SIMULATION AND IMPLEMENTATION OF PULSED ANALOG NEURAL CIRCUITS

TAPAS BANERJEE

SIMULATION AND IMPLEMENTATION OF PULSED ANALOG NEURAL CIRCUITS

By

©Tapas Banerjee, B.E. (Hons.)

A thesis
submitted to the School of Graduate Studies
in partial fulfillment of the requirements for
the degree of Master of Engineering

Faculty of Engineering and Applied Sciences
Memorial University of Newfoundland
St. John's, Newfoundland, Canada A1B 3X5

August, 1993

ISBN 0-315-91635-4

Abstract

Conventional simulation of neural networks implemented using our pulsed analog topology is slow. The commercial circuit simulator HSPICE takes about half a day to simulate even a medium scale network, which is frustrating at the design stage. A new simulator, PULSE, has been developed to relieve the designer from this problem. PULSE provides about two orders of speed improvement over HSPICE while predicting the circuit performance with comparable accuracy. It uses a macromodeling approach in contrast to the transistor level simulation approach in HSPICE. Special features of pulsed analog networks are exploited constantly to reduce the simulation time. The analysis algorithms used are Waveform Gauss-Seidel and Functional Iteration. Circuit equations formulated being very sparse, correct ordering of the equations leads to convergence to the solution in only one Gauss-Seidel iteration. Using PULSE as the simulation tool, a Matrix Associative Memory has also been designed which is in the process of fabrication.

Acknowledgement

I sincerely acknowledge and thank my supervisor Prof. Bruce-Lockhart for all his involvement, patience, criticisms and constant encouragement for my work for the whole duration of my program here. I thank The School of Graduate Studies of Memorial University, the Faculty of Engineering, the Associate Dean of Engineering (Graduate Studies) and his office for the financial support provided. I also thank Lloyd Little for his constant technical support and thank all of my fellow graduate students in the Faculty of Engineering for many sweet memories. Finally, I deeply acknowledge the constant support of my family from overseas.

Contents

Abstract ii

Acknowledgement iii

Contents iv

List of Figures viii

List of Tables xii

List of Symbols xiii

1 Introduction 1

2 Literature Review 6

2.1 Introduction 6

2.2 Review 9

2.2.1 The Circuit Simulation Process 9

2.2.2 Different Types of Simulators 12

2.2.3 Transient Analysis in Conventional Simulators 22
2.2.4 Third Generation Simulators 27
2.3 Concluding Remarks 31

3 Pulsed Neural Networks and Our Approach **32**
3.1 Introduction 32
3.2 Review 36
3.2.1 Synapses 37
3.2.2 Neurons 38
3.2.3 Programmability and Learning 39
3.3 Our topology 40
3.4 Conclusion 45

4 Simulator Design **46**
4.1 Introduction 46
4.2 The Simulation process in PULSE 48
4.3 Development of the Netlister 50
4.4 Modeling 52
4.4.1 Excitatory Synapse 53
4.4.2 Inhibitory Synapse 58
4.4.3 Neuron 59
4.4.4 Non-Inverting Input Neuron 61
4.4.5 Inverting Input Neuron 62

4.5 Analysis and Algorithms 63
4.5.1 Equation Formulation 64
4.5.2 Implicit Integration 65
4.5.3 Waveform Gauss-Seidel Iteration 67
4.5.4 PULSE Analysis Block - Reviewed 68
4.6 Discussion 70
4.7 Conclusions 73

5 Simulation using PULSE **75**

5.1 Simulation of a Simple Network 75
5.2 Simulation of XOR Gate 78
5.3 Simulation of CAM 81
5.4 Matrix Associative Memory 89
5.4.1 Implementation 90
5.4.2 Simulation 96
5.5 Discussion 99
5.6 Conclusion 102

6 Results **103**

6.1 Cost of PULSE simulation 103
6.1.1 Evaluation of Neuron Input Voltages 106
6.1.2 Number of Functional Iterations 108
6.1.3 Number of Gauss-Seidel Iterations 110

6.2 Speed Improvement in PULSE 112
6.3 Concluding Remarks 115

7 Conclusions **116**

References **118**

A Minimum Time-Step for Predictor-Corrector Method **124**

B Source Code of the Simulator **126**

B.1 Data Flow Diagram of PULSE 126
B.2 PULSE Source Code 128
B.3 The Macromodels 152
B.4 The Waveforms Used in PULSE 159

C PULSE Input Files - An Example **161**

C.1 netlist 161
C.2 pulse.inp 162
C.3 pulse.sim 162
C.4 control 162

D Output File Generated by PULSE **163**

E The Netlister **167**

E.1 S/SLG File to provide Formatting Instructions to FNL 167
E.2 S/SLG File to Define Netlist Output of Symbols 168

List of Figures

2.1 (a) The circuit and (b) the input file for the simulation of (a) . 10

2.2 The simulation output 11

2.3 (a) Model of or gate, (b) Principle of event scheduling and selective trace 15

2.4 Steps involved in a transient analysis 23

3.1 A neuron and the connected synapses 33

3.2 (a) The excitatory and (b) the inhibitory synapse circuit . . . 41

3.3 Block diagram of the proposed neural network 42

4.1 PULSE simulation steps 48

4.2 How FNL works 50

4.3 Sample output of the netlister 51

4.4 The excitatory synapse 53

4.5 Plot of exsynapse current I(in μA) against V_{wt}(in $Volts$) (a) HSPICE level 1, (b) HSPICE level 3 55

4.6 The model generated characteristics and the HSPICE characteristic (Fig. 4.5(b)) plotted on top of each other. They totally overlap as the graph shows. 57
4.7 The inhibitory synapse 58
4.8 The standard neuron 59
4.9 The non-inverting input neuron 61
4.10 The inverting input neuron 62
4.11 PULSE analysis block 63
4.12 A typical connection at neuron input node 64
4.13 Waveform Gauss-Seidel Algorithm as in PULSE 68

5.1 A simple network using 3 synapses and one neuron 76
5.2 Simulating a circuit with two excitatory and one inhibitory synapses. First three waveforms are input pulses and the last one is the waveform at the summing node of the neuron. PULSE and HSPICE both outputs are shown for the last waveform. 77
5.3 Implementation of XOR using our topology 78
5.4 Outputs of the XOR circuit as predicted by PULSE and HSPICE for inputs of (a) 01 and (b) 11. 80
5.5 Schematic diagram of the CAM. The synapses are shown as coin-shapes whereas neurons are the trapezoids at the center. . 82

5.6 HSPICE simulation of CAM showing the activation and outputs of neuron $N1$, $N6$ and $N7$ respectively. 83
5.7 PULSE simulation of the CAM circuit showing the activation and outputs of the same neurons. 84
5.8 HSPICE simulation of CAM for input 10101 showing the activation and outputs of neuron $N2$, $N4$ and $N6$ respectively. . 86
5.9 PULSE simulation of the CAM circuit showing the activation and outputs of the same neurons. 87
5.10 Output of CAM chip in response to input 10101 (a) Neuron N4, (b) Neuron N2 88
5.11 Circuit used to implement (a) Positive weights and (b) Negative weights 91
5.12 Schematic diagram of the Matrix Associative Memory 93
5.13 Layout of the associative memory obtained by auto place and route routines (only Metal1 layers are shown). 95
5.14 PULSE and HSPICE simulation of the Matrix Associative Memory for input 10001111 showing the outputs of the circuit. For all three neurons, the first output is predicted by PULSE and the second one is predicted by HSPICE. 97
5.15 PULSE and HSPICE simulation of the Matrix Associative Memory for distorted input 10011111 showing the outputs of the circuit. 98

5.16 HSPICE predicted output for XOR circuit with input 01 (a) Prediction with 1ns time step. (b) Prediction with 2ns time step and (c) Prediction with 2ns time step with NMOS conductance 1% reduced. 100

6.1 Code generated by PULSE 104

6.2 Dependence of PULSE simulation time on the number of synapses present in a CAM circuit 107

6.3 Dependence of PULSE simulation time on the convergence criterion of the functional iteration loop 108

6.4 Dependence of number of functional iterations on circuit activity 109

List of Tables

4.1 Neuron output and discharge pulse details 60

4.2 Worst-case time step 71

5.1 Input-Output patterns for matrix associative memory 90

5.2 Weight voltages used in the associative memory 92

6.1 Time required to evaluate synapse macromodels 106

6.2 Comparing PULSE and HSPICE speed (on a DECStation 5000/200) 114

List of Symbols

Symbol	Description
C	- capacitance connected to a given node
Δt	- timestep in transient analysis
T	- time-period of transient analysis
V_{dc}	- discharge pulse input to the excitatory synapse
V_{ex}	- pulsed input to the excitatory synapse
V_{in}	- pulsed input to the inhibitory synapse
V_{lk}	- leakage input to the excitatory synapse
V_m	- membrane voltage of the neuron
V_{th}	- threshold voltage of the neuron
V_{wt}	- weight voltage input to the synapses

Chapter 1

Introduction

The foundation of research concerning Artificial Neural Networks can be traced back to the 1940's when McCulloch and Pitts (1943) came up with a mathematical theory of neural computation. At that time there existed no mechanism to explain the learning in neurons. Only in the 1960's did Frank Rosenblatt, a Cornell University psychologist, introduce the *perceptron* which had a clear learning rule defined.

Perceptrons were essentially single-layer networks of linear threshold units connected without feedback. The most appealing feature about them was that given a problem, there was a convergence procedure that assured their learning, provided a learning solution existed. But in 1967, Minsky and Papert clearly exposed the deficiencies in perceptrons by showing their inability to solve even simple class of problems such as the XOR problem. It rendered a severe blow to the field of Neural Network Research and literally no interest existed among the researchers during the 1970's.

The perceptrons could however solve these class of problems using a hidden layer between the inputs and the output layer in contrast to using only a single output layer as proposed by Rosenblatt. These kind of networks already existed [Nilsson, 1965], but there was no algorithm present for their learning. In 1986, the *generalized delta rule* introduced by the PDP research group at MIT provided the much required learning algorithm for multilayered networks using neurons with continuous, nonlinear activation functions. It triggered an explosion of interest among researchers and as a result, neural networks experienced a phenomenal growth over the last few years.

Most of the research activities in neural networks is now directed towards theoretical studies and simulation. Computer simulation is slow and the real power of neural networks can only be exploited by implementation of them using special purpose hardware that maps the specific network directly into the architecture using models of neurons and synapses. Neural networks generically draw their computational power not from their processors but from the massive number of interconnections among them. VLSI tends to be a natural choice for implementation of them in this respect owing to its inherent ability to implement large number of interconnections and in favoring repetitive structures as present in the neural networks.

VLSI implementation of neural networks can be carried out using the digital or analog approach. The digital approach is traditional and allows advantages such as precise arithmetic, robustness against noise and readily

available implementation processes. This approach however has its disadvantages. The major operations involved in neural networks are multiply and add. Digital multipliers occupy large space in silicon. The precision required in implementing neural network is low which again doesn't necessitate the use of area-hungry digital implementation. These considerations have attracted neural researchers toward building neural networks in analog VLSI. Analog VLSI implementations are compact compared to their digital counterpart, but have their own drawbacks of susceptibility to noise and process variations.

The drawbacks associated with both digital and analog processes encourages one to take a hybrid approach. In this scheme, called the pulse-stream approach, the neural states are represented as pulses, which provides the system with noise immunity. The synapses and neurons are still built using analog circuits for compactness. In short, this approach combines the best of both the digital and analog worlds. As a matter of fact, the biological neurons are also known to communicate through pulses.

Our approach to implementation of Neural Networks is pulsed analog. However it differs from other adherents of this approach in several aspects. Firstly, our synapses are non-androgynous i.e. they are either excitatory or inhibitory and don't switch between these modes. The synapses are non-linear i.e. the input-output relationship for synapses is not linear. The proposed topology is also auto-scaling, which enables any number of synapses to be

connected to a given neuron.

Our proposed pulsed analog topology has already been verified by building two chips using CMOS3DLM, a 3 micron CMOS process. The first chip implemented the standard cells while the second one implemented a CAM (Content Addressable Memory) and a Maxnet. The testing of them have shown favorable results.

The simulation of these networks, when built in silicon, however causes a problem. A small network consisting of only 10 neuron and 100 synapses consists of 500 transistors. Conventional circuit simulators like SPICE takes a long time to simulate this kind of networks. Simulations running over days during the design phase is unacceptable and that inspired us to build a fast circuit simulator which will permit exploration of complex networks.

In this thesis, a fast simulator is built to address this issue. The design considerations and the methodologies employed are described. The simulator outputs have been bench-marked against HSPICE, a commercial version of SPICE. The speed improvement achieved is at least two orders of magnitude. With extensive use of this simulator (as was its purpose) an associative memory has been built, which is currently in the process of fabrication.

The thesis has been organized as follows. After this brief introduction, the second chapter deals with the literature review. Different approaches in simulator design have been discussed. In the third chapter, implementation aspects of neural networks have been reviewed with stress on pulsed

implementations. The fourth chapter discusses the simulator design. The fifth chapter looks into the simulation of known networks and also deals with the implementation aspects. It also includes the schematic diagram and the layout of the associative memory which is in the process of fabrication. The sixth chapter presents the results obtained and also compares the speed of the developed simulator against the conventional circuit simulator HSPICE. Finally, in the last chapter the thesis is concluded and some areas are pointed out for future investigation.

Chapter 2

Literature Review

2.1 Introduction

Circuit simulators are indispensable verification tools for a VLSI designer. The parasitics present in an integrated circuit (IC) cannot be modeled accurately with the traditional breadboarding technique, rendering this method inappropriate for verifying a chip's performance. A simulator solves that problem by using a mathematical model of the circuit elements and the parasitics. It also provides the designer with the advantage of extensively probing the IC, which is otherwise impossible owing to size constraints.

Frank Brannin is often credited as the developer of the first effective circuit simulation program, TAP, written in the 1960's. TAP was never released in the public domain, but it's development formed the basis of two more simulators that came from IBM - ECAP1 and PREDICT. All these programs reportedly had several problems; they were hard to use, were unfriendly and had severe nonconvergence problems [Pederson, 1984].

The simulator programs described above are commonly referred to as the *first-generation* programs in the literature. SPICE and ASTAP are typical examples of *second generation* simulators. ASTAP [Weeks et al., 1973] was developed at IBM in 1973 for statistical simulation and eventually found widespread use within the IBM corporation. During the same period, SPICE [Nagel, 1975] was developed by Nagel at University of Berkeley and is arguably the best known simulator in the world. Both SPICE and ASTAP are accurate and reliable, but designed to simulate circuits with only a couple of hundred transistors. Hence simulation using SPICE2 and ASTAP is slow for VLSI circuits and led to the development of a new breed of simulators.

Introduction of MOTIS [Chawla et al., 1975] marked the era of so called *third generation* simulators. These simulators use various algorithms such as relaxation techniques, node decomposition, table look-up models etc. to achieve more than an order of speed improvement over second generation simulators while providing outputs with considerable accuracy. SPLICE [Newton, 1979] (developed at Berkeley) and DIANA are typical examples of this generation of simulators. The review section covers these simulators at a greater depth. Recent simulators e.g. RELAX2 [Newton and Sangiovanni, 1984], SPLAX [Saleh and White, 1990] using waveform relaxation techniques reportedly perform up to 50 times faster simulation with accuracy comparable to SPICE2.

All these simulators described above are general-purpose. However, custom-

made simulators are widely used in industry. The main reason is their ability to exploit efficiently the peculiarities of the topology for which they are built in order to speed up the transient analysis further. Our pulsed analog topology is no exception. The simulator for our pulsed circuits can be reduced to a timing simulator (refer to subsection 2.2.2) by using the peculiarities that are particular to our topology (refer to subsection 6.1.3). The timing simulators can provide over two orders of magnitude speed improvement against SPICE thus being computationally more efficient than the general purpose third generation simulators [Newton and Sangiovanni, 1984].

Also once macromodels are developed, our experience is that the designer effort required to install them in a general purpose simulator is commensurate with building a custom simulator from scratch. Since the actual code required for the custom simulator is small in our case and we could be reasonably certain that it would run faster than a general purpose one, it was decided to build our own.

Since the proposed simulator is built for fast transient analysis and uses second and third generation simulator concepts, the review material is organized accordingly. The first section presents the circuit simulation process. Different types of simulators e.g. Timing Simulators, Logic Simulators etc. are introduced next to illustrate the different approaches taken by the designers over the years to solve the problem of circuit simulation. The classic algorithms used by second-generation simulators in performing a transient

analysis are discussed in the following section. Finally, third generation simulators and the algorithms used by them to speed up the transient analysis are presented.

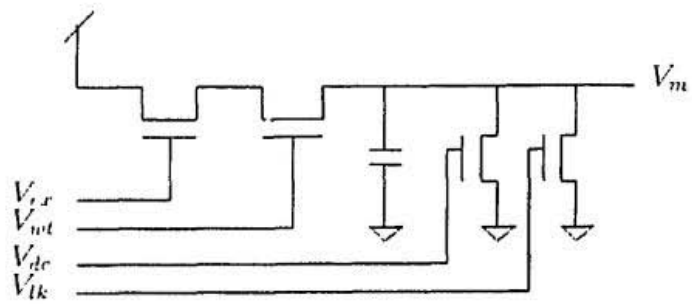
2.2 Review

2.2.1 The Circuit Simulation Process

The circuit simulation process can be best understood with the help of an example. The example presented here is that of simulating a transistor circuit to find the DC current flowing through one of its transistors. An independent voltage variable is swept across the range 0-5V, which in turn changes the current. This simulation provides the designer with a *transfer characteristic*.

This circuit is simulated using HSPICE, a commercial version of SPICE. At the start of the simulation, an input file is presented to the simulator by the user. In many simulators, considerable help is available to the user in building this file. Fig 2.1 shows the input file and the circuit to be simulated.

The input file is divided into many paragraphs. The first paragraph defines the circuit inputs. The second paragraph (it's really one line), asks the simulator to do a DC analysis at each point as the variable V_{in} is swept over the range 0-5V. The next paragraph is the description of the circuit, commonly known as the *netlist*. Each component connection is defined here and the parameters of a complex model of the NMOS transistor is presented that will be used for analysis.



(a)

```
Vdd 1 0 dc 5v
VVex 3 0 5v
VVlk 5 0 1.5v
VVm 2 0 1v
VVwt 4 0
VVdc 7 0 2v

.dc VVwt 0.0 5.0 0.1

C3 2 0 poly 0.1pf
.MODEL Model4 nmos level=3 vto=.7 kp=4.e-05 gamma=1.1 phi=.6
+lambda=.01  pb=.7 cgso=3.e-10 cgdo=3.e-10 cgbo=5.e-10 rsh=25
+cj=.00044  mj=.5 cjsw=4.e-10 mjsw=.3 js=1.e-05 tox=5.e-08
+nsub=1.7e+16  nss=0 nfs=0 tpg=1 xj=6.e-07 ld=3.5e-07 uo=775
+utra=0 vmax=1.e+05  xqc=.5 theta=.13  eta=.05 kappa=1
M5 2 5 0 0 Model4  l=2.06e-05  w=5.4e-06
M6 2 7 0 0 Model4  l=3.e-06  w=5.4e-06
M7 6 4 2 0 Model4  l=3.e-06  w=5.4e-06
M8 1 3 6 0 Model4  l=3.e-06  w=5.4e-06

.print dc I(M7)
.end
```

(b)

Figure 2.1: (a) The circuit and (b) the input file for the simulation of (a)

The last paragraph asks the simulator to present the DC current through the transistor M7 (which is the second transistor from the left, comparing the figure to the netlist) as the output and ends the file.

The simulator reads this file and performs a DC analysis. How this analysis is done will be discussed in depth in the next few sections. Here we are only concerned about the simulation process, hence we will skip through that part. Once the simulation is complete, the simulator presents the output in the format shown below.

```
      volt      current
                   m7
0.              5.694e-12
0.10000         5.695e-12
0.20000         5.695e-12
   ...             ...
   ...             ...
5.00000         1.016e-04
```

Figure 2.2: The simulation output

The left column of the output shows the variable V_{out} and the right column shows the current through transistor M7 and that concludes this simulation run. The user can also request a plot of the output using a *plot* statement in place of the *print* statement, and is allowed to do other kinds of analysis too, e.g. AC or transient analysis.

A simulation run is discussed in this section to familiarize the reader with the simulation process. The next section will discuss different types of sim-

ulators and their analysis methods. The discussion starts with conventional simulators [1] and then describes other simulators, pitting their performances against the conventional ones. This will familiarize one with the breadth of the simulation methods used nowadays to simulate an entire IC, and also will introduce one to the evolution of the simulation techniques as the circuit complexity continued to grow.

2.2.2 Different Types of Simulators

Circuit Simulators

SPICE2 is a typical example of this group of simulators. These simulators perform very accurate DC, AC, frequency domain, noise and sensitivity analysis based on well-known nonlinear bipolar and MOS circuit models. Considering our interest, we will take a look at the transient analysis algorithms of these simulators.

In transient analysis, the node voltages are to be calculated for a time period of 0 to T (at time intervals of Δt). The voltages at time 0 (the initial condition), are calculated using a DC analysis. From then on, the voltage values are calculated sequentially, i.e. the voltages at time point $(t + \Delta t)$, are calculated using the voltages at time point t. The circuit elements are replaced by mathematical models to yield a set of circuit equations of the

[1] The terms *conventional simulator* and *circuit simulator* will be used interchangeably as in the circuit simulation literature.

form

$$\dot{\mathbf{v}} = f(\mathbf{v}, \mathbf{u}, \mathbf{t}) \tag{2.1}$$

where, $\mathbf{v} = [v_1, \ldots, v_n]^T$ are the dependent node voltages and $\mathbf{u} = [u_1, \ldots, u_n]^T$ are the independent node voltages. This typical form of Eqn. 2.1 arises owing to the presence of energy storage elements such as inductors and capacitors. An implicit integration scheme is required to convert this set of differential equations to a set of nonlinear equations of the form

$$j(\mathbf{v}) = 0 \tag{2.2}$$

in order to solve them.

The nonlinearity arises from the nonlinearity inherent in the device model. The well-known Newton-Raphson (NR) iterative method is commonly used to linearize this set of equations. A typical NR iteration is of the form

$$J(v^t)(v^{t+\Delta t} - v^t) = -j(v^t) \tag{2.3}$$

where, v^t is the voltage at timepoint t and $J(v^t)$ is the Jacobian matrix of $j(v^t)$. Since each NR iteration step produces a linear equation, this algorithm generates a set of linear equations from the set of nonlinear equations at each timestep. These equations have to be solved to find the new iterated value and hence to decide whether convergence has been reached (in the solution) or not. The LU factorization method solves these equations. At convergence, the iterated values provide the set of node voltages at time $(t + \Delta t)$. Implicit integration, Newton-Raphson and LU factorization will

be discussed at greater length in the next section since the purpose of this discussion is only to introduce the reader to the transient analysis process in conventional simulators.

As discussed in the introduction of this chapter, the circuit simulators were developed for the simulation of a couple of hundred transistors. Experimental as well as theoretical evidence shows that the computational cost of conventional simulation is

$$Computational\ cost = O(n^y)\ (1.1 \leq y \leq 1.5) \tag{2.4}$$

where n is the number of circuit nodes [Newton and Sangiovanni, 1984]. This super-linearity in time cost is important for circuits with several thousand nodes since at that break-even point the super-linearity starts dominating the time cost. This problem worsens for circuits of larger scale.

Circuit simulators suffer from another major problem. In a given circuit, the time constants of various nodes vary over a large degree. One such case arises when lumped capacitors are present at some nodes of the circuits and small parasitics are present at other nodes. For nodes with smaller time constants, the rate of change in node voltage is much higher which requires smaller timesteps in transient analysis. Although nodes with much larger time constants could be simulated using much higher timesteps, thus reducing the simulation time, this is rendered impossible due to the presence of nodes with much smaller time constants. Hence more timesteps are needed which means more computation in a given time interval T and more simulation

cost.

The above two problems are serious drawbacks for conventional simulators when used at VLSI scale and suggests a need to explore other approaches to simulate these (VLSI scale) circuits economically. The following paragraphs elaborate on that.

Logic Simulators

Logic simulators simulate at least at the gate level in the time domain as compared to the transistor level simulation in circuit simulators. That drastically reduces the number of nodes to be handled in a given simulation. Also, they operate on logic states rather than working with the voltage and current waveforms. The gates are modeled as logic blocks with a delay element added as shown in Figure 2.3(a).

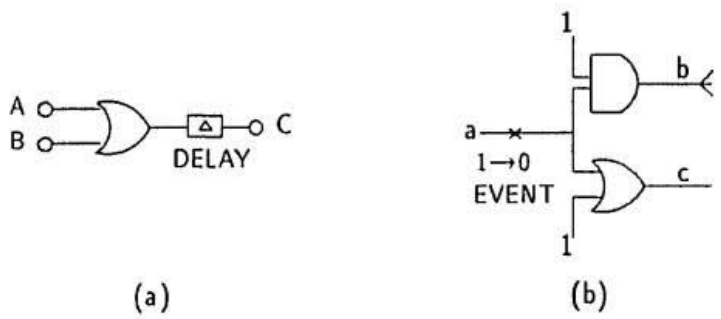


Figure 2.3: (a) Model of or gate, (b) Principle of event scheduling and selective trace

The delay can be *zero* for simple logic verification, or *assignable* by the user (for example, all OR gates can be assigned a delay of $3ns$ for a specific

implementation process) or *precise* i.e. the minimum and maximum possible delay in an element can be specified, which can also provide some timing information [Szygenda and Thompson, 1974]. The signal states recognized are $1, 0, H$ and $*$. If a node is floating, it assumes the 'H' state and undefined nodes assume the '$*$' state.

Logic simulators are typically 10^3 times faster than the circuit simulators. Along with the use of simple models and the fact that they only deal with the logic states, the improvement in their speed can also be attributed to two characteristic algorithms, known as *event scheduling* and *selective trace*.

Selective trace methods do not follow the output of an element if its output did not change when it was evaluated at this timepoint. Assume one of these elements to be A whose output has changed at this point of time and let the delay in all gate models be Δ. Thus only the fan-outs of A can change state after time Δ, and hence are scheduled to happen at that time. This is the principle of *event scheduling*. Fig. 2.3(b) illustrates these principles. Here, only node b changes state when node a switches from 1 to 0, and hence only b is scheduled to happen after time Δ. Since most parts of a large digital IC remain inactive at a given point of time, these techniques provide considerable speed-up.

However, logic simulation has many limitations. The gap between logic simulation models and physical systems is considerable. Logic simulators hardly provide any timing information. Also, interconnection noise, clock

feedthrough etc. cannot be modeled with this type of simulator. Conventionally, these problems are best tackled by circuit simulators, but their lack of speed led to the development of another kind of simulator that tries to enhance circuit simulation speed by relaxation of some constraints. They are called *timing simulators* and are described next.

Timing Simulators

Timing simulators deliberately introduce approximations to achieve greatly improved simulation speed [Chawla et al., 1975]. These simulators characteristically use *equation decoupling* and *single NR iteration* techniques in place of "iteration until convergence" as in the circuit simulators, and piecewise-linear or *table look up models* in place of complex algebraic nonlinear models. In this discussion, we describe these methods and investigate their effect on the performance of the simulator.

The equation decoupling approach can be described as follows. Consider a circuit with N nodes. Assume that we are evaluating v_i at time $(t + \Delta t)$ and the voltages $v_1, v_2, \ldots, v_{i-1}$ at time $(t + \Delta t)$ are already known. At this point of time, if we solve Equation 2.2 for v_i, using the already known values of $v_1, v_2, \ldots, v_{i-1}$ at time $(t + \Delta t)$ and that of $v_{i+1}, v_{i+2}, \ldots, v_N$ at time t, the equation becomes

$$j_i({v_1}^{t+\Delta t}, \ldots, {v_{i-1}}^{t+\Delta t}, \mathbf{v_i}, {v_{i+1}}^{t}, \ldots, {v_N}^{t}) = 0 \qquad (2.5)$$

The above equation assumes all node voltages other than v_i to be con-

stant. Hence in contrast to circuit simulators, after the NR step, we do not get a system of N linear equations in N independent variables, but N decoupled equations, each with one variable. Hence the name *equation decoupling*. This renders the LU factorization step of circuit simulators unnecessary, and enhances the simulation speed at the cost of losing some accuracy at the equation decoupling step.

Timing simulators characteristically use only a single NR iteration at each timestep. This is in direct contrast to the "NR iteration until convergence" approach of the conventional simulators. This (one NR iteration) approach produces correct results only when these node voltages calculated by a single NR iteration are iterated (using Gauss-Seidel type algorithms) until convergence at a given time point (this iterative process is commonly known as the *relaxation* [1] *iteration* and is a very common algorithm in third-generation simulators). But timing simulators perform only one relaxation iteration at a given time point which can lead to considerable error for circuits with tight feedback loops.

While looking at Eqn. 2.5, also notice an inherent limitation of the timing simulators. Assume a node equation of the form

$$C\frac{dv}{dt} = kv^2 \tag{2.6}$$

An implicit integration step (say, *backward euler*), generically produces the

[1] The term *relaxation* comes from relaxing the exactness of the solution. Here this relaxation originates from the equation decoupling step.

following equation

$$v_{t+1} = av'_{t+1} + b \tag{2.7}$$

which in our case becomes

$$v_{t+1} = \frac{ak}{C} {v_{t+1}}^2 + b \tag{2.8}$$

or,

$$ak{v_{t+1}}^2 - Cv_{t+1} + bC = 0 \tag{2.9}$$

This characteristic equation generated by implicit integration algorithms can be written as

$$g(v_{t+1}) = 0 \tag{2.10}$$

Application of k-th NR iteration will then yield a linear equation of form

$${v_{t+1}}^{k+1} = {v_{t+1}}^k - \frac{g({v_{t+1}}^k)}{\frac{\partial g({v_{t+1}}^k)}{\partial v_{t+1}}} \tag{2.11}$$

where ${v_{t+1}}^k$ refers to the v_{t+1} value at k-th NR iteration.

To assure the $\frac{\partial g({v_{t+1}}^k)}{\partial v_{t+1}}$ term to be non-zero, one has to assure the presence of the term v_{t+1} in Eqn. 2.9. That can only be done by assuring the numerical integration step, i.e. by the presence of a $\frac{dv}{dt}$ term.

After the equation decoupling step, timing simulators solve an equation equivalent to Eqn. 2.10. So, they have to assure the presence of the $\frac{dv}{dt}$ term too, which is done by assuming the connection of a capacitance (assuring the presence of the C term in Eqn. 2.6) between every node to ground or to

another node [Refer to Eqn. 2.6]. In most cases parasitic capacitances serve this purpose.

Use of tabe look-up models is another specialty of timing simulators. A big advantage of this technique is its ability to use complex models with no speed sacrifice. Also, since nonlinear element updating is reported to take 80% of CPU time in standard circuit simulation [De Man, 1979], this also provides improvement of speed with negligible loss of accuracy.

Application of these approximation techniques provide timing simulators with a $100 - 200$ times speed improvement over conventional simulators [De Man, 1979]. But by using the equation decoupling technique, the timing simulators neglect the feedback between elements. Hence they are ineffective for circuits with strong bilateral coupling. Also convergence in solution cannot be guaranteed with only one relaxation iteration as stated before. Owing to these reasons, timing simulators are known to provide incorrect results in some situations.

As noted by Newton and Sangiovanni [Newton and Sangiovanni, 1984], "*A circuit designer will use a program that gives the correct simulation result and occasionally gives no result. A circuit designer soon loses confidence in a program that occasionally gives an incorrect answer.*" Even recognizing these shortcomings, it has to be admitted that the timing simulators introduced us to some revolutionary concepts that led to the development of the VLSI age (third generation) simulators.

Mixed-mode Simulators

Independent use of any one of the above simulators does not prove enough for the simulation of a typical LSI IC'. Circuit waveforms are essential at some parts of the circuit, but full circuit simulation is not cost-effective as discussed before. Timing analysis is not possible at some parts of the circuit since some bilateral components may be present. At some parts of the circuit, only logic analysis could be sufficient. To solve this problem, mixed-mode simulators combine the capabilities of the other three kinds of simulators.

SPLICE [Newton, 1979] is a typical example of this class of simulator. It is a hybrid simulation program, which divides a LSI circuit into several parts and according to the need, performs either circuit, timing or logic simulation for that part of the circuit. The circuit, logic and timing simulators communicate with each other by

- **Threshold eleme: ts** - Work as an interface from the Circuit and Timing simulators to the Logic simulators. They provide a '1' to the Logic simulator when the input voltage to it is above a predefined threshold, a '0' when the input voltage is below another predefined threshold and a 'X', when the input voltage is in between.

- **Logic-to-Voltage (LTV) converters** - Work as an interface from the Logic simulators to the Circuit and Timing simulators. They provide voltages corresponding to the '1' and '0' state in the steady state and

provide a ramp during the change of state with slope corresponding to the rise (fall) time of the LTV, when the state change is from '0'('1') to '1' ('0').

This class of simulators uses third-generation simulation techniques in an effort to combine the advantages of circuit, logic and timing simulators. It shares their disadvantages too, but its power is its ability to use the right simulator according to the need of the simulation problem, which makes cost-effective simulation in VLSI possible .

The evolution of different kinds of simulators and the methods used by them to solve the circuit analysis programs have been discussed in this section. In the next section, following the reasoning of the introduction, we will concentrate on transient analysis in conventional simulators.

2.2.3 Transient Analysis in Conventional Simulators

Transient analysis in conventional simulators has already been introduced to the reader in the section on circuit simulators. Fig. 2.4 (in the next page) shows the major steps in a transient analysis. The following paragraphs deal with these steps separately. The algorithms used at each step are briefly described and the relative efficiency of one algorithm over another, whenever present, is pointed out.

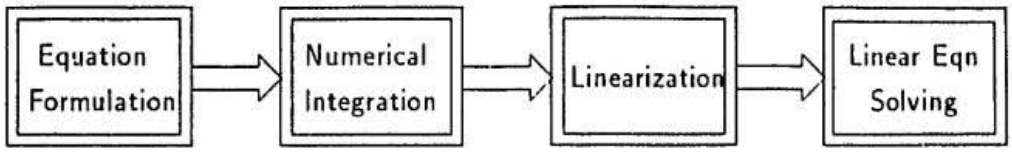


Figure 2.4: Steps involved in a transient analysis

Equation Formulation

Equation formulation is the very first step in any circuit analysis. Circuit simulators use two major approaches - Modified Nodal Analysis (SPICE2) and sparse-tableau formulation (ASTAP). For a circuit with N nodes and m voltage-defined (for example, containing only a voltage source) branches, modified nodal analysis (MNA) formulates $N - m + 1$ equations with the $(N - 1)$ nondatum node voltages and the m voltage-defined branch currents as variables. In contrast to that, the sparse-tableau formulation assembles an exhaustive list of equations that can arise from branch relations, from KCL and from KVL.

The best algorithm is determined by the computational effort required to solve those equations and also the numerical conditioning of the equations formed. When we compare the two methods, the number of equations in sparse-tableau is greater, though the equation formulation time is less, since an exhaustive list of equations is prepared. Obviously, solution of sparse-tableau by conventional means will require considerable effort, and we will need complex reordering mechanisms. Also, if the set of formulated equations

is found to be ill-conditioned, the reordering mechanism that we have to use to make the set well-conditioned is more complex for sparse-tableau. For all these reasons, MNA is considered to be the better of the two.

Numerical Integration

The differential equations formed in the equation formulation step are then passed through a numerical integration routine. Recognizing the fact that in a transient analysis we want to predict $V(t + \Delta t)$ using the already known values of $V(t)$ and Δt (the timestep), integration methods can be divided into two classes. In *explicit* integration methods, $V(t + \Delta t)$ is predicted in terms of $V(\tau)$ where $\tau \leq t$. In implicit integration, $V(t + \Delta t)$ is predicted in terms of $V(\tau)$ where $\tau \leq t + \Delta t$. Though explicit integration is straightforward, it is unstable i.e. the error generated at each step of numerical integration tends to accumulate. Hence, implicit integration methods are preferred and are used in all circuit simulators.

In all implicit integration algorithms other than the trapezoidal integration in its original form [McCalla and Pederson,1971], $V(t + \Delta t)$ is defined in terms of $\dot{V}(t + \Delta t)$. Since we cannot use large timesteps with the trapezoidal method, (more about it in the later chapters) the other algorithms are commonly used.

Assume circuit equations are in the normal form i.e.

$$\dot{x} = f(x, t) \tag{2.12}$$

Hence, $\dot{V}(t+\Delta t)$ is a function of $V(t+\Delta t)$ i.e. $f(V(t+\Delta t))$. Now $f(V(t+\Delta t))$ contains device models, which are nonlinear in most cases. Hence the equation after the integration step is (in most of the cases) nonlinear. So, now we need a linearization step to solve those equations.

Linearization

Newton-Raphson(NR) functional iteration is the most commonly used method in linearization [Kreyszig, 1988]. Here it is introduced by considering a single nonlinear equation.

$$g(v) = 0 \tag{2.13}$$

Expansion of $g(v)$ about a point v_0 in Taylor series and subsequent retention of only first order terms gives,

$$v = v_0 - \frac{g(v_0)}{g'(v_0)} \tag{2.14}$$

This suggests the sequence of iterations to be

$$v^{(t+1)} = v^t - \frac{g(v^t)}{g'(v^t)} \tag{2.15}$$

The generalization of Equation 2.15 to a system of n equations is

$$\mathbf{v}^{t+1} = \mathbf{v}^t - J(\mathbf{v}^t)^{-1} g(\mathbf{v}^t) \tag{2.16}$$

where $J(\mathbf{v}^t)^{-1}$ is the inverse of the Jacobian of g computed at $\mathbf{v}^t$.

$$J(\mathbf{v}^t) = \begin{bmatrix} \partial g_1(\mathbf{v}^t)/\partial v_1 & \cdots & \partial g_1(\mathbf{v}^t)/\partial v_n \\ \vdots & \ddots & \vdots \\ \partial g_n(\mathbf{v}^t)/\partial v_1 & \cdots & \partial g_n(\mathbf{v}^t)/\partial v_n \end{bmatrix} \tag{2.17}$$

Equation 2.16 defines the t-th iteration of the Newton-Raphson scheme. These iterations are carried out at each timepoint of transient analysis and the resultant linear equations solved to predict the value at timepoint $(t+\Delta t)$.

Linear Equation Solving

This set of linear equations (Equation 2.16) can now be solved by Gaussian Elimination or LU decomposition. Gaussian elimination being a well known process [Kreyszig, 1988], we describe the other one.

Given a set of linear equation $Ax = b$, suppose we have a way of factoring the coefficient matrix A into the product of two matrices, L and U, which are lower and upper triangular respectively. Then we have $Ax = LUx = b$. Now by setting $Ux = y$, we can solve the triangular system $Ly = b$ by forward substitution and then can solve the triangular system $Ux = y$ by backward substitution. There could be many ways the matrices L and U are chosen, a unique choice can be made by making the diagonal elements of L and U 1.

Any one of the two methods mentioned above can be used for solution of the linear equations. The system of equations being inherently sparse, *sparse-matrix methods* are almost always used to accelerate the execution times.

This concludes the discussion of the algorithms used in transient analysis in SPICE2 type simulators. The next and the final section discusses the algorithms used by the state-of-the-art simulators that provide remarkable

speed improvement in nonlinear transient analysis using novel algorithms in contrast to the conventional algorithms presented above.

2.2.4 Third Generation Simulators

The third generation simulators came up with the solutions of the two severe drawbacks present in circuit simulators. First the iterative analysis algorithms used in them brings down the computational cost of nonlinear transient analysis from $O(N^{(1.1-1.5)})$ as in circuit simulators to $O(N)$, where N is the number of circuit nodes.

Secondly using equation decoupling (first introduced in the timing simulators), these simulators treat each of the circuit nodes separately and use different timesteps at different nodes to speed up the entire simulation process. These two advantages make them superior to the reliable and accurate second generation simulators.

The review of this class of simulators can be done in two ways. According to the first approach, the simulators could be brought up by name and the methods used in them could be mentioned. The other way is to bring up the characteristic algorithms and describe them. The second approach seems to be more relevant here, algorithms being more important than the simulator itself. Hence this section concentrates on the characteristic algorithms used by the third generation simulators to speed up the transient analysis.

Iterative Timing Analysis

In iterative timing analysis (ITA), as in circuit analysis, a numerical integration scheme converts the differential equations to nonlinear equations; but the method used to solve these nonlinear equations are different; these are nonlinear Gauss-Jacobi and nonlinear Gauss-Seidel. These nonlinear counterparts of the two well known iterative methods (Gauss-Jacobi and Gauss-Seidel) [Kreyszig, 1988] operate on the nonlinear equation level, i.e. the nonlinear equations formed after the nonlinear integration step are solved by the NR method and the solved values are iterated till convergence.

The Gauss-Seidel algorithm can be more readily discussed from our previous discussion of timing simulators. As described before, in a timing simulation, only one relaxation iteration is done after the equation decoupling step. In the nonlinear Gauss-Seidel approach however this relaxation iteration is carried until convergence after the equation decoupling step. Nonlinear Gauss-Jacobi differs from nonlinear Gauss-Seidel only in one aspect. Referring to the Equation 2.5, v_i is calculated using node voltages ${v_1}^t, \ldots, {v_{(i-1)}}^t, {v_{(i+1)}}^t, \ldots, {v_N}^t$ in contrast to the use of ${v_1}^{(t+\Delta t)}, \ldots, {v_{(i-1)}}^{(t+\Delta t)}$, ${v_{(i+1)}}^t, \ldots, {v_N}^t$ as in Gauss-Seidel.

These iteration methods account for the reduced computational cost of third generation simulators. The computational cost of both these methods are

$$\textit{Computational cost} \sim O(n) \tag{2.18}$$

where n is the number of nodes.

Since they are iteration methods, their convergence properties are important to us. These nonlinear methods use NR iteration as discussed in the last paragraph. Say, the set of linear equations formed after the NR iteration is of form $A\mathbf{v} = b$. Then the presence of diagonal dominance in the matrix A gives a sufficient condition for the convergence of both of these methods. Also their rate of convergence is at least linear [Newton and Sangiovanni, 1984].

Let's note a salient point about these algorithms. These methods belong to a class of *relaxation* techniques, i.e. they relax the accuracy of solution to speed up the simulation process. Looking back, timing simulation was the first in using the relaxation technique, with the use of equation decoupling.

Waveform Relaxation Techniques

Both timing analysis and ITA are based on the application of relaxation techniques at the nonlinear equation level. Waveform Relaxation (WR) applies relaxation techniques at the differential equation level. As we can see at this level we are dealing with waveforms e.g. $v_1(t)$, $v_2(t)$; and the relaxation is at the waveform level, hence the name waveform relaxation.

The procedure can be explained in a simple fashion by considering two equations

$$\begin{aligned} \dot{x_1} &= f_1(x_1, x_2, t), \qquad x_1(0) = x_{10} \\ \dot{x_2} &= f_2(x_1, x_2, t), \qquad x_2(0) = x_{20} \end{aligned} \tag{2.19}$$

The first equation is solved assuming x_2 to be constant, hence reducing it to a differential equation in $x_1(t)$. The calculated value is used in the second equation and the process is repeated. These disjointed (now single variable) differential equations are then solved using implicit integration and NR iteration as in conventional simulators. As in all relaxation methods, the convergence of this method is linear [Lelarasmee et al., 1982].

Event Scheduling and Selective Trace

The principle has been already discussed in context to the logic simulators. In the third generation simulators, this method is applied at all types of simulators, even circuit simulators.

Table look-up models

The device models are stored as look-up models rendering algebraic computation unnecessary at every step of the simulation process. We have discussed this approach already in the context of timing simulators.

Before summing up this section, let us note one point - these third generation simulators carry on one limitation that was inherent in timing simulators. They still assume a capacitor connected between every node and the ground or another node. This property is generic in simulators using equation decoupling techniques. But MOS circuits, for which these simulators are mainly intended, provide that by virtue of having parasitic capacitances.

Hence the assumption is justified and the decoupling method that evolves from this assumption provides fast enough simulation to prove the worth of the assumption.

This concludes the review of the third generation simulation algorithms and also effectively concludes the review section of this chapter. The development of the review section had been serial. The more refined simulation techniques have been presented after the presentation of the background material. This effort is aimed at making the research more understandable to the reader.

2.3 Concluding Remarks

The simulation techniques were already matured in the 1970's with the development of reliable and accurate simulators like SPICE2 and ASTAP. Research afterwards sought to speed up the simulation process as the circuit complexity continued to grow. The review followed the continuous development of simulation techniques from TAP developed in 1960's to SPLAX in 1990's, pointing out the salient features of each development. After familiarizing the reader to the subject of VLSI circuit simulation in this chapter, this thesis will now concentrate on the pulsed analog neural networks for which the proposed simulator is built.

Chapter 3

Pulsed Neural Networks and Our Approach

3.1 Introduction

Artificial neural networks are biologically inspired. In a nervous system, *neurons* act as the processors and the *synapses* work as the connecting units. When excited above a predetermined threshold, biological neurons fire a pulse and though this processing power seems extraordinarily simple, when collectively considered, gives rise to a computing power far beyond the reach of even the most modern computers. Artificial neural networks imitate these biological networks in search of that enormous computing power.

In an artificial neural network (from now on referred to as a *neural network*), neurons are simple processors and synapses are the weighted connections by which the neurons control each other's activity. The synapses can be either *excitatory*, used by one neuron to excite another, or *inhibitory*, used by one neuron to inhibit another excited neuron. Fig. 3.1 shows the

conventional modeling of this topology.

Here, the synapses are modeled as simple multiplication factors to the inputs of a neuron. Excitatory synapses are represented by positive multiplication factors, whereas inhibitory synapses are represented by negative multiplication factors. Functionally, the neuron is divided into two parts, a summer and an activation unit.

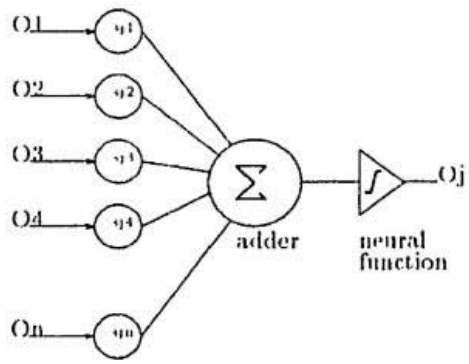


Figure 3.1: A neuron and the connected synapses

Assuming the inputs to be I_i $(i = 1, 2, \ldots, n)$, which could be outputs of other neurons, and the synapses to be multiplication factors of value S_{ji} to the inputs, neuron j adds the product terms to produce a sum of $S_j = \sum_{i=1}^{n} S_{ji} I_i$. The activation unit can be modeled by a threshold function or a sigmoidal function of the form

$$O_j = \frac{1}{1 + e^{-(S_j + \theta_j)/\theta_0}} \tag{3.1}$$

though other kinds of functions are also used. Note that the neurons us-

ing threshold functions will produce digital outputs, whereas neurons using sigmoidal function will provide continuous, analog output for a range of S_j values.

An interesting property of these networks is their learning ability. The multiplication factors (by which the synapses are modeled) are traditionally called the *weights* of the synapses and *learning* takes place by changing these weights. Through the learning process, the neurons learn to produce desired outputs in response to an arbitrary input, which gives these networks their power.

Neural networks can be broadly classified into three categories - supervised learning networks, unsupervised learning networks and associative memories. In a supervised learning network, the network is presented with a chosen set of inputs (called *learning samples*) and also the desired outputs in response to those inputs. Following a learning rule, the network continues to learn till the stage at which it does not make any more mistakes in predicting the outputs. *Backpropagation Nets* are one example of this kind of network. After the learning is over, the network is put into production i.e. random inputs are now presented and the network guesses the output with the help of already learned weights.

The unsupervised learning process differs from the supervised learning process characteristically by not presenting any desired output (to the network) along with the learning samples. The aim of the network here is to

group the inputs into meaningful clusters. In search of this goal, it uses *mutual inhibition*, i.e. the most activated neuron tries to suppress the others to identify the input to be one of its own cluster. The representative networks are *maxnet*, *ART* and the *Kohonen's network*.

Associative memories traditionally can be separated from the other two classes by the use of fixed weights i.e. they hardly learn. These memories can be autoassociative or heteroassociative in nature, i.e. when an input presented is closer to x_i than any other stored patterns, it can produce x_i as output (autoassociative) or y_i as output (heteroassociative), where y_i is the associated response to x_i. The examples are *hopfield nets*, *matrix associative memories* etc.

Later, while discussing the implementation of specific neural networks using the proposed topology, some of these networks are discussed at a greater length. Here, however the concentration is on implementations of neural networks in hardware using a pulsed analog approach rather than the characteristics of a specific network. The Edinburgh university VLSI neural research group has made the most significant breakthroughs in this area and hence their approach is reviewed first to acquaint the reader with the developments in this field over the years. Next, our proposed topology of pulsed analog implementation of neural networks is presented at length and the salient characteristics of it will be highlighted. The simulator described in the next chapter exploits these characteristics extensively.

3.2 Review

In a biological neural network, the neurons communicate with each other through *action potentials*, which are pulses. A biological neuron consists of four major parts – cell body, axon, dendrites and presynaptic terminals. The action potential is generated by the *cell body* and is transmitted through the *axon* that in turn divides into *presynaptic terminals* and transmits this pulse through synapses to the postsynaptic terminals (*dendrites*) of other neurons. The pulsed neural networks use this idea to represent neural outputs as pulses in contrast to the conventional analog or digital neural outputs. The advantages of this approach have been discussed before.

In the last few years, under the leadership of Dr. A. F. Murray, the Edinburgh University group has looked into several neuron and synapse circuit models for efficient implementation of neural networks in pulsed analog form and also at the learning aspects of the neural networks built using this approach. In this section the progress made by them in these areas over the past few years will be summarized.

Implementation of neural networks in hardware aims to achieve a few goals. Minimization of synaptic area is one of them. Another goal is to implement the multiplication and the addition process (Refer to Fig 3.1) in an efficient manner, i.e. reducing the neuron standard cell area. The following discussion continualy refers to these issues to evaluate the effectiveness of the design decisions made by the Edinburgh group.

3.2.1 Synapses

Synapses form one of the two basic building blocks in any neural network implementation. Since each neuron in a network uses several synapses to connect to the other neurons, the synapses dominate by numbers in any neural network. Hence, an efficient and compact implementation of them is an absolute necessity. The synapse design changes made by this group over the years reflects their effort in achieving this goal.

The initial design of their synapses used registers to store the synaptic weights digitally [Murray and Smith, 1987]. The first bit of the weight defined the nature of the synapse, i.e. whether it is excitatory or inhibitory. The purpose of the other bits can be explained as follows. Say, for a particular synapse the other bits of the weight are 110; then if that synapse receives input pulses at a frequency of f, it was designed to pass ($1.2^{-1} + 1.2^{-2} + 0.2^{-3}$)=75% of those pulses. The drawback of this design was the use of digital registers that used up a large amount of area thus preventing one to implement more than 100 synapses per chip and the design subsequently had to be rejected.

The synapses that replaced them used a time-modulation technique [Murray and Smith, 1988]. The technique can be explained as follows. Let the inputs to the synapses be pulses of fixed width dt and of frequency f. In this implementation, the pulse width dt gets linearly multiplied by the synapse weights S_{ij}, where $0 \leq S_{ij} \leq 1$. Hence the pulses that appear at the postsy-

naptic terminal of the neuron are of variable width $S_{ij}.dt$ and of frequency f. These synapses are elegant compared to the clumsy controlling mechanism of the previous design, and also allow one to use an efficient current summing techniques (described later) on their outputs, but each of them used 11 transistors so they were still uneconomical considering the enormous number of synapses present in a network.

Their newest design uses a four transistor synapse that implements the product term $S_{ij}I_i$ using the MOSFET equation in the triode region. The squared term in the MOSFET equation is eliminated by proper choice of transistor $\frac{W}{L}$ ratios and the drain and gate voltages. These synapses also allow the use of a current summing technique and 15,000 of them can be implemented in a single chip [Murray, 1991].

All these designs share some common characteristics. They are androgynous, i.e. they are aimed to implement both excitatory and inhibitory synapses. They are linear in nature and also auto-scaling, i.e. they allow any number of synapses to be connected to a particular neuron.

3.2.2 Neurons

The neuron design changed continuously to accommodate the changes in the synapse design. Neurons are relatively fewer in number in a neural network and hence are allowed to be larger to use more efficient multiplication and summation techniques.

The first design of their neuron was for the digital synapses and it worked

by integrating the pulses coming from the gated synapses and then using an odd number of inverter delays to produce output pulses whose frequency represented the neural activation [Murray et al., 1987]. Their second design was for time modulating synapses. A simple voltage controlled oscillator was used as the neuron in this case. The voltage accumulated at the postsynaptic node, by current summing of the time modulated pulses, was used to control the firing rate of the neuron [Murray et al., 1988]. In the most recent design, an op-amp circuit is used as the neuron for the four transistor synapses described before. Summation of multiplied terms is easily achieved by the op-amp circuit, which incidentally also provides the sigmoidal nonlinearity [Murray, 1991].

3.2.3 Programmability and Learning

The synapses designed by this group were always programmable in the sense that an off-chip learning scheme could be used to train them. The first type of synapse used registers whereas the second and third design had capacitors present at the weight terminal so that the weights could be dumped and periodically refreshed for use in learning.

Murray [Murray, 1992] describes a successful learning scheme for these networks. Though backpropagation is the most popular algorithm for learning, it requires a complex weight updating scheme and hence is not suitable for hardware on-chip learning. Consequently, a variation of backpropagation, with a simple weight updating scheme, has been chosen as the learning

algorithm.

The pulsed approach uses analog circuits that inherently suffer from noise problems. Contrary to popular belief, during the learning phase this noise is reported to enhance the learning [Murray, 1991]. In backpropagation type algorithms the weights are taken to the global minima of the error space by guiding them with the use of , learning rule. During this process, the weights tend to get stuck at the local minimas. Injection of noise reportedly helped the weights to survive the local minimas in the error space thus effectively assisting the learning.

Contrary to this conventional approach of using linear synapses (to stick to the model as in Fig. 3.1), our approach is to use nonlinear synapses. Hence, the relationship of the contribution of our synapses to the weight voltage is nonlinear. Also our excitatory and inhibitory synapses have been designed separately, i.e. they are non-androgynous. The synapse and the neuron circuits and our proposed topology is the focus of the next section.

3.3 Our topology

Figure 3.2 shows the transistor level design of our synapses. The excitatory synapses use four n-transistors, whereas the inhibitory synapses use two. The input $V_{ex}(V_{in})$ for excitatory(inhibitory) synapses is a series of pulses, whose frequency represents the input excitation. The weight voltage V_{wt} determines the effectiveness of the synapses. The other inputs to the synapses can be

best understood in context to the topology that is presented in Fig. 3.3.

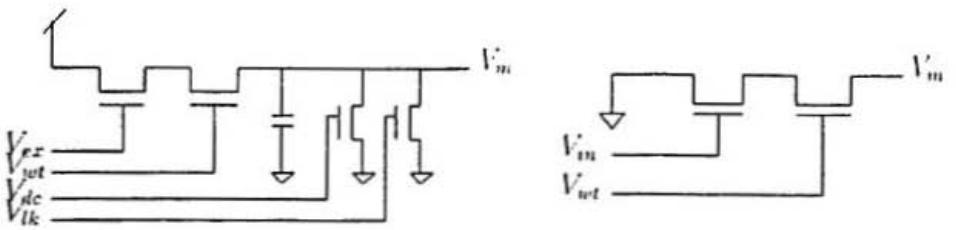


Figure 3.2: (a) The excitatory and (b) the inhibitory synapse circuit

Fig. 3.3 shows the synapses, which are all excitatory in this diagram, connected to the V_{act} node of the neuron. The voltage at the V_{act} or the summing node resembles the membrane voltage of the biological neural cell. This voltage arises in a biological neuron because of the separation of charges across the membrane that acts as a barrier to the diffusion of ions.

The neuron is modeled as a threshold unit; when the membrane voltage $V_m = V_{act}$ crosses the threshold voltage V_{thr}, it fires a pulse O_j. In our topology, the neuron along with firing the output pulse, also fires a discharge pulse of longer duration (than the output pulse) to bring back the membrane voltage to zero.

The membrane effectively integrates the charge dumped by the synapses with its membrane capacitance. This capacitance could be implemented as a lumped component in front of the neuron. Once implemented like this, the number of synapses connected to a neuron is limited because of the problem of saturation.

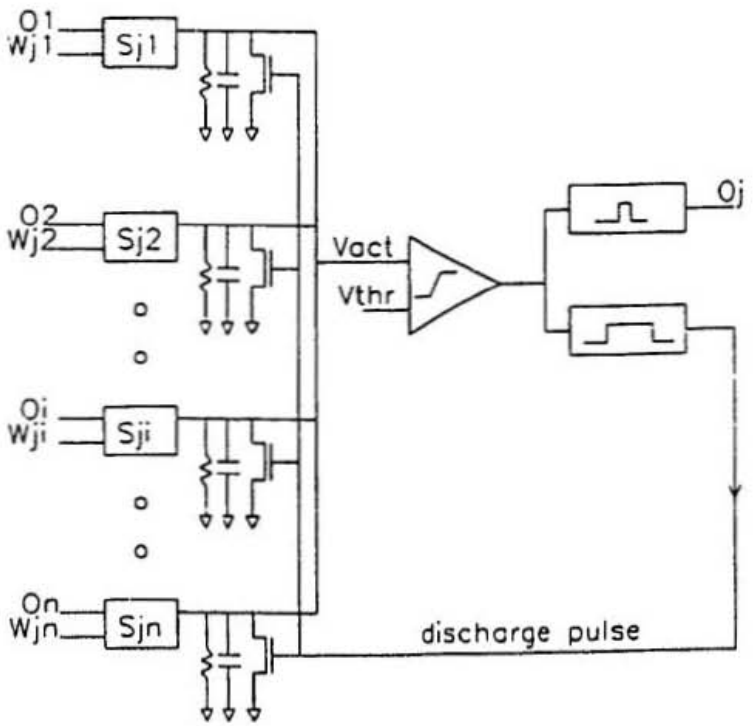


Figure 3.3: Block diagram of the proposed neural network

This problem is taken care of in the proposed architecture by distributing the membrane capacitance over the synapses. Hence this network is auto-scaling, i.e. any number of synapses can be connected to a specific neuron.

The V_{dc} terminal of an excitatory synapse is connected to the discharge pulse output of the neuron to which it is feeding. Scalability is again achieved by distributing the discharge transistors over the synapses. The discharge transistor is also designed to be wider along with a wider discharge pulse from the neuron to guarantee almost instantaneous discharge of the membrane voltage.

In a biological neuron, the output O_j is fired when the membrane voltage rises enough to open the voltage-gated channels to rush in positive ions. Along with the gated channels, the membrane also has some non-gated channels that constantly leak charge. This phenomenon has been modeled in the excitatory synapse by a leakage resistance that is implemented as the transistor with the gate voltage of V_{lk}.

Along with the type of neuron used in Fig. 3.3, we also use two kinds of input neurons. The pulsed analog implementation requires pulsed inputs whereas conventional neural networks use digital or analog voltage inputs. The input neurons are used to provide us with pulsed inputs in response to user-defined voltage inputs. The non-inverting input neurons fire at a higher rate with increase in voltage in contrast to the inverting input neurons firing at a lower rate with increase in voltage. These neuron circuit designs and

their output specifications can be found in the thesis work of D. Bhattacharya [Bhattacharya, 1991].

From the above discussion it is apparent that our design is cell-based; standard cells have been designed to implement the synapses and neurons present in a given neural network. The input neurons are designed to be used in the input layers, whereas the standard neurons are to be used in the hidden and output layers. The leakage voltage of the excitatory synapses and also the threshold voltages of the neurons are provided globally.

Several circuits, e.g. *Maxnet*, *associative memory* etc. have been simulated using this topology and two chips have been built to date in CMOS3DLM (a 3 micron CMOS technology) to confirm our designs. The first chip implements the five standard cells – excitatory and inhibitory synapse, standard neuron and the inverting and non-inverting neuron. The second chip implements a content addressable memory (CAM). The testing of these chips have shown favorable results.

In this section our proposed pulsed analog topology is outlined and the salient characteristics of it have been discussed. Our proposed synapses are nonlinear and we believe that the learning, being essentially a tuning process, will take care of any nonlinearity present in the system. The synapses used are non-androgynous and have been optimized for their particular function. The proposed network is auto-scaling, hence the number of synapses that can be connected to a neuron is not limited by any constraint. Lastly, the

approach is cell-based to allow users to implement a neural network using these cells without bothering about the transistor level details.

3.4 Conclusion

This section introduces the reader to the pulsed analog approach of implementation of neural networks. A brief review of neural network theory has been followed by a conventional pulsed analog approach used for implementation of them. Finally our approach has been presented and the differences have been emphasized. After reviewing the simulator design and the pulsed analog approach in the last two chapters, the next chapter will present the reader with the design level details of the simulator that has been designed for our proposed pulsed analog topology.

Chapter 4

Simulator Design

4.1 Introduction

Our motivation for the development of this simulator is fast behavioral simulation of our pulsed analog topology. Commercial circuit simulators like HSPICE takes around 40 minutes for a single simulation run of even a small neural net implementation of only 48 synapses and 3 neurons, which is frustrating at the design stage. Against this backdrop, our aim is to provide a reasonably accurate prediction of the network behavior along with a radically improved simulation time. The simulator PULSE that emerged as a result of pursuing this goal performed the simulation of the above circuit in 6 seconds.

The guidelines followed in any simulator design are ease of use, efficiency and simplicity [Nagel, 1975]. These guidelines are also followed in the design of PULSE and are elaborated below :

- *Ease of use* refers to the user interface. User input to the simulator has to be in simple form and the output has to be easily comprehen-

sible. Keeping this in mind, PULSE requires user to submit inputs in SPICE format and generates a circuit connectivity description by its own *netlister* from user-drawn *Cadence* schematics. The output is provided in both textual and graphical forms.

- *Efficiency* refers to the dollar cost of the simulation. Typically 50 times speed improvement is provided by PULSE in medium sized circuits which reduces the design cycle time considerably, thus increasing design efficiency.

- *Simplicity* refers to the clarity of the design so that it can be readily modified, enhanced and supported. In this respect, modularity had always been a prime design goal in PULSE. Addition of new elements and enhancement of existing models is easy in PULSE. Simple algorithms have been chosen when the required complexity did not pay off in commensurate improvement of accuracy or speed.

This chapter presents the design details of PULSE. The simulation process in PULSE is first discussed and the various steps are outlined. The netlister development, for the generation of a connectivity description from a user drawn schematic, is then described. Modeling of the synapses and the neurons is described next. Finally, the algorithms used in the simulator are discussed and their choice is justified.

4.2 The Simulation process in PULSE

PULSE performs a *transient analysis* i.e. voltage values are calculated for a user-defined time interval $(0, T)$ at timesteps of Δt, which is again user-defined. Other than these parameters, the simulator also requires definition of input stimuli and specification of the outputs to be provided as simulation results. Circuit connectivity description is also an essential input.

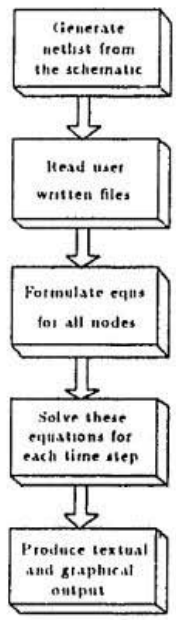


Figure 4.1: PULSE simulation steps

The simulator contains a custom-built flat netlister [1] [2]. This netlister

[1] A *netlist* is a textual description of the connectivity of a given circuit.

[2] A flat netlister works as follows. If an op-amp appears thrice in a circuit and the op-amp contains 50 transistors, the connectivity of all those 50 transistors will be provided every time the op-amp is encountered by the netlister.

allows a user to draw the neural net schematic in Cadence using custom-built symbols of synapses and neurons, which it then converts to a netlist (conforming to SPICE format). This improves the user-friendliness of the simulator since a schematic is easier to provide for a user than a textual description of the connectivity of the circuit.

Once the netlist is generated, the simulator reads a user written file *pulse.inp* to find the stimuli provided by the user. Another user-written file *pulse.sim* specifies the time-interval T and the timestep Δt of the transient analysis. Lastly, the user dictates the outputs that are needed from the simulator by writing a file called *control*. The formats of these files are shown in the appendix.

These input files are read by the simulator to enter a setup phase. In this phase, it formulates equations for all the nodes present in the circuit and writes a C program defining these equations and the algorithm to be used to solve them. In the analysis phase, this generated file is finally compiled along with the simulator-defined models to produce the final output according to the user requirements.

Among the steps shown in Fig. 4.1 the first step requires the development of a custom-built netlister which is described next.

4.3 Development of the Netlister

While designing a circuit simulator, a designer has to always think about the interface to be provided to a user. Here, at Memorial University of Newfoundland, *Cadence* is used for the complete development of an IC starting from its circuit design phase. Cadence, a proven VLSI package, offers various tools that can be used to improve the simulator interface. The *Flat Netlister (FNL)* is one of them.

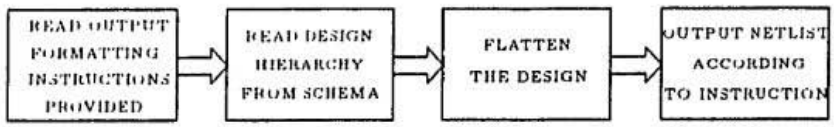
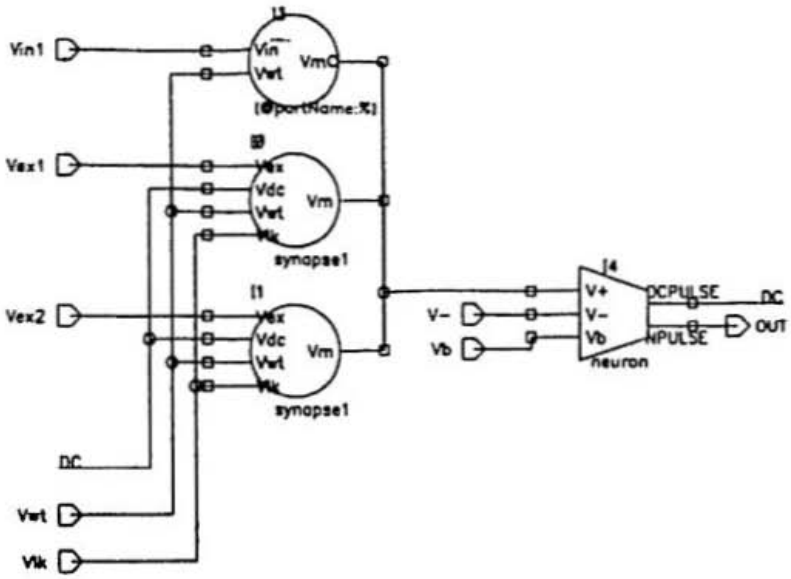


Figure 4.2: How FNL works

FNL provides a generic flat netlist that can be formatted according to the needs of a simulator. This custom-made netlister is built by exploiting that feature of FNL. The above diagram illustrates the process.

A program was written in the Symbol/Simulation Library Generator (S/SLG) language specific to Cadence [Cadence OSS Manual] for the formatting of FNL according to PULSE specific format [1]. The next step was the creation of the symbol library. The symbol for each of the neurons and synapses were created using Cadence symbol editor. Lastly, an S/SLG file was written for defining the netlisted output for all five library elements [2].

[1] The program appears in Appendix E.1
[2] The program appears in Appendix E.2




```
// Netlist Generated by PULSE Netlister //
*net0=/DC
*net1=/Vex1
*net2=/Vex2
*net3=/Vin1
*net4=/Vwt
*net5=/Vlk
*net6=/V-
*net7=/Vb
*net8=/OUT
*net9=/I0.Vm
*neuron(0)=/I4;
SN0 9 6 7 0 8 ;
*insynapse(1)=/I3;
IS1 3 4 9 ;
*exsynapse(2)=/I1;
ES2 2 0 4 5 9 ;
*exsynapse(3)=/I0;
ES3 1 0 4 5 9 ;
```

Figure 4.3: Sample output of the netlister

These three steps constituted the netlister development process. The netlister is implemented in such a way that it can be run inside Cadence. Fig. 4.3 shows a sample output of the netlister. A simple circuit of two excitatory synapses and one inhibitory synapse connected to a neuron has been chosen for this example.

The netlister development process for PULSE is outlined in this section. More detailed discussion of this topic will involve details of the Cadence simulation process, which are unimportant here. The details can be found in [Banerjee, 1993]. At this point, we are done with the first step of Fig. 4.1. Before proceeding with the next steps, we will digress for a while to discuss the way the building blocks, the two synapses and three neurons, are modeled in this simulator to enhance the simulation speed.

4.4 Modeling

Modeling of the components present in the intended circuits is a major part of any simulator development. Here the components present are five – excitatory synapse, inhibitory synapse, standard neuron, inverting input neuron and non-inverting input neuron. Our approach is not to perform a transistor level simulation of them but to build a fast and simple macromodel for each. The transistor level simulation details obtained from HSPICE level 3 simulation details are then incorporated in these macromodels to keep them fairly accurate. The modeling of each of these components is now separately

discussed.

4.4.1 Excitatory Synapse

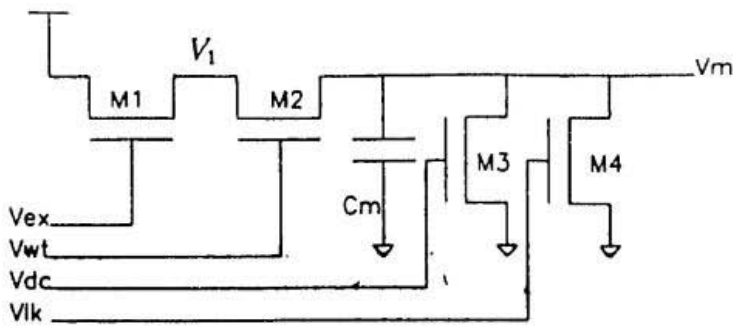


Figure 4.4: The excitatory synapse

The modeling of the excitatory synapse will provide us the amount of current dumped by the synapse in response to a given set of voltages V_{ex}, V_{wt}, V_{dc}, V_{lk} and V_m. The input-output characteristics for this building block can be derived as follows.

According to Fig. 4.4 and our topology

- Current flowing through $M1$ and $M2$ is the same at any point of time.
- Leakage through transistor $M4$ is a function of V_m.
- V_m instantly drops to zero when V_{dc} goes high.

Also I, the current dumped by the synapse, is given by

$$I = I_1 - I_{dc} - I_{lk} \tag{4.1}$$

where, I_1 is the current flowing through $M1$ or $M2$ when V_{ex} is high.

V_m is always connected to a neuron input. With the neuron threshold being $\simeq 1.5V$, V_m hardly goes over $2.5V$ before a discharge pulse brings it down to zero. Hence, when V_{ex}, the pulsed input goes high, $V_{ex} - V_1 > V_{th}$, where V_{th} is the threshold voltage of the N-transistors and V_1 is the source voltage of transistor $M1$. Thus, whenever V_{ex} is high, $M1$ is in saturation.

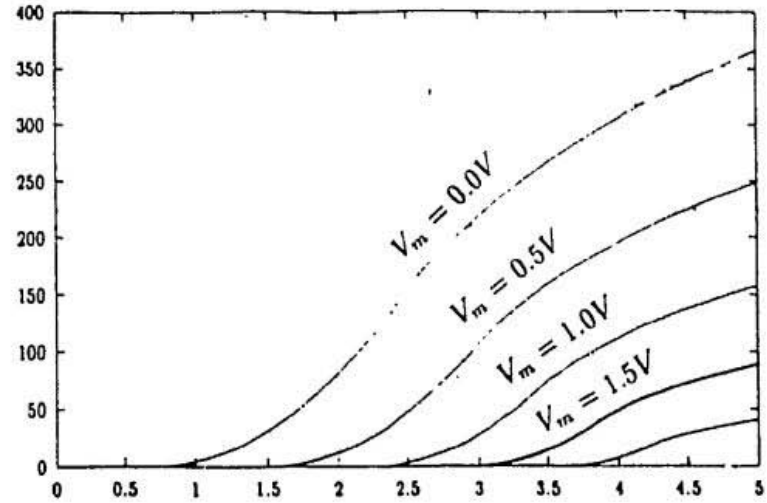
However, $M2$ can be in the linear or saturation region, depending on voltage V_1 for a given V_{wt}.

$$\begin{array}{ll} V_1 > V_{wt} - Vth & : M2 \text{ in saturation} \\ V_1 < V_{wt} - Vth & : M2 \text{ in linear region} \end{array} \tag{4.2}$$

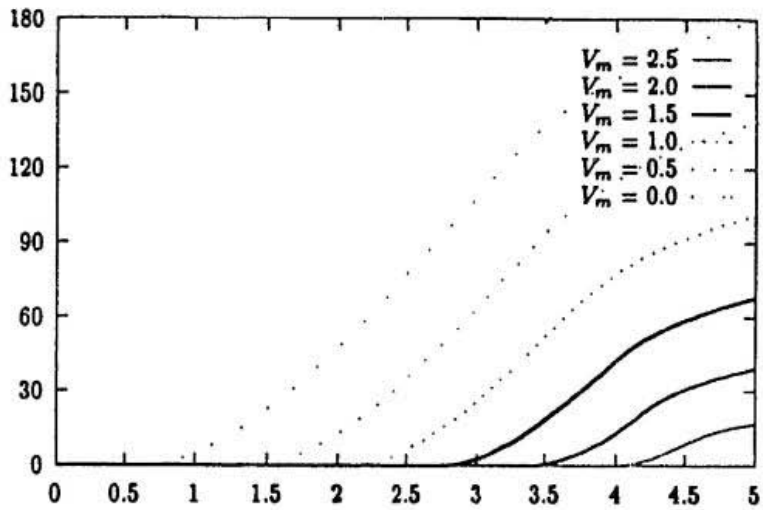
Hence solving for current I_1 in Eqn. 4.1 will involve first determining the condition of $M2$ according to Eqn. 4.2 and then equating the appropriate current equation for $M2$ with current through $M1$ in saturation region. V_1 can be solved from the resulting equation by iteration (the equation being fairly involved) and hence I_1 can be determined.

Following the above procedure, a transfer characteristics can be obtained by varying V_{wt} from 0-5V and finding current I for different values of V_m (assuming V_{dc}=0). The Shichman-Hodges SPICE level 1 model [Geiger et al., 1990] is used for transistors to provide a reasonably accura e simulation. The transfer characteristics thus obtained are shown in Fig. 4.5(a). The parameters for this SPICE level 1 model are taken from the CMOS3DLM databook [1].

[1]CMOS3DLM is a 3μ CMOS process used by us for our chip fabrication.



(a)



(b)

Figure 4.5: Plot of exsynapse current I(in μA) against V_{wt}(in $Volts$) (a) HSPICE level 1, (b) HSPICE level 3

When the same circuit was simulated in HSPICE using level 3 MOS models for verification, the characteristic obtained was as in Fig. 4.5(b). The value of currents obtained were almost half of that in Fig. 4.5(a) at a given (V_{wt}, V_m) point. Obviously it reflects the sophistication of the HSPICE level 3 model as compared to the Shichman-Hodges model.

The characteristic in Fig. 4.5(b) has been incorporated in our excitatory synapse macromodel in the following way. Fig. 4.5(b) has been curve-fitted with a 13-th degree polynomial in V_{wt}. Hence,

$$I = a_0 V_{wt}^{13} + a_1 V_{wt}^{12} + \ldots + a_{12} V_{wt} + a_{13} \tag{4.3}$$

These coefficients a_i were calculated for different V_m's. The resultant values were again curve-fitted with a 13-th degree polynomial in V_m. Hence a_0, a_1 etc. were again expressed in a series of form

$$a_i = b_0 V_m^{13} + b_1 V_m^{12} + \ldots + b_{12} V_m + b_{13} \tag{4.4}$$

These equations Eqn. 4.3 and Eqn. 4.4 constitute the major part of the macromodel for excitatory synapses. The degree of the polynomials had been chosen by visual inspection of the matching present in the model generated characteristics and the HSPICE level 3 characteristics. Fig. 4.6 shows both of these characteristics plotted on the same graph.

Note that the current thus obtained only accounted for the first and the third term in Eqn. 4.1. During the time the discharge pulse gets high, I_{dc} also appears into the picture. The voltage V_{dc} being constant (=5V), for that

period of time, current I_{dc} has also been modeled as a 13-th order polynomial in V_m in the same way.

$$I_{dc} = c_0 V_m^{13} + c_1 V_m^{12} + \ldots + c_{12} V_m + c_{13} \tag{4.5}$$

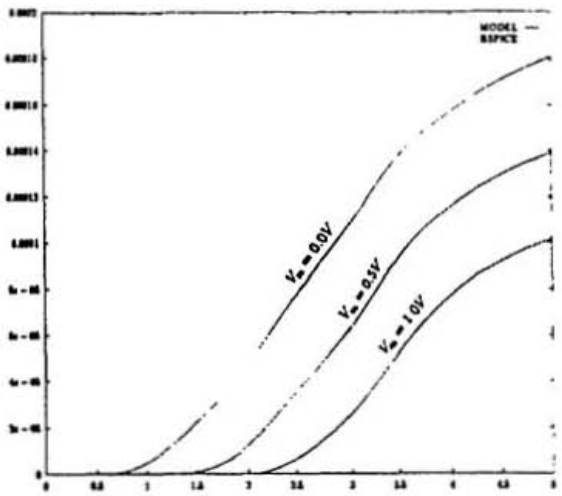


Figure 4.6: The model generated characteristics and the HSPICE characteristic (Fig. 4.5(b)) plotted on top of each other. They totally overlap as the graph shows.

Hence, Eqn. 4.3, 4.4 and 4.5, when combined together, constitutes the final macromodel of excitatory synapse. The model being only the combination of a maximum of three power series, the evaluation is fast. Also, the accuracy is inherently incorporated in this macromodel by assuring its conformance with the HSPICE level 3 simulation results.

4.4.2 Inhibitory Synapse

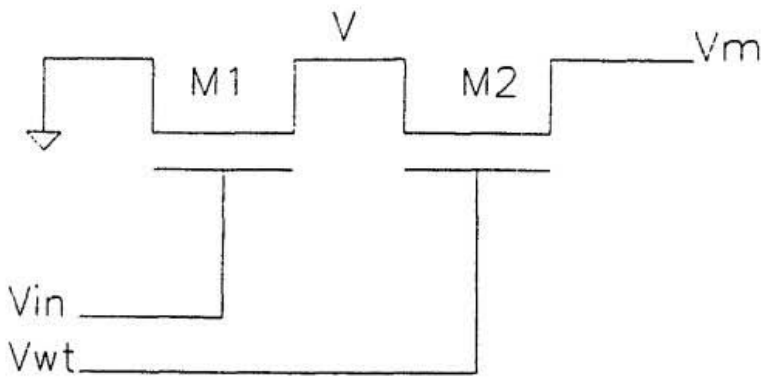


Figure 4.7: The inhibitory synapse

The inhibitory synapse can be modeled using the same approach. In contrast to the excitatory synapse dumping current, the inhibitory counterpart sinks current. Looking at Eqn. 4.1, we can note that the same equation can be used for modeling of inhibitory synapse and only the last two terms in that equation will be absent in this case. Following that observation, the current sinked had been modeled using the same equations 4.3 and 4.4. HSPICE level 3 simulation was once again used for calculating the proper coefficients a_i's and b_i's for this macromodel.

4.4.3 Neuron

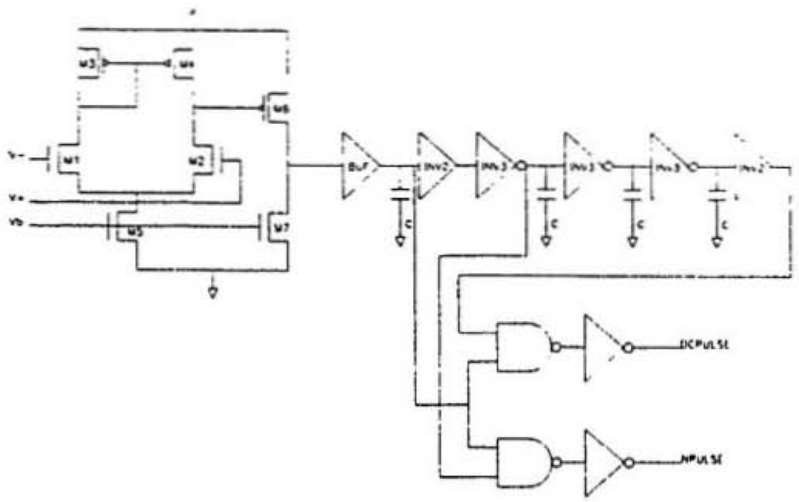


Figure 4.8: The standard neuron

Fig. 4.8 shows the circuit schematic of the neuron. The first part of the circuit containing transistors M1-M7 is the comparator block whereas the rest of the circuit provides the delays required to produce the neuron output and discharge pulse.

The neuron is modeled as a comparator. When the input voltage $V+$, which is really the V_m node of connecting synapses, crosses $V-$, it fires two pulses. The discharge pulse goes back to the excitatory synapses to discharge V_m and the output pulse models neuron excitation.

In this simulator, the pulses have been modeled as trapezoids. Their rise and fall time, delay and on times have been noted from their HSPICE level 3 simulation and are provided in this model. Table 4.1 shows these values in

nanoseconds.

Pulses	Delay	Rise	On	Fall
Output	9.8	1	6.5	1.1
Discharge	10.2	1.3	14.8	1.2

Table 4.1: Neuron output and discharge pulse details

Though ideally this neuron should fire when the input is just above the threshold, it doesn't necessarily behave this way (i.e. it doesn't always fire when the input is just above the threshold) in our topology. The comparator circuit responsible for the firing of this neuron is however known for its accurate trip voltages [Allen and Holberg, 1987]. This behavior (of neuron not firing) is suggested to arise from operating this circuit in our topology at a very high speed. In many cases the neuron inputs stay high over the threshold voltage for only a few nanoseconds. If this duration is 1-2 ns, and the input is just at V_{th}, the neuron doesn't fire.

This phenomenon can be modeled in PULSE simulation by using a higher threshold voltage input (in most cases, 0.2V more) than that is intended to be used in the physical circuit. An accurate modeling of this phenomenon is not essential as the exact timing of the pulses doesn't seem to be meaningful (refer to section 5.5).

4.4.4 Non-Inverting Input Neuron

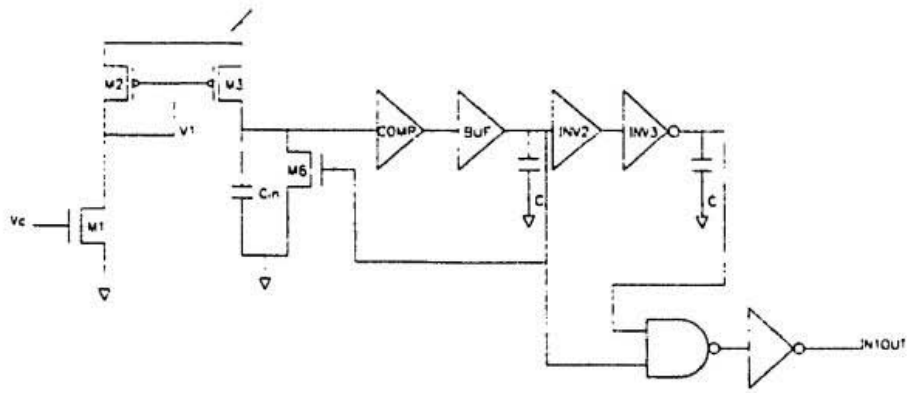


Figure 4.9: The non-inverting input neuron

The non-inverting input neuron circuit consists of the following stages. Transistors M1-M3 builds a ramp generator stage which charges the capacitor C_m. The ramp generator stage is ollowed by a comparator stage as in the standard neuron. When the voltage on the capacitor crosses the comparator threshold voltage, the comparator changes state. The delay stages following the comparator generates a pulse corresponding to this change.

This circuit fires pulses of increasing frequency as the input excitation increases. Hence the model is simply a single input, single output block with the details of its output pulses recorded from its HSPICE simulation.

4.4.5 Inverting Input Neuron

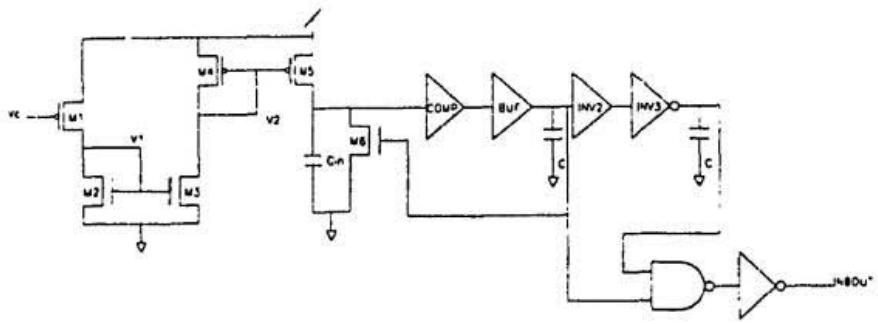


Figure 4.10: The inverting input neuron

Other than the presence of an additional current mirror at the input, the rest of this circuit remains the same as in the non-inverting input neuron. The resultant circuit fires pulses of decreasing frequency as the input voltage rises. Hence the model remains the same i.e. a single input, single output block as in its non-inverting counterpart.

This completes the modeling of the building blocks of PULSE. The next section takes a look at the core of PULSE, the algorithms used.

4.5 Analysis and Algorithms

```
Form equations at each node;
At each time-point
{ repeat Waveform Gauss-Seidel
  { perform implicit integration for neuron inputs;
    use plug-in models for neuron outputs;
  }
  until convergence;
}
```

Figure 4.11: PULSE analysis block

The operation of the analysis block in PULSE can be summarized as Fig. 4.11. PULSE, after the netlisting step, goes to the equation formulation stage. In this stage it forms two kinds of equations. Differential equations are formed for the neuron input nodes that goes through a numerical integration stage utilizing predictor-corrector based implicit integration (commonly known as functional iteration). The other group of equations are formed at neuron output nodes that are solved using plug-in models.

The implicit integration is performed for the first group of equations by treating these equations as single variable differential equations and this process is repeated using Gauss-Seidel iteration (the whole process is commonly known as the *Waveform Gauss-Seidel method*). This section discusses these algorithms in the light of their application to our topology and their implementation in our simulator. This section concludes with an example of the operation of this analysis block.

4.5.1 Equation Formulation

PULSE forms an equation for every node present (except for the input nodes) in the circuit at each time-point. The equations thus formulated can be divided into two subgroups - equations for neuron output nodes and equations for neuron input nodes. The synapses can only be connected from the output of one neuron to the input of another neuron or from the input (stimuli) nodes to the input of one neuron, hence these are the only two types of equations possible.

Out of these two subgroups, the equations for the neuron output nodes are straightforward. These equations define neuron outputs in terms of neuron inputs. The standard neuron is a comparator and the input neurons are modeled as simple VCOs. Hence these equations do not require any sophisticated methods for solving them and from now on we will not bother about them.

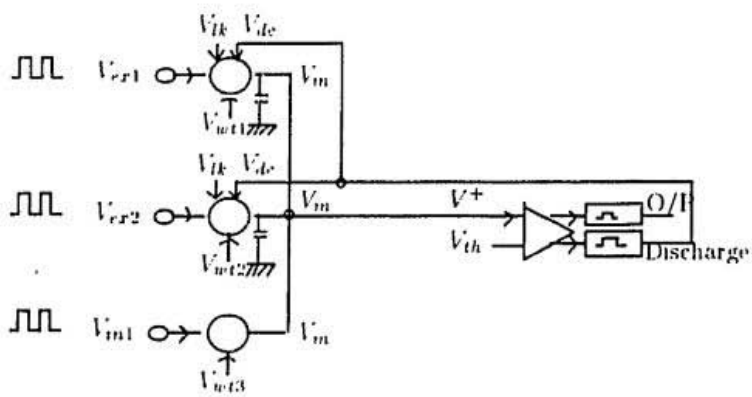


Figure 4.12: A typical connection at neuron input node

The second group of equations are formed at the neuron input nodes. These set of equations require more attention. Fig. 4.12 shows two excitatory synapses and one inhibitory synapse connected to the input node of a neuron. Let I_1 and I_2 be the current dumped by the excitatory synapses and I_3 be the current sinked by the inhibitory synapse. Then the equation at the input node of this neuron can be written as

$$I_1 + I_2 - I_3 = 2C\frac{dV_m}{dt} \tag{4.6}$$

Equations like 4.6 are assembled by the simulator for input nodes of every neuron. These groups of equations resemble the equations formed by the modified nodal equation formulation technique extensively used by the second and third generation simulators, and hence the method and the algorithms for solving these equations are well known (refer to the subsection 2.2.3). The way these proven algorithms apply to our specific topology and a comparison of their efficiency in this application forms the topic of the rest of this chapter.

4.5.2 Implicit Integration

Differential equations of the form of Eqn. 4.6 have to be solved at every time-point for finding the voltages at the neuron input nodes. Implicit integration is the first step in this process. As stated in the introduction of this section, PULSE uses the functional iteration method in this stage. This algorithm as applied to our topology is described below.

Recall that I_1, I_2, I_3 of Eqn. 4.6 are inherently functions of V_m (from the modeling approach of the synapses). According to the method of *waveform relaxation*, equations like 4.6 can be treated as a single variable differential equation in V_m [Lelarasmee et al., 1982] (more about this in the next section). Hence the group of equations like Eqn. 4.6 are essentially of the form

$$\dot{V}_m = f(V_m(t)) \tag{4.7}$$

Lets keep in mind that we are performing transient analysis. Hence, while solving the voltage at timepoint $(t + \Delta t)$, we already know the voltages at timepoint t and before. In this algorithm, an accurate predictor is first used to predict the value of V_m at time $(t + \Delta t)$, ${V_m}^P(t + \Delta t)$, using the value $V_m(t)$ and $V_m(t - \Delta t)$, which is already known.

$${V_m}^P(t + \Delta t) = V_m(t) + \frac{\Delta t}{2}[3f(V_m(t)) - f(V_m(t - \Delta t))] \tag{4.8}$$

This predictor is known as Adams-Bashforth predictor [Chua and Lin, 1975].

Since $V_m(t)$, $V_m(t - \Delta t)$ are already known and $f(V_m(t))$, $f(V_m(t - \Delta t))$ can be found out using them, ${V_m}^P(t + \Delta t)$ can be calculated from this equation. This predicted value is corrected by averaging the slope at points t and $(t + \Delta t)$. Let the corrected value be ${V_m}^1(t + \Delta t)$, calculated using the following equation

$${V_m}^1(t + \Delta t) = V_m(t) + \frac{\Delta t}{2}[f({V_m}^P(t + \Delta t)) + \dot{V}_m(t)] \tag{4.9}$$

On the right-hand side of this equation, $V_m(t)$ and Δt are known and $\dot{V}_m(t)$ can be found out from Eqn. 4.7. Hence $V_{mC}{}^1(t+\Delta t)$ can be calculated.

This correction process is then repeated until convergence [1]. The j-th iteration involves the step

$$V_{mC}{}^j(t+\Delta t) = V_m(t) + \frac{\Delta t}{2}\left[V_{mC}{}^{j-1}(t+\Delta t) + \dot{V}_m(t)\right] \tag{4.10}$$

At convergence, the value of $V_{mC}{}^j(t+\Delta t)$ provides the neuron input node voltage at that timepoint.

4.5.3 Waveform Gauss-Seidel Iteration

Waveform Relaxation and *Gauss-Seidel iteration* have been previously described in the literature review section. The waveform Gauss-Seidel is a combination of these two algorithms and is described in the next page in the form that was implemented in our simulator.

In this special case of the Waveform Gauss-Seidel method (i.e. when applied in our pulsed analog topology), only the node voltage that is to be solved appears as the differential term as can be seen in Eqn. 4.6 and all other node voltages appear inside the f_i term.

Application of this algorithm requires u to be a piecewise linear continuous input. In our case, u is the input pulses. Also C_i is independent of V_i in this case. Under these conditions, Waveform Gauss-Seidel iteration process is guaranteed to converge [White and Sangiovanni, 1986] and the rate

[1] The convergence criterion has been detailed in Appendix A

of convergence is at least linear [Newton, 1984].

Waveform Gauss-Seidel Algorithm for solving Eqns. of form Eqn. 4.6

The superscript k denotes the iteration count, the subscript i denotes the node number, and ϵ is a small positive number.
$k \leftarrow 0$
Assume initial condition $V_i = 0$; $i \in \{1, \ldots, n\}$
repeat {
  $k \leftarrow k + 1$
  foreach ($i \in \{1, \ldots, n\}$) {
    solve
      $C_i \dot{V}_i^k - f_i(V_1^k, \ldots, V_i^k, V_{i+1}^{k-1}, \ldots, V_n^{k-1}, u) = 0$
    for ($V_i^k(t); t \in [0, T]$)
  }
} **until** (all $\| V_i^k - V_i^{k-1} \| \leq \varepsilon$)

Figure 4.13: Waveform Gauss-Seidel Algorithm as in PULSE

Hence applying this algorithm, equations of the form of Eqn. 4.6 can be treated as single variable equations in V_m (refer to the subsection 2.2.4) and hence the claim made in the subsection 4.5.2, i.e. Eqn. 4.6 is of the form of Eqn. 4.7, is justified.

4.5.4 PULSE Analysis Block - Reviewed

The focus of this section is to explain to the reader the flow of the PULSE analysis block. For this discussion we will assume a circuit consisting of N nodes and m standard neurons (there can be any number of input neurons present in this circuit) .

Out of these N nodes present in this circuit, we are only concerned with n standard neuron input nodes and n standard neuron output nodes. The other $(N - 2n)$ nodes are circuit input nodes or input neuron output nodes and hence are already known from user provided stimulus values and plug-in models.

From the netlist, the connectivity of the $2n$ (standard neuron input and output) nodes (that we are interested in) is available in an arbitrary order. Hence the equations are also formed in an arbitrary order in PULSE. Say the first of these $2n$ equations evaluate the input node of neuron 1 (by using functional iteration). Also assume that the second equation evaluates the output node of neuron 3 (by using plug-in models). Now if neuron 1 input node depended on neuron 3 output voltage and neuron 3 output changed at this point of time, the input node voltage of Neuron 1 will be calculated incorrectly being unable to take into account the changed voltage of neuron 3 output.

This problem is circumvented in PULSE by passing these $2n$ equations through a Gauss-Seidel iterative loop. While passing through these loop, (in this case) the second iteration of the Gauss-Seidel loop takes into account the changed value of neuron 3 output correctly thus evaluating the correct neuron 1 input voltage. Extending this logic, when this Gauss-Seidel loop reaches convergence, all the nodes in this circuit are correctly evaluated independent of the order in which the equations were formed. This explains the presence

of the outer Gauss-Seidel loop in Fig. 4.11.

The flow of the PULSE analysis block and the function of the algorithms used in this block has been explained in this section. The next section explains the choice of the algorithms used in this block in comparison to the other algorithms available.

4.6 Discussion

Two algorithms are used in PULSE – the predictor-corrector based implicit integration and the waveform Gauss-Seidel iteration. Another crucial part in any simulator design, the equation formulation stage, is simple in PULSE, and provides equations in the same form as required in standard circuit simulators. Hence no complex algorithm is required at that stage.

Once the equations are formulated, the second step in the PULSE analysis is predictor-corrector based implicit integration. In conventional circuit simulation, other implicit integration algorithms (such as *backward euler*) followed by Newton-Raphson iteration algorithm serve the same purpose. The reason the predictor-corrector based algorithm is not used (in conventional simulators) is its inherent limitation on timestep [1] (to assure its convergence) that can be used in transient analysis. In Appendix A, this limitation of timestep has been calculated for our topology. The worst-case timestep limit is tabulated below which has been found to be dependent on the ratio

[1] In this discussion, if not stated explicitly, the term *timestep* refers to the algorithmic timestep and not the user-defined timestep.

of excitatory and inhibitory synapses connected to a neuron input node. In

Ratio of excitatory and inhibitory syn	Worst case timestep (in ns)
3	1.6ns
2	1.6ns
1	1.2ns
1/2	0.73ns
1/3	0.52ns

Table 4.2: Worst-case time step

neural network implementations, we usually come across circuits that have a ratio of excitatory to inhibitory synapses greater than or close to 1. Hence in that case the *worst-case* timestep size by which we are limited is 1.2ns. Also, our input and output pulse widths being 6.5ns, timesteps of more than the pulse width (≈6ns) will never be used by the user in transient analysis of pulsed analog networks built using our topology.

To begin with, the commonly used methods of backward euler followed by Newton-Raphson allows one to use this larger user-defined timestep value of 6ns. Let's take a look at the computational complexity that will be required for this timestep improvement. In this method, after the usual numerical integration step, the local truncation error of the implicit integration algorithm has to be computed at every timepoint and the timestep has to be decreased accordingly (a simulator running at timestep of 6ns from user point of view could be running at 0.5ns inside all the time and printing values at 6ns intervals) to keep the truncation error of integration algorithm small at each timepoint. At all these timepoints one has to run Newton-Raphson

iterations. Additional complexity is required to assure the convergence of Newton-Raphson without which the simulation run may fail. Since these integration algorithms require old timepoint values and at time t=0 there are none present, they will also require some starting mechanism. The Runga-Kutta algorithm has to be implemented to take care of that.

After considering these additional complexities, one is not even assured that running at (6/1.2)=5 times larger user-defined timestep will pay off by that amount of speed improvement. One reason for this is the additional computation that is required (as described in the last paragraph) at each timepoint. Another and more important reason is that whenever just one node voltage in the entire circuit will change significantly (our circuit topology being pulsed, that will happen more than 75% of the time during a simulation), the timestep has to be reduced significantly to assure the convergence of the Newton-Raphson iterations [Nagel, 1975]. Hence though the simulator will look like it is running at a 6ns timestep to the user, internally it will run at much reduced timesteps for most of the time, which will bring down the speed of simulation drastically.

Hence, in this particular application, after all these computational complications, the conventional method of implicit integration followed by Newton-Raphson iterations does not provide us with commensurate improvement in speed. In contrast to that, the predictor-corrector method is much simpler to implement, provides the same accuracy and needs no such complex measures

as required by the other method for a successful simulation run. Hence it was chosen as the implicit integration algorithm for PULSE.

The use of the Gauss-Seidel method is common in third generation simulators. The other iterative method commonly used is the Gauss-Jacobi method. However, Gauss-Seidel converges faster than Gauss-Jacobi and the error possible in Gauss-Seidel is always less than or equal to that in Gauss-Jacobi [Newton, 1984]. Also Gauss-Seidel allows one to run the simulation with only one copy of the output variable (thus reducing memory requirements) as compared to two copies required in Gauss-Jacobi. Owing to these advantages, Waveform Gauss-Seidel is more efficient than Waveform Gauss-Jacobi and hence was chosen as the iterative solution algorithm in PULSE.

This section explained the reasoning involved in the choice of the algorithms in PULSE. The motivation in development of this simulator had been a fast, behavioral simulation of our pulsed analog topology that would allow one to run as many simulations as necessary to confirm the hardware implementation during a tightly scheduled design cycle. The choice of simple, efficient algorithms in PULSE allowed us to reach that goal.

4.7 Conclusions

This chapter concentrated on the different design and implementation aspects of PULSE. The different steps involved in a PULSE simulation run have been described. The building of a custom-made netlister for PULSE has been

outlined. The modeling of the different building blocks have been presented. Finally, the algorithms used in PULSE have been discussed and their choice has been justified. After looking at the design details of the simulator in this chapter, we will now use it as a tool to explore the implementations of different neural networks using our topology.

Chapter 5

Simulation using PULSE

PULSE was developed for fast and reasonably accurate simulation of neural networks implemented using our topology. The accuracy aspect of PULSE simulation will be the topic of our discussion in this chapter. To elaborate on this, both PULSE and HSPICE will be used to simulate some common neural networks to show its accuracy. We will start with simple circuits (with only three synapses and one neuron) and progressively grow in complexity. PULSE will be used for simulation of a tested chip and we will implement and simulate a *Matrix Associative Memory*.

5.1 Simulation of a Simple Network

A simple neural network reflecting the characteristics of our pulsed analog circuits can be built by connecting two excitatory and one inhibitory synapses to the summing input of a neuron as in Fig. 5.1. The weight voltage for all the synapses are 3.4V and the neuron threshold voltage is 1.5V.

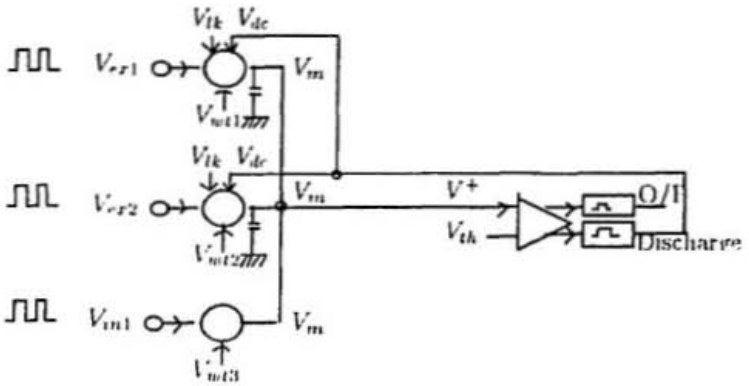


Figure 5.1: A simple network using 3 synapses and one neuron

Fig. 5.2 (in the next page) shows a simulation result of this circuit. The input pulses to the excitatory and the inhibitory synapses and the input voltage at the neuron summing node are shown in this figure.

The waveform at the summing node of the neuron can be explained as follows. Whenever an input pulse (V_{ex1} or V_{ex2}) arrives at an excitatory synapse, charge is dumped on to the capacitor at the neuron input (from now on the input voltage of neurons is referred to as V_m, corresponding to the membrane voltage of the biological neural cell), which explains the sudden rises in V_m. When an input pulse appears at the inhibitory synapse (V_{in1}), charge is sinked and hence the voltage drops.

The membrane voltage never reaches the threshold voltage for the neuron model (which is 1.7V for a input threshold voltage of 1.5V, as explained in the last chapter about neuron modeling), hence the neuron never fires and the cycle continues.

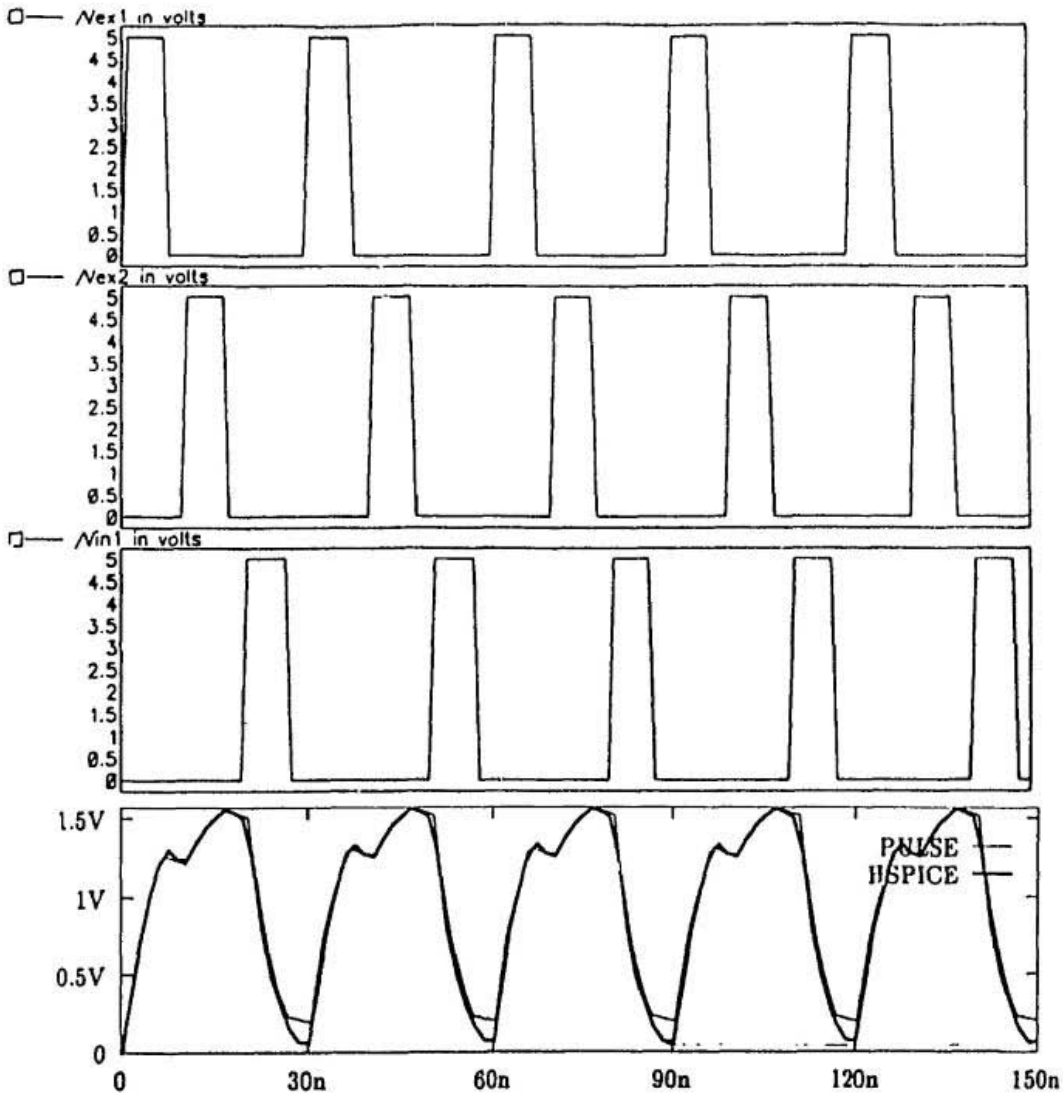


Figure 5.2: Simulating a circuit with two excitatory and one inhibitory synapses. First three waveforms are input pulses and the last one is the waveform at the summing node of the neuron. PULSE and HSPICE both outputs are shown for the last waveform.

Note that this circuit does not do anything, but the simulation of this circuit introduces the reader to the general behavior of the circuits built using our topology. In this simulation, the PULSE and HSPICE prediction of V_m can be seen to be quite close. The waveform only differs at voltages close to zero which can be explained as follows.

After removal of the input pulse to the inhibitory synapse, PULSE predicts the drop in V_m to be owing to the leakage present in the excitatory synapses whereas HSPICE predicts the inhibitory synapse to be active for some time even after the removal of the input pulse, hence the difference. Otherwise for most of the time, the two waveforms overlap. The difference present (in the waveforms) though is unimportant, the interest being the accurate prediction of voltages closer to the threshold voltage.

5.2 Simulation of XOR Gate

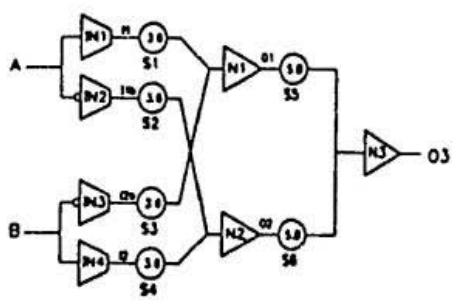


Figure 5.3: Implementation of XOR using our topology

We will now simulate another circuit that is a common benchmark for the implementation topologies of neural networks, the XOR circuit. The im-

plementation of XOR in our topology is shown in Fig. 5.3 [Bhattacharya, 1991]. The synapses used are all excitatory (represented by circles) and the figure shows their weight voltages. The weight voltages for $S1, \ldots, S4$ have been adjusted such that the neurons $N1$ and $N2$ can only fire when both its inputs are active, whereas $S5$ and $S6$ weight voltages are such that neuron $N3$ will fire whenever any one of its inputs is active. The input neurons $IN1, \ldots, IN4$ are used in the input layer, the small circles showing inverting input neurons. The standard neurons $N1$, $N2$ and $N3$ are used in the second (commonly referred to as the *hidden layer*) and the output layer.

Fig. 5.4 shows the simulation of this circuit for 500ns with inputs of 01 and 11. For inputs of 11, both the outputs can be seen to be zero. For inputs of 01 however, the output is 1 and the comparison of PULSE and HSPICE outputs show that they are identical except for a slight difference in timing in case of the first output pulse.

This difference in timing (for the first pulse) is due to a spike that appears at the output of the input neurons when they start firing. PULSE neglects that spike whereas HSPICE takes that into account. Otherwise, PULSE is as accurate as HSPICE here.

The first chip that was built by us consisted of the neuron and the synapse standard cells. The second chip implemented a pattern classifier followed by a MAXNET to build a content-addressable memory [Bhattacharya, 1991]. In the next section we will simulate that circuit using PULSE.

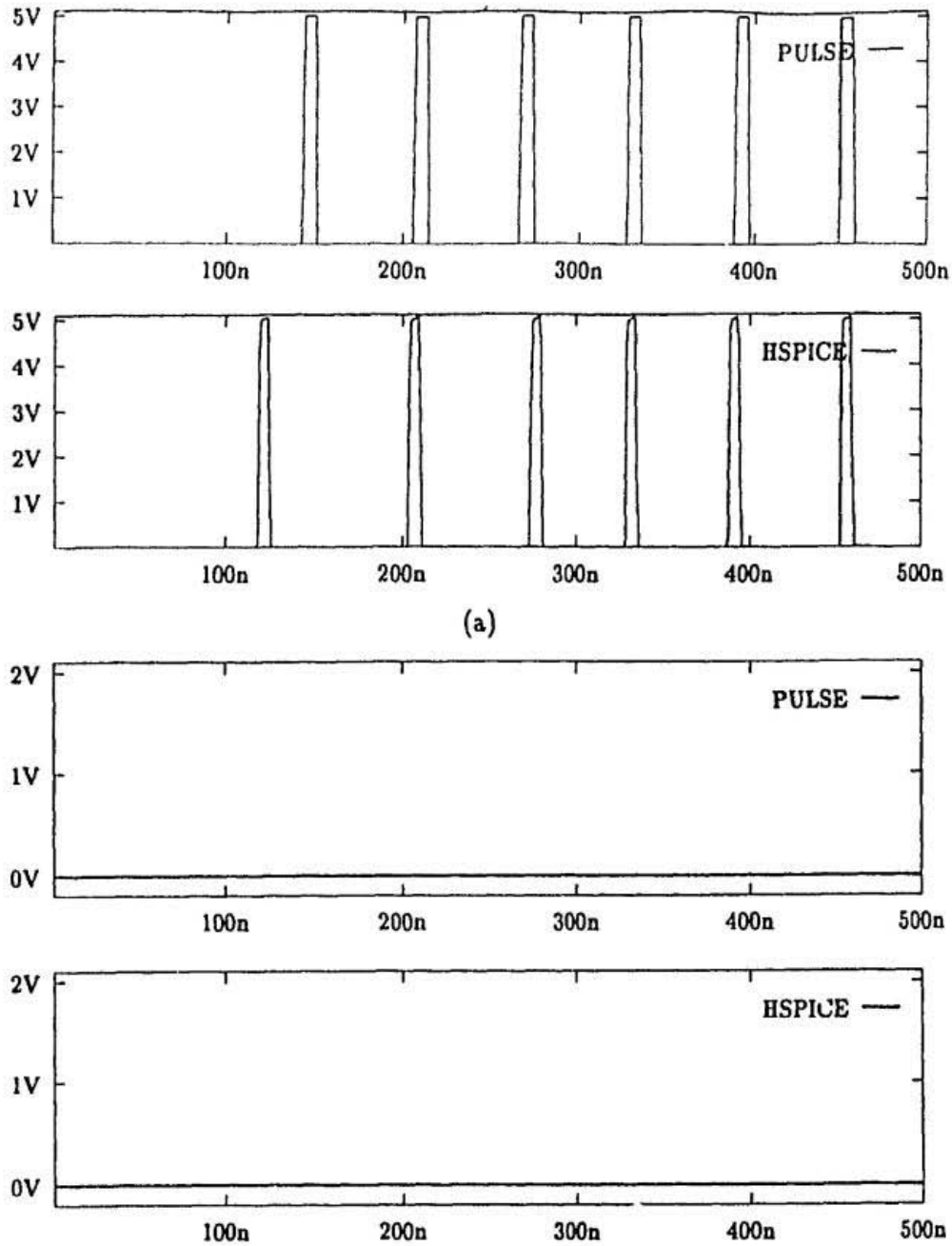


Figure 5.4: Outputs of the XOR circuit as predicted by PULSE and HSPICE for inputs of 01 and 11.

5.3 Simulation of CAM

The content-addressable memory (CAM) circuit is shown in Fig 5.5. The upper half of the circuit is a pattern classifier. The other half of the circuit is a bank of inhibitory synapses that implement the MAXNET.

The MAXNET part can be switched in and out of the circuit by making the CNT bit 1 or 0 which allows the neuron outputs to pass through the AND gates. If the neuron output passes the AND gate (CNT=1), the inhibitory synapses receive inputs. In that case, using those inhibitory synapses, an active neuron can drain charges from the summing nodes of other neurons thus implementing the winner-take-all or MAXNET circuit.

This circuit processes five bit patterns. *Input neuron* (inverting and non inverting) outputs are inputs to this chip. Patterns are stored in a hardwired fashion using excitatory synapse connections in a given column.

Depending on the *Hamming Distance* (H.D.) of the presented pattern from the stored patterns, a number of excitatory synapses are activated thus dumping charge at the summing input of the neuron (as the H.D. increases, a smaller and smaller number of excitatory synapses get activated in that particular column). If the membrane voltage V_m of a neuron crosses the threshold voltage, it fires and identifies the input to be close to the stored pattern in its column. When CNT=1, as the neuron fires, it also drains charge from summing node of other neurons thus inhibiting them.

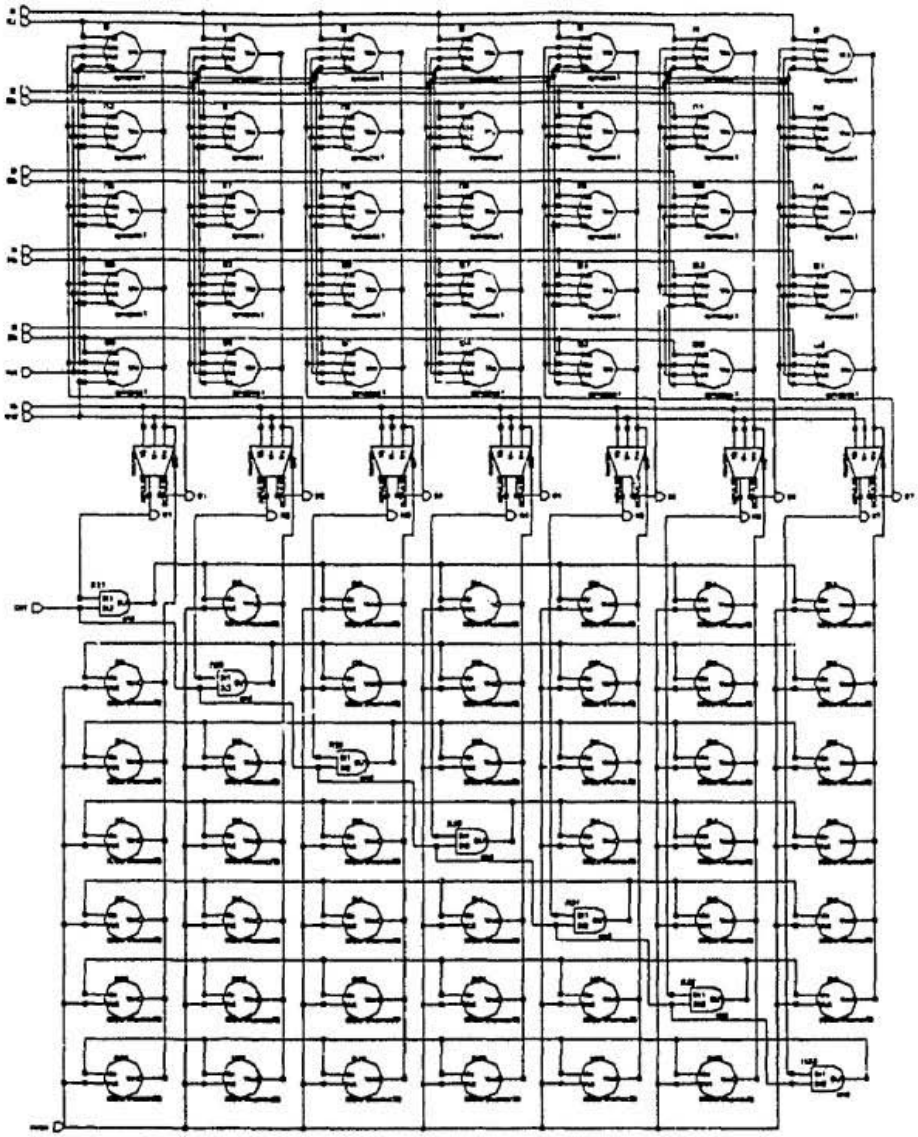


Figure 5.5: Schematic diagram of the CAM. The synapses are shown as coin-shapes whereas neurons are the trapezoids at the center.

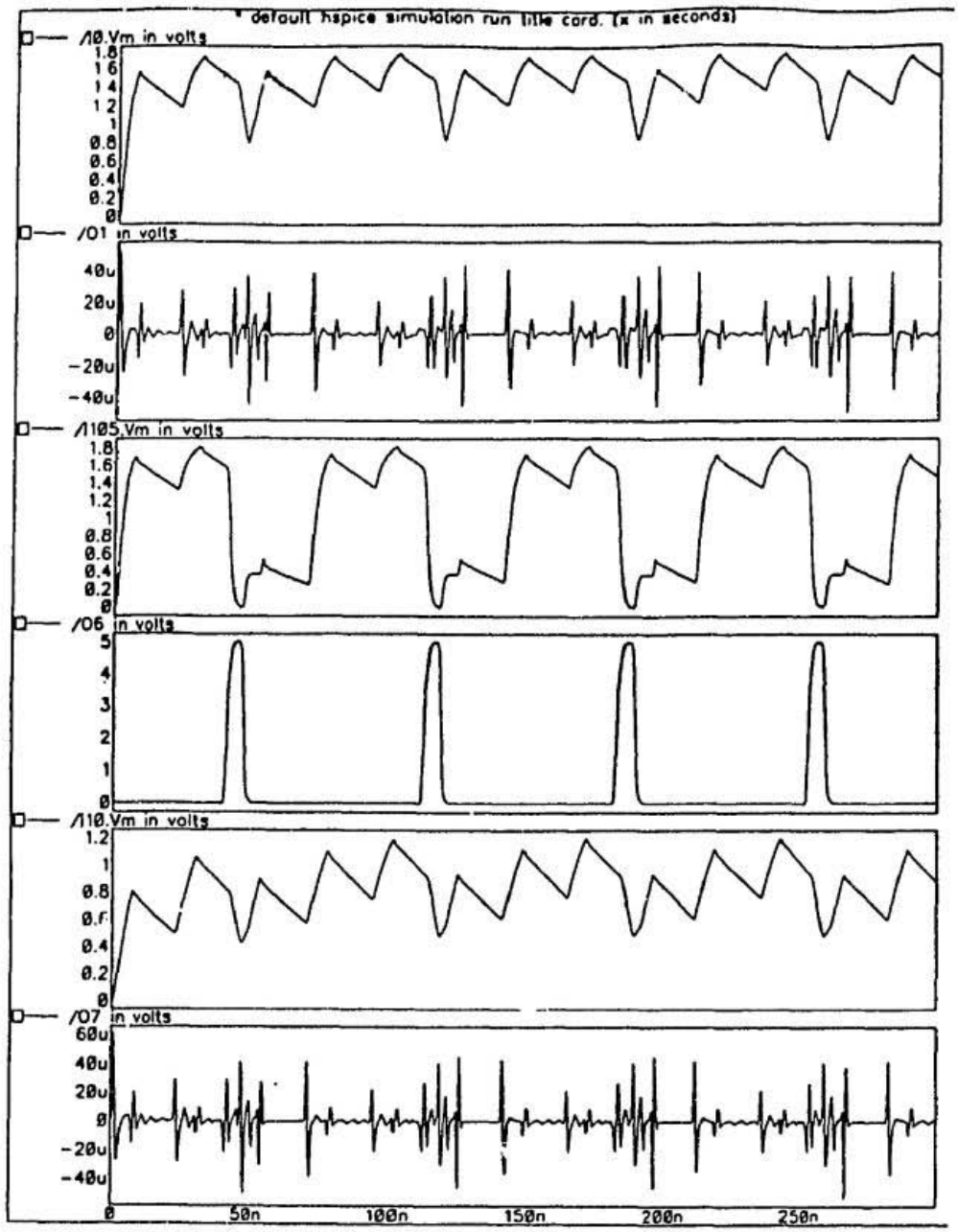


Figure 5.6: HSPICE simulation of CAM showing the activation and outputs of neuron $N1$, $N6$ and $N7$ respectively.

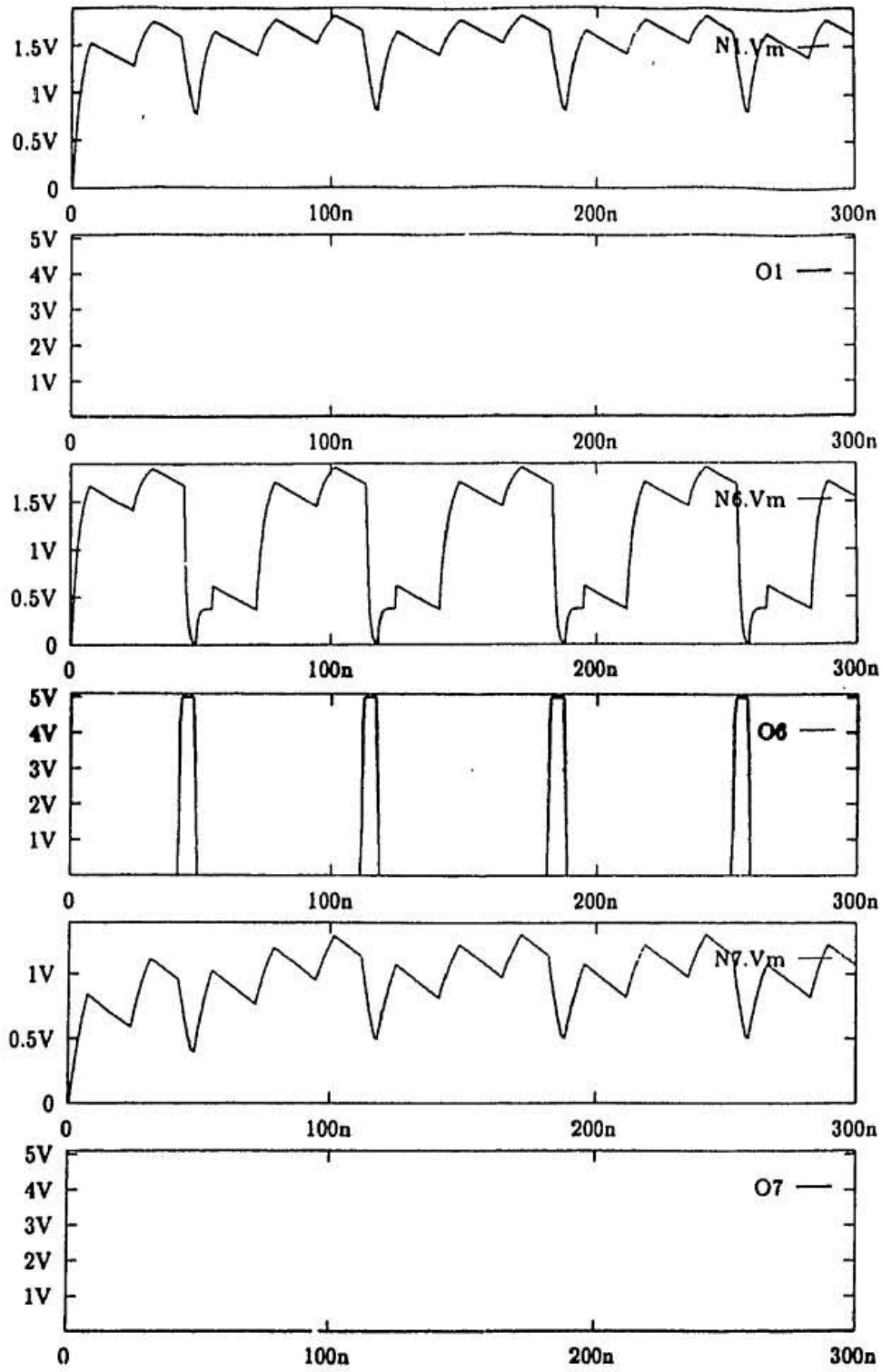


Figure 5.7: PULSE simulation of the CAM circuit showing the activation and outputs of the same neurons.

Fig. 5.6 shows the HSPICE simulation results of this winner-take-all circuit with a *presented pattern* of 01100, which is closest to the stored pattern 00100. Fig. 5.7 shows the simulation results for the same circuit using PULSE. Comparing the PULSE and HSPICE outputs for the given input pattern one can only see minute differences in their performances.

The patterns stored in this chip are 00000, 11111, 11110, 10101, 01010, 00100 and 11011. Hence the other close patterns to the presented pattern (01100) are 00000 ($N1$), 11110($N3$) and 01010($N5$) which are all H.D. of 2 apart from 01100. All other patterns are further apart (H.D.>2).

The simulation results show membrane voltages and outputs of three neurons $N1$, $N6$ and $N7$. The pattern corresponding to neuron $N6$ being the closest, it fires and does not allow any other neuron to fire although one can see the membrane voltage of neuron $N1$ rising close to the threshold voltage (which is 1.6V input corresponding to 1.8V in our neuron model) suggesting that it is the next closest pattern.

Fig. 5.8 and 5.9 shows the simulation of the same network with an input of 10101 which is a stored pattern. Again the PULSE and HSPICE simulation results can be seen to be close. The outputs $O2$, $O4$ and $O6$ are predicted by PULSE as accurately as by HSPICE. Some differences in the prediction of the membrane voltage can be seen that arises owing to minor timing differences present in the two simulators. The next paragraph will elaborate on that.

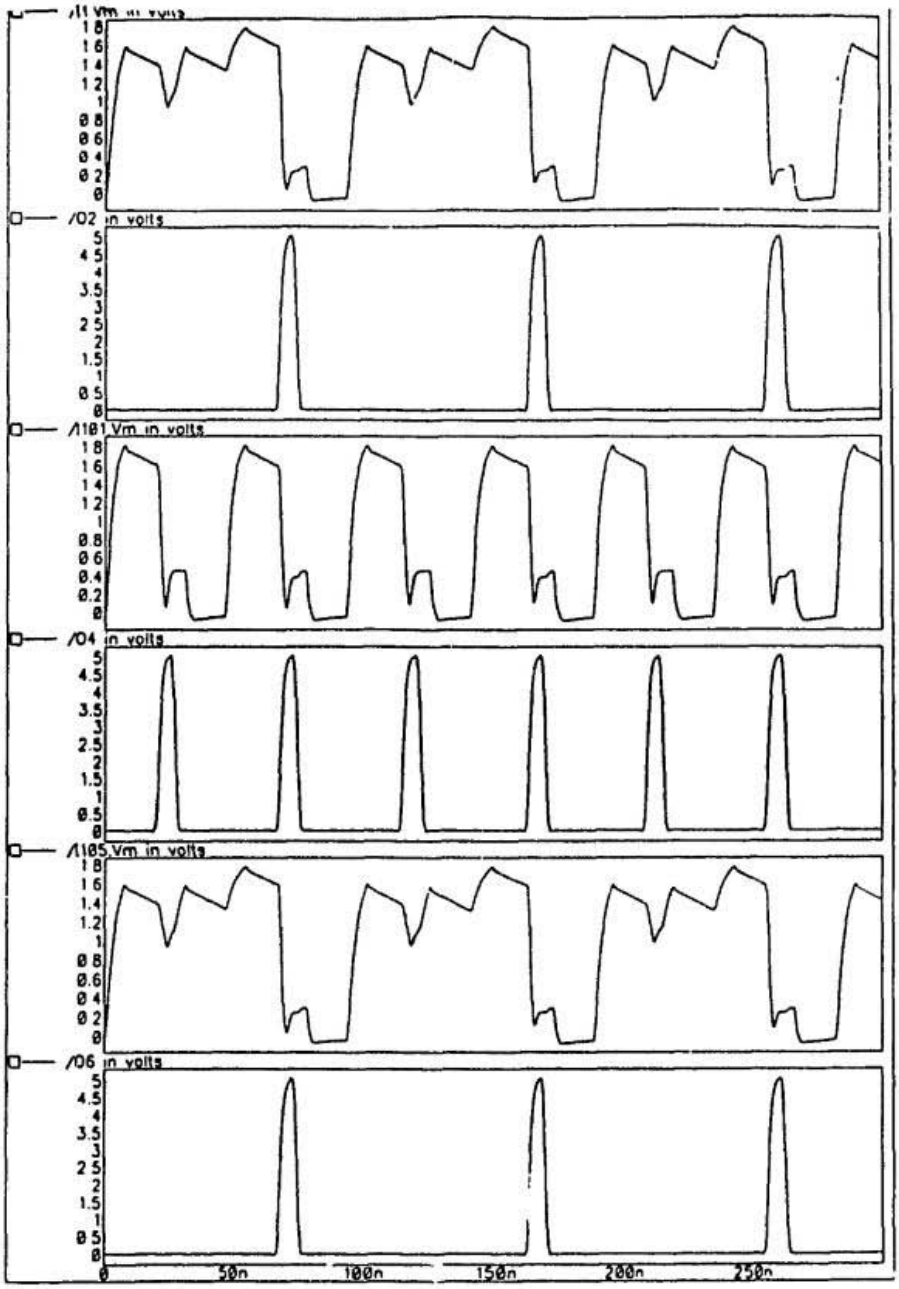


Figure 5.8: HSPICE simulation of CAM for input 10101 showing the activation and outputs of neuron $N2$, $N4$ and $N6$ respectively.

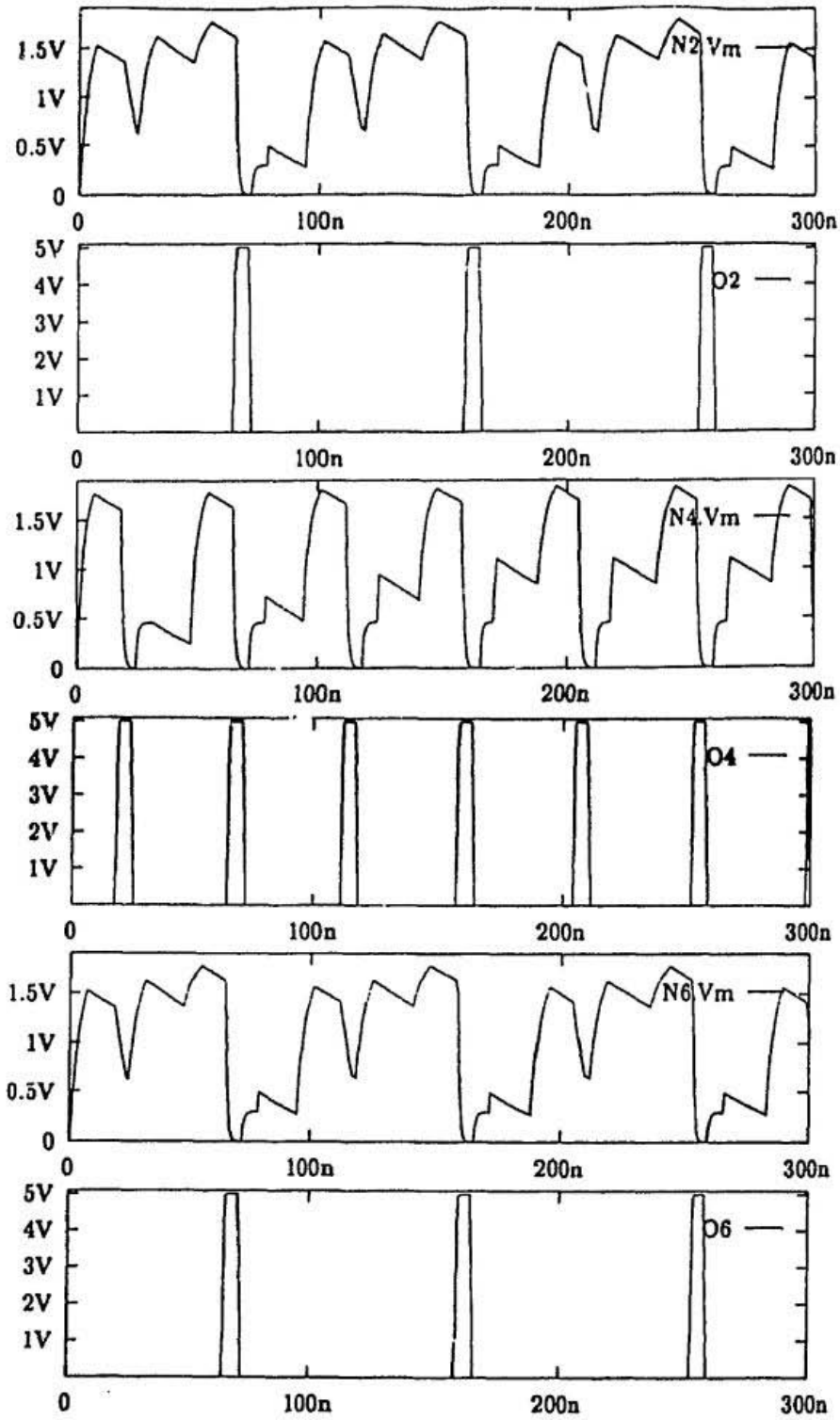
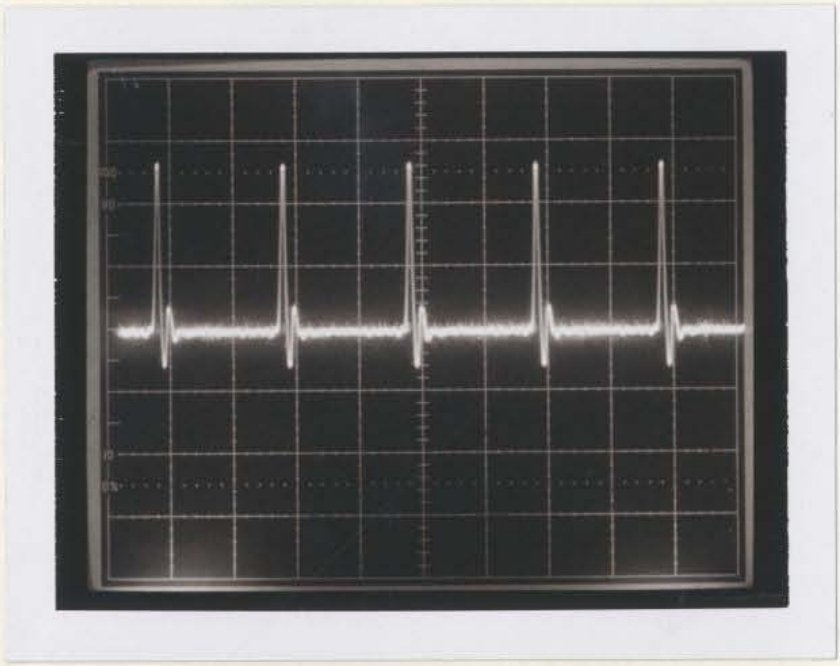
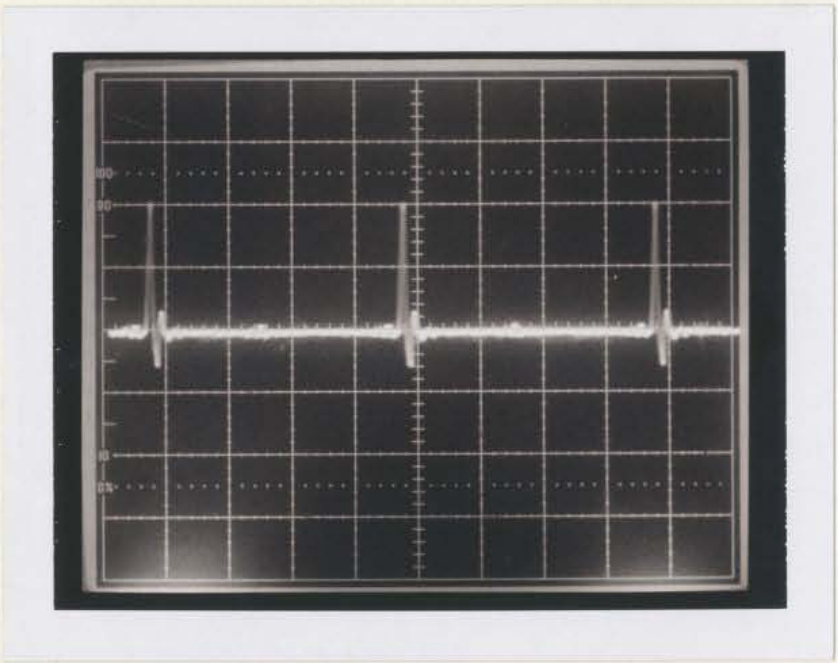


Figure 5.9: PULSE simulation of the CAM circuit showing the activation and outputs of the same neurons.



(a)



(b)

Figure 5.10: Output of CAM chip in response to input 10101 (a) Neuron N4, (b) Neuron N2

Looking at the membrane voltage, one can see some differences in the PULSE and HSPICE predicted waveforms just after the neuron firing. This occurs owing to a minor timing difference in the PULSE and HSPICE predicted output pulses. PULSE predicts the output pulse one or two nanoseconds before, hence the discharge pulse corresponding to the neuron output pulse also ends a few nanoseconds before the HSPICE predicted discharge pulse. Hence, HSPICE predicted V_m goes down again to zero after a small rise due to the presence of the discharge pulse, the rise in V_m being due to the input pulses at the excitatory synapses whereas PULSE predicted waveform goes down only linearly due to the leakage (since the discharge pulse has already gone down) after the rise.

Other than this small difference, which does not affect the output pulse prediction as can be seen comparing Fig. 5.8 and 5.9, PULSE predicted waveforms can be seen to be as accurate as the HSPICE predicted waveforms. The CAM chip has been tested and the results obtained are shown in Fig. 5.10. The relative rate of firing of the two neurons N4 and N2 in the CAM chip can be compared to Fig. 5.8 and Fig. 5.9 to justify the validity of the simulations.

5.4 Matrix Associative Memory

Given an associated pair of pattern $(\mathbf{x}, \mathbf{y})$, where $\mathbf{x}$ is a M bit input pattern and $\mathbf{y}$ is a N bit output pattern, one can form an associative memory by connecting the N neurons to the M inputs by synapses whose weights are

given by matrix $T = \mathbf{y}\mathbf{x}^t$ [Pao Y. H., 1988]. The memory thus built is commonly known as the *Matrix Associative Memory*.

5.4.1 Implementation

Our aim is to build this memory for eight 8-bit input patterns, each Hamming Distance of 4 away from each other. Three neurons will be used to encode the output patterns. Table 5.1 shows the input output pairs.

INPUTS								OUTPUTS		
-1	1	1	-1	1	1	1	-1	-1	-1	-1
-1	-1	1	1	-1	1	1	1	-1	-1	1
-1	1	-1	1	1	-1	1	1	-1	1	-1
1	-1	-1	-1	1	1	1	1	-1	1	1
-1	-1	-1	-1	-1	-1	1	-1	1	-1	-1
-1	-1	-1	1	1	1	-1	-1	1	-1	1
-1	1	-1	-1	-1	1	-1	1	1	1	-1
-1	-1	1	-1	1	-1	-1	1	1	1	1

Table 5.1: Input-Output patterns for matrix associative memory

For these given input-output pairs, the matrix T can be calculated as

$$\mathrm{T} = \begin{bmatrix} -2 & 2 & 2 \\ -2 & 2 & -6 \\ -2 & -2 & 2 \\ -2 & -2 & 2 \\ -2 & 2 & 2 \\ -2 & -2 & 2 \\ -6 & -2 & -2 \\ -2 & 6 & 2 \end{bmatrix}$$

When an input pattern X is presented to this memory, contribution from the second input bit of X to neuron 1 by T_{21} can be calculated as $X_2.T_{21}$. Hence,

1. If the 2nd bit of X is +1, the contribution is (+1)×(-2)=-2.

2. If the 2nd bit of X is -1, the contribution is (-1)×(-2)=+2.

Now, in our pulsed analog implementation the +1 inputs are 5V, whereas the -1 inputs are 0V. These inputs are passed through the input neurons to generate the input pulses that drive the corresponding synapses thus dumping (sinking) charge at (from) neuron input. Now the above algorithm suggests addition of charge to the neuron input when the input is +1 and removal of charge from the neuron input when the input is -1.

This suggests a requirement of an androgynous synapse that will *switch* between excitatory and inhibitory states depending on the input presented to it. In contrast to that, our synapses are strictly non-androgynous. Remember that this algorithm is originally designed for a software implementation, where this switching process is simply a multiplication. Also there is no evidence of a biological analogue of an androgynous synapse.

To implement this circuit, one can get around this problem by using the modules shown below to build the weights in matrix T.

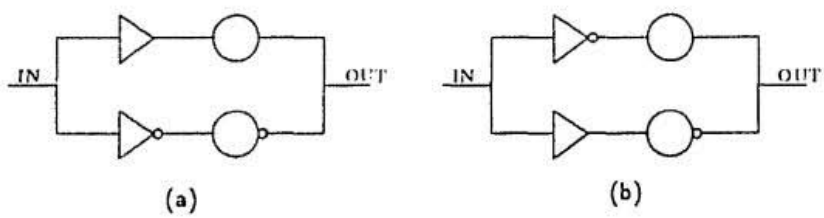


Figure 5.11: Circuit used to implement (a) Positive weights and (b) Negative weights .

In Fig. 5.11(a), the triangles show the input neurons whereas the circles show the synapses. When an input of 1 i.e. 5V occurs, the upper neuron of Fig. 5.11(a) is active thus adding charge to the neuron input. When the input is -1 i.e. 0V, the lower neuron is active, thus activating the inhibitory synapse to drain charge. Hence a positive weight in matrix T is effectively implemented by this circuit. Fig. 5.11(b) can be explained in the same fashion.

The implementation of this network is shown in Fig. 5.12. To save area on the chip, the input neurons are shared by the synapses corresponding to the same input bits. The weight voltages used for the excitatory and inhibitory synapses to implement the weights of 2 and 6 are tabulated below.

Weight	*Exsyn wt voltage*	*Insyn wt voltage*
2	3.6*V*	1.7*V*
6	5.0*V*	3.6*V*

Table 5.2: Weight voltages used in the associative memory

The criterion used to calculate the weight voltages is as follows. The maximum current (say I) is delivered by the excitatory synapse with a weight voltage of 5V. Hence that has been used to implement the maximum weight i.e. 6, for the excitatory synapses. Excitatory synapses with a weight voltage of 3.6V delivers current of magnitude $I/3$ at the threshold voltage of 1.0V. Hence that was used to implement an weight of 2 (1/3rd of 6) for the excitatory synapses.

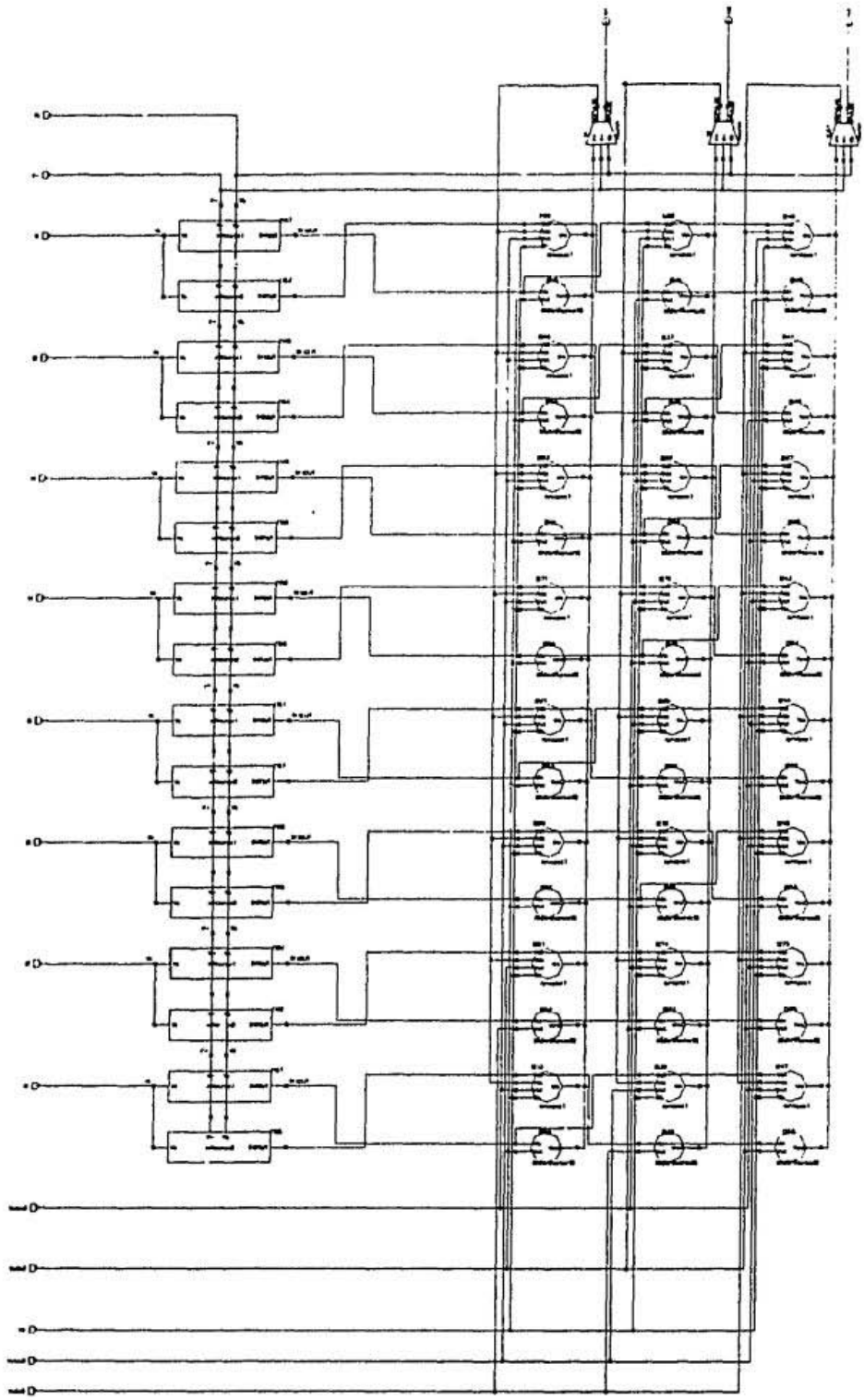


Figure 5.12: Schematic diagram of the Matrix Associative Memory

PAGINATION ERROR.

TEXT COMPLETE.

ERREUR DE PAGINATION.

LE TEXTE EST COMPLET.

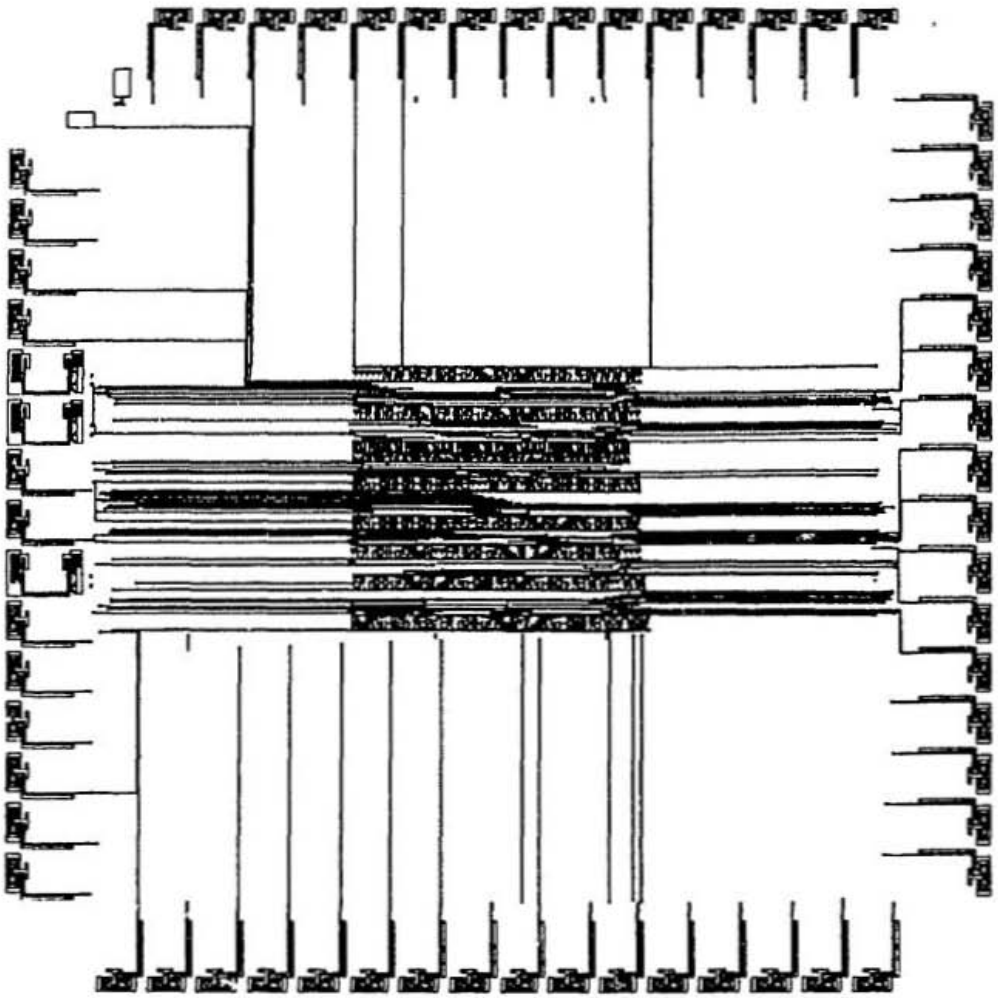


Figure 5.13: Layout of the associative memory obtained by auto place and route routines (only Metal1 layers are shown).

5.4.2 Simulation

The simulation results for this network using PULSE and HSPICE are shown in Fig. 5.14 and 5.15. The first simulation (Fig. 5.14) shows the three neuron outputs for a stored input of 10001111 (here -1's are represented as 0's for convenience) whose corresponding output is 011. Both PULSE and HSPICE simulations reflect correct operation of this circuit.

The second simulation (Fig. 5.15) shows the three neuron outputs for a distorted input pattern of 10011111 (closest to the stored pattern of 10001111). The content-addressability [1] property of this memory is reflected in the outputs which is again 011, i.e. the memory responded correctly even when given a distorted pattern.

Differences in performance of PULSE and HSPICE is noticeable here though they both predict the same outputs. This difference will be recognized and discussed in the next section.

[1] This property of memories refers to their ability to respond correctly while given a partial cue.

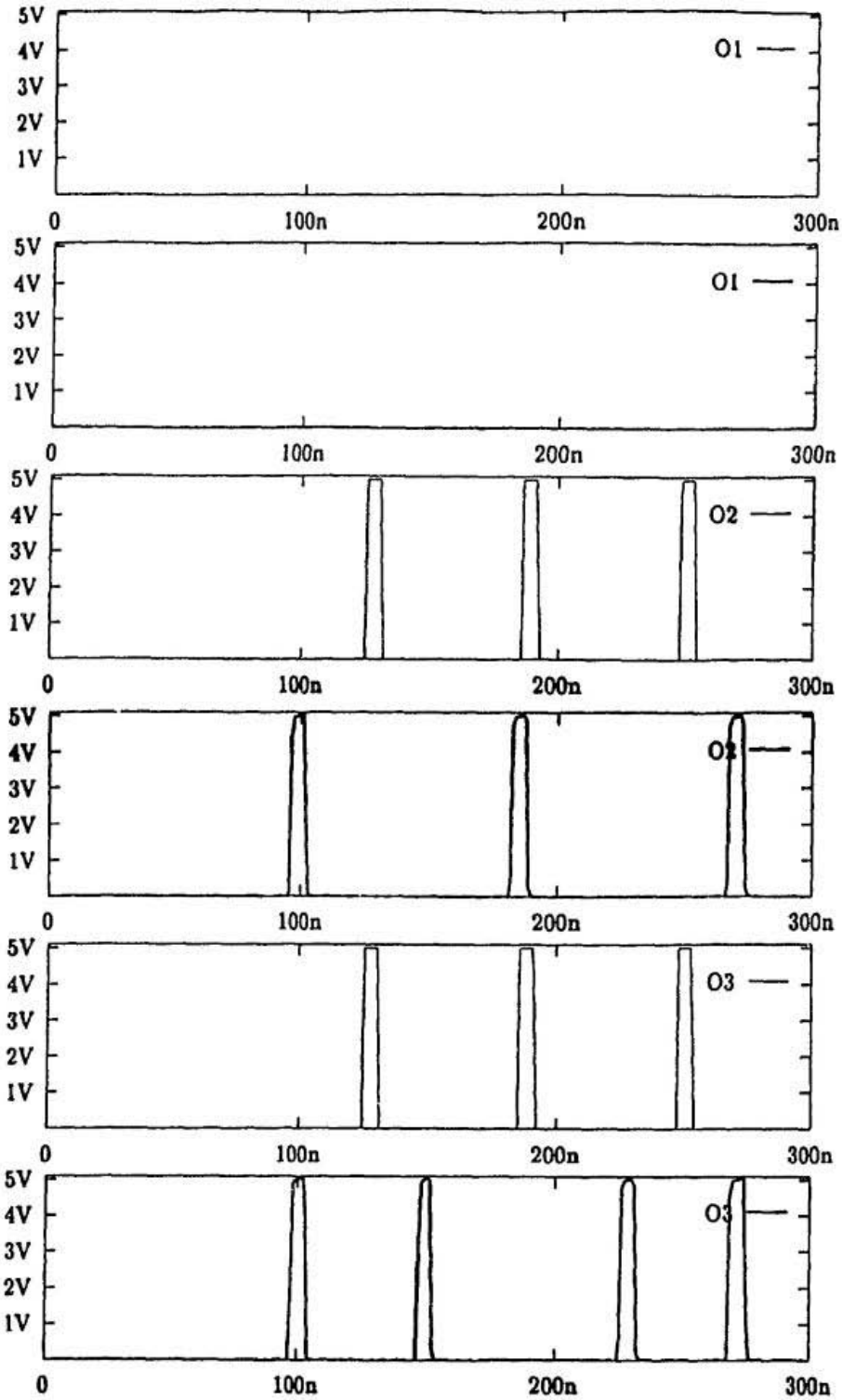


Figure 5.14: PULSE and HSPICE simulation of the Matrix Associative Memory for input 10001111 showing the outputs of the circuit. For all three neurons, the first output is predicted by PULSE and the second one is predicted by HSPICE..

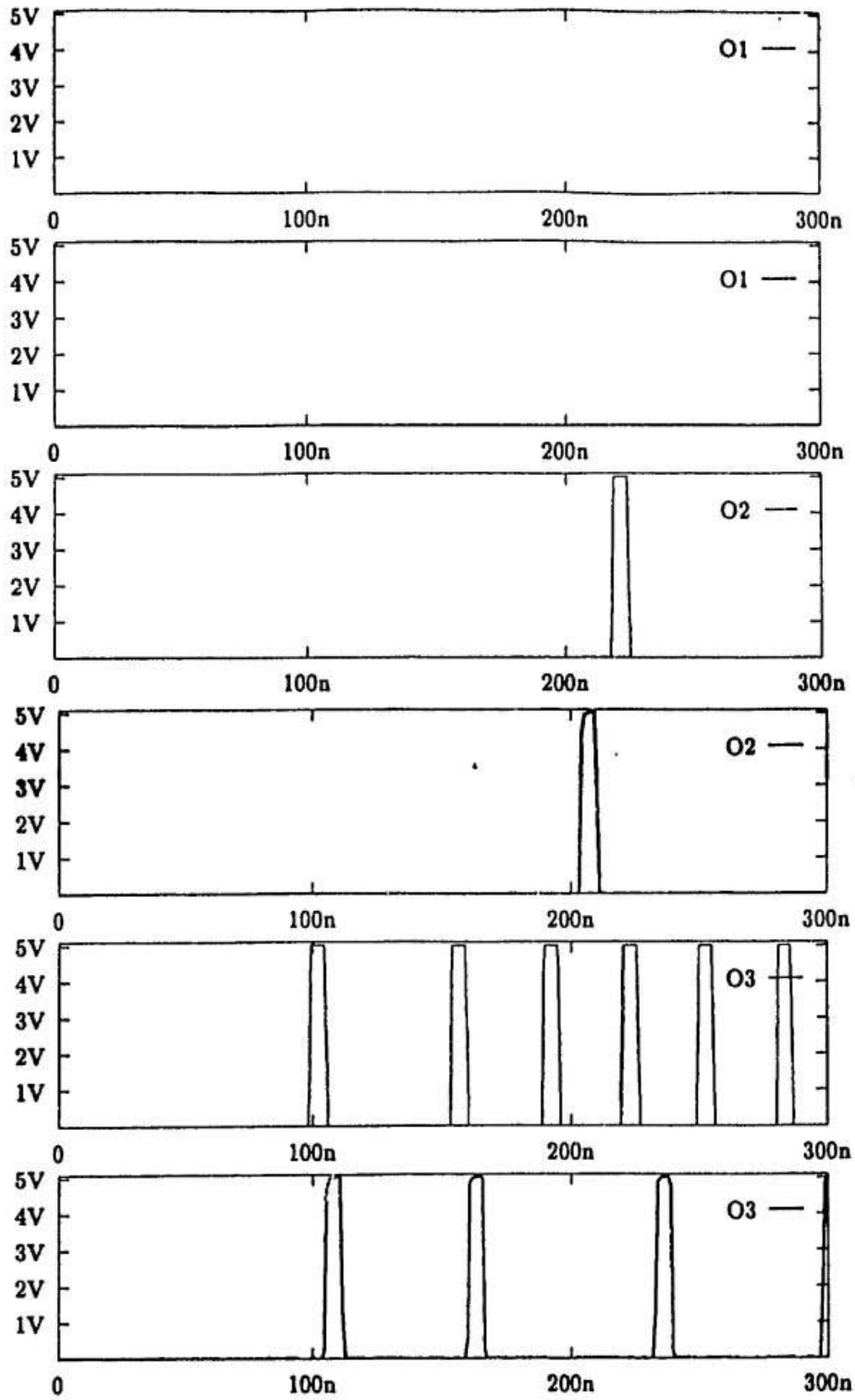


Figure 5.15: PULSE and HSPICE simulation of the Matrix Associative Memory for distorted input 10011111 showing the outputs of the circuit.

5.5 Discussion

PULSE is built for simulation of pulsed analog neural networks. Hence to judge the performance of this simulator, one has to take into account the peculiarities of the intended network and the user expectations. The information in a pulsed network is in the form of *pulses*, both at the input or output of the network. The user is interested in knowing whether the circuit output is firing or not and if two outputs are firing then at what relative rate.

Considering these facts, one might suggest the possibility of standard, inherently faster logic simulation for these networks since the inputs and outputs are both digital pulses. But a mixed-mode simulation becomes necessary because, though the input-output informations are digital pulses in pulsed analog circuits, the internal processing is analog. Hence circuit simulation becomes necessary for some parts of the circuit, whereas digital simulation is sufficient for other parts.

The precision present in HSPICE to simulate a given circuit contributes to the *generality* property of that simulator. In contrast, the accuracy constraint is more relaxed in case of PULSE since it only analyzes pulsed networks. The user is satisfied in knowing whether the output is firing or not rather than knowing exactly at what time point the pulse occurs.

Also, exact prediction of the pulse timings seem to be impossible from our experience. This fact is illustrated in Fig. 5.16 that shows HSPICE simulation outputs of the XOR circuit with an input of 01.

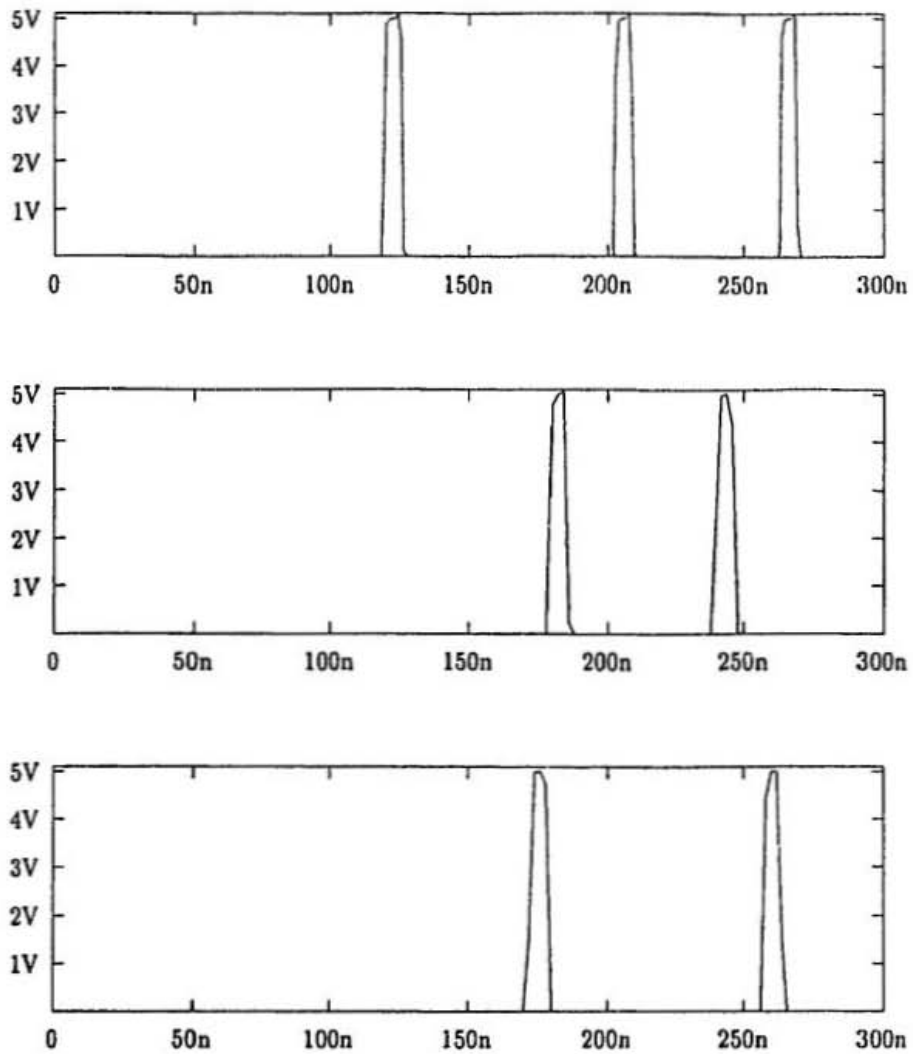


Figure 5.16: HSPICE predicted output for XOR circuit with input 01 (a) Prediction with 1ns time step, (b) Prediction with 2ns time step and (c) Prediction with 2ns time step with NMOS conductance 1% reduced.

Fig. 5.16 shows that varying the timestep from 1ns to 2ns completely changes the pulse timings, and with 2ns timestep HSPICE only predicts 2 output pulses in 300ns of time period as compared to 3 pulses predicted with 1ns timestep. In Fig. 5.16(*c*), the NMOS transistor conductance is reduced by only 1%. The pulse timings vary again even with such a small change in the transistor parameters which can happen during chip fabrication. This is not a problem related to this specific circuit but is general to all the circuits built using our topology.

This anomaly in HSPICE outputs can be explained as follows. Assuming that the input voltage to a neuron is close to its threshold voltage, the firing of that neuron depends on whether the next input pulse to the excitatory synapse connected to the neuron will be able to carry the neuron's input voltage to above its threshold or not. If the input voltage is carried to a point really close (but still less than) its threshold, the neuron would not fire.

Comparing outputs of two simulators simulating the same pulsed circuit, due to a small roundoff error present in one of these simulators, say a neuron which is supposed to fire at a given time point didn't fire. From that time-point onwards the two simulator outputs are bound to behave differently. This difference in behavior is present in HSPICE even when the simulation is done under different conditions.

Thus one can see that predicting the exact timing of the pulses seems to

be impossible in this class of networks. One should really aim to predict the relative rate of firing of the outputs, or whether the output is firing or not, than bothering about the exact timing of the pulses which is impossible. The simplifications in PULSE models allows it to predict that information for our topology in considerably reduced time as compared to the unwanted exact and time-consuming analysis of HSPICE. Hence PULSE is more efficient for the simulation of this class of circuits from user's point of view.

5.6 Conclusion

This chapter has established PULSE as a reliable and efficient simulator for pulsed analog networks by benching it against HSPICE and recognizing its ability to exploit the peculiarities of the pulsed networks to enhance its speed. The next chapter will focus on this speed improvement aspect of PULSE and will separate out the contributions of different factors responsible for it.

Chapter 6

Results

The theme of this thesis is the development of PULSE and implementation of new neural networks using PULSE as the tool for verification. Development of PULSE and its reliability and effectiveness in simulation of pulsed neural networks had been the focus of discussion in the previous two chapters. Implementation of an associative memory has also been described. In that way a part of the results have already been presented. To sum up the results, in this chapter, we will discuss the speed improvement achieved in PULSE over commercial circuit simulators like HSPICE and the various factors responsible for the cost of PULSE simulation.

6.1 Cost of PULSE simulation

As described in the *Simulator Design* chapter, PULSE simulation uses two iteration loops. One of them is the outer Gauss-Seidel iteration loop. All the node voltages found by using the macromodels and the node-equations are

iterated until convergence in this loop.

The other loop is nested inside this outer Gauss-Seidel loop. This loop implements the functional iteration algorithm used for the evaluation of the neuron input node voltages. A typical code generated by PULSE illustrates this point.

```
do
{ t+=delt;
  V[0]=stddcpulse(10.2,0,13,0,2);
  V[1]=stddcpulse(10.2,0,13,0,0);
  ..
  do
  { ..
    temp=Vlast2[12]+0.5*(store1[2]+((exsynapse(V[15],V[8],V[1],
    V[7],V[12])+exsynapse(V[14],V[8],V[1],V[7],V[12]))*delt*
    1e-09)/(2*0.15e-12));
  }
  while((fabs(temp-V[12]))>0.0001);
  V[12]=temp;
  if (flag2!=20) for (i=0;i<20;i++) if ((fabs(V[i]-Vlast[i]))
  <0.0001) flag2++;
}
while ((flag2<20) || (flag1!=1));
```

Figure 6.1: Code generated by PULSE

Here the outer *do* loop is the Gauss-Seidel loop whereas the inner *do* loop is the *Functional Iteration* loop (for one of the neurons present in the circuit). In this way every neuron contributes one functional iteration loop within the outer Gauss-Seidel loop.

Hence if m is the total number of Gauss-Seidel iterations at a given timepoint and p is the average number of functional iterations required in every Gauss-Seidel iteration, the simulation of a circuit containing N neurons has a total time cost of

$$T = mT_1 + mpNT_2 \tag{6.1}$$

for every timepoint. Here T_2 is the average time required to evaluate a neuron input node during a single functional iteration step and T_1 is the time required to evaluate all other nodes in the circuit other than the neuron input nodes during a Gauss-Seidel iteration.

T_1 only involves the time required to perform the logic evaluations to calculate the neuron output node voltages. In contrast to that, T_2 involves evaluation of all the *synapse macromodels* connected to the neuron. Since $pNT_2 \gg T_1$ (synapses being the dominating component in any neural network), the exact equation 6.1 can be approximated as

$$T \simeq mp(NT_2) \tag{6.2}$$

Since PULSE uses fixed timestep, (the timeperiod being an integral multiple of the timestep), PULSE simulation time as a first order approximation can be seen to be the product of three factors. Any change in one of these factors will affect the simulation time proportionally. Keeping this in mind, these terms and their dependence on the network parameters will be discussed next.

6.1.1 Evaluation of Neuron Input Voltages

This refers to the (NT_2) term in Eqn. 6.2. Remember that T_2 was defined to be the average time required to evaluate a neuron input node. This term (NT_2) can be more exactly represented as

$$NT_2 = \sum_{i=1}^{N} t_{EX} N_{EXi} + t_{IN} N_{INi} \tag{6.3}$$

where t_{EX} (t_{IN}) is the time taken to evaluate the excitatory (inhibitory) synapse macromodel and N_{EX_i} (N_{IN_i}) is the number of excitatory (inhibitory) synapses connected to the i-th neuron. Eqn. 6.3 reflects the scale of the network and hence the order of the computation cost to simulate a given network can be found by studying this term.

The magnitude of time factor t_{EX} (t_{IN}) present in Eqn. 6.3 varies at different time points depending on whether a particular synapse is receiving pulsed input at that given point or not. At the time point when the synapse is not receiving any activation, the time required to evaluate that synapse macromodel is much less, as Table 6.1 shows.

Time Taken	Input Pulse ON	Input Pulse OFF
t_{EX}	272 μs	15.8 μs
t_{IN}	272 μs	1.6 μs

Table 6.1: Time required to evaluate synapse macromodels

Since it is complex to find out the proportion of synapses that are active at a given point of time, one can see that it is difficult to predict the order of

the total simulation time taken by PULSE (while simulating a given neural network). The simulation time from Eqn. 6.3 depends strongly on the activity of the circuit and the connectivity of the circuit (the number of neurons present and how the synapses are connected to it). However from Eqn. 6.3, one can intuitively predict that the time taken for regular networks [1] under strong activation (t_{EXON} and t_{INON} dominating over t_{EXOFF} and t_{INOFF}) will have a *close to* linear dependence on S, where S is the total number of synapses (*excitatory + inhibitory*) present in a circuit.

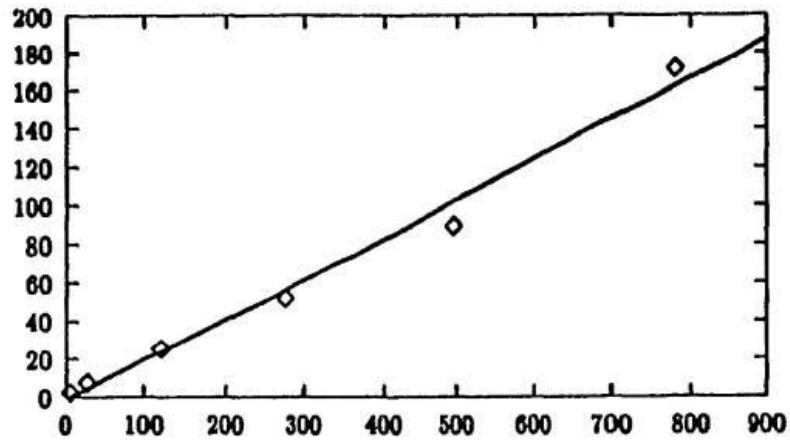


Figure 6.2: Dependence of PULSE simulation time on the number of synapses present in a CAM circuit

To verify this intuition, a netlist generator program was written to generate the netlist (circuit connectivity) of a MAXNET, a regular network with N number of stored patterns each *Hamming Distance* of $N/2$ away from the

[1] Regular networks refer to those type of networks where the number of excitatory and inhibitory synapses connected to different neurons remain the same. Many artificial neural networks fall under this category.

other. A stored pattern was put as input and the run time of PULSE was measured. This experiment was repeated for N= 2, 4, 8, 12, 16 and 20. Figure 6.2 shows the results obtained which ratified our intuition.

Hence though the *exact* order of simulation time of PULSE involves evaluating Eqn. 6.3, a first order approximation of PULSE simulation time for regular networks under strong activation can be estimated as $O(S)$, where S is the number of synapses present in the network.

6.1.2 Number of Functional Iterations

This refers to the term p in Eqn. 6.2. The number of iterations required in the functional iteration loop depends strongly on the convergence criterion and hence on the accuracy required by the user. As the accuracy requirement increases, the simulation time increases nonlinearly, as Fig. 6.3 shows.

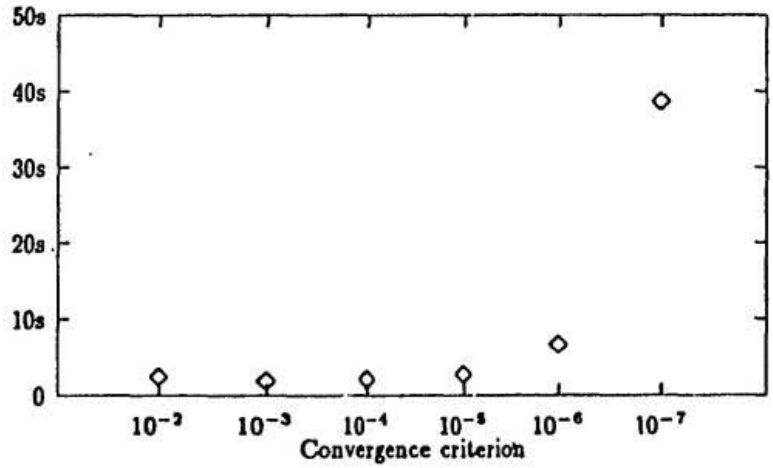


Figure 6.3: Dependence of PULSE simulation time on the convergence criterion of the functional iteration loop

In PULSE, the convergence criterion is 0.0001 i.e. the loop is exited when the last calculated voltage and the present voltage differs only by 0.0001V and this criterion has been set to conform to HSPICE results.

Once the convergence criterion is set, the number of functional iterations required to find the input voltage of neuron X depends on the input activation and firing rate of neuron X. This variation is shown for the output neuron of XOR circuit in Fig. 6.4.

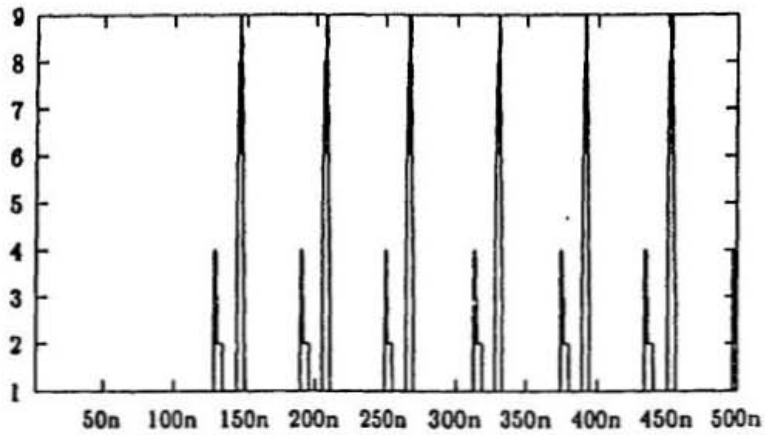


Figure 6.4: Dependence of number of functional iterations on circuit activity

When the discharge pulse arrives, a greater number of iterations are required to calculate the neuron input voltage owing to the sudden drop in membrane voltage. This corresponds to the period right after the neuron firing and corresponds to the large spikes in Fig. 6.4. The small spikes correspond to the sharp rises in neuron input voltages and occur due to the arrival of input pulses to the input synapses connected to the neuron.

One has to recognize the fact that Fig. 6.4 corresponds to one neuron in the given network and there are many others that are not firing at the given point of time. Since p is averaged over all neurons, p never reaches the peak value of 9 as shown above. A typical value of p appears to be 2 for highly activated networks of medium scale.

6.1.3 Number of Gauss-Seidel Iterations

This refers to the term m in Eqn. 6.2. The Gauss-Seidel iteration process can be explained with reference to the following equations

$$\begin{aligned} 2x_1 - 3x_2 &= -5 \\ x_1 + x_2 &= 5 \end{aligned} \tag{6.4}$$

In this algorithm, starting with an assumption of x_1=x_2=0, x_1 is evaluated from the first equation assuming x_2 to be constant and using this new value of x_1, x_2 is calculated from the second equation. This iterative process is then repeated until convergence.

Now considering circuit node voltages to be the variables, if x_2 is dependent on x_1 and x_1 is independent of x_2 then only one evaluation will be required to solve the above set of equations. This will occur when the system of equations is sparse e.g. the coefficient of x_2 in the first equation is zero.

In circuit simulation, the set of equations generated are inherently sparse. That suggests that the sequence in which the equations are generated in a simulator is important. If the second equation was encountered first and was solved for x_2, another iteration would have been required for solving this set

of equations. Hence from Eqn. 6.2, the simulator time would be doubled. Extending this logic, for a circuit with n nodes, incorrect ordering can lead to a n-fold increase in simulation time in the worst case.

Keeping this in mind and taking advantage of the fact that the system of equations generated is very sparse, PULSE reduces the number of Gauss-Seidel iterations drastically. While ordering the set of equations, PULSE puts the input (independent) node equations before the other nodes. Among the two classes of dependent node equations, the neuron input node voltages depend on other neuron outputs. Hence, neuron output node equations are also evaluated before neuron input node equations in PULSE to ensure correct ordering of the equations.

Taking the above measures, PULSE brings down m to a value of 1. Another iteration is performed after the first iteration just to guarantee the convergence of the Gauss-Seidel loop and also to update the status of the neurons for which its input has crossed the threshold at this point of time. As stated in the last paragraph, neuron inputs are evaluated after the neuron outputs in PULSE and hence during the first iteration it is impossible to know this change in status.

This concludes the analytic study of the cost of PULSE simulation. The three major factors responsible have been described and their effects have been evaluated. Exploiting the network properties, the effect of one of those factors has been diminished to speed up the simulator. The following section

will present the statistics of PULSE and HSPICE simulations for some common networks to show the speed improvement that has been achieved over HSPICE.

6.2 Speed Improvement in PULSE

PULSE has be already proven to be a reliable simulator for pulsed analog networks built using our topology. This section discusses on the speed improvement that has been achieved in PULSE.

Here, the speed improvement of PULSE is presented by comparing it with HSPICE as is the standard in the circuit simulation literature. The rationale behind this is simple. SPICE is the most widely available simulator and thus is a good benchmark. Although it would have been nice to compare PULSE with some of the third generation simulators, the implementation of our macromodels in a third generation simulator is so time-consuming that it was actually one of our motivations behind building PULSE.

The speed improvement in PULSE is difficult to present in an analytical framework owing to the constraint that though PULSE simulation time can be found to be close to Eqn. 6.1, there is no way to predict the HSPICE simulation time. This problem is common and hence simulator speed improvements are always reported by citing the improvements achieved in typical circuits. Following the same track, Table 6.2 shows the statistics of PULSE and HSPICE simulation for common neural networks simulated in Chap. 5.

As reflected in the table, HSPICE gets slower as the number of neurons (input and standard) increases in a circuit. For example in the two circuits where the number of neurons dominate, i.e. the XOR and the associative memory circuit, HSPICE is much slower than PULSE. In the other two cases, since the circuit contains fewer number of neurons, HSPICE speed improves (the circuits simulated being devoid of input neurons). But still a two order of speed difference can be seen in the performance of the two simulators.

The above observation can be readily explained. The neurons in PULSE are modeled as logic blocks. In contrast to that HSPICE performs circuit simulation for the whole neuron circuit to predict its output. The neuron circuit being bulky, considerable time is wasted by HSPICE in simulating that circuit and hence it gets slower with increasing number of neurons.

Also referring to the literature review section, HSPICE simulation time is a nonlinear function of n, where n = no. of nodes present in the circuit. This stems from HSPICE using sparse matrix solution methods for solving the equations generated from Newton-Raphson steps. To avoid this, PULSE uses Gauss-Seidel iteration methods (as in Third Generation Simulators) which takes $O(n)$ time to solve a set of n linear equations. Though the individual contribution of this algorithm can not be separated out in the speed improvement achieved in this simulator, its effectiveness in keeping the PULSE simulation time down while simulating large networks can be acknowledged theoretically.

PULSE STATISTICS FOR THE CIRCUITS
SIMULATED IN CHAPTER 5

- *Simple Neural Network*
  Simulated for 150ns
  Standard Neuron - 1
  Synapses - 3 (2 Ex, 1 In)

	HSPICE	PULSE
CPU Time (in seconds)	47	0.4

Speed Improvement - 117 times

- *XOR Network*
  Simulated for 500ns
  Standard Neuron - 3
  Synapses - 6 (6 Ex)
  Input Neurons - 4 (2 Std, 2 Inv)

	HSPICE	PULSE
CPU Time (in seconds)	600	1.56

Speed Improvement - 384 times

- *Content Addressable Memory*
  Simulated for 300ns
  Standard Neuron - 7
  Synapses - 77 (35 Ex, 42 In)

	HSPICE	PULSE
CPU Time (in seconds)	734	6.56

Speed Improvement - 112 times

- *Associative Memory*
  Simulated for 300ns
  Standard Neuron - 3
  Synapses - 48 (24 Ex, 24 In)
  Input Neurons - 16 (8 Std, 8 Inv)

	HSPICE	PULSE
CPU Time (in seconds)	2572	6.77

Speed Improvement - 428 times

Table 6.2: Comparing PULSE and HSPICE speed (on a DECStation 5000/200)

From Table 6.2, the speed improvement achieved can be seen to vary from 100 to 400 times which shows PULSE to be at least two orders faster than HSPICE while simulating pulsed networks built using our topology. This order of improvement in simulation time was a major goal in building this simulator and the results presented in this section and the previous chapter shows that this goal has been achieved.

6.3 Concluding Remarks

This chapter provides an analytical study of PULSE simulation time. The speed of the two simulators HSPICE and PULSE has been compared. PULSE is seen to be at least two orders faster than HSPICE while providing the same accuracy as in HSPICE. This increases designer efficiency and makes exhaustive simulation of a network possible.

Chapter 7

Conclusions

This thesis describes the development of a simulator that provides a two order of magnitude speed improvement over HSPICE for our pulsed analog topology. This speed-up achieved reduces the simulation time from days to minutes which makes exhaustive simulation of large neural networks p ssible.

While building this simulator, both second and third generation simulation techniques have been used. *Waveform Gauss-Seidel*, a third generation technique has been used to decouple the circuit equations formed (by the simulator) whereas *functional iteration* has been used to solve those decoupled equations. Functional iteration, an old technique in circuit simulation, is not commonly used in present day circuit simulators since it imposes timestep limit in transient analysis. As argued in this thesis, this is not a serious constraint in our topology and hence this simple and accurate algorithm has been used to replace two major steps in conventional circuit simulation, numerical integration and NR iteration.

By proper ordering of the circuit equations and exploitation of the peculiarities of pulsed analog topology, the number of Gauss-Seidel iterations has been reduced to one in this simulator. Also accurate macromodels have been used for the synapses thus reducing the computation time significantly as compared to the transistor level simulation performed in any circuit simulator.

The aim of building PULSE was fast and accurate simulation of large networks built using our topology and exploration of new neural networks. Though most of the research period was spent in building and validating the simulator, the advantage of having this tool is already evident in implementation of a matrix associative memory. Extensive simulation of this network was only possible because we had this simulator. Doing the same number of simulations with HSPICE would have taken months; they were done in one day using PULSE.

Also in case of analog neural nets the circuit output information is present in the running average of the pulses in contrast to the presence or absence of pulses at the output as in digital networks. To calculate the running average, one has to simulate the network for significant duration (say, 2000ns) which amounts to a day's simulation with HSPICE for even small networks. Hence for simulation of analog networks, HSPICE is too expensive.

Some interesting points have been observed during the whole work. As described in Chapter 5, the prediction of the exact timing of the output

pulses in this kind of networks seem to be impossible, i.e. this phenomenon looks inherently chaotic. Since this phenomenon seems be an inherent characteristic of our pulsed topology, it is required to be formally investigated to make our understanding of the topology clearer.

Also, though this simulator has been experimentally seen to be able to simulate larger circuits than it is possible to build using our standard cells and CMOS3DLM process, the size of the code generated by the simulator increases as the circuit size increases which is not desired. That suggests more and more memory requirement as we will think of building larger and larger circuits spanning say a set of chips. At that point of time, one has to really take a hard look at the circuit representation in this simulator. A linked list representation of the circuit connectivity seems to be one of the choices to decrease the memory requirement of this simulator though the best solution seems to be unclear at this moment.

However at this point of time, PULSE provides the circuit designer with a significant edge. Armed with a simulator that is two orders faster than the circuit simulators available, a designer now has the power to explore new network topologies i.e. he is now able to examine the implications of our circuit design approach on large scale networks that was impossible using HSPICE. In that way, development of this simulator opened a new door in implementation of pulsed analog networks using our topology.

References

Allen P. E. and Holberg D. R. (1987), *CMOS Analog Circuit Design*, Holl, Rinehart and Winston, New York.

Banerjee T. (1993), "Building a Cadence Flat Netlister for One's Own Simulator," Canadian Microelectronics Corporation (CMC) Application Report.

Bhattacharya D. (1991), "Design and Analysis of Auto Scaling Pulsed Analog Neural Circuits," *M. Eng. thesis*, Memorial University of Newfoundland, St. John's, Newfoundland, Canada, 1991.

Cadence Open Simulation System (OSS) Reference Manual

Chawla B. R., Gummel H. K. and Kozak P. (1975). "MOTIS - An MOS Timing Simulator ," *IEEE Transactions on Circuits and Systems*, Vol. 22, No. 12, pp. 901-910.

Chua L. O. and Lin P. M. (1975), *Computer-Aided Analysis of Electronic Circuits : Algorithms and Computational Techniques*, Prentice-Hall, Inc, Englewood Cliffs, N. J.

CMOS3DLM Cell Library, Canadian Microelectronics Corporation Report. 1989.

De Man H. J. (1979). "Computer-Aided Design for Integrated Circuits:Trying to Bridge the Gap," *IEEE Journal of Solid-State Circuits*, Vol. 14, No. 3, pp. 613-621.

Geiger R. L., Allen P. E. and Strader N. R. (1990), *VLSI Design Techniques for Analog and Digital Circuits*, McGraw-Hill, Inc, N. Y.

Kreyszig E. (1988), *Advanced Engineering Mathematics*, 6th Edition, John Wiley and Sons, N. Y.

Lelarasmee E., Ruheli A. E., Sangiovanni-Vincentelli (1982), "The Waveform Relaxation Method for Time Domain Analysis of Large Scale Integrated Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 1, No. 3, pp. 131-145.

McCalla W. J. and Pederson D. O. (1971). "Elements of Computer-aided Circuit Analysis," *IEEE Transactions on Circuit Theory*, Vol. 18, No. 1, pp. 14-26.

Murray A. F. (1991), "Analogue Noise-enhanced Learning in Neural Network Circuits," *Electronics Letters*, Vol. 27, No. 17, pp. 1546-1548.

Murray A. F. (1992), "Multilayer Perceptron Learning Optimized for On-Chip Implementation - A Noise-Robust System," *Neural Computation*, Vol. 4, No. 3, pp. 366-381.

Murray A. F. (1991), "Silicon Implementation of Neural Networks," *IEE Proceedings-F*, Vol. 138, No. 1, pp. 3-12.

Murray A. F. and Smith A. V. W. (1987), "Asynchronous Arithmetic for VLSI Neural Systems," *Electronics Letters*, Vol. 23, No. 12, pp. 642-643.

Murray A. F. and Smith A. V. W. (1988), "Asynchronous VLSI Neural Networks using Pulse-stream Arithmetic," *IEEE Journal of Solid-State Circuits*, Vol. 23, No. 3, pp. 688-697.

Nagel L. W. (1975), "SPICE2: A Computer Program to Simulate Semiconductor Circuits," Memorandum No. UCB/ERL M520, 9 May 1975.

Newton A. R. (1979). "Techniques for the Simulation of Large-Scale Integrated Circuits," *IEEE Transactions on Circuits and Systems*, Vol. 26, No. 9, pp. 741-749.

Newton A. R. and Sangiovanni-Vincentelli A. L. (1984). "Relaxation-Based Electrical Simulation," *IEEE Transactions on Computer-Aided Design*, Vol. 3, No. 4, pp. 308-330.

Nilsson N. J. (1965), *Learning Machines*, McGraw-Hill, New York, NY.

Pao Y. H. (1988), *Adaptive Pattern Recognition and Neural Networks*, Addison-Wesley Publishing Company, Inc., Massachusetts.

Pederson D. O. (1984). "A Historical Review of Circuit Simulation," *IEEE Transactions on Circuits and Systems*, Vol. 31, No. 1, pp. 103-111.

Saleh R. A. and White J. K. (1990). "Accelerating Relaxation Algorithms for Circuit Simulation using Waveform-Newton and Step-size Refinement," *IEEE Transactions on Computer-Aided Design*, Vol. 9, No. 9, pp. 951-958.

Szygenda S. A. and Thompson E. W. (1975). "Digital Logic Simulation in a Time-based Table-driven Environment," *Computer*, March 1975, pp. 24-36.

Weeks W. T., Jimenez A. J., Mahoney G. W. , Mehta D., Qassemzadeh H. and Scott T. R. (1973). "Algorithms for ASTAP - A Network-Analysis Program," *IEEE Transactions on Circuit Theory*, Vol. 20, No. 6, pp. 628-634.

White J. K. and Sangiovanni-Vincentelli A. (1987), *Relaxation Techniques for the Simulation of VLSI Circuits*, Kluwer Academic Publishers, Norwell, Massachussettes.

Appendix A

Minimum Time-Step for Predictor-Corrector Method

This appendix refers to our discussion in Chapter 4 about the *worst-case* time-step that can be used in PULSE while using the predictor-corrector algorithm. The maximum timestep that can be used with our predictor-corrector algorithm is [Chua and Lin, 1975]

$$h < \frac{2}{|\partial f/\partial x|} \tag{A.1}$$

Hence, the worst-case time-step will occur at the maximum absolute value of $|\partial f/\partial x|$. To explain the method of finding this worst case value, we will refer to Fig. 4.8.

Let the sum of the currents at the neuron input node of Fig. 4.8 be I, hence $I = I_1 + I_2 - I_3$. A comparison of Eqn. 4.6 and Eqn. 4.7 reveals that in our case, $\partial f/\partial x$ is $(1/2C)\partial I/\partial V_m$.

Hence, the worst case time-step will correspond to the maximum value of

$|\partial I/\partial V_m|$. From definition of I, one can write Eqn. A.1 as

$$h < \frac{2mC}{|mx - ny|} \tag{A.2}$$

where, m and n are respectively the number of excitatory and inhibitory synapses connected to the neuron input node and x and y are such values of $\partial I/\partial V_m$ for excitatory and inhibitory synapses that $|mx - ny|$ is maximum at a given V_m.

Equation A.2 can also be written as

$$h < \frac{2C}{|x - py|} \tag{A.3}$$

where, $p=\frac{n}{m}$. from equation A.3 the worst-case time values were calculated which are tabulated below (this table is repeated in Chapter 4).

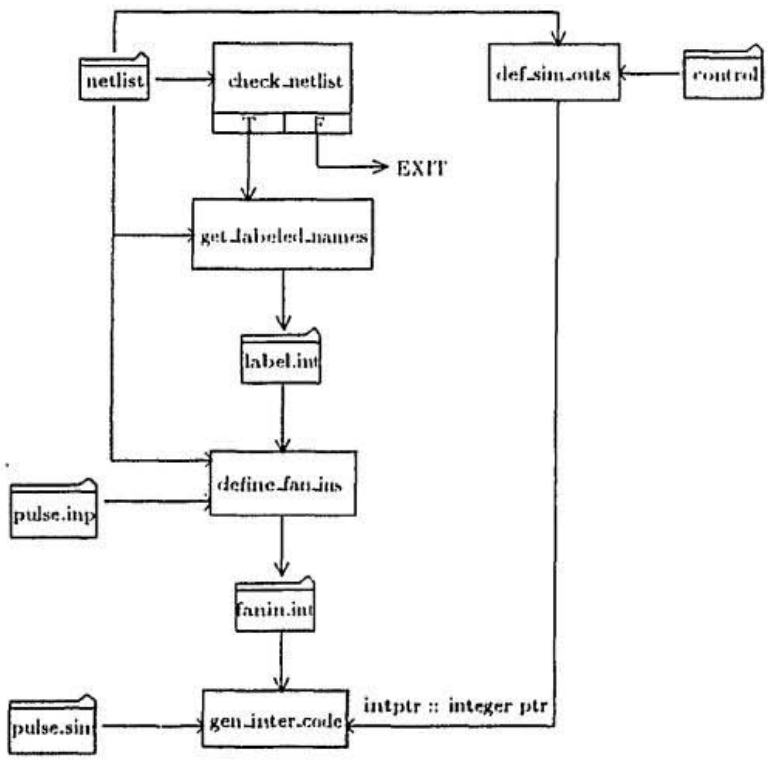
Ratio of excitatory and inhibitory syn	Worst case timestep (in ns)
3	1.6ns
2	1.6ns
1	1.2ns
1/2	0.73ns
1/3	0.52ns

Table A.1: Worst-case time step

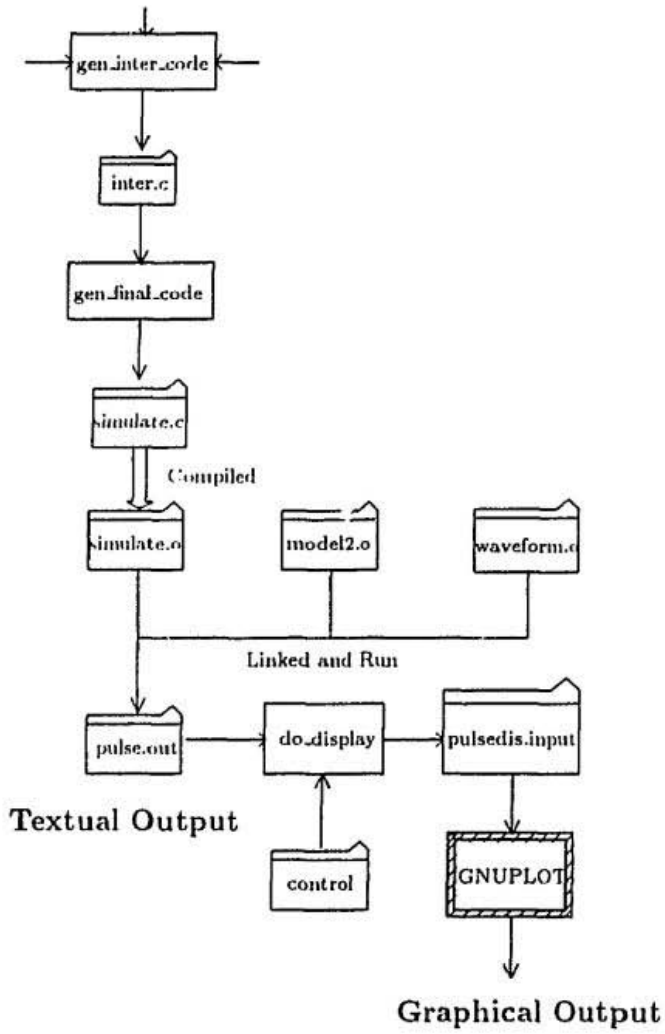
Appendix B

Source Code of the Simulator

B.1 Data Flow Diagram of PULSE



(Continued in the next page . . .)


gen_inter_code
inter.c
gen_final_code
simulate.c
Compiled
simulate.o
model2.o
waveform.o
Linked and Run
pulse.out
do_display
pulsedis.input
Textual Output
control
GNUPLOT
Graphical Output

B.2 PULSE Source Code

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <ctype.h>

 /* THIS IS THE SOURCE CODE FOR SIMULATOR PULSE. TO RUN THIS
    SIMULATOR, USER FIRST PROVIDES FOUR INPUT FILES -
    i)   THE CIRCUIT CONNECTIVITY DESCRIPTION, CALLED netlist.
    ii)  A FILE DEFINING THE INPUTS TO THE CIRCUIT, CALLED pulse.inp.
    iii) A FILE DEFINING THE TIME-PERIOD AND TIMESTEP OF TRANSIENT
    ANALYSIS, CALLED pulse.sim.
    iv)  A FILE DEFINING THE OUTPUTS THAT THE USER WANTS, CALLED contr
ol.

    AFTER READING THESE FILES, THIS PROGRAM GOES THROUGH SOME INTERMED
IATE
    STEPS BEFORE WRITING THE FINAL CUSTOMIZED FILE SIMULATE.C FOR THE
GIVEN
    CIRCUIT. THIS FILE SIMULATE.C IS A C PROGRAM DEFINING THE NODE EQU
ATIONS
    OF THE CIRCUIT AND THE ALGORITHMS REQUIRED TO SOLVE THOSE EQUATION
S.

    THIS FILE (simulate.c) IS FINALLY COMPILED BY THIS PROGRAM ALONGWI
TH TWO
    MORE FILES -
    i)  MODEL2.C (the file defining the table lookup synapse models an
d the
    neuron models)
    AND
    ii) EXTRY1.C (the file defining the waveforms required by the simu
-
    lator, till now only PULSE waveform is defined)
    TO RUN THE SIMULATION.

    AT THE END OF THE SIMULATION RUN, GRAPHIC OUTPUT IS PRODUCED FOR T
HE
    USER-REQUESTED NODE VOLTAGES USING GNUPLOT. TEXTUAL OUTPUT IS ALSO
    PRODUCED AND SAVED IN THE FILE pulse.out. FINALLY THE CPU TIME TAK
EN
```

```
    IN THE SIMULATION RUN AND THE BREAKDOWN OF THAT TIME IN SIMULATOR
SETUP
    TIME AND SIMULATOR RUN TIME IS DISPLAYED IN THE OUTPUT. */


int net=0,stdno=0,min=1000;
float timeperiod=0.;

/*  net    = THE TOTAL NUMBER OF NODES PRESENT IN THE CIRCUIT
    stdno = TOTAL NUMBER OF STANDARD NEURONS PRESENT
    min    = IN THE NETLIST, Std Neuron's ARE FOLLOWED BY ARBITRARY NO.
's. THIS
    VARIABLE STORES THE LOWEST OF THOSE NUMBERS.
*/

void check_netlist();
void get_labeled_names();
void define_fan_ins();
int *def_sim_outs();
void gen_inter_code(int *);
void gen_final_code();
void do_display();

main()
{ char response;
  clock_t start,start2,end;

  start=clock();            /* Mark the starting time point */

  check_netlist();
  get_labeled_names();      /* Go through the intermediate steps befor
e */
  define_fan_ins();         /* writing the final file simulate.c
  */
  gen_inter_code(def_sim_outs());
  gen_final_code();

  printf("Do you want to run it (y/n) ? ");scanf("%c",&response);
  if (response=='y')

  /* Compile the file simulate.c alongwith two other files, remove the
       intermediate files and the output file of the last simulation r
un */
```

```
  { system("gcc -o extry waveform.o model2.o simulate.c -lm ");/*rm la
bel.int;
 rm fanin.int;rm inter.c;rm pulse.out*/


    start2=clock();             /* Mark the end of simulator setup tim
e  */
    system("extry > pulse.out");
    end=clock();                /* Mark the end of simulator run time
   */

    printf("\nCPU time taken is %gs\n",(end-start)*1e-06);
    printf("\nSetup time is %gs\n",(start2-start)*1e-06);
    printf("Simulator run time is %gs\n",(end-start2)*1e-06);

    do_display();                   /* Display the outputs using gnup
lot     */

  }
}

/*************************************************************************
*******/
/*                              if_file_err
      */
/*
      */
/* If the input file is not present in the current directory, this fun
ction */
/* detects that, prints an error message and exits the program. All th
e     */
/* modules in this program uses this function before opening any file
for   */
/* reading.
      */
/*************************************************************************
*******/


void if_file_err(FILE *flptr,char filename[100])
{
  if (!flptr)
  {  printf(" ** ERROR : Can't open file %s **\n\n", filename);
     exit(0);
```

```
  }
}

/*************************************************************************
*******
                    syntax_check (used by function check_netlist)

 This function checks the syntax of the connectivity description of a
given
 element. The inputs to this function are as follows
   *s = the connectivity description of an element,
   no_of_arg = the number of nodes present in the element
   line_no = The line number of "netlist" file where s occurs

  This function first checks whether the string entirely consists of A
lpha-
  numeric characters and a single ; or not. Presence of two continuous
 blanks
  is an error. Also if no_of_arg doesn't tally with the no. of nodes p
resent
  in the element description, an error is raised. In all cases, the li
ne
  no. of the netlist where the error occurs is indicated in the error
message.

*************************************************************************
*****/

void syntax_check(char *s, int no_of_arg, int line_no)
{ int i=0,j=0,flag=1; /* flag is set to 0 in case of an error */

  for (i=2;i<=(strlen(s)-3);i++) /*The first two characters define the
 element*/
  { if (!isspace(s[i])) {if (!isdigit(s[i])) {flag=0;}}
    if ((isspace(s[i])) && (!isspace(s[i+1]))) {j++;} /*j=no of blanks
 in line*/
    if ((isspace(s[i])) && (isspace(s[i+1]))){flag=0;}
  }
  if (s[i]!=';') {flag=0;}
  if (j!=(no_of_arg+1)) flag=0;

  if (!flag) {printf("\n\n ** ERROR : Element format at line %i of net
list file is wrong **\n\n", line_no); exit(0);}
}
```

```
/*************************************************************************
*******/
/*                                check_netlist
      */
/*
      */
/* This function checks the validity of the input netlist. The netlist
 file */
/* is assumed to be of the following form

   // Netlist Generated by PULSE Netlister //  <---  The first line is
 assumed
                                                    to be a comment a
nd is
                            ignored (this line cannot start with string
 *net).
   *net0=/I1
    ........
   *net<no>=/netname                          <---  Followed by a lis
t of
    ........                                        node names and nu
mbers.
   *net19=/I6

   and a list of element connectivity descriptions. This description i
s
   different for each elements and is described below. The first line
of the
   description is a comment in each case as is evident from the leadin
g * and
   can be left out.

   (a)   *neuron(<index>)=/I<instance no.>
         SN<index> <n1> <n2> <n3> <n4> <n5> ;

      This description refers to a neuron with <n1> and <n2> as its in
put and
   threshold nodes. <n3> refers to its bias voltage. <n4> and <n5> ref
ers to
   its discharge and output pulse nodes respectively.

   (b)   *inNeuron1(<index>)=/I<instance no.>
         SI<index> <n1> <n2> <n3> <n4> ;
```

```
      This description refers to a standard input neuron with <n1> and
  <n2> as
    its input and threshold nodes. <n3> refers to its bias voltage. <n4
 > refers
    to its output node.

    (c)    *inNeuron0(<index>)=/I<instance no.>
           II<index> <n1> <n2> <n3> <n4> ;

      This description refers to a inverting input neuron with <n1> an
d <n2> as
    its input and threshold nodes. <n3> refers to its bias voltage. <n4
 > refers
    to its output node.

    (d)    *synapse1(<index>)=/I<instance no.>
           ES<index> <n1> <n2> <n3> <n4> <n5> ;

      This description refers to an excitatory synapse with <n1> and <
n2> as
    its input and discharge nodes. <n3> refers to its weight voltage.
<n4> and
    <n5> refers to its leakage and output node respectively.

    (e)    *synapsein(<index>)=/I<instance no.>
           IS<index> <n1> <n2> <n3> ;

      This description refers to an inhibitory synapse with <n1> and <
n2> as
    its input and weight nodes. <n3> refers to its output node.

/* The checks performed are now described in sequence. First this func
tion  */
/* checks whether a list of nets exist in the netlist or not. Then it
checks*/
/* whether the elements present in the netlist are known to the simula
tor   */
/* or not. If the element is known, then the correctness of the syntax
 is   */
/* checked with the help of function syntax_check. Finally,it checks w
hether*/
/* a net appearing in the list of nets appears also in the connectivit
y     */
```

```
/* description or not and vice versa.
      */
/*
      */
/* In case of the first two errors, the errors are fatal and this func
tion  */
/* prints an error message and exits the program. In the last case,it
issues*/
/* an warning and continues execution
      */
/* DEFINE EXACT NET LIST
                              */
/*****************************************************************
*******/

void check_netlist()
{ FILE *flptr;
  int *buf,i,n;
  char s[100],scopy[100],s1[8];

  flptr=fopen("netlist","r");
  if_file_err(flptr,"netlist");

  while (fgets(s,100,flptr))              /* Calculates no. of nets pr
esent */
  { if (!(strncmp(s,"*net",4))) net++;};  /*       in the given netlist
      */
  rewind(flptr);

  if (!net) { printf("\n\n **** ERROR -  A list of the nets not presen
t in netlist ****\n\n",s1[2]); exit(0);}    /* If no nets are present,
 it's an error */

  buf=(int *) malloc(net*sizeof(int));
  for (i=0;i<net;i++) *(buf+i)=0;

  fgets(s,100,flptr);i=1;
  while (fgets(s,100,flptr))
  { i++;
    strcpy(scopy,s);
    if (strncmp(s,"*",i))
    { strcpy(s1,strtok(s," "));
      if ( (!(strncmp(s1,"SN",2))) || (!(strncmp(s1,"SI",2))) ||
   (!(strncmp(s1,"II",2))) || (!(strncmp(s1,"ES",2))) ||
```

```
   (!(strncmp(s1,"IS",2))) )   /* checks whether the element i
s known */
      { if ((!(strncmp(s1,"SN",2))) || (!(strncmp(s1,"ES",2))))
          syntax_check(scopy.5,i);
        if ((!(strncmp(s1,"II",2))) || (!(strncmp(s1,"SI",2))))
          syntax_check(scopy,4,i);
        if (!(strncmp(s1,"IS",2)))
          syntax_check(scopy,3,i);

        strcpy(s1,strtok(0," "));
while(strncmp(s1,";",1))
{ if (n>=net) printf("\n\n ** WARNING - Net No. %i does not ap
pear in the list of nets **\n\n",i);
          n = atoi(s1);
  *(buf+n)=1;
  strcpy(s1,strtok(0," "));
}
      }
      else { printf("\n\n **** ERROR -  Unknown Element %s present in
netlist ****\n\n",s1); exit(0);}
    }
  }

  for (i=0;i<net;i++) {if (!(*(buf+i))) printf("\n\n ** WARNING - Net
No. %i does not appear in the connectivity description **\n\n",i);}
}


/*************************************************************************
*******/
/*                              get_labeled_names
      */
/*
      */
/* Assuming a valid netlist, this function finds out the user-labeled
termi-*/
/* nals from the netlist and maps their names to corresponding node nu
mbers.*/
/* This mapping is written down in a file label.int.
      */
/*
      */
/* The node-names are of three types -
```

```
   (i)  Vdd! and gnd!
   (ii) User labeled node names (doesn't contain a . or a ! and are ve
ry few
        in number in a big circuit)
   (iii)Other names (may be automatically generated and is of the form

        <substring1>.<substring2>)

    The user-labeled node names are found from the netlist by detectin
g the
   absence of . or ! in the nodename. The output file label.int define
s all
   the labeled nodes in the network using tuples of the form
      */
/*       2=IN4;
      */
/* where 2 is the node number of the input and IN4 is its name.
      */
/*************************************************************************
*******/

void get_labeled_names()
{
  char oneline[80];
  FILE *flptr,*flptr2;

  flptr=fopen("netlist","r");   /* READS THE NETLIST FILE       */
  if_file_err(flptr,"netlist");
  flptr2=fopen("label.int","w"); /* WRITES I/P MAPPING FILE */

  while (fgets(oneline,80,flptr))
  { if (!(strncmp(oneline,"*net",4)))    /* checks starting of node=*n
et   */
    { /* if no . or ! present in the node entry then it is I/P or othe
r user */
      /* named node
       */

      if ((!strchr(oneline,'.')) && (!strchr(oneline,'!')))
      { strtok(oneline,"t");
fputs(strtok(0,"/"),flptr2);       /* s1 gets the node number
as 5=*/
fputs(strtok(0,"\n"),flptr2);      /* s2 gets say IN4
     */
```

```
fprintf(flptr2,";\n");              /* hence the entry is 5=IN4
     */
      }
    }
  }
  free(oneline);
  fclose(flptr);
  fclose(flptr2);

  printf("\n OUTPUT FILE label.int GENERATED SUCCESSFULLY !!\n\n");
}

/*********************************************************************
*******/
/*                                define_fan_ins
     */
/*
     */
/* Here we are trying to write down the fan-ins present at every node
of the*/
/* network in file fanin.int. In simulation, the fan-in of a node is t
he    */
/* set of all inputs that drive that node, therefore defining its volt
age.  */
/* The nodes present in the circuit are only of three types.
     */
/*
     */
/* They are i) inputs to the circuit (including weights, thresholds, a
nd    */
/* bias voltages, ii) neuron input nodes and iii) neuron output nodes.
     */
/*
     */
/* In case of neuron input nodes, we will get several entries in our o
utput */
/* file (fanin.int) due to the presence of several synapses. For neuro
n    */
/* o/p nodes, we will get single entries which is either NPULSE or DCP
ULSE  */
/* output of a given neuron.
     */
/*
     */
```

```
/* This function raises a warning if a node is present in pulse.inp an
d not */
/* present in label.int.
      */
/*

   The entries in the file pulse.inp are only of two forms

      Vnodename node-voltage;
   or Vnodename pulse(delay,rise,on,fall);
      */

/* Algorithm
      */
/* ---------
      */
/* i) Read the value or type (currently only pulse) of all inputs from
      */
/* pulse.inp.
      */
/*
      */
/* ii) Open the netlist file. If Vdd! and gnd! nodes are present in th
e net */
/* list, then map Vdd! and gnd! to 5v and 0v respectively.
      */  /*
                */
/* ii) Sweep through the netlist, ignoring comment lines. (Everything
else  */
/* is an element connectivity description.) During the sweep
      */
/*
      */
/* (a) Count the number of neurons since they have to be identified to
 keep */
/* track of their firing sequence.
      */
/*
      */
/* (b) Call function elmnt_descr for each element entry which writes i
n the */
/*     file fanin.int -
      */
/*
```

```
      */
/*      For neuron entries: 2 entries, of the form
<no> stdnpulse(V[<n1>],V[<n2>],delay,rise,on,fall,<num
>);
<nd> stddcpulse(delay,rise,on,fall,<num>);
where <no> and <nd> are the node numbers of the neuron output
and
discharge output respectively, <n1> and <n2> are the neuron in
put and
threshold node nos. respectively, delay,rise,on,and fall are t
he
corresponding times for the pulse being defined, and <num> is
the
number of the neuron. All other characters are exactly as type
d.

        For i/p neuron entries: 1 entry of the form
                <ns> stdinnrn(V[<n1>],V[<n2>]); (for standard i/p neur
on)
             or <ni> invinnrn(V[<n1>],V[<n2>]); (for inverting i/p neu
ron)
        where <ns> and <ni> are the output node numbers of the standar
d and
        inverting input neurons respectively. <n1> and <n2> are the ne
uron
        input and threshold node nos. respectively.

        For excitatory synapse entries: 1 entry of the form
                <no> exsynapse(V[<n1>],V[<n2>],V[<n3>],V[<n4>],V[<no>]
);
        where <no> is the node number of the excitatory synapse output
. <n1>
        and <n2> are the node numbers of the excitation and weight inp
uts to
        the excitatory synapse, whereas <n3> and <n4> are the node num
bers
        of the discharge and leakage input to the excitatory synapse.


        For inhibitory synapse entries: 1 entry of the form
                <no> insynapse(V[<n1>],V[<n2>],V[<no>]);
        where <no> is the node number of the inhibitory synapse output
. <n1>
        and <n2> are the node numbers of the inhibition and weight inp
```

```
uts to
        the inhibitory synapse.
      */

/*  The format of the entries in the output file fanin.int (as describ
ed    */
/*  above) follows  the model defined in model2.c.
      */
/*
      */
/*************************************************************************
*******/


void elmnt_descr(char *,FILE *); /* A declaration, definition follows
later */

void define_fan_ins()
{
  int i,m,flag;
  char s[100],pinp[100],ss[20],s2[100],*nodeno;
  FILE *flptr,*flptr2,*flptr3,*flptr4;

  flptr=fopen("label.int","r");
  if_file_err(flptr,"label.int");
  flptr2=fopen("fanin.int","w");
  flptr3=fopen("pulse.inp","r");
  if_file_err(flptr3,"pulse.inp");
  flptr4=fopen("netlist","r");
  if_file_err(flptr4,"netlist");

  while (fgets(pinp,100,flptr3))
  { strcpy(s2,pinp);
    while (fgets(s,100,flptr))
   { nodeno=strtok(s,"=");
     strcat(strcpy(ss,"V"),strtok(0,";"));
     flag=0;
     if(!strcmp(strtok(pinp," "),ss)) break;
     flag=1;
   }
   if (!flag) { strtok(s2," ");
fprintf(flptr2,"%s %s;\n",nodeno,strtok(0,";"));
      }
   else printf("\n\n **** WARNING - Input %s does not appear in the ne
```

```
tlist ****\n\n",strtok(s2," "));
    rewind(flptr);
   }

 fclose(flptr);
 fclose(flptr3);

 for (i=0;i<net;i++)                    /* IF NETLIST CONTAINS Vdd or GND
 */
 { fgets(s,100,flptr4);                 /* NET DECLRN TAKE CARE OF THAT
 */
   if (strchr(s,'!'))
   {
     if (strchr(s,'g'))                 /* IF 'g' DOES NOT OCCUR IN THE N
ET */
     { strtok(s,"t");                   /* THEN STRCHR RETURNS NULL
   */
       fputs(strtok(0,"="),flptr2);
       fprintf(flptr2," 0;\n ");
     }
     if (strchr(s,'v')!=0)
     { strtok(s,"t");
       fputs(strtok(0,"="),flptr2);
       fprintf(flptr2," 5;\n  ");
     }
   }
 }

 while (fgets(s,100,flptr4))            /* The SN's can have any number aft
er them*/
 { if (!strncmp(s,"SN",2))              /* Idea is to find the minimum of t
he no.s*/
   {                                    /* so that, that can be subtracted
from   */
   /* all those numbers, to make their
 no.   */
    strtok(s,"N");                      /* start from zero. Since array for
 them  */
    m=atoi(strtok(0," "));              /* is going to start from zero.
       */
    if (m<min) min=m;
   }
 }
 rewind(flptr4);
```

```
 while (fgets(s,100,flptr4))      /* IN NETLIST FILE,IF IT DOESN'T STAR
T WITH */
 {                                /* A *, THEN CAN BE VDD,GND OR ACTIVE
 ELEME */
   if (strncmp(s,"*",1))          /* NT INFO, FIRST IS ALREADY TAKEN CA
RE OF. */
   elmnt_descr(s,flptr2);         /* TAKE CARE OF THE SECOND POSSIBILIT
Y,THAT */
 }                                /* IS-ACTIVE ELEM USING COMMON FUNCTI
ON.ABOVE */
 free(s); free(pinp); free(ss);
 fclose(flptr2);
 fclose(flptr4);

 printf("\n OUTPUT FILE fanin.int GENERATED SUCCESSFULLY !!\n\n");
}

/*************************************************************************
*********
                  elmnt_descr (used by function define_fan_ins)

 It creates element description for each element connectivity descript
ion that
 is input to it. The connectivity description appears in the netlist f
ile.

*************************************************************************
********/
void elmnt_descr(char *s,FILE *flptr)
 /* TAKE A STRING, PARSE IT AND PUT REQUIRED STRING IN FANIN.INT */
 /* FOR ALL VALID ACTIVE ELEMENTS, NOW ONLY SY AND NEURONS          */
{
  char *s1,*s2,*s3,*s4,*s5;
  int n;

  if (!strncmp(s,"SN",2))
  {
    stdno++;
    strtok(s,"N");
    n=atoi(strtok(0," "));
    s1=strtok(0," "); s2=strtok(0," ");
    s3=strtok(0," "); s3=strtok(0," "); s4=strtok(0," ");
    fprintf(flptr,"%s stdnpulse(V[%s],V[%s],9.8,0.8,5,1.4,%i);\n",s4,s
```

```
1,s2,n-min);
    fprintf(flptr,"%s stddcpulse(10.2,0,13,0,%i);\n",s3,n-min);
  }
  if (!strncmp(s,"ES",2))
  {
    strtok(s," ");
    s1=strtok(0," "); s2=strtok(0," ");
    s3=strtok(0," "); s4=strtok(0," "); s5=strtok(0," ");
    fprintf(flptr,"%s exsynapse(V[%s],V[%s],V[%s],V[%s],V[%s]);\n",s5,
s1,s3,s2,s4,s5);
  }
  if (!strncmp(s,"IS",2))
  {
    strtok(s," ");
    s1=strtok(0," "); s2=strtok(0," "); s3=strtok(0," ");
    fprintf(flptr,"%s insynapse(V[%s],V[%s],V[%s]);\n",s3,s1,s2,s3);
  }
  if (!strncmp(s,"SI",2))
  {
    strtok(s," ");
    s1=strtok(0," "); s2=strtok(0," ");
    s3=strtok(0," "); s3=strtok(0," ");
    fprintf(flptr,"%s stdinnrn(V[%s],V[%s]);\n",s3,s1,s2);
  }
  if (!strncmp(s,"II",2))
  {
     strtok(s," ");
     s1=strtok(0," "); s2=strtok(0," ");
     s3=strtok(0," "); s3=strtok(0," ");
     fprintf(flptr,"%s invinnrn(V[%s],V[%s]);\n",s3,s1,s2);
  }
}

/*************************************************************************
********
                              def_sim_outs

   Reads the user-requested outputs from the file control. The format
of the
   control file is like this

   *9;             <-- User request for voltage at node 9
   *10;
   /N1;            <-- User request for voltage at node named N1
```

```
   i.e. the user can define the requested o/p's by node number or name
s.
   This function then builds a table of these node numbers. Finally it
 puts
   the number of o/ps requested by the user on top of the table and pa
sses
   the pointer of the top of the table to the next function.

   Consider the example that user requests o/p at node 23,45 and 64 i.
e. 3 o/ps.

   Table built -- > |  3 | --> Pointer to this entry is o/p of this fu
nction
    | 23 |      and is passed to the function gen_inter
_code
    | 45 |
    | 64 |
     ----

**********************************************************************
******/


int *def_sim_outs()                              /* Reads user request for o
utput */
{ char s[100],news[100],*s1,*s2;
  FILE *flptr,*flptr2;
  int *ptr,*buf,i=0;

  flptr=fopen("control","r");
  if_file_err(flptr,"control");
  flptr2=fopen("netlist","r");
  if_file_err(flptr2,"netlist");

  while (fgets(s,100,flptr)) i++;          /* counts no. of lines in con
trol  */
  ptr= (int *) malloc((++i)*sizeof(int));/* keep one more space to sto
re count*/
  buf=ptr;                                 /* keep top of array in buf
       */
  *ptr=i;
  rewind(flptr);
```

```
  while (fgets(s,100,flptr))
  { if (!strncmp(s,"*",1))                /* two types of nodes, by nod
e no. */
     *(++ptr)=atoi(strtok(strtok(s,"*"),";"));
    if (strncmp(s,"/",1)==0)              /*        or by name
     */
    { s1=strtok(s,";");
      for (i=0;i<=net;i++)
      { fgets(news,100,flptr2);
if (strncmp(news,"*",1)==0)
{ s2= strtok(news,"t");
  s2= strtok(0,"=");
  if (strcmp(s1,strtok(0,"\n"))==0) *(++ptr)=atoi(s2);
}
      }
    }
    rewind(flptr2);
  }
  rewind(flptr);
  free(s);free(news);
  fclose(flptr2);
  fclose(flptr);
  ptr=buf;

  return(buf);
}

/*************************************************************************
********
      gen_inter_code

  This function generates an intermediate C program inter.c which defi
nes the
  node equations for the circuit to be simulated and implements the Ga
uss -
  Seidel Relaxation algorithm used to solve those circuit equations. T
he node
  equations are built by extensively using the fanin connections defin
ed in
  the file fanin.int. The file inter.c also contains the data structur
e
  required to simulate a given circuit. It's best to take a look at th
e
  heavily commented sample file inter_sample.c (a copy of a sample int
```

```
er.c )
  present in this directory before peeping in this function.

  The pseudo-code for the generated file is given below :-

  do
  {  increment time by a timestep;
     store the node voltages calculated from last timepoint;
     do
     {  store the node voltages calculated in the last iteration;
        calculate all neuron outputs from plug-in logic equations;
        for all neuron inputs
        do
        {  if first iteration, use predictor to guess the node voltage;
           store last iterated value;
           calculate new iterated value;
        }
        while the two values do not converge;
     }
     while all the nodes haven't converged;
   }
   while (time <= timeperiod);

 ********************************* ***************************** *******
*******/


void gen_inter_code(int *intptr)
{
  FILE *flptr,*flptr2,*flptr3,*flptr4;
  char s[80],*s3;
  int i,j,n,*count,count2,countex,*buf;
  float p,q;

  flptr=fopen("fanin.int","r");
  if_file_err(flptr,"fanin.int");
  flptr2=fopen("inter.c","w");
  flptr3=fopen("pulse.sim","r");
  if_file_err(flptr3,"pulse.sim");

  fgets(s,80,flptr3);            /* READS IN .SIM FILE PARAMETERS */
  if (!strcmp(strtok(s," "),".tran"))
  { p=atof(strtok(0," "));
    q=atof(strtok(0,";"));
```

```
    timeperiod=q;
   }
   else {printf(" ** ERROR : Format of .sim file is not correct!! **\n\
n");
printf(" Example format for PULSE.SIM file : .tran .5 200;\n\n
");
exit(0);
        }
   fprintf(flptr2,"#include <stdio.h>\n\n");           /* WRITES INTER.
C FILE */
   fprintf(flptr2,"#include <math.h>\n\n");
   fprintf(flptr2,"double t=0,delt=%2.3f,tmark[%i];\n\nint flag[%i];\n\
n",p,stdno,stdno);
   fprintf(flptr2,"extern double pulse(double ondly,double riset,double
 ontime,double fallt,double offtime);\n\n");
   fprintf(flptr2,"extern double exsynapse(double aVex,double aVwt,doub
le aVdc,double aVlk,double Vm);\n\n");
   fprintf(flptr2,"extern double insynapse(double aVin,double aVwt,doub
le Vm);\n\n");
   fprintf(flptr2,"extern double stdnpulse(double aVin,double aVth,doub
le ondly,double riset,double ontime,double fallt,int n);\n\n");
   fprintf(flptr2,"extern double stddcpulse(double ondly,double riset,d
ouble ontime,double fallt,int n);\n\n");
   fprintf(flptr2,"extern double stdinnrn(double aVc,double aVth);\n\n"
);
   fprintf(flptr2,"extern double invinnrn(double aVc,double aVth);\n\n"
);
   fprintf(flptr2,"main()\n{\n");
   fprintf(flptr2,"  int i,j,flag1,flag2;\n");
   fprintf(flptr2,"  double store1[%i],store2[%i],temp,V[%i],Vlast[%i],
Vlast2[%i],sum=0.0;\n",stdno,stdno,net,net,net);
   fprintf(flptr2,"  for (i=0;i<%i;i++) { tmark[i]=0;flag[i]=0;store1[i
]=0;store2[i]=0;}\n",stdno);
   fprintf(flptr2,"  for (i=0;i<%i;i++) { V[i]=0;Vlast[i]=0;Vlast2[i]=0
;}\n",net);
   fprintf(flptr2,"  fflush(stdout);\n");
   fprintf(flptr2,"  do\n  {\n    t+=delt;j=0;\n");
   fprintf(flptr2,"    for(i=0;i<%i;i++) Vlast2[i]=V[i];\n",net);
   fprintf(flptr2,"    flag2=0;flag1=0;\n");
   fprintf(flptr2,"    do\n    { j++;\n      if (flag2!=%i) flag2=0;\n"
,net);
   fprintf(flptr2,"      for(i=0;i<%i;i++) Vlast[i]=V[i];\n    ",net);
   fprintf(flptr2,"  ");
   count=(int *) malloc(net*sizeof(int));
```

```
  for (i=0;i<net;i++)      /* COUNTS HOW MANY TIMES A PARTICULAR NODE
*/
    *(count+i)=0;                /*        OCCURS IN FANIN.INT
 */
  while (fgets(s,80,flptr))
  { n=atoi(strtok(s," "));
    *(count+n)= *(count+n)+1 ;
  }
  rewind(flptr);
  for (i=0;i<net;i++)
  { if (!(count+i))                        /* COUNT=0 MEANS NEITHER IT W
AS   */
    { flptr4=fopen("label.int","r");  /* MENTIONED IN PULSE.INP NOR AS
  */
      if_file_err(flptr4,"label.int");/* OUTPUT NET IN NETLIST. SO, IT
'S*/
      while (fgets(s,80,flptr4))         /* AN INPUT FOR SURE AND NOT
DEFI-*/
      { if (atoi(strtok(s,"="))==i)      /* NED IN PULSE.INP
     */
{ printf(" ** ERROR : Input %s not defined in file pulse.inp *
*\n\n",strtok(0,";"));
  printf(" Example format for PULSE.INP file : VIN1 5;\n\n");
  exit(0);
}
      }
      fclose(flptr4);
    }

    if (*(count+i)==1) /* IF ONE OCCURRENCE THEN JUST TAKE THE STRING
AND PUT */
    { while (fgets(s,80,flptr))                  /* IN INTER.C
   */
      { n=atoi(strtok(s," "));
if(n==i) { fprintf(flptr2,"V[%i]=",i);
   fputs(strcat(strtok(0,";"),";\n      "),flptr2);
 }
      }
      rewind(flptr);
    }
  }

  for (i=0;i<net;i++)
  { if (*(count+i)>1)                  /* IF COUNT>1, FOR NOW WE ASSUME
```

```
 NODE */
    { count2=0; countex=0;            /*  IS ONLY MADE UP OF EX AND IN
 SYN  */
      while (fgets(s,80,flptr))
      { n=atoi(strtok(s," "));
if (n==i) { count2++;
    s3=strtok(0,";");
    if (count2==1) fprintf(flptr2,"V[%i]=Vlast2[%i]+((
",i,i);
    if (!strncmp(s3,"ex",2)) /* IF NODE HAS EX SYN THE
N NEEDS +*/
    { if (count2==1) fputs(s3,flptr2);/* SINCE ADDS CU
RRENT, ALSO */
      else { fputs("+",flptr2);/* KEEP COUNT SINCE CAP
 INCREASES */
     fputs(s3,flptr2); /* WITH IT PROPORTIONAL
LY          */
   }
      countex++;
    }
    if (!strncmp(s3,"in",2))  /* SAME AS ABOVE,PUT- FO
R CURRENT */
    { fputs("-",flptr2);        /* TAKEN OUT, BUT NO C
AP INCREASE */
      fputs(s3,flptr2);
    }
    if (!strncmp(s3,"stdn",4)) /* SAME AS ABOVE,PUT+FO
R STDNPULS*/
    { fputs("+",flptr2);
      strcat(s3,"*78e-06");
      fputs(s3,flptr2);
    }
    if(count2==*(count+i))
       fprintf(flptr2,")*delt*1e-09)/(%i*0.15e-12);\n
     ",countex);
  }
      }
      rewind(flptr);
    }
  }

  fprintf(flptr2,"if (flag2!=%i) for (i=0;i<%i;i++) if ((fabs(V[i]-Vla
st[i]))<0.00i) flag2++;\n",net,net);
  fprintf(flptr2,"    }\n    while ((flag2<%i) || (flag1!=i));\n",net)
```

```
;
  fprintf(flptr2,"    printf(\"");
  fprintf(flptr2,"%%g ");
  for (i=1;i<*intptr;i++) fprintf(flptr2,"%%g ");
/*for (i=0;i<*intptr;i++) fprintf(flptr2,"%%f ");*/
  fprintf(flptr2,"\\n\",(t*1e-09)");buf = intptr;
  for (i=1;i<*buf;i++) fprintf(flptr2,",V[%i]",*(++intptr));
  fprintf(flptr2,");\n");
  fprintf(flptr2,"  }\n  while(t<%5.3f);\n}\n",q);
  free(s);
  fclose(flptr);
  fclose(flptr2);
  fclose(flptr3);
  printf("\n INTER.C GENERATED SUCCESSFULLY !!\n\n");
}

void gen_final_code()
{ FILE *flptr,*flptr2;
  char oneline[5000],twoline[5000],*buf;
  int nodeno,i;
  flptr=fopen("inter.c","r");     /*Reads the intermediate C file */
  if_file_err(flptr,"inter.c");
  flptr2=fopen("simulate.c","w");     /* Writes final file */

  while(fgets(oneline,5000,flptr))
 { if (strstr(oneline,"]+("))
   { strtok(oneline,"[");
     nodeno=atoi(strtok(0,"]"));
     strtok(0,"+");
     free(oneline);
     strcpy(twoline,strtok(0,";"));
     fprintf(flptr2,"      if (j==1) V[%i]=Vlast2[%i]+0.5*(3*store1[%i
]-store2[%i]);\n",nodeno,nodeno,i,i);
     fprintf(flptr2,"      i=0;\n      if (flag2<%i){ do\n      { i++;
\n        if (i!=1) V[%i]=temp;\n",net,nodeno);
     fprintf(flptr2,"        temp=Vlast2[%i]+0.5*(store1[%i]+",nodeno,
i);
     buf=twoline;
     while (*buf!='\0') fputc(*buf++,flptr2);
     fprintf(flptr2,");\n");
     fprintf(flptr2,"      if (i>10000) {fprintf(stderr,\"\\n\\n ******
  No convergence - try with smaller timestep ******\\n\\n\");\n
            exit(0);}\n");
     fprintf(flptr2,"      }\n      while((fabs(temp-V[%i]))>0.001);\n
```

```
",nodeno);
     fprintf(flptr2,"      V[%i]=temp;}\n",nodeno);
     fprintf(flptr2,"      if (flag2==%i) {store2[%i]=store1[%i];\n",n
et,i,i);
     fprintf(flptr2,"      store1[%i]=(",i);
     buf=twoline;
     while (*buf!='\0') fputc(*buf++,flptr2);
     fprintf(flptr2,");\n      flag1=1;}\n");
     i++;
   }
   else fputs(oneline,flptr2);
 }
 free(oneline);free(twoline);
 fclose(flptr);
 fclose(flptr2);
 printf("\n FINAL.C GENERATED SUCCESSFULLY !!\n\n");
}

void do_display()                    /*does the display*/
{ FILE *flptr1,*flptr2;
  int counter=1;
  char oneline[80],twoline[80];

  flptr1=fopen("control","r");   /* READS THE CONTROL FILE TO KNOW
*/
  if_file_err(flptr1,"control"); /* WHICH OUTPUTS ARE TO BE DISPLAYED
*/
  flptr2=fopen("pulsedis.input","w");

  fprintf(flptr2," set terminal X11\n set format xy \"%%g\"\n set xlab
el \"t(in s)\"\n set ylabel \"V\\\\\\(in Volts\)\" \n plot [t=0:%5.3fe-
09] ",timeperiod);

  while (fgets(oneline,80,flptr1))
  { if ((!(strncmp(oneline,"/",1))) || (!(strncmp(oneline,"*",1))))
    { strcpy(twoline,strtok(oneline,";"));
      if (counter==1) fprintf(flptr2," \"pulse.out\" using 1:%i title
\"%s\" with lines",++counter,&twoline[1]);
      else fprintf(flptr2,", \"pulse.out\" using 1:%i title \"%s\" wit
h lines",++counter,&twoline[1]);
    }
  }
  fprintf(flptr2,"\n pause -1  \"  Hit return to continue ...\"\n");
  free(oneline); free(twoline);
```

```
  fclose(flptr1);
  fclose(flptr2);
}
```

B.3 The Macromodels

```
#include <math.h>

extern double t,delt,tmark[]; /* Supposed to be defined in netlist4.c
*/

extern int flag[];

extern double pulse(double ondly,double riset,double ontime,double fal
lt,double offtime);

extern double exsynapse(double aVex,double aVwt,double aVdc,double aVl
k,double Vm) /*OUTPUT in A*/
{
  int i,j;
  double I1,Idc,sum,a[14],Vmarray[14],Vwtarray[14];

  static double leakcoeff[14]={ -6.603051934256143e-07,1.0271713577150
63e-05,-6.963168316429258e-05,2.686739238106694e-04,-6.446822966975835
e-04,9.800099067403378e-04,-9.016468635105736e-04,4.056735443102641e-0
4,3.509327673376104e-05,-1.282451851932237e-04,5.516554485257316e-05,-
1.615123804007882e-05,8.452632463713466e-06,-2.327727950646864e-10};

  static double discoeff[14]= { -1.601284486756084e-06,2.4165241379361
21e-05,-1.611046416958130e-04,6.257991711893582e-04,-1.573562772290460
e-03,2.692477056156958e-03,-3.209451404119135e-03,2.682522438299104e-0
3,-1.557765884323081e-03,6.068197617998657e-04,-1.307471339332492e-04,
-7.523948155120020e-05,2.606179952161005e-04,4.966812190786698e-10};

  static double coeff[14][14]={
 { -1.031930954808331e-09,-2.236835027325452e-08,4.960705759973295e-07
,-3.694837598086475e-06,1.477215757419324e-05,-3.576816459453455e-05,5
.431890342467863e-05,-5.108985080230358e-05,2.796031753203229e-05,-7.6
42238810641949e-06,6.848197749382061e-07,-2.773217400575023e-08,2.5667
06403239300e-08,-3.791300627928867e-09},
 {  4.440046418678889e-08,5.489448925647399e-07,-1.487967849432084e-05
```

,1.153355207426963e-04,-4.696274337992896e-04,1.150246738257972e-03,-1
.762141521235776e-03,1.670135713973943e-03,-9.208501289095139e-04,2.53
5196227673102e-04,-2.266843493367775e-05,7.923082251947870e-07,-8.4033
61467601280e-07,1.251202622269522e-07},
{ -8.105462818409834e-07,-5.081068492385639e-06,1.943957174260033e-04
,-1.584079975106061e-03,6.590479366470540e-03,-1.635538627142836e-02,2
.530458994552041e-02,-2.419139165686240e-02,1.345148424944125e-02,-3.7
34997502061011e-03,3.337254543516843e-04,-9.542836462338302e-06,1.2191
20141991981e-05,-1.827760249381630e-06},
{ 8.393932300647426e-06,1.639490908029626e-05,-1.446709362374472e-03,
1.258052216462729e-02,-5.370970150215657e-02,1.353351675328465e-01,-2.
117577633216307e-01,2.044363222770373e-01,-1.147799814889808e-01,3.219
201505222989e-02,-2.879810823108894e-03,6.119268220074650e-05,-1.03149
3393400954e-04,1.557496878175554e-05},
{ -5.514837606026973e-05,7.043725583151204e-05,6.718447365060018e-03,
-6.383901006951882e-02,2.813261471779898e-01,-7.216925365912068e-01,1.
143977149471640e+00,-1.116882640026025e+00,6.340855960819900e-01,-1.79
9698343577352e-01,1.616012319196882e-02,-2.054747982962562e-04,5.63474
4402603770e-04,-8.591909235406857e-05},
{ 2.421682935146632e-04,-9.198947462370994e-04,-1.994752843782460e-02
,2.157884050222244e-01,-9.898667963442510e-01,2.594506304034621e+00,-4
.175434535163080e+00,4.129710473597894e+00,-2.375002697115931e+00,6.83
7285302111370e-01,-6.184583920359052e-02,1.729447671487126e-04,-2.0766
18069726012e-03,3.220435793519317e-04},
{ -7.265960603720162e-04,4.237555259844561e-03,3.671236148085379e-02,
-4.910375904159711e-01,2.374693952049423e+00,-6.390758375570652e+00,1.
047149745426328e+01,-1.051482566798235e+01,6.139101227823355e+00,-1.79
7824596394723e+00,1.646680679910993e-01,1.421124399101191e-03,5.233224
234016207e-03,-8.375412758341282e-04},
{ 1.492253942574126e-03,-1.121599475951268e-02,-3.599961510538718e-02
,7.418338204423325e-01,-3.861134250533759e+00,1.074446183328351e+01,-1
.799260534089419e+01,1.839395064955905e+01,-1.093285246173577e+01,3.26
8942750307334e+00,-3.055494122124450e-01,-6.553975868594050e-03,-8.928
784715200866e-03,1.514804487373199e-03},
{ -2.064318579137274e-03,1.845860021479958e-02,3.586891551410636e-03,
-7.129076824972758e-01,4.145410444790667e+00,-1.205626808511887e+01,2.
074257701586726e+01,-2.166812891376764e+01,1.315715991828990e+01,-4.03
6092209479749e+00,3.895775750639108e-01,1.361828232602385e-02,9.948054
633577699e-03,-1.874343620840948e-03},
{ 1.854440501264744e-03,-1.886147175670775e-02,3.265819341486776e-02
,3.965337730403778e-01,-2.790358034921483e+00,8.632616566586590e+00,-1
.537748897332012e+01,1.649579765918335e+01,-1.027942035014721e+01,3.25
5639305582785e+00,-3.303817071015601e-01,-1.558642185827251e-02,-6.705
393913531543e-03,1.517519092082872e-03},

```
 {  -1.012800101997430e-03,1.139166757504039e-02,-3.505633885438296e-0
2,-9.907662543557520e-02,1.072277245917127e+00,-3.646158308783695e+00,
6.807026183122772e+00,-7.551087326942140e+00,4.857666038455164e+00,-1.
601395475698781e+00,1.753190645273069e-01,9.436884620712642e-03,2.3493
01133888509e-03,-7.269535708620266e-04},
 { 2.989916918412895e-04,-3.648702134898104e-03,1.488075324075222e-02,
-1.831337618716480e-03,-1.963146556392899e-01,7.897707860686255e-01,-1
.578673483628524e+00,1.830042427652221e+00,-1.224699768577713e+00,4.24
3184999203852e-01,-5.171736162141719e-02,-2.486939606658406e-03,-3.389
234070612998e-04,1.789515265103695e-04},
 { -3.781696520599198e-05,4.938988660455531e-04,-2.398941344520955e-03
,3.800521002841823e-03,1.090351858435061e-02,-6.631913602668045e-02,1.
484477001324116e-01,-1.830032633818878e-01,1.286518450902641e-01,-4.72
3314746818176e-02,6.514737067812121e-03,2.095309722803457e-04,9.969171
851780565e-06,-1.788340084439765e-05},
 {  1.109328092375133e-06,-1.535261972412140e-05,8.423859077776905e-05
,-2.079586735152370e-04,6.021594850196061e-05,1.017531908397153e-03,-2
.913570801845314e-03,3.968436909398971e-03,-2.980948357478795e-03,1.17
0387999465542e-03,-1.829345410952289e-04,-2.316157248589741e-06,2.9182
52678441764e-07,3.931288593614648e-07},
                          };

  Vmarray[0]=1;
  for (i=1;i<14;i++) Vmarray[i]=Vmarray[i-1]*Vm;

  if (aVex==5)
  { Vwtarray[0]=1;
    for (i=1;i<14;i++) Vwtarray[i]=Vwtarray[i-1]*aVwt;

    for (i=0;i<14;i++)
    { sum=0;
      for (j=0;j<14;j++)
        sum+=coeff[i][j]*Vmarray[13-j];
        a[i]=sum;
    }
    sum=0;
    for (i=0;i<14;i++)
      sum+=a[i]*Vwtarray[13-i];
    I1=sum;
    if (I1<0) I1=0;
  }
  else
  { sum=0;
    for (i=0;i<14;i++)
```

```
      sum+= leakcoeff[i]*Vmarray[13-i];
    I1= -sum;
  }
  if (aVdc==5)
  { sum=0;
    for (i=0;i<14;i++)
      sum+= discoeff[i]*Vmarray[13-i];
    Idc= sum;
  }
  else Idc = 0;

  return(I1-Idc);
}

extern double insynapse(double aVin,double aVwt,double Vm)
{
  int i,j;
  double I,sum,a[14],Vmarray[14],Vwtarray[14];
  static double coeff[14][14]={
 {  2.177421397682281e-08,-4.099037523459378e-07,3.362195883198151e-06
,-1.582157038389652e-05,4.726426749602979e-05,-9.356127455272212e-05,1
.242283518534696e-04,-1.094853416647668e-04,6.200051579657870e-05,-2.1
32027684216276e-05,4.152594302645939e-06,-4.680534839874958e-07,3.7965
95226996644e-08,-3.178326304227981e-12},
 { -6.796058118414131e-07,1.288716449999704e-05,-1.063660404904711e-04
,5.032897840006304e-04,-1.511011487319583e-03,3.004936756274800e-03,-4
.007104815718331e-03,3.545559368096041e-03,-2.014536200240832e-03,6.94
0630118174520e-04,-1.349784546379309e-04,1.514396271179674e-05,-1.2484
39243622917e-06,1.040438308199698e-10},
 { 9.328905586795263e-06,-1.784370225202192e-04,1.483527825172087e-03,
-7.064368715389898e-03,2.133072197940125e-02,-4.264369564776774e-02,5.
714408739200507e-02,-5.078928846935329e-02,2.896743756243009e-02,-1.00
0180147377283e-02,1.941598155939491e-03,-2.165707029021997e-04,1.81822
2939255472e-05,-1.508404815896006e-09},
 { -7.403273377467977e-05,1.430899516504552e-03,-1.199979320213371e-02
,5.756816525203354e-02,-1.749793233084323e-01,3.519291340154487e-01,-4
.742368188890835e-01,4.236608865492772e-01,-2.426858718109301e-01,8.40
0561302634316e-02,-1.627279722037804e-02,1.801548364710234e-03,-1.5438
30510362061e-04,1.275295914645522e-08 },
 { 3.755032075643166e-04,-7.351518423735087e-03,6.229678676833955e-02,
-3.015110062165590e-01,9.235645783380702e-01,-1.870557758535991e+00,2.
536906456079810e+00,-2.279745937717613e+00,1.312498215078681e+00,-4.55
67608981175 91e-01,8.804841809880300e-02,-9.652843315469180e-03,8.46372
7242341413e-04,-6.967847849580855e-08 },
```

```
 { -1.270395829563339e-03,2.527788991064903e-02,-2.169647468559937e-01
,1.061295957174184e+00,-3.280818142413291e+00,6.699501030562952e+00,-9
.154334681399092e+00,8.282928275113015e+00,-4.796854370909696e+00,1.67
1370484793371e+00,-3.220149079776888e-01,3.484724049140217e-02,-3.1335
05873071273e-03,2.576168530977180e-07},
 { 2.903439724051806e-03,-5.900403814017672e-02,5.146744549838416e-01,
-2.550578940038298e+00,7.972179387414515e+00,-1.643834915833805e+01,2.
266045258468414e+01,-2.066912921650463e+01,1.205402368158606e+01,-4.21
8614756108000e+00,8.101120521197160e-01,-8.614213595059243e-02,7.95679
1014982633e-03,-6.559935574866137e-07 },
 { -4.440927695783746e-03,9.287493500154487e-02,-8.272819858822575e-01
,4.167527015546636e+00,-1.320375505639862e+01,2.754642873049280e+01,-3
.837395304848845e+01,3.533856501973460e+01,-2.078322459744306e+01,7.31
4455112042512e+00,-1.399652647125268e+00,1.452843557073401e-01,-1.3778
41046135225e-02,1.148592442094132e-06},
 { 4.402884530280746e-03,-9.596645066192971e-02,8.794812100712543e-01,
-4.525814286928335e+00,1.458508732483426e+01,-3.086903227584012e+01,4.
355248061121107e+01,-4.057307218371277e+01,2.410827022541753e+01,-8.54
6718467826050e+00,1.629917791162814e+00,-1.636116099472059e-01,1.58378
1645682025e-02,-1.356844552314438e-06},
 { -2.654965914687481e-03,6.176368976048895e-02,-5.897098398563506e-01
,3.123535849413030e+00,-1.029093393719300e+01,2.217799432823669e+01,-3
.178407142439369e+01,3.003065669596910e+01,-1.807312887821575e+01,6.47
0041962827985e+00,-1.231132026145507e+00,1.179991395857574e-01,-1.1437
29929393002e-02,1.039839463139946e-06},
 { 8.602909526240915e-04,-2.254349539347759e-02,2.296244711498549e-01,
-1.267991742519216e+00,4.304663782860188e+00,-9.497186051680638e+00,1.
388160658337729e+01,-1.334780738645695e+01,8.163077490316200e+00,-2.96
1804066460231e+00,5.642658263843501e-01,-5.094272731374962e-02,4.70929
7260546332e-03,-4.827354950876515e-07},
 {-1.108721889084980e-04,3.904950299208463e-03,-4.480836263848028e-02,
2.643546983829601e-01,-9.372202371689592e-01,2.134581776969955e+00,-3.
200699549073321e+00,3.146515504428619e+00,-1.963939437961037e+00,7.259
486872571945e-01,-1.395542459808271e-01,1.186572844235746e-02,-9.71625
2172497211e-04,1.203669640447359e-07},
 {-5.636241407918107e-07,-2.085030779649814e-04,3.246102723209916e-03,
-2.168923472020330e-02,8.245219501086713e-02,-1.966876130901464e-01,3.
052984676550778e-01,-3.088129326923654e-01,1.977680673494919e-01,-7.49
1553265586906e-02,1.467992327546836e-02,-1.197249791288621e-03,8.02040
1812943790e-05,-1.273874375791981e-08},
 {3.599225218101810e-07,-7.481117709879855e-07,-3.594148073022414e-05,
3.441221313974942e-04,-1.508095048006624e-03,3.893692754426601e-03,-6.
370481450501985e-03,6.705983831501377e-03,-4.442443958801378e-03,1.737
108737727755e-03,-3.505778912754730e-04,2.816079454029621e-05,-1.43162
```

```
4691582681e-06,3.080560298639292e-10},
                          };

   if (aVin==5)
   { Vmarray[0]=1;
     Vwtarray[0]=1;
     for (i=1;i<14;i++) Vmarray[i]=Vmarray[i-1]*Vm;
     for (i=1;i<14;i++) Vwtarray[i]=Vwtarray[i-1]*aVwt;
     for (i=0;i<14;i++)
     { sum=0;
       for (j=0;j<14;j++)
         sum+=coeff[i][j]*Vmarray[13-j];
         a[i]=sum;
     }
     sum=0;
     for (i=0;i<14;i++)
       sum+=a[i]*Vwtarray[13-i];
     I=sum;
     if (I>0) I=0.;
   }
   else I = 0;

   return(-I);
}

extern double insynapcap(double aVin,double aVwt,double Vm)
{ insynapse(aVin,aVwt,Vm); }     /* Since it just adds more cap, otherw
ise */
                                  /* same as insynapse */

 extern double stdnpulse(double aVin,double aVth,double ondly,double r
iset,double ontime,double fallt,int n)
{
      /* COUNT THE NO. OF STDNEURON FROM .NETLIST FILE AND */
      /*   GENERATE ARRAYS WITH REQD. NO. OF ELEMENTS      */
      /*   IN MAIN TO STORE TMARK AND FLAG                 */

 if (aVth==0) aVth=5;            /* TO TAKE AWAY INITIAL ASSIGNMENT PRO
BLEM */
 if((aVin<aVth)&&(flag[n]==0))
 {
   return(0);
 }
 else
```

```
 {
  if (flag[n]==0) {tmark[n]=t;flag[n]=1;} ;
  if ((t-tmark[n])<=ondly)
     return(0);
  if (((t-tmark[n])>ondly)&&((t-tmark[n])<=(ondly+riset)))
     return((5.0/riset)*(t-tmark[n]-ondly));
  if (((t-tmark[n])>(ondly+riset))&&((t-tmark[n])<=(ondly+riset+ontime
)))
     return(5.0);
  if (((t-tmark[n])>(ondly+riset+ontime))&&((t-tmark[n])<=(ondly+riset
+ontime+fallt)))
     return(5.0-((5.0/fallt)*(t-tmark[n]-ondly-riset-ontime)));
  if ((t-tmark[n])>(ondly+riset+ontime+fallt))
  { flag[n]=0;
    return(0);
  }
 }
}

extern double stddcpulse(double ondly,double riset,double ontime,doubl
e fallt,int n)
{
  if(tmark[n]>0)
  {
    if ((t-tmark[n])<=ondly) return(0);
    if (((t-tmark[n])>ondly)&&((t-tmark[n])<=(ondly+riset)))
       return((5.0/riset)*(t-tmark[n]-ondly));
    if (((t-tmark[n])>(ondly+riset))&&((t-tmark[n])<=(ondly+riset+onti
me))) return(5.0);
    if (((t-tmark[n])>(ondly+riset+ontime))&&((t-tmark[n])<=(ondly+ris
et+ontime+fallt)))
       return(5.0-((5.0/fallt)*(t-tmark[n]-ondly-riset-ontime)));
    else {tmark[n]=0;return(0);}
  }
  else return(0);
}

extern double stdinnrn(double aVc,double aVth)
{ aVc*=2;
  if ((ceil(aVc)-aVc)>(aVc-floor(aVc))) aVc=0.5*floor(aVc);
  else aVc=0.5*ceil(aVc);
  if(aVc== 5.0) return(pulse(77.3,1.5,4.2,1.3,24.2));
  if(aVc== 4.5) return(pulse(76.5,1.6,3.3,2.3,22.7));
  if(aVc== 4.0) return(pulse(76.4,1.7,4.1,1.3,24.5));
```

```
  if(aVc== 3.5) re' ırn(pulse(65.2,1.7,3.9,1.7,27.));
  if(aVc== 3.0) return(pulse(65.2,2.4,3.6,1.9,29.6));
  if(aVc== 2.5) return(pulse(53.4,1.9,2.8,1.1,26.15));
  if(aVc== 2.0) return(pulse(64.3,2.5,3.2,2.3,49.9));
  if(aVc== 1.5) return(pulse(62.,2.6,3.4,1.7,92.3));
  if(aVc== 1.0) return(pulse(323.6,1.5,4.5,2.,335.));
  if(aVc  <1.0) return(0);
}

extern double invinnrn(double aVc,double aVth)
{  aVc*=2;
   if ((ceil(aVc)-aVc)>(aVc-floor(aVc))) aVc=0.5*floor(aVc);
   else aVc=0.5*ceil(aVc);
   if(aVc==0.0) return(pulse(53.1,2.5,4.1,1.3,22.5));
   if(aVc==0.5) return(pulse(55.6,1.4,4.1,2.1,24.2));
   if(aVc==1.0) return(pulse(60.4,1.8,4.2,1.3,26.7));
   if(aVc==1.5) return(pulse(60.7,1.3,4.5,1.1,30.));
   if(aVc==2.0) return(pulse(69.8,1.6,4.,2.,36.2));
   if(aVc==2.5) return(pulse(90.8,1.4,4.4,1.6,46.6));
   if(aVc==3.0) return(pulse(74.9,2.3,4.,1.6,72.4));
   if(aVc==3.5) return(pulse(63.5,1.9,3.2,2.5,150.5));
   if(aVc==4.0) return(pulse(143.2,1.9,2.9,2.,844.));
   if(aVc >4.0) return(0);
}
```

B.4 The Waveforms Used in PULSE

```
#include <stdio.h>
#include <math.h>

extern double t;

extern double pulse(double ondly,double riset,double ontime,double fal
lt,double offtime)
{
  double reqd1,reqd2;
  reqd2=ontime+offtime+riset+fallt;
  if (t<ondly) return(0);
  else
  {
    reqd1=t-ondly-floor((t-ondly)/reqd2)*reqd2;
    if ((reqd1>(riset+ontime+fallt))||(reqd1<0.0)) return(0);
```

```
    else
    {
      if ((reqd1>=0) && (reqd1<riset)) return((5.0/riset)*reqd1);
      if ((reqd1>(riset+ontime)) && (reqd1<=(riset+ontime+fallt)))
        return(5.0-((5.0/fallt)*(reqd1-(riset+ontime))));
      else return(5);
    }
  }
}
```

Appendix C

PULSE Input Files - An Example

These were the input files used to simulate the XOR circuit in Chapter 5.

C.1 netlist

```
// Netlist Generated by PULSE Netlister //
*net0=/I10.DCPULSE
*net1=/I18.DCPULSE
*net2=/A
*net3=/B
*net4=/Vwt1
*net5=/V-
*net6=/Vb
*net7=/Vlk
*net8=/Vwt2
*net9=/OUT
*net10=/I6.Vm
*net11=/I8.Vm
*net12=/I16.Vm
*net13=/I11.DCPULSE
*net14=/I10.NPULSE
*net15=/I11.NPULSE
*net16=/I0.IN1OUT
*net17=/I1.IN0OUT
*net18=/I2.IN1OUT
*net19=/I3.IN0OUT
```

```
*neuron(0)=/I18;
SN0 12 5 7 1 9 ;
*neuron(1)=/I11;
SN1 11 5 6 13 15 ;
*neuron(2)=/I10;
SN2 10 5 6 0 14 ;
*synapse1(3)=/I17;
ES3 15 1 8 7 12 ;
*synapse1(4)=/I16;
ES4 14 1 8 7 12 ;
*synapse1(5)=/I9;
ES5 19 13 4 7 11 ;
*synapse1(6)=/I8;
ES6 18 13 4 7 11 ;
*synapse1(7)=/I7;
ES7 17 0 4 7 10 ;
*synapse1(8)=/I6;
ES8 16 0 4 7 10 ;
*inNeuron0(9)=/I3;
II9 2 5 6 19 ;
*inNeuron0(10)=/I1;
II10 3 5 6 17 ;
*inNeuron1(11)=/I2;
SI11 3 5 6 18 ;
*inNeuron1(12)=/I0;
SI12 2 5 6 16 ;
```

C.2 pulse.inp

```
VVlk 1.5;
VA 5;
VB 5;
VVwt1 3.6;
VVwt2 5;
VV- 1.65;
VVb 2;
```

C.3 pulse.sim

```
.tran 1.0 300;
```

C.4 control

```
*9;
```

Appendix D

Output File Generated by PULSE

From the input files provided (as in Appendix C), pulse builds an output file which gets compiled alongwith the macromodels and the waveform file to run the simulation. The file shown below was generated while simulating the XOR circuit, and hence it corresponds to the input files presented in Appendix C.

```
#include <stdio.h>

#include <math.h>

double t=0,delt=1.000,tmark[3];

int flag[3];

extern double pulse(double ondly,double riset,double ontime,double
fallt,double offtime);

extern double exsynapse(double aVex,double aVwt,double aVdc,double
aVlk,double Vm);

extern double insynapse(double aVin,double aVwt,double Vm);

extern double stdnpulse(double aVin,double aVth,double ondly,double
 riset,double ontime,double fallt,int n);

extern double stddcpulse(double ondly,double riset,double ontime,
```

```
double fallt,int n);

extern double stdinnrn(double aVc,double aVth);

extern double invinnrn(double aVc,double aVth);

main()
{
  int i,j,flag1,flag2;
  double store1[3],store2[3],temp,V[20],Vlast[20],Vlast2[20],sum=0.0;
  for (i=0;i<3;i++) { tmark[i]=0;flag[i]=0;store1[i]=0;store2[i]=0;}
  for (i=0;i<20;i++) { V[i]=0;Vlast[i]=0;Vlast2[i]=0;}
  do
  {
    t+=delt;j=0;
    printf(" ");
    for(i=0;i<20;i++) Vlast2[i]=V[i];
    flag2=0;flag1=0;
    do
    { j++;
      if (flag2!=20) flag2=0;
      for(i=0;i<20;i++) Vlast[i]=V[i];
      V[0]=stddcpulse(10.2,0,13,0,2);
      V[1]=stddcpulse(10.2,0,13,0,0);
      V[2]=5;
      V[3]=5;
      V[4]=3.6;
      V[5]=1.65;
      V[6]=2;
      V[7]=1.5;
      V[8]=5;
      V[9]=stdnpulse(V[12],V[5],9.8,0.8,5,1.4,0);
      V[13]=stddcpulse(10.2,0,13,0,1);
      V[14]=stdnpulse(V[10],V[5],9.8,0.8,5,1.4,2);
      V[15]=stdnpulse(V[11],V[5],9.8,0.8,5,1.4,1);
      V[16]=stdinnrn(V[2],V[5]);
      V[17]=invinnrn(V[3],V[5]);
      V[18]=stdinnrn(V[3],V[5]);
      V[19]=invinnrn(V[2],V[5]);
      if (j==1) V[10]=Vlast2[10]+0.5*(3*store1[0]-store2[0]);
      i=0;
      if (flag2<20){ do
      { i++;
        if (i!=1) V[10]=temp;
```

```
        temp=Vlast2[10]+0.5*(store1[0]+((exsynapse(V[17],V[4],V[0]
,V[7],V[10])+exsynapse(V[16],V[4],V[0],V[7],V[10]))*delt*1e-09)/(2
*0.15e-12));
      if (i>10000) {printf("\n\n  No convergence - try with smaller
 timestep\n\n");
                    exit(0);}
      }
      while((fabs(temp-V[10]))>0.0001);
      V[10]=temp;}
      if (flag2==20) {store2[0]=store1[0];
      store1[0]=(((exsynapse(V[17],V[4],V[0],V[7],V[10])+exsynapse(
V[16],V[4],V[0],V[7],V[10]))*delt*1e-09)/(2*0.15e-12));
      flag1=1;}
      if (j==1) V[11]=Vlast2[11]+0.5*(3*store1[1]-store2[1]);
      i=0;
      if (flag2<20){ do
      { i++;
        if (i!=1) V[11]=temp;
        temp=Vlast2[11]+0.5*(store1[1]+((exsynapse(V[19],V[4],V[13]
,V[7],V[11])+exsynapse(V[18],V[4],V[13],V[7],V[11]))*delt*1e-09)/(2
*0.15e-12));
      if (i>10000) {printf("\n\n  No convergence - try with smaller
timestep\n\n");
                    exit(0);}
      }
      while((fabs(temp-V[11]))>0.0001);
      V[11]=temp;}
      if (flag2==20) {store2[1]=store1[1];
      store1[1]=(((exsynapse(V[19],V[4],V[13],V[7],V[11])+exsynapse(
V[18],V[4],V[13],V[7],V[11]))*delt*1e-09)/(2*0.15e-12));
      flag1=1;}
      if (j==1) V[12]=Vlast2[12]+0.5*(3*store1[2]-store2[2]);
      i=0;
      if (flag2<20){ do
      { i++;
        if (i!=1) V[12]=temp;
        temp=Vlast2[12]+0.5*(store1[2]+((exsynapse(V[15],V[8],V[1],V
[7],V[12])+exsynapse(V[14],V[8],V[1],V[7],V[12]))*delt*1e-09)/(2*0.15
e-12));
      if (i>10000) {printf("\n\n  No convergence - try with smaller
timestep\n\n");
                    exit(0);}
      }
      while((fabs(temp-V[12]))>0.0001);
```

```
      V[12]=temp;}
      if (flag2==20) {store2[2]=store1[2];
      store1[2]=(((exsynapse(V[15],V[8],V[1],V[7],V[12])+exsynapse(V
[14],V[8],V[1],V[7],V[12]))*delt*1e-09)/(2*0.15e-12));
      flag1=1;}
      if (flag2!=20) for (i=0;i<20;i++) if ((fabs(V[i]-Vlast[i]))<0.00
01) flag2++;
    }
    while ((flag2<20) || (flag1!=1));
    printf("%g %f \n",(t*1e-09),V[9]);
  }
  while(t<300.000);
}
```

Appendix E

The Netlister

E.1 S/SLG File to provide Formatting Instructions to FNL

```
; Making the netlist representation properties editable
replaceRepProp=t
; nlpglobals/pulse gets replaced by the design, which then gets netlisted.
; These lines below defines properties for nlpglobals/pulse i.e. of the
; netlist.
defRepProp(nlpglobals pulse
; If length of a line in netlist is > 70, start another line with prefix +
NLPLineLength=70
NLPLinePrefix="+"
; While encountering each instance in the schematic, generate a comment
; like *neuron(1)=/I5
NLPsingleLineCommentString="*"
NLPElementComment=nlpExpr("*[@BlockName]([@ElementNumber])=[@InstPathName];")
; generate a comment for each net as *net2=/OUT
NLPcreateNetString=nlpExpr("*net[@NodeNumber]=[@NetPathName]\\n")
; generate netlist for each instance by replacing the value of the property
; NLPElementPostamble from Appendix E.2. Generate an error if the element
; format is not defined in the file in Appendix E.2.
NLPcompleteElementString=nlpExpr("[@NLPElementPostamble:%:**No element format
 property found for element [@InstPathName]]\\n")
NLPnetlistHeader=nlpExpr("// Netlist Generated by PLSE Netlister //\\n")
)
```

E.2 S/SLG File to Define Netlist Output of Symbols

```
simRep( stdneuron symbol pulse
   NLPElementPostamble=nlpExpr("[@NLPElementComment:%\n]SN[@ElementNumber]
 [|V+] [|V-] [|Vb] [|DCPULSE] [|NPULSE] ;")
)
simRep( exsynapse symbol pulse
   NLPElementPostamble=nlpExpr("[@NLPElementComment:%\n]ES[@ElementNumber]
 [|Vex ] [|Vdc] [|Vwt] [|Vlk] [|Vm] ;")
)
simRep( insynapse symbol pulse
   NLPElementPostamble=nlpExpr("[@NLPElementComment:%\n]IS[@ElementNumber]
 [|Vin ] [|Vwt] [|Vm] ;")
)
simRep( sineuron symbol pulse
   NLPElementPostamble=nlpExpr("[@NLPElementComment:%\n]SI[@ElementNumber]
 [|Vc] [|V-] [|Vb] [|IN1OUT] ;")
)
simRep( iineuron symbol pulse
   NLPElementPostamble=nlpExpr("[@NLPElementComment:%\n]II[@ElementNumber]
 [|Vc] [|V-] [|Vb] [|IN0OUT] ;")
)
; For each of the three symbols in the symbol library, the value of the
; variable NLPElementPostamble is the format which is to be used for their
; netlist. The formats for each of these symbols are defined in this file.
; The value of the variable NLPElementComment is assigned in Appendix E.1.
```