makes APL a powerful interactive calculator.
Functions	Most of the special characters on the keyboard are built-in APL functions. At last count there were about 30 or so of them. Built-in functions are either monadic or dyadic, as described previously. The operand of a monadic function always appears on the right of the function. The function appears between its operands if it is dyadic. The operands of most functions can be scalars, vectors, or n-rank matrices. The multiplication function × is an example of a dyadic function. The expression 3×4 evaluates to 12 as one would expect. The right to left evaluation order must be considered when typing an expression. For example, 3×6−4 gives 6 as a result, not 14 as it would in other languages. The expression is evaluated by subtracting 4 from 6 and then multiplying 3 and the result. At first glance this seems stupid but it would be impossible to assign a reasonable precedence to all 30 functions. The results of all operations 'fall out' the left side.

Operators APL operators apply dyadic functions between the elements of vectors and matrices. For example, the result of the expression

```
+/3 9 2 1 2
```

is 3+9+2+1+2 or 17. These operators can perform some surprisingly powerful operations. For instance, inner products are calculated with the '.' operator and the '×' and '+' functions.

User Functions

User defined functions are much like built-in functions except that they are identified by name instead of with a symbol. They can take zero, one, or two operands. In this version of APL, user defined functions are created by editing with a standard UNIX editor like vi. The first line of a function contains the function header: the function name, the dummy arguments, local variables, and indication of explicit function value assignment. The function is loaded when vi is terminated. To execute a user defined function, just type its name. As with built-in functions, the operands of a dyadic user-defined function are placed on either side of the function itself.

4. Basic Operations

4.1. Assignment

The most basic operation is assignment to a variable. The left pointing arrow does this. The statement A← 2 8 4 assigns the vector 2 8 4 to the variable A. Statements such as B← A are also valid. Just typing a variable name will print out its contents.

4.2. Mathematical Functions

We will start with a description of the basic mathematical dyadic functions. These are evaluated right to left with no precedence. The result of each operation is always available on the left. These functions can operate between any operands of the same shape. They all operate on an element by element basis so the shape of the result is the same as the shape of the operands. If one of the operands is a scalar, it is automatically repeated to be the same as the shape of the other operand.

Mathematical Operations		
function	monadic form	dyadic form
+	Nothing changes	The operands are added together
−	Sign of operand is changed	The operands are subtracted
×	Signum of operand	The operands are multiplied
÷	Inverse of operand	The operands are divided
=		1 for equal elements, 0 for unequal
>		1 for lhs greater, 0 otherwise
<		1 for lhs less, 0 otherwise
≥		1 for lhs ≥ , 0 otherwise
≤		1 for lhs ≤ , 0 otherwise
≠		1 for non equal elements, 0 for equal
^		logical and operation
v		logical or operation
~		logical not
*		lhs to the power of rhs
⌈	ceiling	maximum of lhs and rhs
⌊	floor	minimum of lhs and rhs

4.3. Matrix Manipulation

It is necessary to know a few matrix manipulation commands to use the vector and matrix capabilities of APL. The most useful are:

Matrix Manipulation		
function	monadic form	dyadic form
ρ	Gives the shape of its argument. The shape is a vector consisting of the size of each of the dimensions.	Reshapes the second argument according to the first. A matrix of the shape of the left hand argument will result. Extra elements are discarded. If more elements are needed, the right hand argument will be repeated.
⍉	The elements of the argument are transposed. (rows become columns, columns become rows)	
⍳	The numbers from the index origin (usually 1) to the argument are put in a vector. This is a very useful function for generating a sequence of numbers.	
↑		The first lhs elements are taken from the rhs. Each element in the lhs corresponds to one dimension of the rhs.
↓		The first lhs elements are dropped from the rhs. Each element in the lhs corresponds to one dimension of the rhs.
/	Reduce operator. The function on the lhs is applied between the elements of the rhs.	The rhs is compressed according to the lhs. A zero in the lhs will cause the corresponding element of the rhs to be removed.
\	Scan operator. The function on the lhs is applied successively between the elements of the rhs.	The rhs is expanded according to the lhs. A one in the lhs will insert a 0 into the rhs at the corresponding point.

Another important operation is matrix subscripting. Subscripts are put in square brackets and are separated by semicolons. A subscript can be a vector or a scalar. If a subscript is missing (but the semicolon is there) the entire dimension for that subscript is included. Consider for example the matrix

$$M = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

We have the following relationships:

$$M[2;3] = 7$$

$$M[1\ 2;4] = 4\ 8$$

$$M[;3\ 4] = \begin{pmatrix} 3 & 4 \\ 7 & 8 \\ 11 & 12 \end{pmatrix}$$

Note that the first subscript corresponds to the first dimension in the matrix, namely the rows. Specifying a vector for a subscript selects all the rows (or columns) specified.

5. A Few Advanced Functions

There are many more functions than have been described here. Also, the functions here can deal with higher order matrices. It is hard to understand the results of some APL operations without reading a proper text or manual on the language. There are a few more functions which will be described here because they are particularly useful for simulation with the APLSIM simulator.

5.1. Conversion into Another Base

APL provides two functions for base conversion. The first is encode. It converts a base 10 number (or a vector of them) into whatever base is specified. For example

$$2\ 2\ 2 \top 6 \quad \textit{gives} \quad 1\ 1\ 0 \quad \textit{or} \quad 6 \quad \textit{in binary}$$

The left hand argument gives the base for each position in the new number. In this case the three 2's indicated a three bit binary number was desired. If the righthand argument is a vector, a matrix will result.

$$2\ 2\ 2 \top 6\ 4\ 1\ 2\ 7 \ \textit{gives} \ \begin{pmatrix} 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

The binary numbers are read down the columns. This format is compatible with the APLSIM bus variables. If the bus is defined so that the LSB is on the right, assigning this matrix to the bus will generate the binary inputs for 6, 4, 1, 2, and 7 in sequence.

5.2. Conversion into Base 10

The decode function converts a vector of numbers into base 10. Actually it should handle a matrix too, but at the moment it does not.

$$2\ 2\ 2 \perp 1\ 1\ 0 \quad \textit{gives} \quad 6$$

6. System Commands

System commands are commands to control the operation of the APL interpreter. They start with a) immediately followed by a keyword. Some commands have an argument like a file name. A list of the currently defined APL system commands follows. The APLSIM simulator adds more system commands.

)CLEAR	Clear all information in the workspace. The network and all other information is lost. Simulator constants and flags will remain. (not yet working)
)SAVE	The current workspace, including all network information, is saved in the file specified. (not yet working)
)LOAD	A previously saved APL workspace is recalled from the file specified. The network which was loaded at the time the workspace was saved is restored to its former state. (not yet working)
)COPY	The contents of the specified file is copied into the workspace like)LOAD except that any non-conflicting symbols are added and the original information in the workspace is not destroyed. (not yet working)
)OFF	This command terminates the session. The window is removed from the screen.
)CONTINUE	The current workspace is saved and the session is terminated. The next time the interpreter is started the workspace will automatically be reloaded. (not yet working)
)READ	The specified file is read as a function definition. The file is an ASCII file created with an editor.
)VI	The UNIX **vi** editor is run on the specified file. APL assumes you are editing a function and tries to perform a)READ on the file when the editor is exited.
)CSH	A new UNIX **csh** shell is started. This allows you to execute normal UNIX commands inside the APL window. To get back to APL, terminate the shell with a control/D.
)ERASE	The specified variable name or function is erased and the memory it was using is released. Do not try this on APLSIM constants, flags, and nodes.
)LIB	This is much like the UNIX **ls** command. It lists the contents of the current directory.
)DROP	The specified file is deleted from the current directory. This does not have to be a file created from inside APL.
)WIDTH	The number of columns on the screen is set. This is necessary because APL indents all wrapped lines.
)VARS	The names of all variables in the workspace are printed. Note that this does not include APLSIM flags, constants, or nodes.
)FNS	The names of all functions in the workspace are printed.

7. APL Keyboard Layout

¨	¯	<	≤	=	≥	>	≠	∨	∧	−	÷	$
1	2	3	4	5	6	7	8	9	0	+	×	◊

?	⍵	∊	⍴	~	↑	↓	⍳	○	⋆	{	}
Q	W	E	R	T	Y	U	I	O	P	←	→

⍺	⌈	⌊	_	∇	∆	∘	'	⎕	(	)	⊣
A	S	D	F	G	H	J	K	L	[	]	⊢

⊂	⊃	∩	∪	⊥	⊤	|	;	:	\
Z	X	C	V	B	N	M	,	.	/

References

[1] C.M. Baker and C. Terman, "Tools for Verifying Integrated Circuit Designs," *LAMBDA*, forth quarter, 1980.

[2] Randal E. Bryant, "An Algorithm for MOS Logic Simulation," *LAMBDA*, forth quarter 1980, pp. 46-53.

[3] C.J.R. Fyson and K.G. Nichols, "Simple Characterization of Logic Gates for Use in Macrosimulation," in *Proc. IEEE International Conference on Circuits and Computers, ICCC 82*, 1982, pp 189-192.

[4] R.I. Gardner, J.W. Grundmann, D.S. Pass, K.C. Swanson, "Multilevel Simulation in VISTA: Architecture and Performance," in *Proc. IEEE International Conference on Circuits and Computers, ICCC 82*, 1982, pp. 26-29.

[5] M.H. Heydemann, G.D. Hachtel, M.R. Lightner, "Implementation Issues in Multiple Delay Switch Level Simulation," in *Proc. IEEE International Conference on Circuits and Computers, ICCC 82*, 1982, pp. 46-49.

[6] Dwight D. Hill, "Edsim: A Graphical Simulator Interface for LSI Design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. CAD-2, no. 2, April 1983, pp. 57-61.

[7] M.L. James, G.M. Smith, J.C. Wolford, *Applied Numerical Methods for Digital Computation with Fortran and CSMP*, Second Edition, Harper and Row, Publishers, 1977.

[8] J.E. Kleckner, R.A. Saleh, and A.R. Newton, "Electrical Consistency in Schematic Simulation," in *Proc. IEEE International Conference on Circuits and*

Computers, ICCC 82, 1982, pp. 30-33.

[9] Giovanni Mancini and Nicholas Rumin, "CAMUS — A Mixed-Mode Network Analysis Program," in *Proc. IEEE International Conference on Circuits and Computers, ICCC 82*, 1982, pp. 176-179.

[10] K.D. Mann, P.G. Gulak, E.Shwedyk, "A Shuffle Exchange Implementation of Viterbi's Algorithm Using CMOS VLSI", Technical Digest, 1985 Canadian Conference on Very Large Scale Integration, 1985, pp. 22-29.

[11] J. Mavor, M.A. Jack, P.B. Denyer, *Introduction to MOS LSI Design*, Addison-Wesley, London, 1983.

[12] Carver Mead and Lynn Conway, *Introduction to VLSI Systems*, Addison Wesley, Reading, Mass., 1980.

[13] Metheus, *Netlist Verification Tools Manual* 700 Series, Metheus-CV Inc., Hillsboro OR., 1984.

[14] Laurence W. Nagel, "SPICE2: A Computer Program to Simulate Semiconductor Circuits," Memorandum No. ERL-M520, College of Engineering, University of California, Berkeley, May 1975

[15] National Semiconductor, *CMOS Databook*, National Semiconductor Corporation, Santa Clara, California, 1981.

[16] A.R. Newton, "Techniques for the Simulation of Large-Scale Integrated Circuits," *IEEE Transactions on Circuits and Systems*, Vol. CAS26, no. 9, September 1979, pp. 741-749.

[17] A.R. Newton and A.L. Sangiovanni-Vincentelli, "Relaxation-Based Electrical Simulation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. CAD-3, no. 4, October 1984, pp. 308-331.

[18] Rathin Putatunda, "Automatic Calculation of Delay in Custom Generated LSI/VLSI Chips," in *Proc. IEEE International Conference on Circuits and Computers, ICCC 82*, 1982, pp. 193-196.

[19] Steven M. Rubin, "An Integrated Aid for Top-Down Electrical Design," Fairchild Laboratory for Artificial Intelligence Reseach, Palo Alto California.

[20] Y.P. Wei, I.N. Hajj, and T.N. Trick, "A Prediction-Relaxation-Based Simulator for MOS Circuits," in *Proc. IEEE International Conference on Circuits and Computers, ICCC 82*, 1982, pp. 34-37.