1995

BiCMOS implementation of the hierarchy for pattern extraction artificial neural network.

Kai Yiu. Hung
University of Windsor

BiCMOS Implementation of the Hierarchy for Pattern Extraction Artificial Neural Network

by

Hung, Kai Yiu

A Thesis

Submitted to the Faculty of Graduate Studies through the

Department of Electrical Engineering in Partial Fulfillment

of the Requirements for the Degree of

Master of Applied Science

at the

University of Windsor

Windsor, Ontario, Canada

May, 1995

ISBN 0-612-10937-2

Name HUNG KAI YU

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

ELECTRONICS AND ELECTRICAL
SUBJECT TERM

0	5	4	4

SUBJECT CODE

U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

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

EDUCATION

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

LANGUAGE, LITERATURE AND LINGUISTICS

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

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy	0422
Religion	
General	0318
Biblical Studies	0321
Clergy	0319
History of	0320
Philosophy of	0322
Theology	0469

SOCIAL SCIENCES

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

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

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

EARTH SCIENCES

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

HEALTH AND ENVIRONMENTAL SCIENCES

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

PHYSICAL SCIENCES

Pure Sciences

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

Applied Sciences

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

PSYCHOLOGY

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

Dedicated with love

to my parents.

Abstract

This thesis presents the architecture and the algorithm of the Hierarchy for Pattern Extraction (HyPE) artificial neural network. The training algorithm and the recalling algorithm of the HyPE artificial neural network are rewritten into C based on a Smalltalk prototype. A switching tree minimization program is introduced that provides logic minimization capable of handling a higher transistor tree height and merges several transistor trees. The Northern Telecom 0.8μ Bipolar Complementary Metal Oxide Semiconductor (BATMOS) technology is used to implement the designs in this thesis. There are two final dynamic neuron designs that have been verified and fabricated. One neuron uses the True Single Phase Clocking (TSPC) Latch and the other neuron uses the Ultra Fast Dynamic Current Steering (UCDCS) latch at the output of the dynamic functional block. The verification of the functional blocks for both neuron designs is done using SPICE simulation. The highest clocking speed applied to the TSPC neuron and the UCDCS neuron are 50MHz and 66MHz, respectively. Additionally, by isolating one of the transistor trees from the functional block, the clocking speed up to 333MHz can be achieved. Finally, a test chip including these two final dynamic neuron designs has been fabricated.

Acknowledgements

I would like to express my sincere thanks and appreciation to my supervisor, Dr. G. A. Jullien, and my co-supervisor, Dr. W. C. Miller for their tremendous support and guidance throughout the progress of this thesis. I would also like to thank to Bruce Erickson and Alagu Annaamalai for guiding me on the research at the beginning. I would also need to thank Subramanian Kumar for working out the algorithm and programming. I need to thank Marjan Shahkarami, John and Dimitris Phoukas for helping me from the layout design problem to writing the thesis. Finally I have to express my deepest gratitude to my parents for their patience, support and encouragement on every aspect of my life.

Table of Contents

List of Figures

List of Tables

List of Symbols

Explaination of Subscripts

e.g. $X_{n,i,j,l}$

i	runs across the neurons in a present level.
j	is the level or layer(input(0), alpha(1), Beta(2), Gamma(3), Basal Ganglia)
l	is the presentation method (1 is the initial presentation, 2 is the regular presentation, 3 is presentation with the target and non-target pattern)
n	runs across the neurons in the upper level.
t	is the time step (Pattern application step).

List of Notations

AVE	Average number of inputs from new virgin alpha neuron to each regular Beta neuron (**integer**).
$\beta_{initial}$	number of the regular neuron in beta level after the initial wake period (**integer**).
BTF	Beta Threshold factor (**integer**).
$\mathbf{C}_{ijt}$	Neuron Connectivity with the previous level (**boolean vector**).
C_{nijt}	The element of $\mathbf{C}_{ijt}$ (**boolean**).
$\mathbf{E}_{j-1,t}$	Regular (for all neurons in previous level, 0 if virgin, 1 if regular, for alpha level elements of $\mathbf{E}$ are all '1's) (**boolean vector**).
$E_{n,j-1,t}$	The element of $\mathbf{E}_{j-1,t}$ (**boolean**).
$\mathbf{F}_{j-1,t}$	Firing status record of neurons in previous level (**boolean vector**).

$F_{n,j-1,t}$	The element of $\mathbf{F}_{j-1,t}$ **(boolean)**.
$\mathbf{FP}_{jt}$	The firing population **(integer vector)**.
FP_{ijt}	The element of $\mathbf{FP}_{jt}$ **(integer)**.
G_{ijt}	Firing status **(boolean)**.
$\mathbf{IH}_{jt}$	The input history **(integer vector)**.
IH_{ijt}	The element of $\mathbf{IH}_{jt}$ **(integer)**.
$\mathbf{IP}_{jt}$	The input population **(integer vector)**.
IP_{ijt}	The element of $\mathbf{IP}_{jt}$ **(integer)**.
$\mathbf{IR}_{jt}$	The input record **(integer vector)**.
IR_{njt}	The element of $\mathbf{IR}_{jt}$ **(integer)**.
k	a constant factor by which the pain factor is increased each time pain is experienced.
k_1	contants for the input population.
kk_2, kk_3, kk_4	contants for the input population.
LP_l	The maximum number of times that a set of inputs will be used in the l'th presentation **(integer)**.
LL	The population of virgin neuron in alpha level **(integer vector)**.
LPF	The distribution of firing population of beta level, $\mathbf{FP}_{2,t}$ **(integer vector)**.
m_{jt}	The number of connection of a neuron in each level **(integer)**.
M	The minimum gamma firing **(integer)**.
N_j	The number of neurons in j'th level **(integer)**.

NA	The Novelty Arousal value (**integer**).
NAC	Novelty Arousal Criterion (**integer**).
NAM	The Novelty Arousal Monitor Value (**boolean**).
NV_j	The size of virgin neurons in j'th level (**integer**).
$\mathbf{P}_t$	The input vector t (**boolean vector**).
P_{jl}	The limit number of firing of virgin neuron in l'th presentation at j'th level (**integer**).
PF	The pain factor (**real**).
PP	The Pain & Pleasure (**boolean**).
R_{ijt}	Am I a regular before (1 is regular 0 is virgin) (**boolean**).
S_j	The size of neurons in j'th level (**integer**).
S_{jt}	This is the set of all neurons in the j'th level $(S_{jt} = S_{R_{jt}} \cup S_{V_{jt}})$.
$S_{R_{jt}}$	This is the set of all regular neurons in the j'th level.
$S_{V_{jt}}$	This is the set of all virgin neurons in the j'th level.
SR_j	The size of regular neurons in j'th level (**integer**).
T_{ijt}	Threshold of Neuron (**integer**).
TNT	TargetNonTarget (1 if target pattern, 0 if non-target pattern) (**boolean**).
TP_l	The number of training pattern used in the l training presentation (**integer**).

Functions

Boolean Vector Sum

$\sum_B \mathbf{V}$	This counts the number of Boolean 1's in the Boolean vector $\mathbf{V}$.

Vectorize Boolean Outputs

$\underset{i}{\Psi}\ [G_{ijt}]$	This produces a vector of boolean outputs across the i index.

Normalizes vector

$\eta\,(\mathbf{IH}_{jt})$	This produces a normalized vector output of $\mathbf{IH}_{jt}$.

Vectorize Integer Outputs

$\underset{i}{\mathbf{VI}}\ [B_{ijt}]$	This produces a vector of integer outputs across the i index.

Random Number

$\Re[low,high]$	This produces an integer number between low and $high$.

Generate Neuron

$\mathbf{N}[T_{ijt},R_{ijt},\mathbf{C}_{ijt},G_{ijt}]$	This allocate resource for a neuron.

Chapter 1

Introduction

1.1 Motivation of the thesis

The Hierarchy for pattern extraction (**HyPE**) artificial neural network was originally proposed by Andrew Coward [1] in the early nineties. It was soon realized that it had the potential to become a complete digital network. A join project between Northern Telecom (NT) and the VLSI Research Group, University of Windsor, was proposed at the same time in order to explore the possibility of implementing HyPE in BATMOS, which was at the time the state of the art fabrication technology available at NT. Our starting basis was a prototype Smalltalk implementation of the HyPE algorithm written by Andrew Coward. As a proof of concept, a single neuron was selected for VLSI implementation. It was not too long before it was realized that the use of **tcells**, the BATMOS standard library elements, was very expensive in terms of space for this kind of application. Therefore a full custom dynamic logic style was adopted for the implementation of the neuron. This dynamic style is based on a recently proposed timing scheme, the True Single Phase Clock (**TSPC**), for which novel storage elements using the full advantages of BiCMOS technologies have been designed by the VLSI Group.

1.2 Background

The topic of the Artificial Neural Networks (ANNs) has been investigated for a long time. ANNs are widely used in different applications such as pattern recognition, optimization, product of predicting financial analysis and more.

Most of the ANNs work with analog neuron which are easy to program but not so easy to implement in silicon. Recently, researchers have developed digital neural networks, such as the adaptive probability neural network [6][7][8] and the HyPE artificial neural network. Although the digital ANN is easy to implement in silicon, it has the disadvantage that the size of the ANN can be several times larger than the analog ANN.

The HyPE artificial neural network is the focus of this thesis. The purpose of the HyPE artificial neural network is to provide the recognition of suggested patterns sent into the network. Since the HyPE artificial neural network is still under modification and it requires a long time to verify one set of parameters, we need a faster process in order to train the network. This thesis focuses on finding a way to improve the performance of the training process, using the Bipolar Complementary Metal Oxide Semiconductor (**BiCMOS**) technology provided by Northern Telecom. It also provides an implementation of the general neuron that involves all the functionality of each neuron in the architecture and is the most representative sub-function to implement under the algorithm of interest.

1.3 Thesis objective

The original objective of this thesis was to reverse engineer the algorithm from the Smalltalk program which was the only resource available to us in the beginning A rigorous description of the algorithm was to be written as part of this investigation. The other objective was to implement in the VLSI medium the full architecture (or some well defined part of it) in order to improve the training performance of the artificial neural network. As a side effect of this process, a new implementation of the switching tree minimization procedure had to be written in order to generate the full custom implementations of the functional blocks of the neuron.

1.4 Thesis Organization

This thesis includes six chapters. Chapter 1 provides the background of this research and provides the structure of the each chapter in this thesis.

Chapter 2 introduces the hierarchy for pattern extraction (HyPE) artificial neural network. It provides the background, architecture and the algorithm of the HyPE artificial neural network. This chapter focuses on the architecture and the algorithm of the HyPE based on the original Smalltalk program written by L.Andrew Coward[1]. Both of them are discussed deeply step by step and compared with the modern artificial neural network.

Chapter 3 presents the graphical switching tree minimization algorithm and the two latch designs that are used in this thesis. The graphical switching tree minimization algorithm is proposed by Bryant[14][19]. A switching tree minimization program written in c programming language is presented. The two latch designs are the true single phase clocking (TSPC) latch[12][15] and the ultra fast dynamic current steering (UCDCS) latch proposed by J.C. Czilli[12].

Chapter 4 presents the pipeline neuron implementation approach and the single block neuron implementation approach for the general neuron that can improve the training process of the HyPE artificial neural network. After that it goes through the other directory of approaches that are based on using the standard cell in 0.8μ BATMOS library (tcell) to design the neuron and using the dynamic logic with switching tree minimization to design the neuron. The two designs of 3-input single block dynamic neuron are chosen to use in the implementation of the general neuron. A test cell has been designed and submitted for fabrication.

The last chapter, Chapter 5, concludes the work done in this thesis. It provides suggestions for improving the architecture and the algorithm of the HyPE artificial neural network and the way to implement the neuron design for the possible future direction of the research in HyPE project.

Chapter2

Hierarchy for Pattern Extraction

2.1 Introduction

This chapter discusses the artificial neural network that Andrew L. Coward brings out from the Brain operation system (The Brain Model). This chapter is divided into four sections. Section one is the introduction of this chapter. The second section is the background of the brain model. The third section is the architecture of the Hierarchy for Pattern Extraction (HyPE). The fourth section is the algorithm used in HyPE. The last section is the summary of the HyPE network.

2.2 Background of ANN

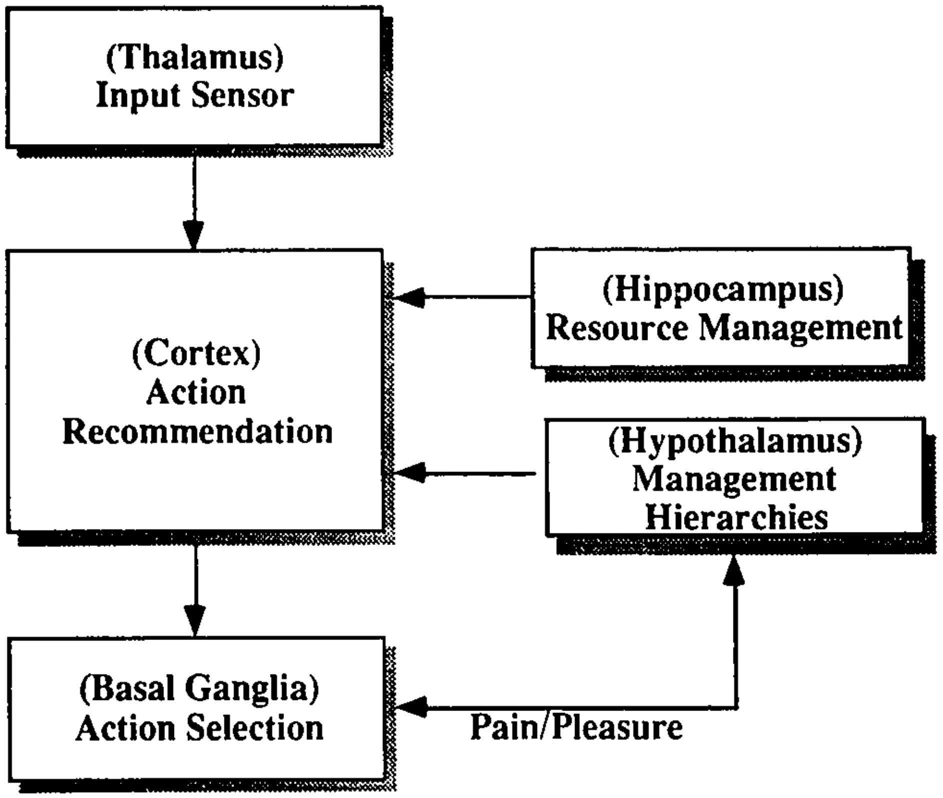
Artificial Neural Networks (ANNs) have been considered as an area of active research for a long time. Most of the developed artificial neural networks are based on the non-linearity neuron which uses a non-linearity function such as a sigmoid function. These kinds of artificial neural networks have some similarities, such as the non-linearity neuron, fixed network size, ease of simulation, and difficulty to fabricate in Very Large Scale Integrated (VLSI). Although the most current VLSI circuit technology is used, it still faces some difficulty, regarding fabrication. The original design in the design environment and after the fabrication may have difference dimensions. These differences may be small but they could create a big problem due to the nature of artificial neural networks in VLSI. Since most of the artificial neural networks use a non-linearity function at the output of a neuron, Either fault tolerance of the fabrication process in the training process or to reduce the sensitivity of the artificial neural network need to be included.

There are several ways to bypass this problem. One is using discrete weight and sigmoid function in the software training process. When the artificial neural network is recalling, the hard-limit function replaces the sigmoid function as the non-linearity function of the artificial neuron. Since the artificial neural network uses the discrete weight and hard-limiting function, it adds great flexibility to the design. Although the fabrication process creates some dimension differences, the design can still handle that. This method was applied on two 3μ Complementary Metal-Oxide Semiconductor (CMOS) designs [3][4], on an optical coupled neural network and a 1.2μ CMOS design[5] on programmable optical coupled neural network. Another approach is the Hierarchy for Pattern Extraction (HyPE) artificial neural network architecture that created by Andrew L. Coward[1][21] using the Smalltalk-V programming language in 1990. The other one is the adaptive probability neural network[6][7][8][9]. The HyPE artificial neural network is the architecture, that this thesis focuses on.

2.3 Architecture of HyPE

The diagram in Figure 2.1 shows the brain model of Andrew design. Input layer **(Thalamus)** is the sensor of the network. It presents sensory input signals to the middle layer. The middle Layer **(Cortex)** is a multilevel pattern declarative memory storage that keeps the characteristic of a target group and gives action recommendation to the network. There are three levels of neurons, Alpha level, Beta level and Gamma level. Each of these levels may contain more than a thousand neurons. It depends on the training process and the training pattern. Output layer **(Basal Ganglia)** is the action selection section. It contains a single artificial neuron. There are two management systems taking care of the training process. One of them is resource management (**Hippocampus**). It contains the mapping of the middle layer neuron resources and assigns additional neurons if the network requires more memory storage. **Hypothalamus** is the other kind of control management. It controls the novelty arousal that sets the threshold value of each resource neuron (virgin neuron) in the middle layer. It also controls the pleasure and pain signal releasing to the output layer neuron.

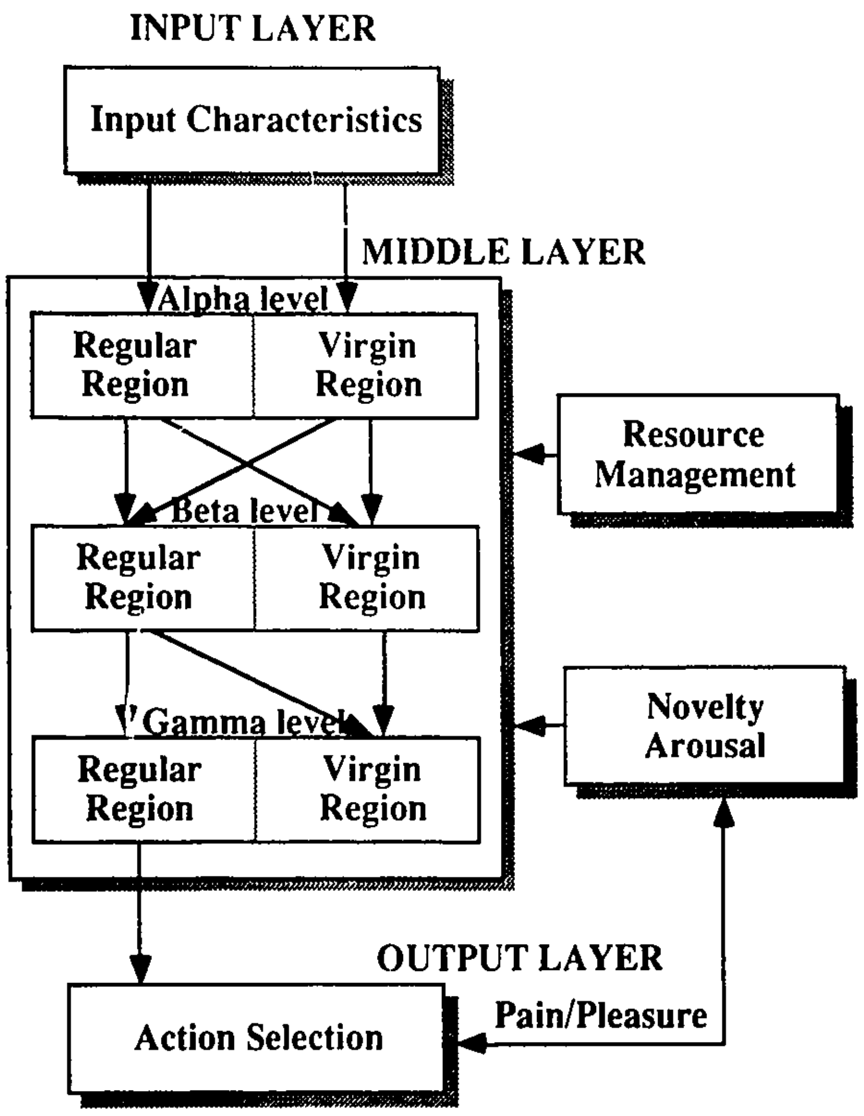
Figure 2.1 The Diagram of the Brain Model


(Thalamus) Input Sensor
(Hippocampus) Resource Management
(Cortex) Action Recommendation
(Hypothalamus) Management Hierarchies
(Basal Ganglia) Action Selection
Pain/Pleasure

In the middle layer, each level contains two regions: the action recommendation region and the unused resource region. The unused resource region (virgin region) contains unused neurons and those are called virgin neurons. Neurons in the action recommendation region (regular region) are called regular neurons. These neurons are the memory storage of the target pattern that give the information, justifying whether the input pattern belongs to the target group or not. The characteristic and the functionality of these neurons will be discussed in more detail in the next section.

HyPE is a feedforward multilayer hierarchy artificial neural network implementing the Brain Model. The sensor gives the input signal to the input layer. Input signals of the alpha level, in middle layer, are the output signals of the input layer. The output signals of the alpha level neurons will penetrate to the beta level neuron as their input signals. The output signals of the beta level neurons will penetrate to the gamma level neurons and so are the output signals of gamma level. They will become the input signals of the basal ganglia layer neuron. The artificial neural network does not have any connection cross over levels or layers. e.g. there is no connectivity behaviour between alpha level's neurons and gamma level's neurons. It is shown in Figure 2.2.

Figure 2.2 HyPE Architecture



The arrows in the Figure 2.2 represent the direction of the signal and show all the possible connections between the layers and the levels. There are two things to notice in this figure. First, there is no connection between the beta virgin region and the gamma regular region. The missing connection is based on the present architecture. It can be modified easily by a small modification in the algorithm and the connection can be restored. The other missing connectivity is the gamma virgin region and the output layer. It is also based on the architecture, however it cannot be replaced unless there is a big modification on the algorithm.

2.3.1 Neuron Model

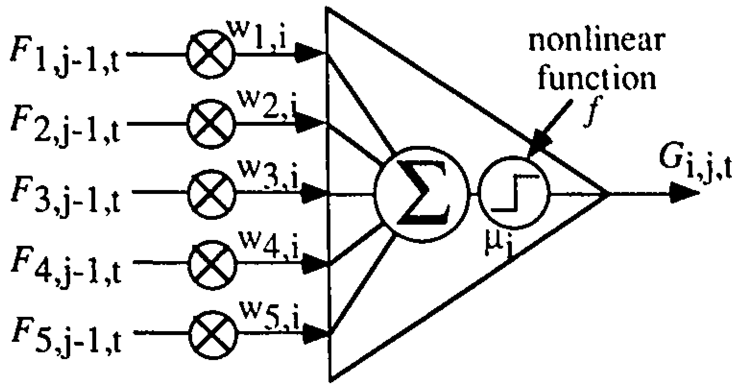
In the HyPE architecture, two types of neurons are used. One is called regular neuron that has already recognized a specific pattern. The other one is called virgin neuron that has not yet recognized any pattern but has been configured to have a high potential to fire and then imprint. Imprinting is a process that transfers a virgin neuron to a regular neuron. This process will be discussed in more detail later. There are three levels of neurons in the middle layer and a neuron in output layer, Basal Ganglia layer. Basically, each neuron has a similar characteristic and functionality except for the output layer neuron. Each neuron contains a threshold value and connectivity to the upper level. The threshold value and connectivity of a neuron normally are unique. This will be clear in the algorithm section.

The activity of a neuron depends on the threshold value and connectivity of the neuron. Threshold value for each neuron is an integer number. Connectivity of a neuron is based on the history of the probability of upper level neurons' activity. It will be discussed in more detail in the algorithm. There is no weight on each connection. It is either connected or not. From a modern artificial neural network perspective, there are only two different kinds of weight for each connection. Those are 0 and 1. It is much simpler than the modern artificial neuron. The modern artificial neuron needs to have a weight on each input connection and needs to have a nonlinear activation function after the summation to determine the activity of the neuron. Figure 2.3 shows a schematic diagram of a

McCulloch-Pitts[2] neuron. This model can be equated as the following with notations that used in the HyPE neuron.

$$G_{i,j+1,t} = f_i\left(\sum_n w_{ni}G_{n,j,t} - \mu_i\right) \tag{2.1}$$

Figure 2.3 Diagram of a McCulloch-Pitts neuron

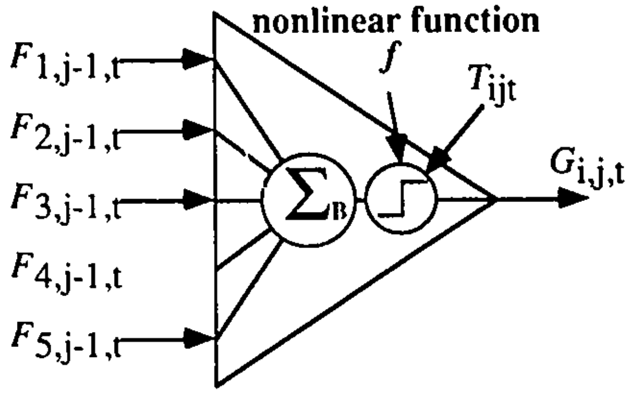


The firing status, G_{ijt}, represents the output signal of the i'th neuron in the j'th level in the t'th pattern of time process. The upper level firing status, $F_{n,j-1,t}$, represents the output signal of the n'th neuron in the $(j-1)$'th level in the t'th pattern of time process. It is same as the firing status,$G_{i,j-1,t}$. w_{ni}, represents the weight between the i'th neuron in the current level and the n'th neuron in the upper level. It can be a positive value or a negative value. μ_i represents the threshold value of the i'th neuron in the j'th level. The nonlinear activation function, f, in McCulloch-Pitts' model is a step function, however it can be modified by using the other nonlinear activation function such as, the signum , sigmoid and threshold logic functions.

In HyPE, since there is no weight on the input connections, the firing status of the artificial neuron is determined from the sum of active input connections, and then compared with the threshold value of the neuron. If the sum of active input connections is greater or equal to the threshold value, the neuron fires (the firing status is 1). For example, if there are 3 active input connections, and the threshold value of the neuron is 3, then the neuron fires. In the general case the firing status of a neuron can be modeled in the HyPE as follows:

$$G_{i,j,t} = f\left(\sum_B [C_{n,i,j,t} \wedge F_{n,j-1,t}] - T_{i,j,t}\right) \qquad (2.2)$$

Figure 2.4 Schematic Diagram of the neuron in HyPE



$\sum_B$ is a function that sum up the boolean numbers. The threshold, T_{ijt}, represents the threshold of the i'th neuron in the j'th level in the t'th pattern of processing time. In here j would be declared more clearly. j is defined as the subscript notation for level or layer. In HyPE architecture, input layer is 0, alpha level is 1, beta level is 2, gamma level is 3 and the basal ganglia is 4. j is a positive integer value. The connectivity, C_{nijt}, represents the connection between the n'th neuron in the $(j-1)$'th level and the i'th neuron in the j'th level. As the diagram shows, the artificial neuron in HyPE architecture is not fully connected to the upper level or layer. The connectivity is defined by the firing population on the upper level. This will be defined more clearly in the next section. Since the threshold, T_{ijt}, is an integer value and so is the sum of the input signals, the signal that passes past through the nonlinear activation function, f, is no longer seen as a continuous value. This is one of the advantages of the HyPE architecture.

The model presented above is for a general neuron. It includes neurons in alpha level, virgin neurons in beta level and neurons in gamma level. Regular beta neurons need to have an additional condition to fire:

$$G_{ijt} = f\left\{\begin{matrix}\left(\sum_B [C_{n,i,j,t} \wedge F_{n,j-1,t}] - T_{ijt}\right) \wedge \\ \left(2\sum_B [C_{n,i,j,t} \wedge F_{n,j-1,t} \wedge E_{n,j-1,t}] - T_{ijt}\right)\end{matrix}\right\} \tag{2.3}$$

$E_{n,j-1,t}$ represents the status of the neuron, i.e. whether it is regular neuron (1) or a virgin neuron (0). The additional condition is applied because the regular beta neuron may have some connections from the virgin alpha neuron. This additional condition protects the over firing with the additional active virgin input neuron.

The neuron in the output layer, Basal Ganglia, is a unique neuron in the HyPE architecture. Its connectivity is controlled by the hierarchy management, hypothalamus, with the pain and pleasure feedback. It will be discussed in more detail in the algorithm description. The firing status of this neuron is determined by any active input connectivity occupied. In other words, if it has any active input connections, the neuron is firing. The model of the firing status of Basal Ganglia neuron is shown as in the following:

$$G_{1,4,t} = f\left(\sum_B [C_{n,1,4,t} \wedge F_{n,3,t}] - 1\right) \tag{2.4}$$

The firing status, $G_{1,4,t}$, is the Basal Ganglia neuron firing status.

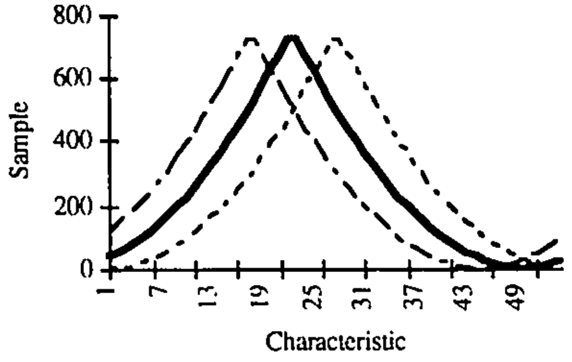
2.4 The HyPE Algorithm

The HyPE algorithm is used to recognize a target group's patterns from the other group of patterns. Due to the complexity of the algorithm, it needs to be divided into 5 parts for further discussion. First is the input pattern for training and then is the overall training process. After that we have the detail of the initialization process, wake process, sleep process and the recalling process.

2.4.1 Training Pattern

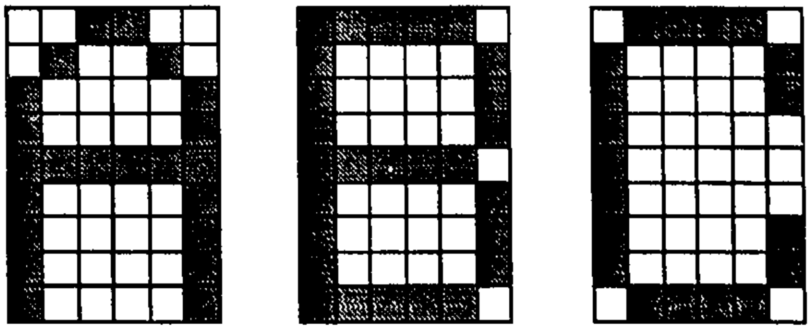
Patterns that are presented to the network are generated from sampling frequency distribution. The training process requires three groups of patterns, so each of the groups needs to create one distribution. Group distribution is the sampling of the group and the summing up the characteristic of the group. An example is given in Figure 2.5. These are the distributions of the 3 groups of object that have the bell shape sampling frequency. Each of the patterns that are presented to the network are generated from those distributions by randomly selecting 21 characteristic out of the 54 from the example of group distribution in Figure 2.5. The C programming source code, "C Code for generating input patterns" on page 77, is used to generate the 500 input patterns.

Figure 2.5 Characteristic distribution of Groups A,B,C



Here is a simple example of 3 characters A, B, C (Figure 2.6). Each character contains 54 characteristic components. Each characteristic component at the edge of the characters has 100 sampling frequency and the other characteristic component has 1 sampling frequency. A group of patterns for each character can be generated by randomly selecting from the sampling frequency distribution in this example.

Figure 2.6 Character A, B, C

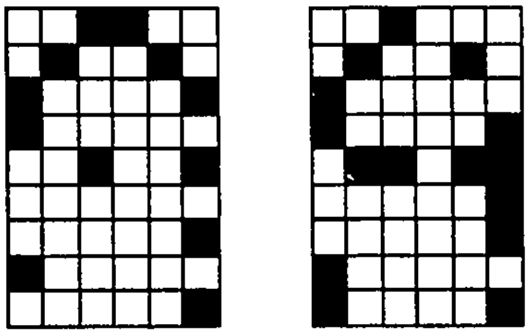


▨ 100 sampling frequency □ 1 sampling frequency

There are two examples of patterns that are generated from the distribution of character A in Figure 2.7. These patterns have about 10% distortion from the original character. Using the examples in Figure 2.7, the patterns that are presented to the network are as follows:

001100|010010|10001|100000|001001|000000|000001|100000|100001
001000|010010|100000|100001|011011|000001|000001|100000|100001

Figure 2.7 Examples from Character A's pattern



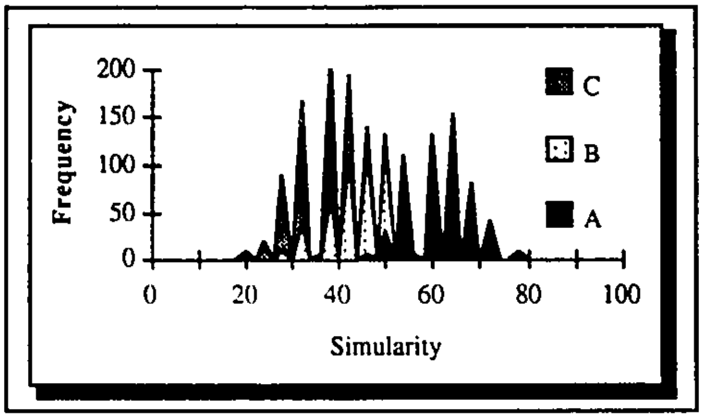
■ 1

□ 0

Using these 500 patterns from each of the characters, one can show the similarity of those patterns. Character A is defined as the target pattern for this network. The similarity of those patterns is the sum of the sampling frequency of each of the characteristics that are presented in those patterns and then normalized with respect to the sum of the sampling frequency of the characteristics in the Character A. e.g. The sum of sampling frequency of Character A in Figure 2.6 is 2232. The similarity of those examples in Figure 2.7 is 53.763% and 67.2%, respectively.

Figure 2.8 shows the similarity distribution of 500 patterns for each of the characters versus the frequency. The similarity distribution provides an idea of possible distinguish ability of the network. A group of character A has an average of 62% similarity, and the similarity of character B and character C is 44% and 36%, respectively.

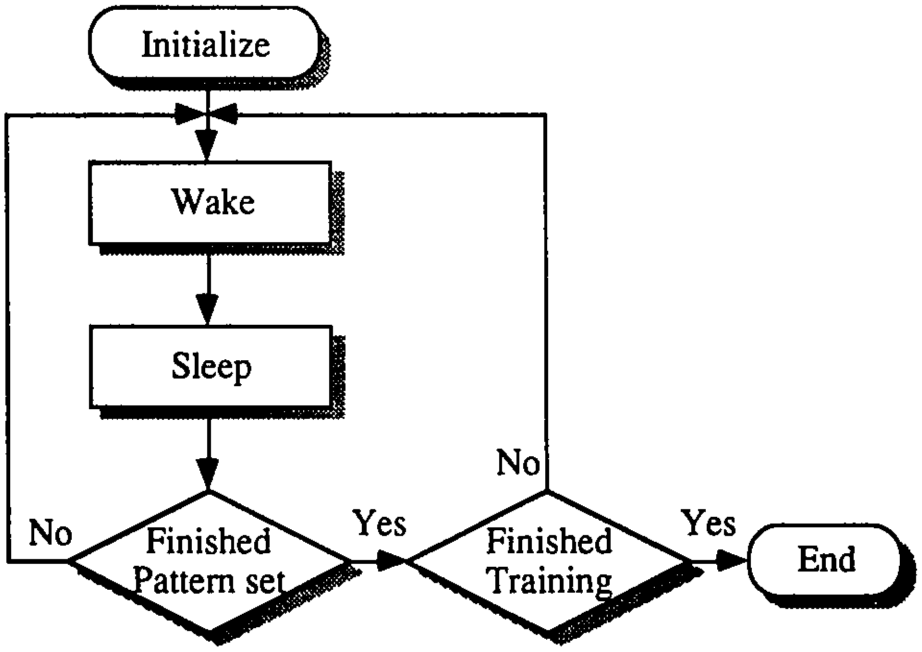
Figure 2.8 Plot of the Similarity Distribution of Character A, B, C



2.4.2 Overall Training Algorithm

The overall training algorithm, shown as Figure 2.9, is divided into three parts: the initialization algorithm, the wake algorithm and the sleep algorithm.

Figure 2.9 The Flow Chart of the Overall Training Algorithm



In the training algorithm, one needs three groups of patterns to train the neural network. One is the target group. The other two are the non-target groups. These patterns are used in three presentation methods. The first presentation method is to initialize the neural network focus on the target group of patterns. The second presentation method is to enhance the recognition of the target group's patterns. The third presentation method is to remove some of the error recognition of the non-target groups' patterns. It is done by putting some non-target groups' patterns into the artificial neural network. Table 2.1 shows the information of the number of patterns presented to the artificial neural network and the maximum number of times that these patterns are presented to the network. In the third presentation method, patterns that are presented to the network are interlacing. Using the example of character A, B and C, the order of patterns presented to the network is the pattern of character A, the pattern of character B, the pattern of character C, and then back to other pattern of character A and so on. Also the set of character A that is presented in the third presentation method is different from the first and the second presentation method.

Table 2.1. Presentation Method's Information

Presentation Method	Group 1 (Target)	Group 2 (Non-target)	Group 3 (Non-target)	presenting times
1	4	0	0	1
2	20	0	0	20
3	10	10	10	20

2.4.3 Initialization Algorithm

Initialization algorithm is a process that creates a set of neurons in each level in the middle layer. Since the algorithm of HyPE is a digital artificial neural network, it has unlimited potential to learn if there are enough accessible neuron resources. The initialization algorithm is to supply the amount of neuron resources to the artificial neural network for the first presentation method.

Table 2.2. Information of Initializing Each Level

Level (j)	Neurons (NV_j)	Inputs (m_{jt})
Alpha	150	15
Beta	150	17
Gamma	150	14

Table 2.2 show the number of neuron resources in each level when the artificial neural network initializes. It also shows the maximum number of input, m_{jt}, these neurons can have.

Connectivity of a neuron is randomly selected from the upper level's neurons. Since the artificial neural network does not have any regular neuron in the initial state, it randomly selects from the unit distribution of the population of virgin neurons. The following is the process to select the input connections to each neuron:

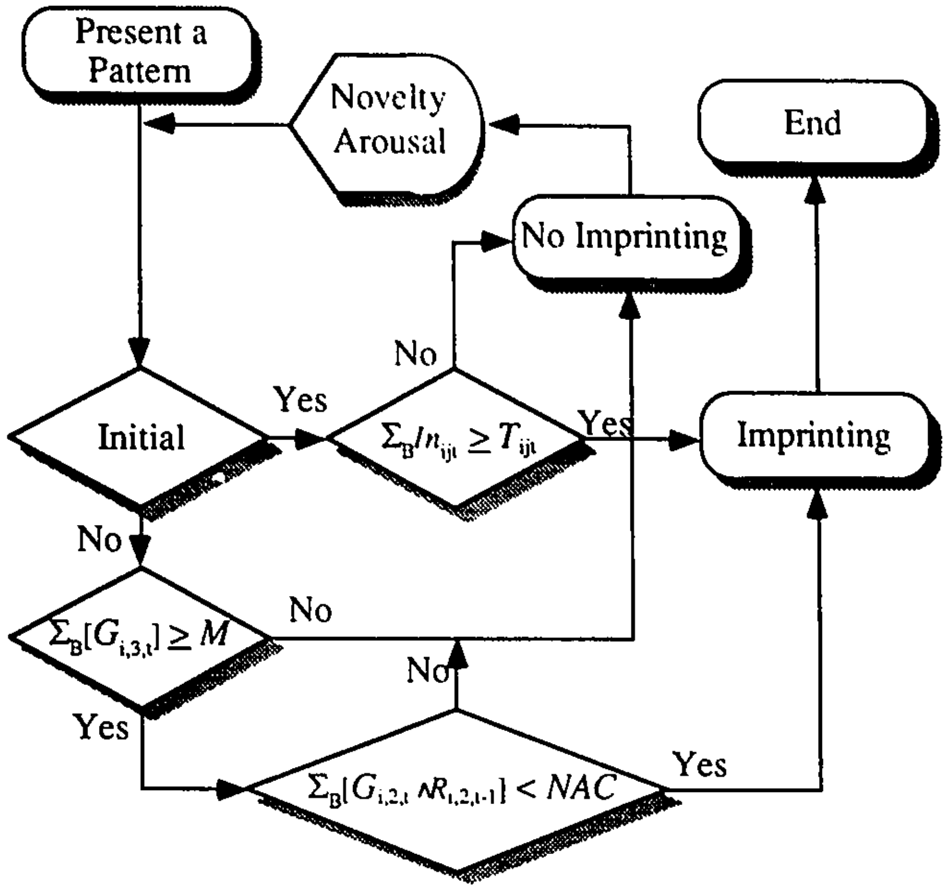
$$C_{\Re[1,N_{j-1}],i,j,0} \leftarrow 1; i \in S_{j,t} \tag{2.5}$$

m_{jt} represents the maximum number of connections. $\Re[1,N_{j-1}]$ is a random number that randomly selects among 1 and N_{j-1}. N_{j-1} is the number of neuron in the $j-1$ 'th level or layer. Because the connections are randomly selected, it is possible to have some duplicated connections. Therefor the number of input connection may be less than m_{jt} show in Table 2.2.

2.4.4 Wake Algorithm

Wake algorithm is a process that trains the neurons in the middle layer to store the information of presented patterns. Flow chart in Figure 2.10 shows the wake algorithm.

Figure 2.10 Flow chart of the Wake Algorithm


Present a Pattern
Novelty Arousal
End
No Imprinting
No
Yes
Initial
$\Sigma_B In_{ijt} \geq T_{ijt}$
Yes
Imprinting
No
$\Sigma_B[G_{i,3,t}] \geq M$
No
No
Yes
$\Sigma_B[G_{i,2,t} \wedge R_{i,2,t-1}] < NAC$
Yes

A few points in the flowchart need to be discussed. When a pattern is presented to the artificial neural network, it will check to see that the neural network is in the initial state. Initial state is the one which the network is still using the first presentation method that focuses the neural network to recognize the target group. It will check that virgin neurons are active or not. If not, the novelty arousal value will increase. This value controls the threshold value of the virgin neuron in each level in middle layer. Table 2.3 shows how the novelty arousal value effects the threshold value of virgin neuron in each level. Since the neural network is at the initial state, novelty arousal value is set to 1 at the beginning. Therefore the threshold values of virgin neuron in alpha level, beta level and gamma level are 7, 7 and 6, respectively. At the second and third presentation method, the novelty arousal value is set to 0 at the beginning. This is going to let the artificial neural network check that the pattern is already recognized. That is done because no virgin neuron will have more the 50 input connections.

Table 2.3. Novelty Arousal controlled Threshold of Virgin Neuron in Level

Novelty Arousal	Alpha Level	Beta Level	Gamma Level
0	50	50	50
1	7	7	6
2	6	6	6
3	5	5	6

M is the minimum number of gamma firing before completing the training of the input pattern. $\sum_B [In_{ijt}]$ defines the number of active inputs. Novelty arousal criterion, NAC, is given by:

$$NAC = PF\frac{\sum_B \mathbf{R}_{jt}}{k1}\left(k2 + k3\frac{\sum_B \mathbf{R}_{jt} - \beta_{initial}}{\beta_{initial}}\right) \tag{2.6}$$

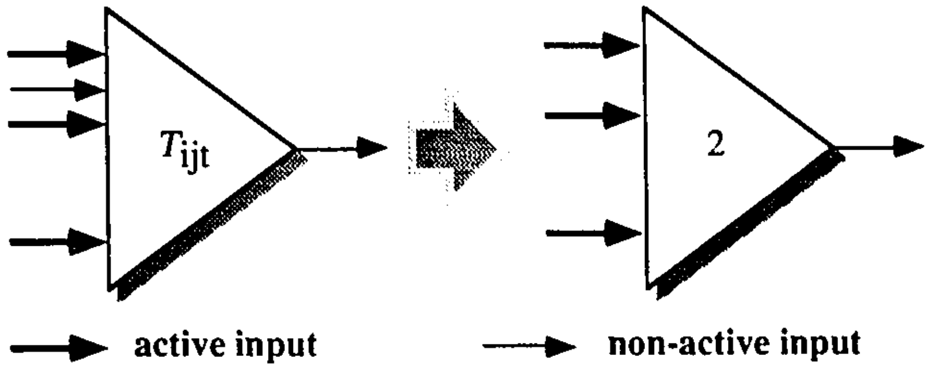
PF is the pain factor. It increases 5% every time when pain is experience. Pain is a signal that a non-target pattern makes the basal ganglia neuron fire. $\sum_B \mathbf{R}_{jt}$ is the size of the regular region in beta level. $\beta_{initial}$ is the initial beta level regular region size. It is defined right after the first presentation method. $k1, k2, k3$ are the constants in this expression. The value of $k1, k2, k3$ is 300, 0.25 and 1.25, respectively.

Imprinting is the process that converts a virgin neuron to a regular neuron when the virgin neuron is active in the training process. When a virgin neuron fires and the neural network goes into the imprinting process, the non-active input of the neuron will be removed and the threshold value of the neuron will set as follows:

$$T_{ijt} = \sum_{B} |In_{ijt}| - 1 \tag{2.7}$$

An example of imprinting a neuron is given in Figure 2.11. In the example, the virgin neuron has 3 active input connections. When it is imprinted, the non-active input has been removed and the threshold is set to 2. The threshold value of the neuron will normally be fixed at this time but there is a special case on beta regular neuron. It will be discussed further in the sleep algorithm.

Figure 2.11 An Example of Imprinting a Neuron



There is a limitation for imprinting for each level in different presentation method. Table 2.4 shows the limitation for imprinting neuron.

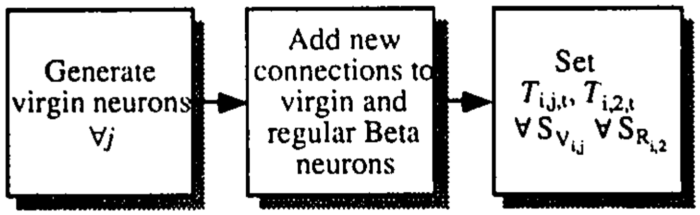
Table 2.4. Limitation of Imprinting Virgin Neuron in each Level

Presentation	Alpha	Beta	Gamma
Method 1 (initial)	none	50	4
Method 2	30	30	15
Method 3	30	30	15

2.4.5 Sleep Algorithm

Sleep algorithm, shown as Figure 2.12, is a process that reorganizes resource neurons of the artificial neural network. It includes removing and adding new virgin neurons in the levels. Beta regular neurons will have an average number of new input connections from the alpha virgin neurons just generated. The threshold of all virgin neurons and the threshold value of beta regular neurons need to be reassigned.

Figure 2.12 Flow Chart of the Sleep Algorithm


Generate virgin neurons $\forall j$
Add new connections to virgin and regular Beta neurons
Set $T_{i,j,t}$, $T_{i,2,t}$ $\forall S_{V_{i,j}}$ $\forall S_{R_{i,2}}$

Virgin neurons after the last wake process are reorganized and new virgin neurons are added to the network. The process of generating new virgin neuron is similar to the initialization algorithm, but the limitation of virgin neurons and the number of their input connections are different from it. Since the imprinted virgin neurons are part of the basic neurons, in the artificial neural network, the limitation of new virgin neuron is much less. On the contrary, each new virgin neuron may have more input connections. Table 2.5 shows the configuration of number of neurons and number of their input connections in sleep algorithm.

Table 2.5. The configuration of Sleep Algorithm

Level	Neurons (NV_j)	Inputs (m_{jt})
alpha	30-80	20
beta	30-80	24
gamma	30-80	26

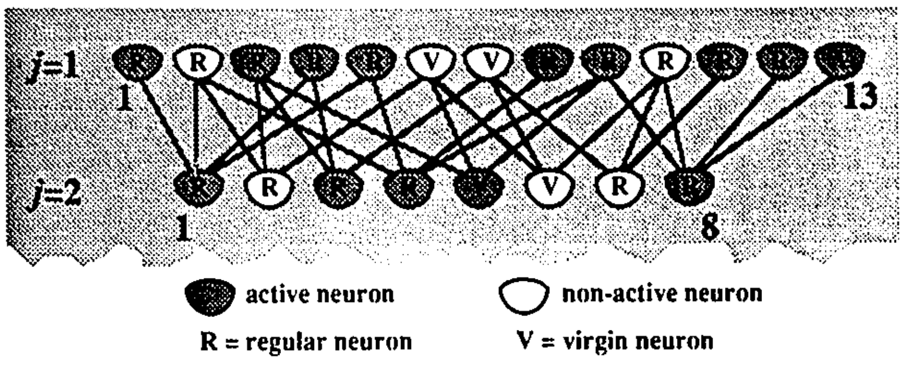
Number of new virgin neurons, NV_j, in the j'th level basically is given by:

$$NV_j = 15 \times \frac{NV_j + SR_j - S_j}{10} \tag{2.8}$$

where SR_j represents number of regular neuron and S_j represents number of neuron in the j'th level after wake process. If NV_j is less then 30, it will set to 40. If NV_j is greater than 80, it will reset to 80. Also the size of the each level effects the number of virgin neuron go to be generated. If the sum of NV_j and SR_j is less than 200, NV_j will increase until the size of each level is at least 200.

The input connection of neuron is randomly selected from the input populations, $\mathbf{IP}_{jt}$. $\mathbf{IP}_{jt}$ is the population of the input history, $\mathbf{IH}_{jt}$. $\mathbf{IH}_{jt}$ is based on the last $\mathbf{IH}_{j,t-1}$ and the input record, $\mathbf{IR}_{jt}$. The input record, $\mathbf{IR}_{jt}$, is the population of active regular neuron in the $(j-1)$'th level that involves the activity of regular neuron in the j'th level. The algorithms of generating those populations, histories and records are in Appendix A. An example of those input connections to the beta level is given in Figure 2.13.

Figure 2.13 An Example of Finding $\mathbf{IR}_{jt}$, $\mathbf{IH}_{jt}$ and $\mathbf{IP}_{jt}$



Assuming the input history, $\mathbf{IH}_{2,t-1}$, is as the following:

$$\mathbf{IH}_{2,t-1} \Leftrightarrow [205, 88, 270, 45, 90, 0, 0, 114, 4, 80, 60, 44, 0]$$

From the example in Figure 2.13, the input record, $\mathbf{IR}_{2,t}$, is as the following:

$$\mathbf{IR}_{2,t} \Leftarrow [1, 0, 1, 2, 2, 0, 0, 1, 2, 0, 0, 1, 0]$$

Using the following equation in the algorithm and replacing the constant, k_1, to 500

$$IH_{n,2,t} \leftarrow \left[\frac{k_1\,(500)}{\sum_{s \in S_{j-1,t}} IH_{s,2,t-1}} \times IH_{n,2,t-1} + \frac{k_1\,(500)}{\sum_{s \in S_{j-1,t}} IR_{s,2,t}} \times IR_{n,2,t} \right] ; n \in S_{R_{1,t}}$$

$$\therefore \mathbf{IH}_{2,t} \Leftrightarrow [153, 44, 185, 122, 145, 0, 0, 107, 102, 40, 30, 72, 0]$$

Since $j = 2$, the following equations apply to the input population, $\mathbf{IP}_{2,t}$. The constants, kk_3 and kk_4, are replaced with 100 and 1000, respectively.

$$\text{Since j=2:} \begin{cases} IP_{njt} \leftarrow \left[\sum_{s=1}^{n} IH_{sjt} + \frac{kk_3\,(100)}{\sum_B E_{i,j-1,t}} \right] ; n \in S_{R_{j-1,t}} \\ IP_{njt} \leftarrow \left[\sum_{s=1}^{n} IH_{sjt} + \frac{kk_4\,(1000)}{\sum_B E_{i,j-1,t}} \right] ; n \in S_{V_{j-1,t}} \end{cases}$$

$$\therefore \mathbf{IP}_{2,t} \Leftrightarrow [163, 217, 412, 544, 699, 1032, 1365, 1482, 1594, 1644, 1684, 1766, 2099]$$

The input population, $\mathbf{IP}_{2,t}$, will be used to generate new input to those new virgin neurons in beta level.

The new input connections to the beta regular neurons are based on the distribution of firing population, **LPF**, of beta regular neuron and the distribution of the virgin neuron, **LL**, in alpha level. Firing population, $\mathbf{FP}_{jt}$, is the population history of active regular neuron in the j'th level. The number of new connections to the alpha level is the product of average, AVE, number and the number of beta regular neuron, SR_2.

2.4.6 Recalling Process

After the artificial neural network is trained, it needs a recalling process to test the network. When a pattern is presented to the trained artificial neural network, the neural network will give a suggestion whether the pattern belongs to the target group or not, based on the information stored in the middle layer in the training process.

The trained neural network has some neurons occupied in each level that are no longer useful for information retrieving. Because of the efficiency of recalling, those neurons are removed from the network before trying to recall any pattern. Program, in Appendix B "recall program" on page 91, is the source code of the recalling process. It first reads in the configuration of trained artificial neural network, then removes those useless regular neurons or virgin neurons in each level and at last it presents a list of patterns to the network. Those patterns are coming from a data file and they can be the target group's patterns or the non-target group's patterns.

2.5 Summary

This Chapter includes the architecture and the algorithm of Hierarchy for Pattern Extraction (HyPE). HyPE is for single group of pattern recognition. It is a multi-level artificial neural network. The learning algorithm was invented by Andrew L. Coward based on his Brain Model. After going through the algorithm, it is easy to understand that HyPE is a complete digital artificial neural network. HyPE is a non-fully connected artificial neural network that has no weight at all the connections. Since the training algorithm applies the random selection connectivity from the distribution, the result artificial neural network will be different from one to the other with the same initial setting. Therefore, it needs at least 10 runs to verify the setting of those parameters in the network. Since the artificial neural network normally has about 1000 neurons in each level, the verification time for each changing is time consuming. Basically, the disadvantages of the HyPE artificial neural network are that it is a complex and time consuming. The advantages of HyPE artificial neural network are its flexiblity and the fact that it is a complete digital network, therefore it is appropriate for VLSI implemention.

Chapter3

Switching Tree Minimization & Latch Designs

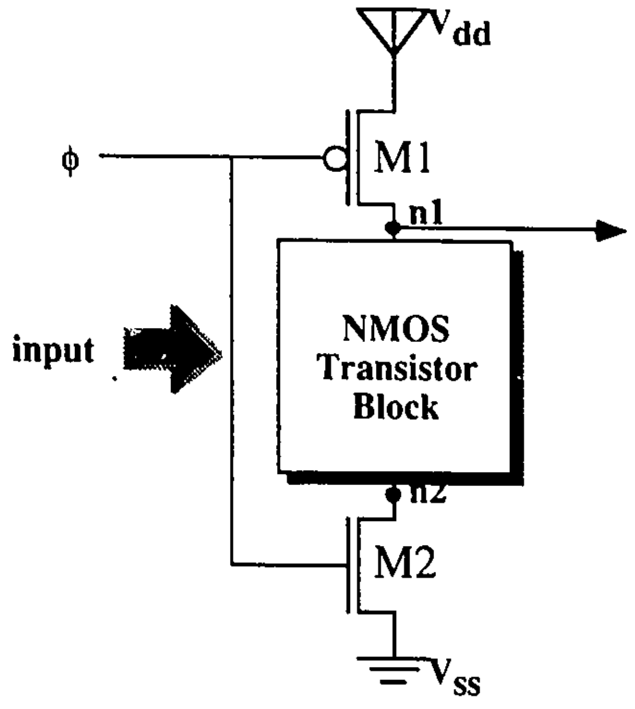
3.1 Introduction

The chapter discusses the minimization process and the latch designs that are going to be applied into the implementation of HyPE. The minimization method is based on Bryant's work. There are two latch designs that will be discussed. One is the True Single Phase Clock (TSPC) latch design and the other is Ultra-Fast Completely Dynamic Current Steering (UCDCS) latch which was modified based on the Completely Dynamic Current Steering (CDCS) latch by Czilli[12].

3.2 Switching Tree Minimization

Switching tree minimization algorithms are trying to that minimize the number of transistor in a decision tree block in dynamic logic[13] as shown in Figure 3.1. The topic has been actively pursued over the last few years[10][11][17][19][20].

Figure 3.1 The Dynamic Logic Diagram



Dynamic logic has one PMOS, one NMOS clock transistor and an NMOS transistor block. The functionality of dynamic logic involves two phases: the pre-charge phase and the evaluation phase. In the pre-charge phase, the clock, ϕ, is low, the M1 PMOS-transistor is on and the M2 NMOS-transistor is off. M1 charges up the internal parasitic capacitor, so node n1 stays at a high logic state. In the evaluation phase, the clock, ϕ, goes high, so M1 is off and M2 is on. If there is a conducting path within the NMOS transistor block between n1 node and n2 node, the internal parasitic capacitor will discharge and n1 will go low. If not, the internal parasitic capacitor will not discharge and n1 will stay high.

3.2.1 Minimization Algorithm

Switching tree minimization applied in this thesis, is based on the algorithms proposed by Bryant[14][19]. There are more than one switching tree minimization approaches. For example, Jullien has suggested a graph based reduction technique that effectively implements those tree blocks with minimizing the number of transistors[10][11][19][20].

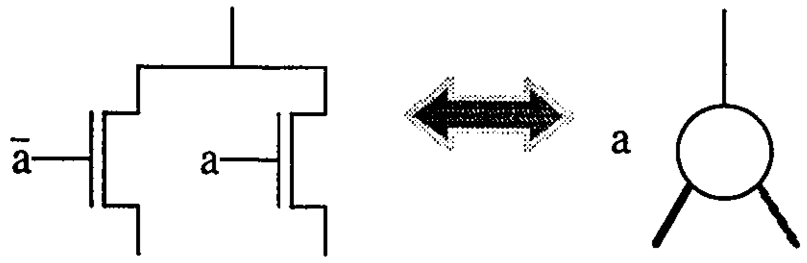
These algorithms are applied to the truth table, as shown in Table 3.1 , where a, b, and c are the input variables. F is the output of the boolean function.

Table 3.1. An Example of Truth Table

a	b	c	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Bryant presents a new data structure to represent boolean functions and a set of rules to manipulate those boolean functions. It can be applied to dynamic logic, domino logic, cascode voltage switch logic (CVSL), etc. The basic element in his paper is a node with 3 terminals. It is equivalent to two transistors with complemented and non-complemented inputs to their gates, as shown in Figure 3.2.

Figure 3.2 Graphicai Symbol of Tree Component



In the figure above, a, represents the non-complemented input to the gate. $\bar{a}$, represents the complemented input to the gate. The letter next to the graphical symbol represents the non-complemented input variable. Using the example truth table shown in Table 3.1 , the full tree of transistors is shown in Figure 3.3 and the graphical representation with the basic element is shown in Figure 3.4.

Figure 3.3 Full Tree of Transistor

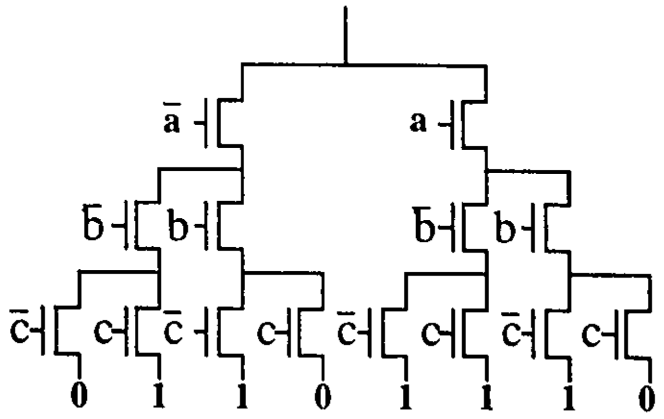
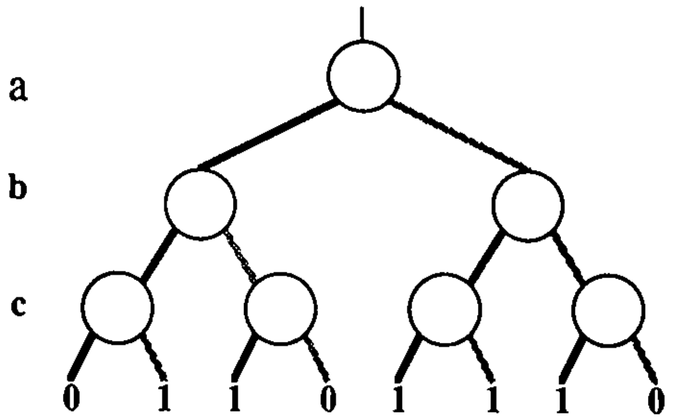


Figure 3.4 Graphical Representation of Full Tree of Transistor

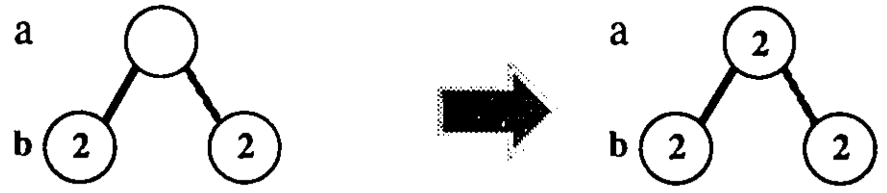


There is a set of rules for minimizing the graphic node presentation that is proposed by Bryant[14][19]. After the procedure of minimization, a key table will be generated. Two terms need to be clarified before discussing those rules: A parent node is the root of the current tree. Children nodes are the top nodes of the sub-tree.

Here are Bryant's Rules:

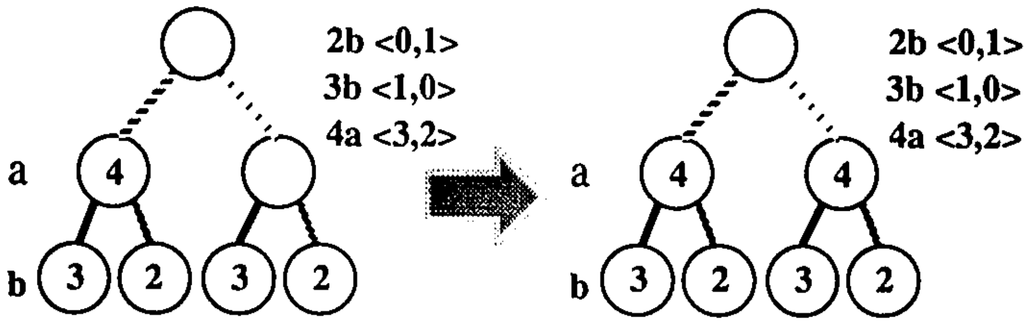
Rule 1:If the children node numbers are equal, the parent node number will be equal to the children node numbers. This is shown as Figure 3.5.

Figure 3.5 Rule 1 of Switching Tree Minimization



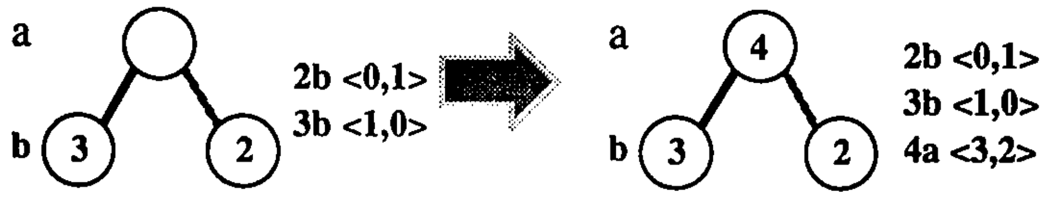
Rule 2a:If the children node numbers are not equal and there is an entry representing the same kind of node in the key table, the same node number in the key table will be assigned to the parent node. Figure 3.6 is the example of the rule.

Figure 3.6 Rule 2 of Switching Tree Minimization



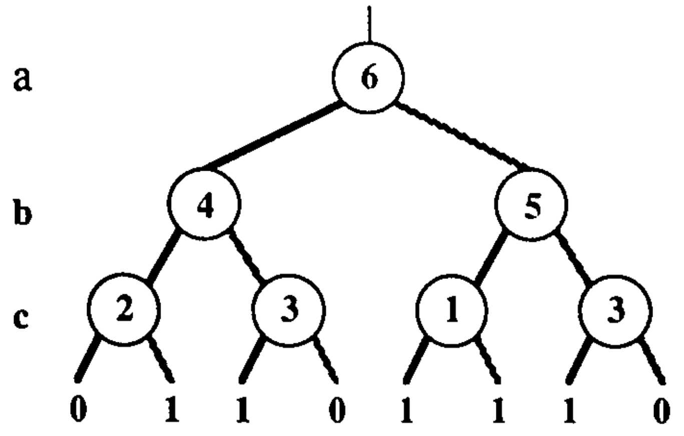
Rule 2b:If the children node numbers are not the same and it does not have an entry representing the same kind of node structure in the key table, a new entry will be created and added in the key table. The new node number will be assigned to the parent node. It is shown in Figure 3.7.

Figure 3.7 Rule 3 of Switching Tree Minimization



From the example of the truth table in Table 3.1 , the key table of that boolean function is shown in Figure 3.8. The graphical representation has assigned all the node numbers in the left hand side. In the right hand size is the key table of the boolean function.

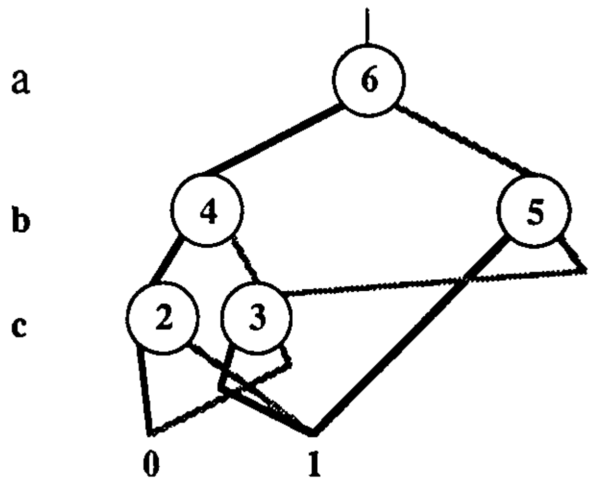
Figure 3.8 The Graphical Representation with Node Number and the Key Table



2c <0,1>
3c <1,0>
4b <2,3>
5b <1,3>
6a <4,5>

After every nodes have been assigned, the original graphic representation of full tree can be removed and the key table is complete. The new graphic representation tree is constructed by using the key table. The order of the construction is opposite from the assigning node number. It starts at the latest assigned node. Most of the time, the latest assigned node is the top of the full tree.

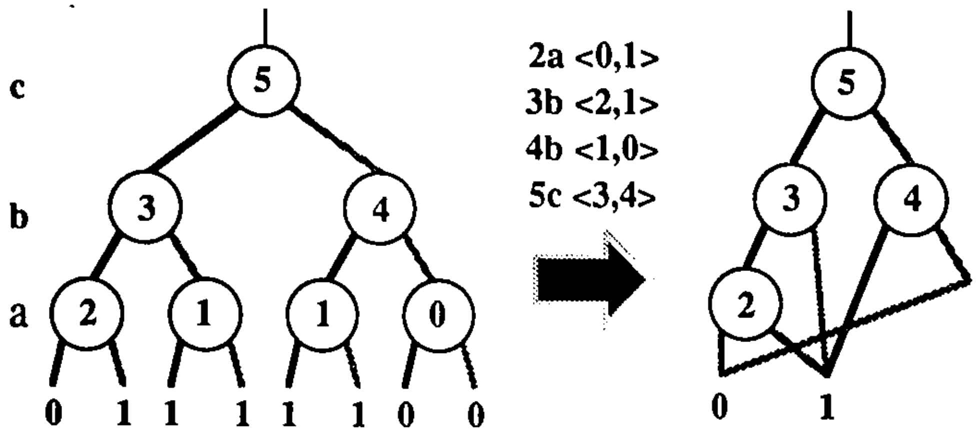
Figure 3.9 Merged Graphical Representation Tree



There are two points that are importance in optimizing the minimization. One is the order of variables. For example, in Figure 3.10, the left handside graphical representation diagram is with the reversed order of variables from the above example and the right handside is the merged graphical representation. Comparing with the merged tree in

Figure 3.9, it has reduced two more transistors from the same boolean function by just reversing the order of variables. Concerning the variable ordering, the better minimization comes with the order of the most changing variable at the bottom of the switching tree and going up. The most stable variable is placed at the top of the switching tree.

Figure 3.10 The Reversed Variable Order Minimization



The other point is the "don't care" case. The "don't care" case is the condition that the output of a boolean function can be either 1 or 0. Consider one of the boolean function outputs in Table 3.1 to be a "don't care" case. "X" is the "don't care" case which can represent "1" or "0".

Figure 3.11 An Example of "Don't Care" Case

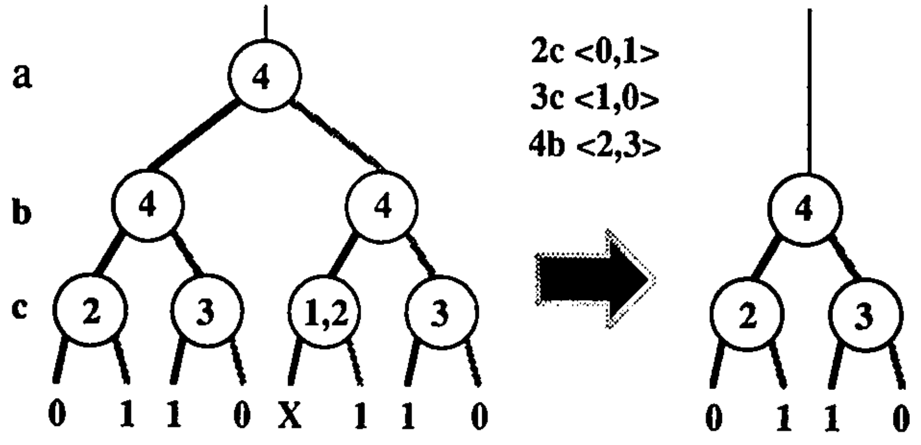


Figure 3.11 is the minimizing process of the "don't care" case. Since "X" can be "1" or "0", the node with the "don't care" can be assigned either "1" or "2". The left half of the sub-tree can be similar to the right half of the sub-tree if the "don't care" case is considered as a "0". In this example, it is not only minimizing the count of transistor but it also reducing the tree height from 3 to 2. In regard to the "don't care" case, the best minimization is resulted by trying all possible combinations of "don't care" cases in the boolean function. For example, if there are 20 "don't care" cases in the function, it will have $2^{20} \Leftrightarrow 1048576$ of full switching tree to minimize for each variable ordering needed to be minimized. It is a very time consuming procedure which is further complicated if there are many "don't care" cases. There is another approach to optimizing the minimization due to the "don't care" cases. More transistors can be reduced if the merging of the sub-trees occurs at the top of the switching tree[19]. By assigning "1" and "0" to "don't care" cases, the sub-trees are tried to be merged from the top of the switching tree. If the sub-tree cannot be merged, then we go down one level and try to merge again. This process goes on until the bottom of the tree is reached.

3.2.2 Switching Tree Minimization Program

The switching tree minimization program in Section B.4. is the minimization of transistor switching tree. It uses the set of minimization rules for CVSL discussed above. The program can merge up to 8 full transistor switching trees. If memory storage in fixed disk is enough, it can be expanded more until no more memory storage is available. For example, the truth table of 8 full transistor switching tree with 19 variables needs about 8 MByte of memory storage under the UNIX system. The program requires about 12 MByte of memory storage. It is a simple program that cannot compare with the other optimized switching tree minimization software since it does not do any variable reordering or tries to work with "don't case" cases so the variable ordering and "don't care" cases need to be taken care of before putting the truth table into the program. Here is an example of how the program minimizes the switching tree as following. In this example, there are 4 full switching trees with 3 variables needed to merge together. The process is from the left to the right. t_1, t_2, t_3, t_4 are the names of the full switching trees.

a, b, c are the three input variables, where a is at the bottom of the tree and c is the top of the tree.

$$\begin{bmatrix} t_1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ t_2 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 \\ t_3 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ t_4 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The merged entries of each node at a, b and c level are shown as following from left to right. The key tables of each level are shown at the right side of the entries.

$$'a\begin{bmatrix} t_1 & 2 & 0 & 1 & 2 \\ t_2 & 0 & 1 & 1 & 3 \\ t_3 & 1 & 2 & 0 & 3 \\ t_4 & 3 & 3 & 0 & 0 \end{bmatrix}; \begin{matrix} 2a <1,0> \\ 3a <0,1> \end{matrix} \qquad 'b\begin{bmatrix} t_1 & 4 & 6 \\ t_2 & 5 & 7 \\ t_3 & 6 & 8 \\ t_4 & 3 & 0 \end{bmatrix}; \begin{matrix} 4b <2,0> \\ 5b <0,1> \\ 6b <1,2> \\ 7b <1,3> \\ 8b <0,3> \end{matrix} \qquad 'c\begin{bmatrix} t_1 & 9 \\ t_2 & 10 \\ t_3 & 11 \\ t_4 & 12 \end{bmatrix}; \begin{matrix} 9c <4,6> \\ 10c <5,7> \\ 11c <6,8> \\ 12c <3,0> \end{matrix}$$

Figure 3.12 The Merged Tree Graphical Representation

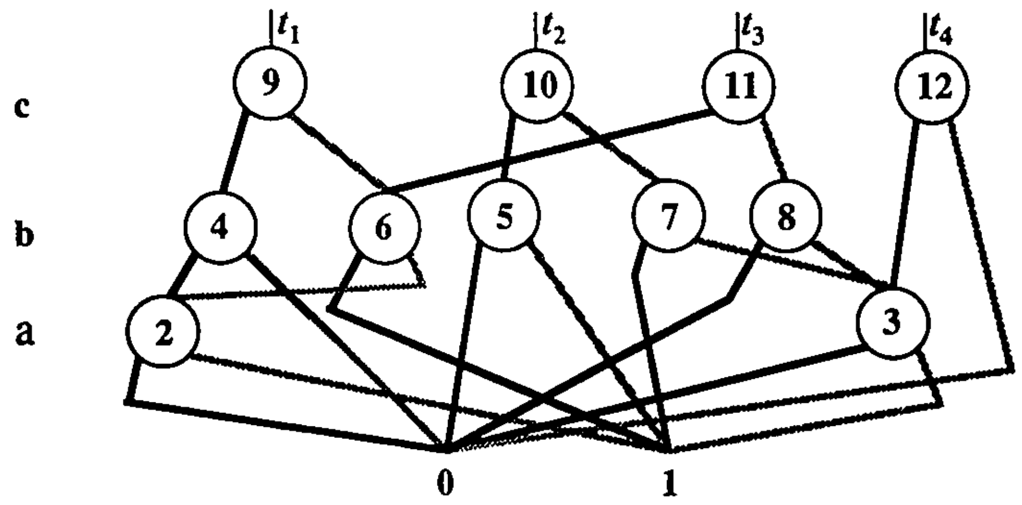


Figure 3.12 is the structure after merging four full trees together. Depending on whether a complemented or the non-complemented output needs to be created, either connections to "1" or connection to "0" will be removed from the NMOS transistor block.

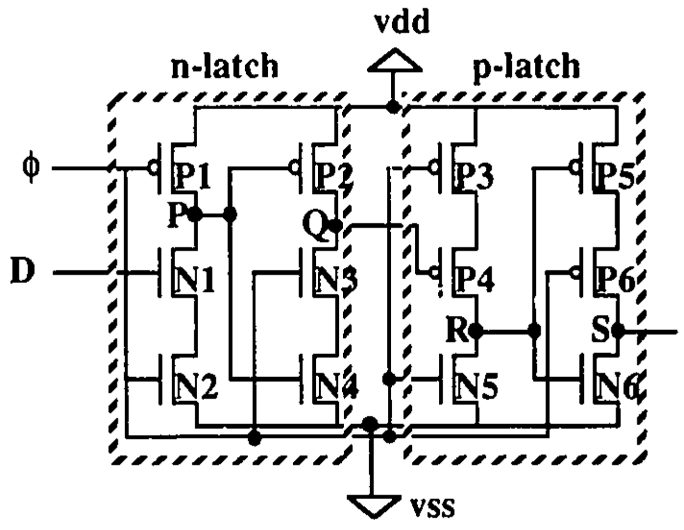
3.3 Latch Design

There are two kinds of latches involved in the implementation of HyPE. One is the True Single Phase Clocking (TSPC) latch. It is a voltage sensing latch. Another is the Ultra Fast Dynamic Current Steering (UCDCS) latch. It is a pseudo single phase clocking latch. The reason for choosing these latches in the implementation is that both latches can be used in a single clocking system. They need only one external clock input signal although the ultra fast dynamic current steering latch needs a complemented clock. It is much easier to control than the other latches which need two clock signals. It is also cost efficiency, by reducing the area that is needed for the additional clock wire.

3.3.1 True Single Phase Clocking Latch (TSPC)

True Single Phase Clocking (TSPC) latch, is shown in Figure 3.13. It is a precharged dynamic pipelined structure. TSPC latch contains two parts, a master TSPC n-latch and a slave TSPC p-latch, as shown in Figure 3.13.

Figure 3.13 True Single Phase Clocking (TSPC) Latch



In the master n-latch, when the clock, ϕ, is low, P1 is on. The internal parasitic capacitor is charged to vdd. The n-latch is in the precharge phase. Node Q is at the previous status. When the clock, ϕ, is high, n-latch is in the evaluate phase. The output of node Q is the same as the input signal, D. The status of each node in the master n-latch are shown in Table 3.2 . Q(t-1) in the table represents the previous status.

Table 3.2. Status of Master n-latch in TSPC Latch

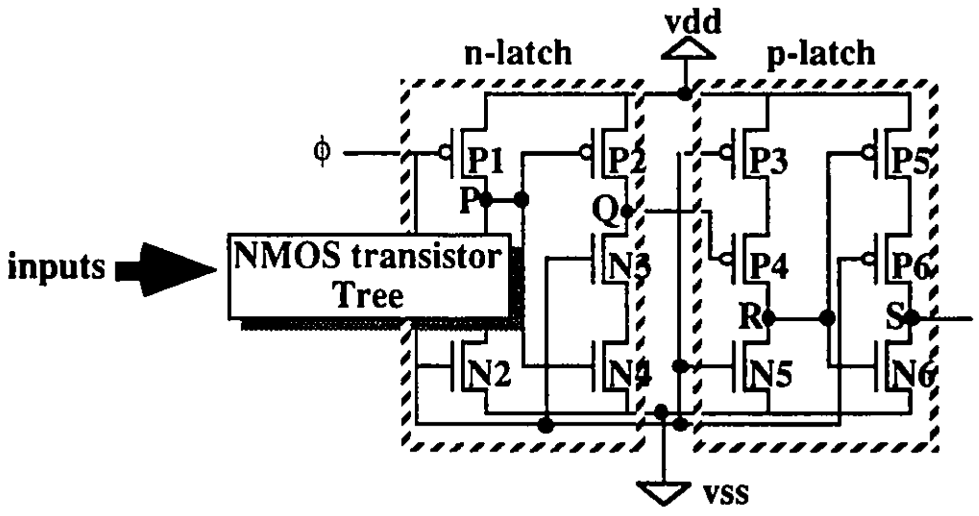
ϕ	D	P	Q
low	1/0	1	Q(t-1)
high	1	0	1
high	0	1	0

The slave p-latch is just the opposite to the n-latch. When the n-latch is in the precharge phase, the p-latch is in the evaluate phase. When the n-latch is in the evaluate phase, the p-latch is in the precharge phase. Table 3.3 shows the status of node in the p-latch.

Table 3.3. Status of Slave p-latch in TSPC Latch

ϕ	Q	R	S
high	1/0	0	S(t-1)
low	1	0	1
low	0	1	0

Figure 3.14 Replace the N1 transistor with the NMOS Transistor Tree

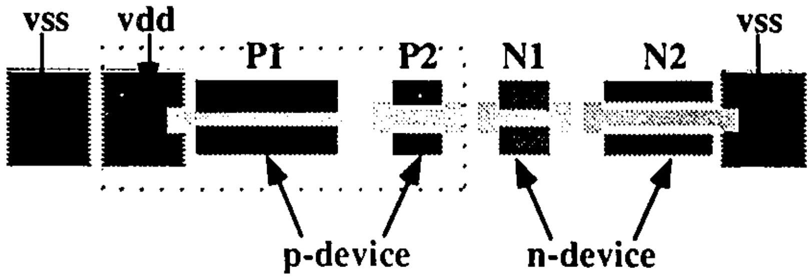


The NMOS transistor tree replaces the N1 transistor, as shown in Figure 3.14. If the non-complemented output signal is required, all the connections to "1" are removed since the NMOS transistor tree block is connected to the ground, vss. If the complemented output signal is required, connections to "0" are removed. The complemented and non-complemented output signals cannot be obtained at the same time if there is more than one full transistor tree merging together.

3.3.1.1 TSPC Design Using in tcell Library

Since the BATMOS technology is used, the TSPC latch had to be designed with tcells, that were the only standard cells available at the time. The first thing that needs to be clarified is the structure of the tcell. Tcell in BATMOS library is created by combining cells, as shown in Figure 3.15.

Figure 3.15 One of the Basic Cell of tcell in BATMOS



This cell has all the requirement for constructing any kind of logic gate. In the middle of the cell is the gate for the NMOS or the PMOS. It can contain up to 2 PMOS transistors and up to 2 NMOS transistors. It depends on the p-device or the n-device expanding to the other side of gate. Because each device can expand to the other side, there are 16 basic cells similar to the cell in Figure 3.15. The device length is fixed. P1, P2, N1, and N2 are 11.8µm, 3µm, 3µm and 8.8µm, respectively. The size of this basic cell is 70.4µm x 7.2µm.

Figure 3.16 Transistor Level Division of TSPC Latch

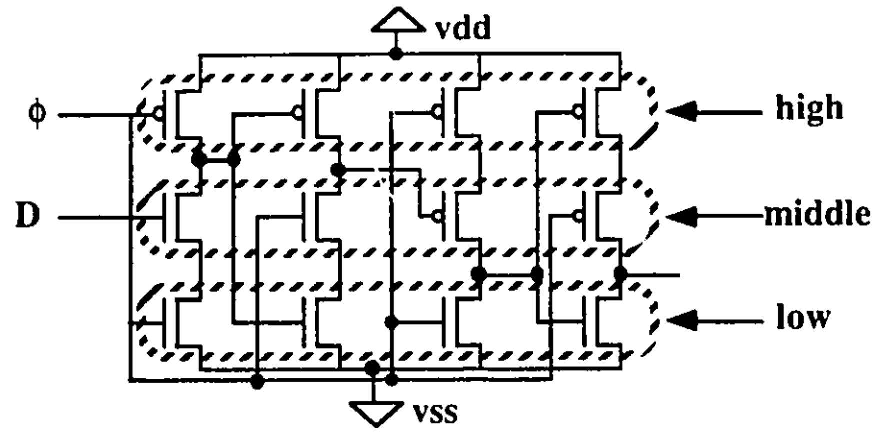


Figure 3.16 is the level division of the TSPC latch's transistors. It is used for trying out different combination of transistor sizes that the basic cell of the tcell library can provide.

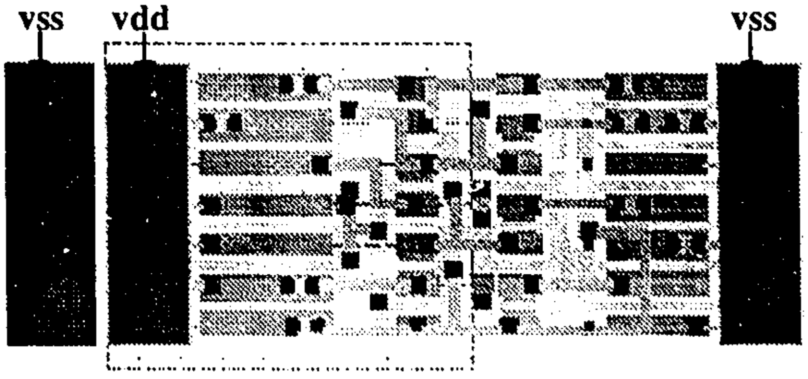
Table 3.4. Different Combinations of Transistor Size

PMOS (μm)		NMOS (μm)		Output at Rise (ns)			Output at Fall (ns)		
high	mid	mid	low	2V	3V	4V	2V	3V	4V
3	11.8	3	8.8	0.65	0.8	1.0	1.45	1.27	1.07
3	11.8	8.8	3	0.44	0.58	0.74	2.48	2.1	1.76
11.8	3	3	17.6	1.05	1.18	1.44	1.2	1.08	0.93
11.8	3	3	8.8	0.84	0.95	1.11	1.19	1.06	0.91
11.8	3	3	8.8ll8.8	1.02	1.2	1.45	1.21	1.08	0.95
11.8	3	3	3	0.66	0.76	0.87	1.22	1.11	1.00
11.8	3	3ll3	8.8	0.82	0.94	1.11	1.21	1.12	1.03
11.8	3	8.8	11.8	0.88	1.01	1.21	1.33	1.21	1.1
11.8	3	8.8	3	0.66	0.75	0.87	1.49	1.36	1.16
11.8	3	8.8	8.8	0.8	0.92	1.1	1.34	1.22	1.1

Table 3.4 list different combinations of transistor sizes and shows the performance of the output rising edge and the performance of the output falling edge. As it is shown, the optimum size for high level of PMOS, the middle level of PMOS, the middle level of NMOS and the low level of NMOS are 11.8mm, 3mm, 3mm and 8.8mm, respectively. The rising edge and the falling edge will be the closest and the rising time and the falling

time will be the least. The layout design is shown in Figure 3.17. The layout design in tcell is 70.4μm x 29μm.

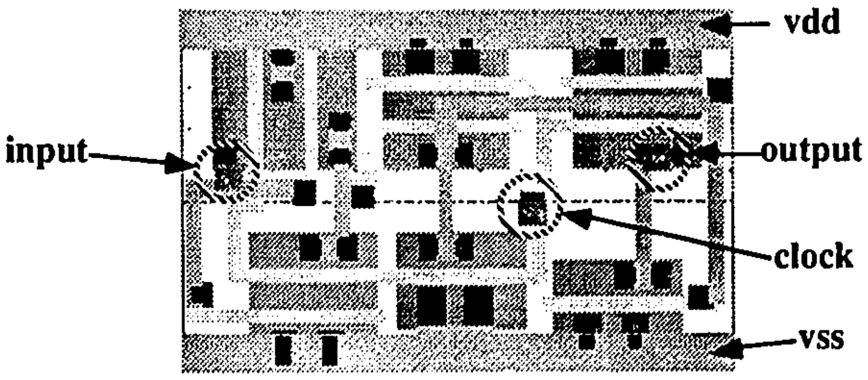
Figure 3.17 TSPC Latch Designed in tcell Format



3.3.1.2 A Full Custom TSPC latch Design

As it can be seen, the tcell design is area inefficient. A full custom TSPC latch is required if standard cells are not used in the design. The following TSPC latch, as shown Figure 3.18, uses unit transistor size for PMOS and NMOS. The layout area is 34.9μm x 22.8μm.

Figure 3.18 TSPC Latch Customized Layout Design

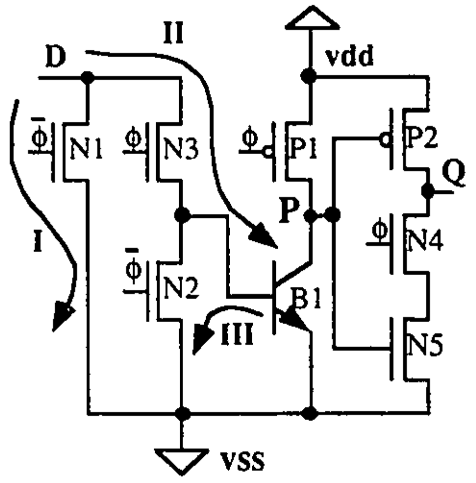


The customized TSPC latch layout design is for the dynamic NMOS transistor block design, so the input, as in Figure 3.18, is fed into the node P in Figure 3.14. The N1 transistor is put into the design afterwards.

3.3.2 Ultra-Fast Dynamic Current Steering Latch

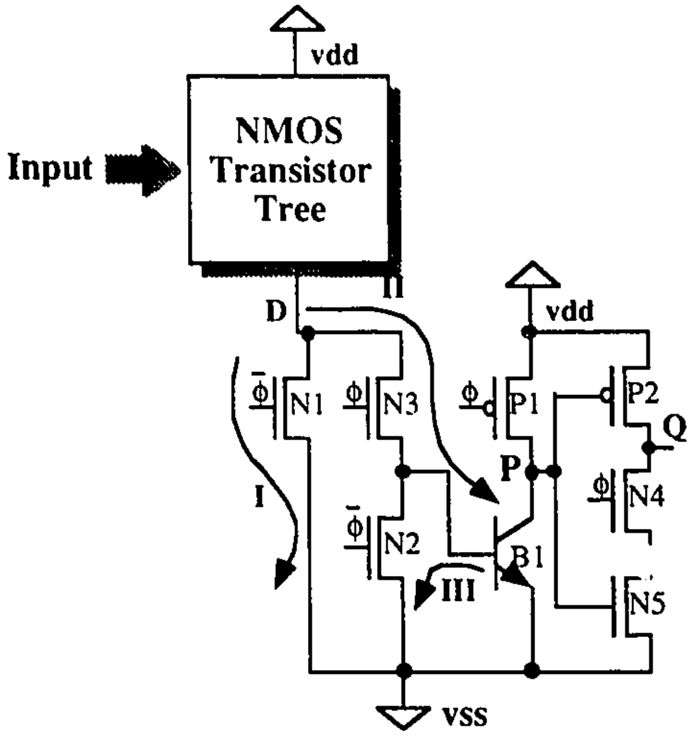
Ultra-Fast Completely Dynamic Current Steering (UCDCS) latch is modified from the completely dynamic current steering (CDCS) latch by Czilli[12] in BiCMOS 0.8μm design technology, using the n-latch based on the CDCS latch design as the master and using the TSPC p-latch as slave. Since the BiCMOS technology can handle the bipolar junction transistors (BJTs) which offer higher switching rates, the modification of the CDCS n-latch is used to improve the pulldown speed of the dynamic node Q.

Figure 3.19 Master n-latch of UCDCS Latch



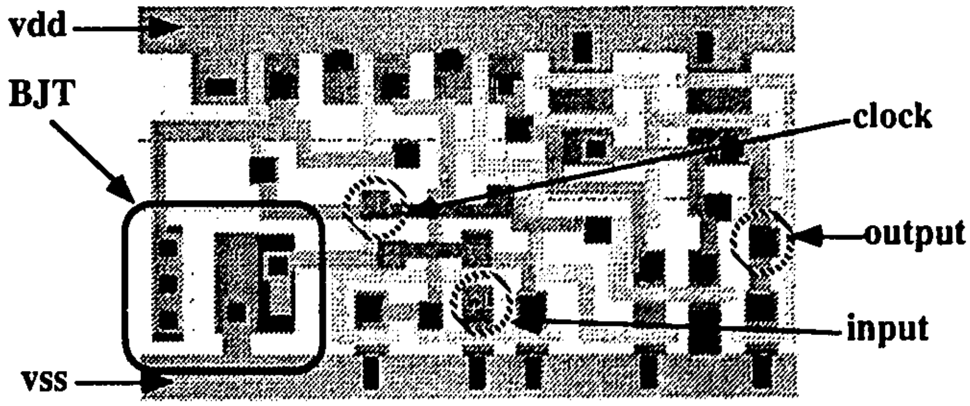
When clock, ϕ, is low, transistor N1, N2, and P1 are on and transistor N3 is off. The current goes through transistor N1 from the path I. Since transistor N2 is on, the base of bipolar transistor B1 is connected to ground, vss, through path III, that keeps the bipolar transistor B1 off. Also transistor P1 is on and node P is changed to vdd. Since transistor N4 is off, node Q is maintained in its last state for the n-latch. This is the precharge phase of the current steering n-latch. When the clock is high, all transistors are off except transistor N3 and N4. At this point, if there is a current coming from input D, then the current follows the path indicated as II into the base of the bipolar transistor B1 and turn on the device, thus discharging node P, turning on the transistor P2 and turning off the transistor N5. So node Q is charged to vdd. If no current comes from input D, the bipolar transistor B1 will not turn on and node P will be held at the vdd level. Since node P is held at vdd, it turns on transistor N5 and turns off transistor P2. Now node Q has a path go through transistor N4 and N5 to the ground, vss, so node Q will be kept low. That is the evaluate phase of the current steering n-latch.

Figure 3.20 Input Replaced by NMOS Transistor Tree in n-latch



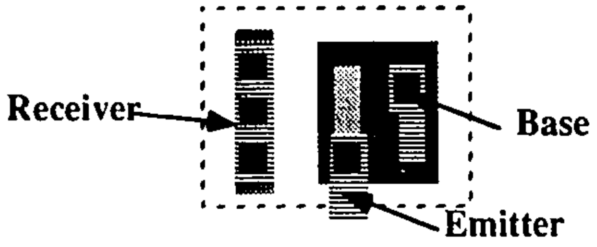
The NMOS transistor tree is similar to the TSPC latch with one exception. If the non-complemented output signal is required, all connections to "0" are removed. If the complemented output signal is required, all connections to "1" are removed. This is opposite to the TSPC latch.

Figure 3.21 UCDCS Latch layout Design



The layout design is based on Czilli's work and it has been reorganized. Since the tcell library does not have the Bipolar Junction Transistor (BJT), no tcell UCDCS latch has been design. The BJT is from the basic cell in BATMOS library. The polarity of the BJT is shown as Figure 3.22.

Figure 3.22 BJT Transistor in BiCMOS



3.4 Summary

This chapter discusses the switching tree minimization algorithm and two latch designs that are used in the implementation process. The switching tree minimization algorithm is based on the set of rules proposed by Bryant. It includes three rules. Also discussed are two important points that can effect the minimization process. Those are the variable ordering and the "don't care" case of the transistor tree. This chapter also includes a simple program that can handle the minimization process with theoretically unlimited tree size or unlimited number of full transistor trees needed to merge together. This program only minimizes the truth table that has already considered the variable ordering and "don't care" cases that exist in the logic function.

The latches outlined in this chapter are the True Single Phase Clocking (TSPC) latch and the Ultra Fast Dynamic Current Steering (UCDCS) latch. Each of them includes two parts: the master n-latch and the slave p-latch. TSPC latch is a true single phase clocking latch. It requires only one input clock signal. UCDCS latch is a pseudo single phase clocking latch that also requires one input clock signal with it complemented. The reason of choosing the UCDCS latch is that it takes the advantage of BiCMOS technique by using the bipolar junction transistor to increase the pulldown speed of the transistor switching.

Chapter4
VLSI Implementation of HyPE

4.1 Introduction

As discussed in Chapter 2, the Hierarchy for Pattern Extraction (HyPE) artificial neural network uses a complex procedure to train the network. After converting the original source code from the Smalltalk object-oriental programming language running on the Mac to the C programming language on UNIX, the training process for ten runs takes about six hours in a SUN Sparc2 workstation. The original Smalltalk program takes about one day to get ten runs of simulation. Although it already shows about four times of improvement, it is not good enough to provide a procedure to tune up the HyPE artificial neural network. Therefore it needs to be implemented to in VLSI in order to accelerate the training procedure. It normally needs about two thousand neuron resources in the middle layer to fulfill the purpose of the network training and it has a complex and random algorithm to create these connections for each neuron.

From the point of view of implementation, it is impractical to have the entire design in a single chip. For example, a self learning artificial digital neural network has been designed using the Wafer-Scale LSI by M. Yasunaga *et al.*[16]. Going through the training procedure, a specific part of the artificial neural network that is used frequently and can be implemented in VLSI is investigated. Since the neuron structures are so similar in each level and are the most time consuming in the algorithm, it is sufficient to create a general neuron that fits in all three levels as a co-processor which works with the host computer. Because each neuron can have over 100 connections, it is not possible to implement all the connection at the same time. Two neuron design approaches have been provided.

4.2 Pipeline Neuron Design Approach

Referring to Eqn. (2.2) and Eqn. (2.3) each neuron in the middle layer has the threshold, T_{ijt}, the connectivity, $\mathbf{C}_{ijt}$, the firing status, G_{ijt} and the "regular or virgin" status, R_{ijt}. After completing many simulations, of the HyPE artificial neural network, it is concluded that the threshold value of each neuron will not exceed 128. When designing the general neuron for the HyPE, 8 bits of storage can handle the threshold value of every neuron. The most significant bit is the sign bit of the threshold value. The connectivity is a vector of boolean numbers: "1" means connecting to the a particular neuron in the upper level and "0" means no connection to the particular neuron. The firing status needs only one bit to be presented and so it is "regular or virgin" status.

Table 4.1. Neuron Parameter Setting

Neuron Parameter	Size (bit)	Comments
T_{ijt}	8	most significant bit is the sign bit.
$\mathbf{C}_{ijt}$	X	Size of upper level.
G_{ijt}	1	firing status.
R_{ijt}	1	regular or virgin.

Since the count of connectivity is flexible, the majority function cannot be satisfactorily applied to it. Because it is not possible to have all input connections feeding into a physical layout neuron design, input connections feed into the physical neuron by section, e.g. 8 bits per period. Following the neuron firing algorithm, when the count of active input connections is greater than or equal to the threshold value of the neuron, the neuron fires except from beta regular neurons. Beta regular neurons need an additional condition. The active regular input connections have at least half of the threshold value. Following is a block diagram of a pipeline neuron design.

Figure 4.1 Pipeline Neuron Design

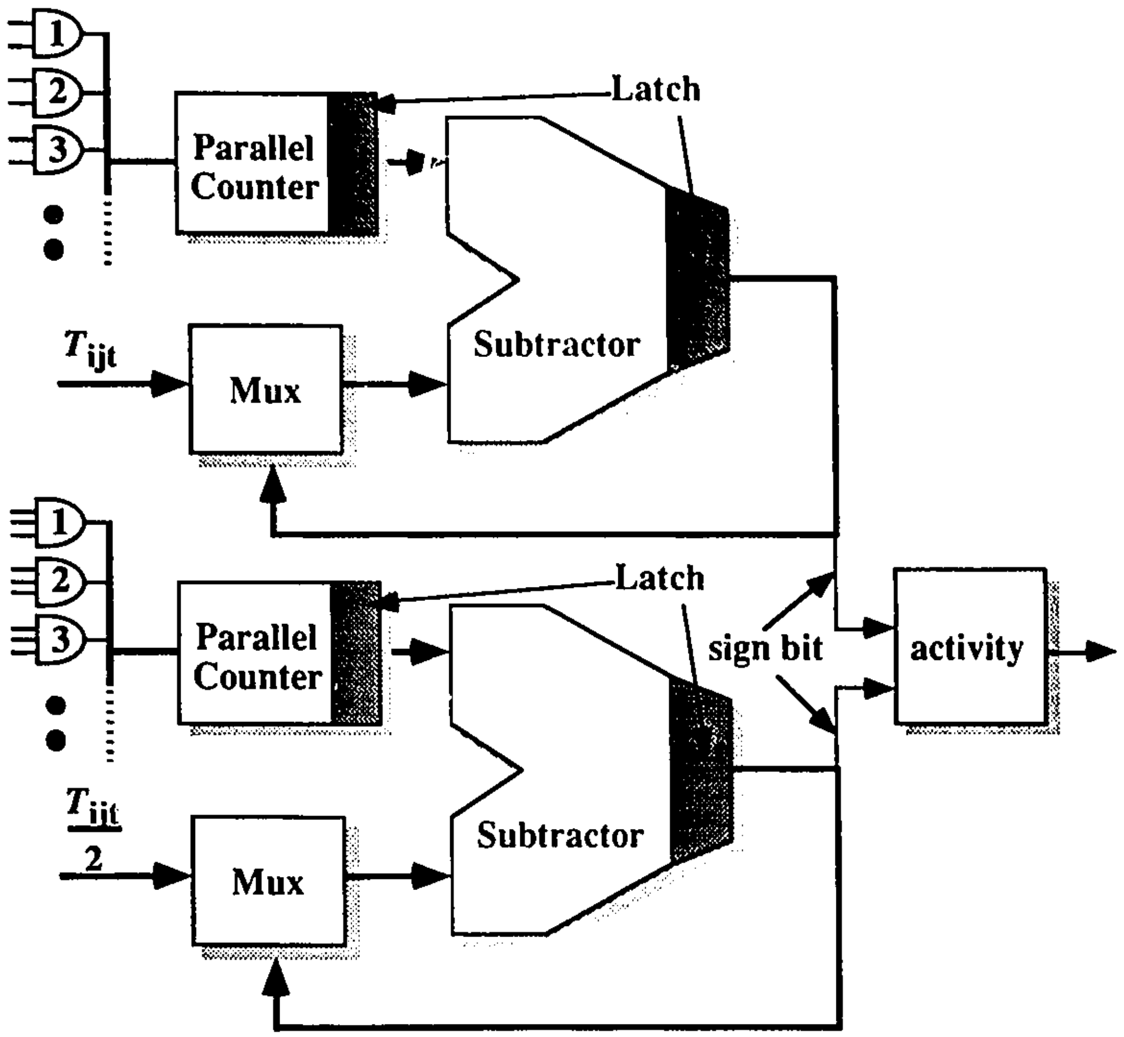


Figure 4.1 shows a pipeline general neuron design. The number of AND logic gate depends on the number of input feeding into the neuron every time. The inputs of the AND logic gates in the upper half are the connectivity, C_{ijt}, of the neuron and the firing status, $F_{n,j-1,t}$, of upper level neurons. The additional input to the lower half is the status of "regular or virgin", $E_{n,j-1,t}$. It applies to the beta regular neuron. The input to the parallel counter is either the active input connections or the active regular input

connections. The functionality of the parallel counter is to count the number of active inputs from the AND logic gate. Latches, put at the output of the parallel counter and the output of the subtractor, are used to maintain the previous status of the block for a stable input to the follower (e.g. the subtractor is the follower of the parallel counter). The firing status of a neuron depends on the number of active inputs, and it is greater than or equal to the threshold (or half of the threshold), therefore the pipeline neuron design is continually subtracting the active input until the recursive value (original value is the T_{ijt} or $T_{ijt}/2$) becomes a negative value or there is no more active input to that neuron. If the recursive value is negative, the sign bit is "1". It will go into the activity selection unit to check the activity of the neuron. The Multiplexer unit is the switch that chooses either to use the threshold, T_{ijt}, (or $T_{ijt}/2$) or the recursive value from the output of the subtractor as the input to the subtractor. The activity selection unit is based on the additional requirements from regular beta neuron. If it is a regular beta neuron, upper half and lower half of the general neuron must have their sign bit output from the subtractor as "1" to fire the neuron. If it is not a regular beta neuron, just the upper half of the general neuron has a sign bit as "1" that will fire the neuron.

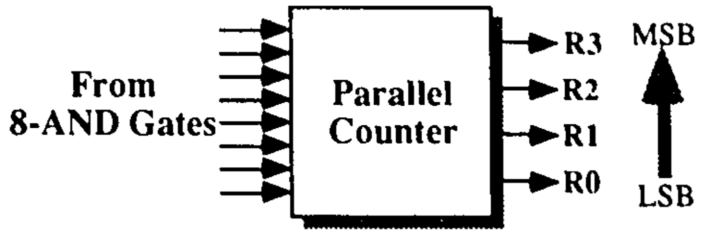
4.2.1 Logic Gate Neuron Design

The logic gate neuron design is based on the structure of the pipeline neuron design. Since the only standard cell library available in BATMOS is the tcell, every design in logic gate will be in tcell format for ease in designing and routing. Few schematic pipeline neuron designs have been done. The neuron design starts with the 8-input neuron. Because 8-bit is a basic unit of computer space (byte), it is a good starting point. It is followed by the 4-input pipeline neuron design, and then a slower design of 8-input neuron. The first two neuron designs are bigger and faster. The last neuron design is much smaller and slower.

4.2.1.1 8-input Pipeline Neuron Design

8-input neuron design requires 8 AND gates as the input to the parallel counter. The resulting parallel counter will have 8-input pins and 4-output pins, as shown in Figure 4.2.

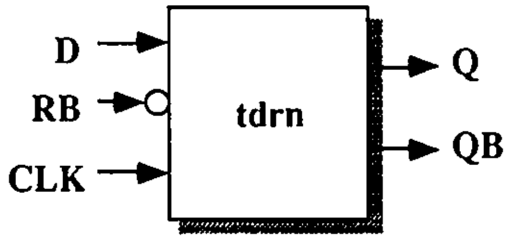
Figure 4.2 8-bit Parallel Counter



R0, R1, R2 and R3 are the output of the 8-bit parallel counter. We have the freedom of minimizing it. The software that is used to minimize the number of logic gates is EXPRESSO ver.2.3 from UC Berkeley. After minimization, the parallel counter needs 362 logic gates which are the simple AND and OR logic gates. Figure C.1 is the schematic of the 8-bit parallel counter. 255 boolean vectors need to be generated with AND gates.

The Latch design used here is the resettable D-type negative edge flip-flop (tdrn), as shown in Figure 4.3, provided in the BATMOS tcell library. The layout of the tdrn flip-flop is shown in Figure C.7. The 8-input parallel counter neuron design requires 24 tdrn flip-flop.

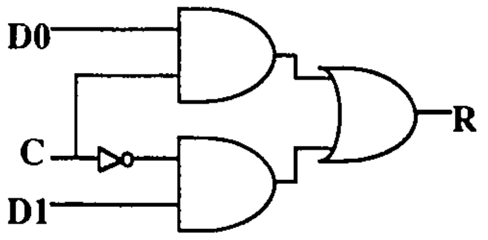
Figure 4.3 The Resettable D-type Negative-edge Flip-Flop(tdrn)



D is the input of the flip-flop. The reset input, RB, is the complement of reset. That means if RB is "0", output, Q, is set to "0" and the complement of output, QB, is set to "1". Output, Q, maintains the status of the input D from one falling edge of the clock (CLK) until the next falling edge of CLK.

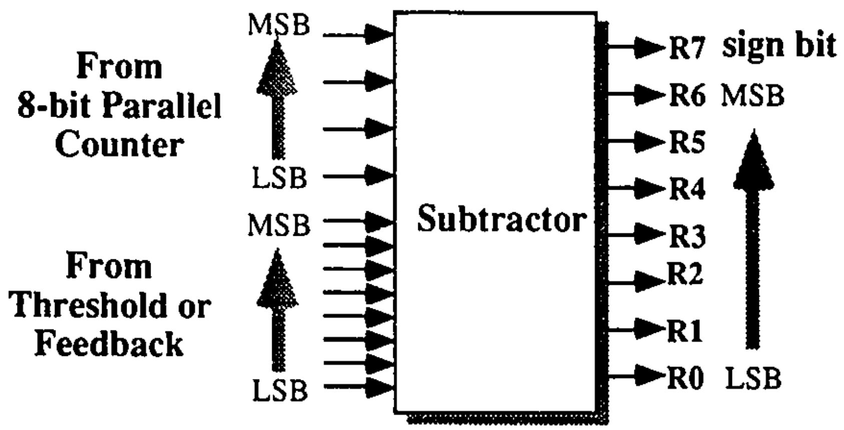
The Multiplexer block is a parallel 2-to-1 multiplexer. It contains 8 parallel unit of 2-to-1 multiplexer. Each 2-to-1 multiplexer contains two AND gates and an OR gate that is shown in Figure 4.4.

Figure 4.4 Schematic of the 2-to-1 Multiplexer



If the selection bit, C, is "1", input D0 will propagate through the multiplexer to the output R. If not, input D1 will propagate to the output.

Figure 4.5 Diagram of the 8-4 Subtractor

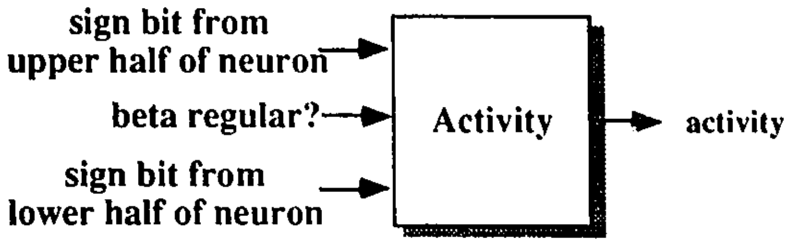


The 8-4 subtractor functional block, as shown in Figure 4.5, is used to continuously feed in the result, 4-bit signals, from the 8-bit parallel counter. It uses the same minimization procedure in EXPRESSO. Since 7 out of 16 combination inputs from the parallel counter are "don't care" cases, the functional block of 8-4 subtractor is minimized much better than the 8-bit parallel counter. The output of the 8-4 subtractor will be stored in the latch until next clock period and it will feedback as the input of the block. The sign bit, R7, will feed into the activity selection unit. Figure C.2 is the schematic of the 8-4 subtractor.

Same as the 8-bit counter, the 8-4 subtractor requires 106 boolean vectors to generate a full functional block.

The activity block is a functional block, as shown in Figure 4.6, that determines the activity of the neuron. It predicts the activity of the neuron on the sign bit from the upper half of the pipeline neuron, the lower half of the pipeline neuron and the algorithm of neuron activity.

Figure 4.6 Diagram of the Activity Block

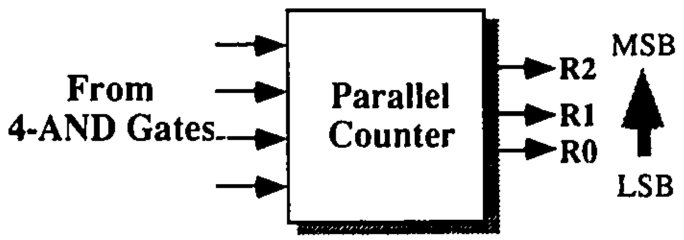

sign bit from upper half of neuron
beta regular?
sign bit from lower half of neuron
Activity
activity

Using Auto Place & Route in the Cadence Edge^{TM} environment, the dimension for the 8-input neuron is 1617µm x 2050µm. It involves of 1131 logic gates. The layout of the 8-input neuron is shown in Figure C.3.

4.2.1.2 4-input Pipeline Neuron Design

The 4-input pipeline neuron design is similar to the 8-input pipeline neuron. The difference is in the design of the parallel counter and the subtractor. The pipeline neuron requires 4 AND gates as the input to the parallel counter. The 4-bit parallel counter design, as shown in Figure 4.7, has 4 input pins and 3 output pins.

Figure 4.7 Diagram of the 4-bit Parallel Counter



It uses the same minimization procedure as the 8-bit counter or the 8-4 subtractor. 15 boolean vectors need to be implemented to provide the functionality of the 4-bit counter. The schematic of the 4-bit counter is shown in Figure C.4.

Figure 4.8 Diagram of 8-3 Subtractor

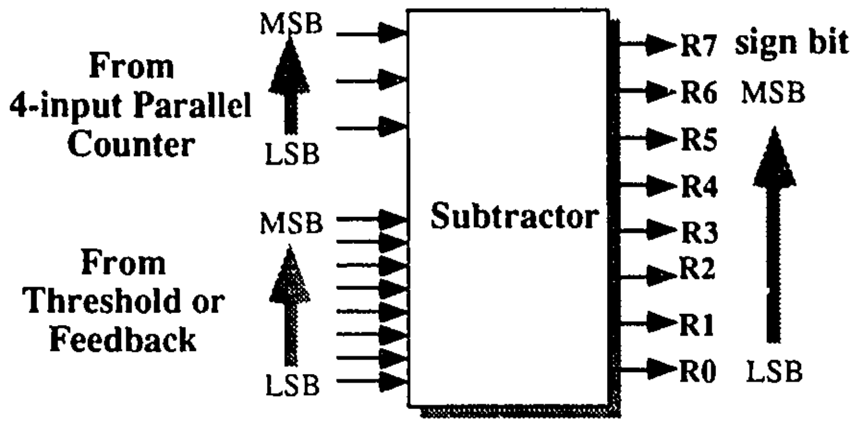


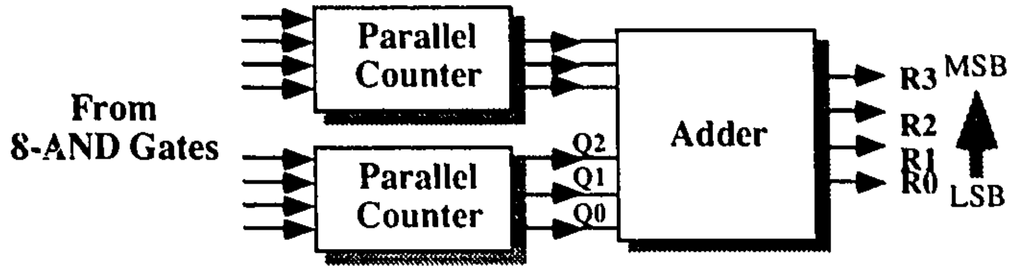
Figure 4.8 shows the 8-3 subtractor that is used in the 4-input pipeline logic gate neuron design. It contains a similar structure as the 8-4 subtractor. It has 11 input pins where 3 inputs come from the 4-bit parallel counter and 8 of them come from the threshold or the feedback from the output of the subtractor. The 3 inputs coming from the 4-bit parallel counter have 3 out of 8 combinations that are "don't care" cases, so better minimization can be achieved from the program. 74 boolean vectors are required to provide the function of the 8-3 subtractor block. The schematic of the 8-3 subtractor is shown in Figure C.5. The final 4-input pipeline neuron requires 945µm x 912µm silicon area. It consists of 409 logic gates.

4.2.1.3 Slow 8-input Pipeline Neuron Design

The slow 8-input pipeline neuron design is similar to the 8-input pipeline neuron design Since the 8-input pipeline neuron requires too much silicon area, another minimization approach is applied. The slow 8-input pipeline neuron design focuses on minimizing the count of logic gates. This new approach is applied to the parallel counter and the subtractor.

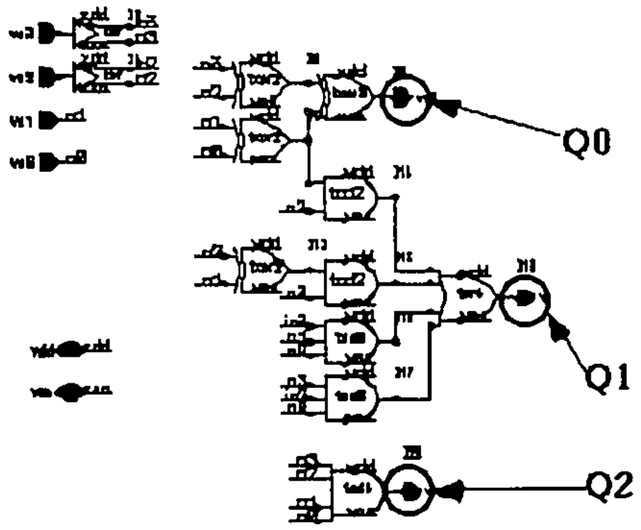
The new design of 8-bit parallel counter, as shown in Figure 4.9, includes three functional blocks, two 4-bit parallel counters and a 3-3 adder. First the inputs from the AND gates are divided into two groups which feed into the 4-bit parallel counter. Next step is to add up the result from the parallel counter.

Figure 4.9 Diagram of a Slow 8-bit Parallel Counter



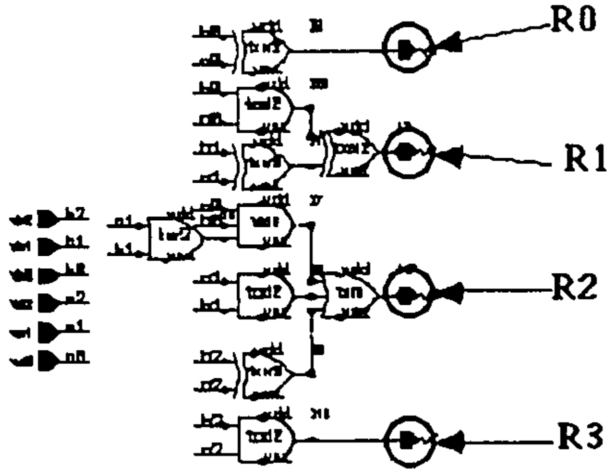
The 4-bit parallel counter no longer uses the same design as the 4-input pipeline neuron. The minimization goes through the customary method using the K-map, as shown in Figure 4.10.

Figure 4.10 Schematic of the Slow 4-bit Parallel Counter



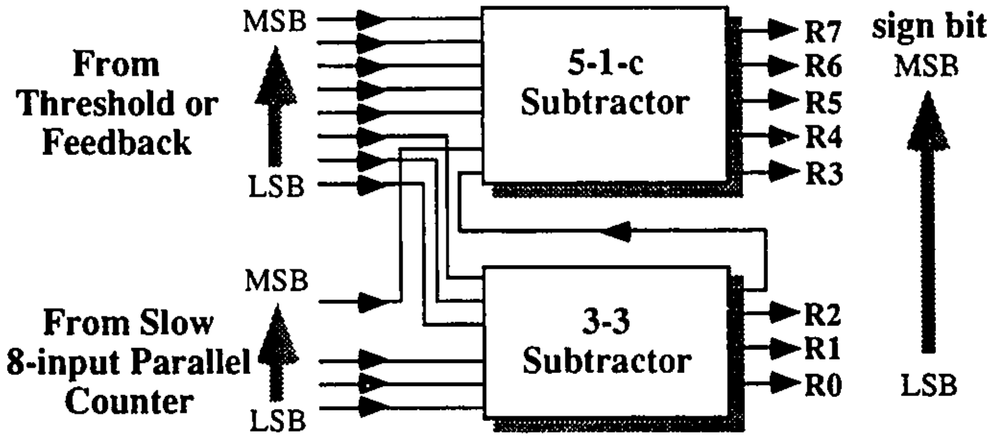
Comparing the 4-bit parallel counter with the 4-input pipeline neuron, the former needs 27 gates to provide the function of 4-bit parallel counting and the latter needs 12 gates to provide the same functionality.

Figure 4.11 Schematic of the 3-3 Adder



The 3-3 adder, shown in Figure 4.11, adds both inputs from the 4-bit parallel counter. Since it involves "don't care" cases at the input of the parallel counter, greater minimization can be achieved.

Figure 4.12 Diagram of Slow 8-4 Subtractor



The new 8-4 subtractor, shown as Figure 4.12, is also divided into two functional blocks, a 3-3 subtractor and a 5-1-c subtractor. The 3-3 subtractor is a subtraction functional block that subtracts two 3-bit inputs. The most significance bit of the 3-3 subtractor will propagate to the 5-1-c as the carry bit. The 5-1-c subtractor is a subtraction block that subtracts a 5-bit input and a one bit input with carry. The schematic of the 5-1-c subtractor and the 3-3 subtractor are shown in Figure 4.13 and Figure 4.14.

Figure 4.13 Schematic of 5-1-c Subtractor

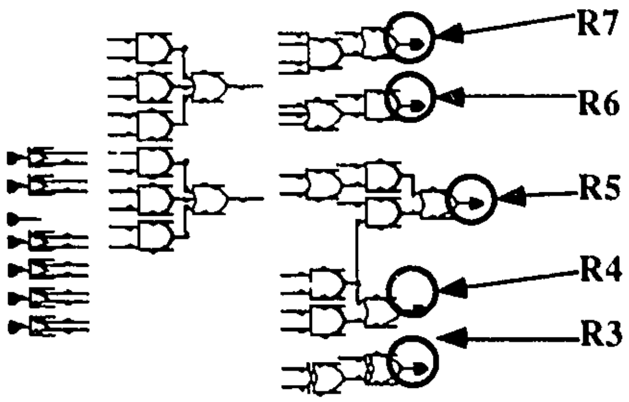
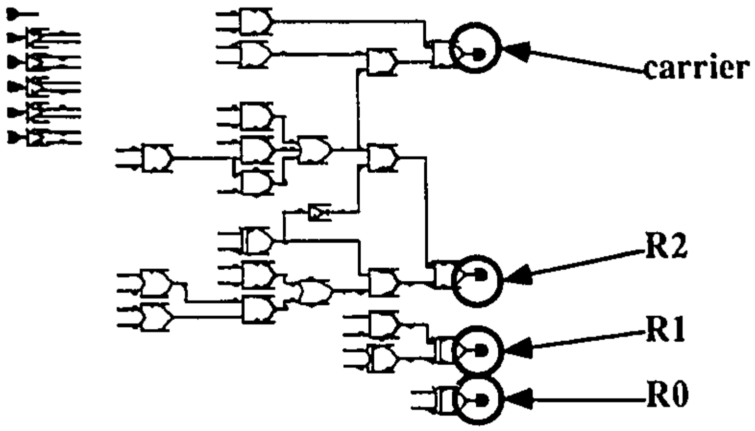


Figure 4.14 Schematic of 3-3 Subtractor



The design approach of this slow 8-input pipeline neuron is to minimize the use of silicon area. The dimension of the design is about 794μm x 840μm. It needs 269 gates to provide the same function as the 8-input pipeline neuron. The minimization gain is due to using the exclusive-OR gate that tcell library provides and merging the logic gate customizing.

4.2.2 Pipeline Dynamic Neuron Design

Two pipeline dynamic neuron designs that have been worked on are the 8-input switching tree pipeline neuron and the 7-input switching tree pipeline neuron.

4.2.2.1 8-input Switching Tree Pipeline Dynamic Neuron

The 8-input switching tree pipeline neuron is used in the pipeline neuron design in Figure 4.1. The difference between the 8-input pipeline neuron design in Section 4.2.1.1 and the 8-input switching tree pipeline neuron is as follows. The former uses only the logic gate of tcell library and the latter uses the switching tree to create the functional block of the parallel counter and the subtractor, and it uses the reorganized logic to produce the function of multiplexer and the activity block.

Figure 4.15 8-bit Switching Tree Parallel Counter

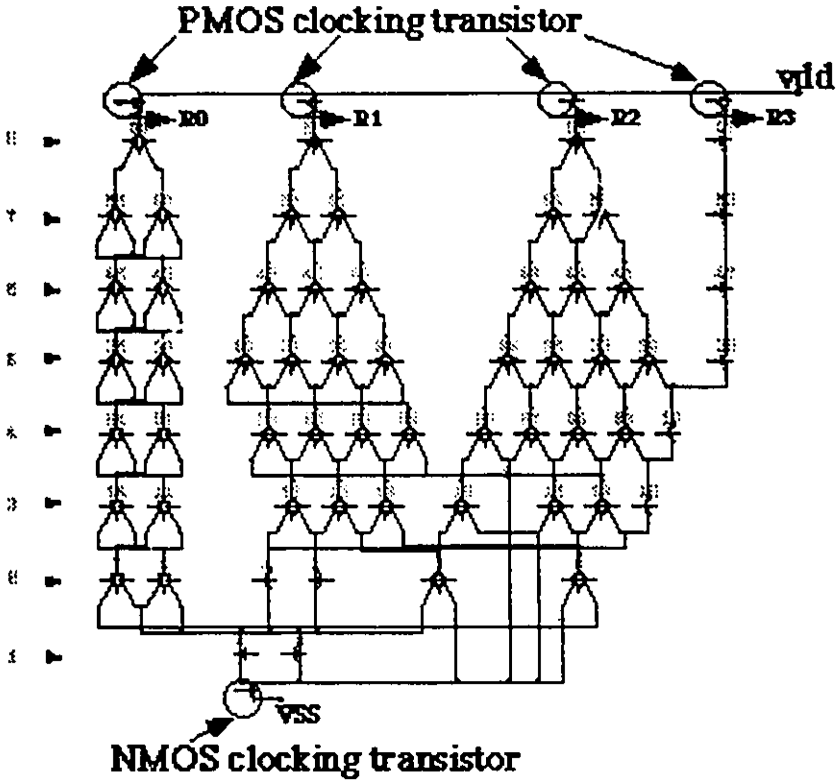
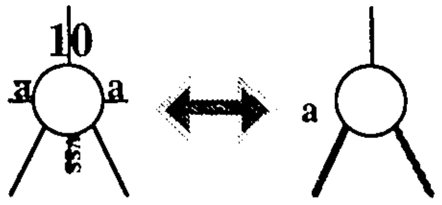


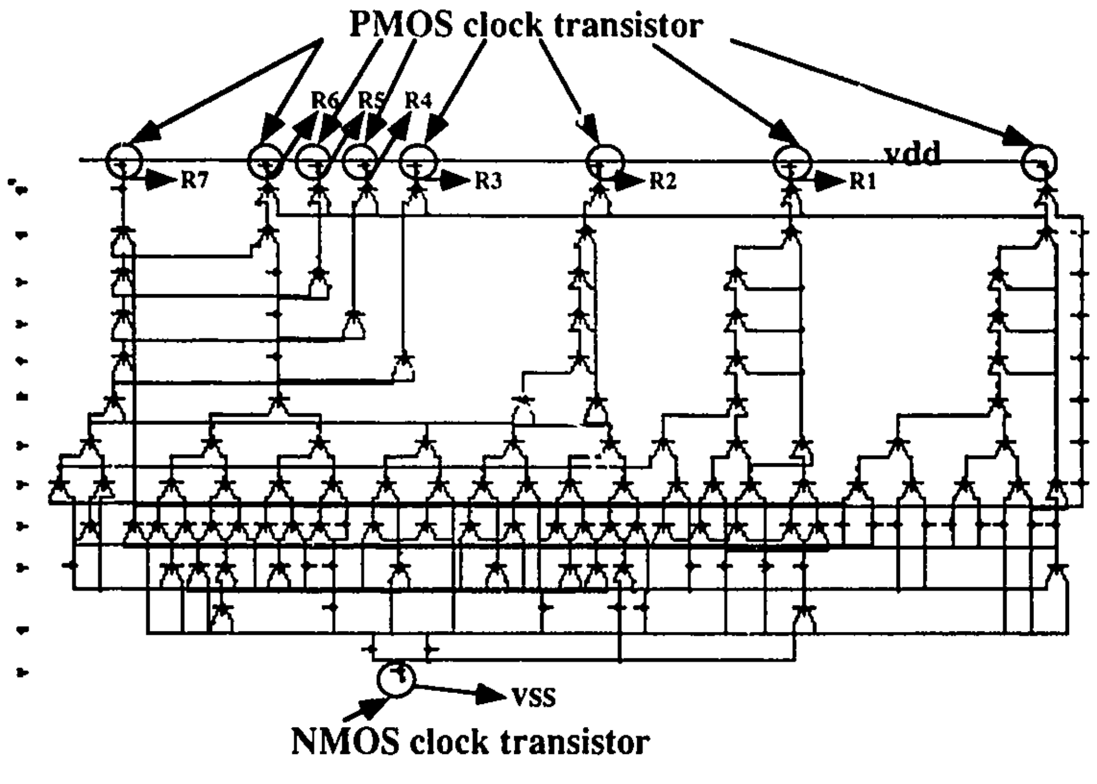
Figure 4.15 shows the schematic of the 8-bit parallel counter. This design will only give the complement of the output since there are four merged switching trees. It uses the simple dynamic logic, shown in Figure 3.1. The clock transistors are shown in the figure. Since it is an 8-bit parallel counter, it requires four outputs. R3 is the most significant bit and the R2 is the second significant bit and so on. The schematic symbol of the transistor pair is equivalent to the graphical symbol which is shown in Figure 4.16.

Figure 4.16 Schematic Symbol and the Graphical Symbol



After the minimization, the switching tree's height remains 8 high. The minimization program, discussed in Section 3.2.2, reduces the switching tree to about 60 pairs of transistor. Since the complemented outputs and the non-complemented outputs cannot be produced from the same tree at the same time, the output must be selected before creating the schematic. The layout of the 8-bit parallel counter is shown in Figure C.13.

Figure 4.17 Schematic of the 8-4 Switching Tree Subtractor

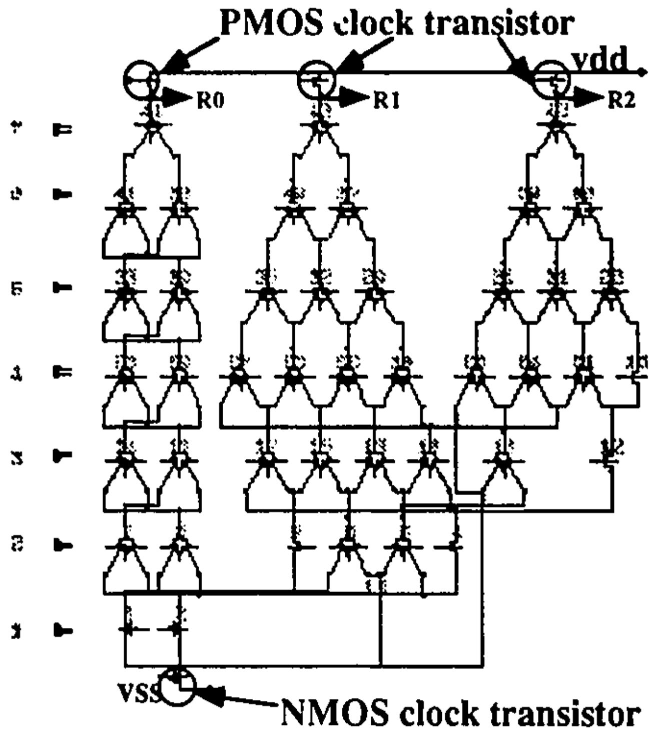


The 8-4 subtractor, shown in Figure 4.17, uses the same strategy as the 8-bit switching tree parallel counter. It has been reduced to 134 pairs of transistor using the minimization program discussed in Section 3.2.2. The layout of the 8-4 subtractor is shown as Figure C.14.

4.2.2.2 7-input Switching Tree Pipeline Neuron Design

The 7-input pipeline neuron design tries to fully use the 3 output bits of the parallel counter to get a better minimization in the switching tree of the functional block. The 7-bit parallel counter has total 44 pair of transistor. The schematic of the 7-bit parallel counter is shown in Figure 4.18 and the layout is shown in Figure C.15.

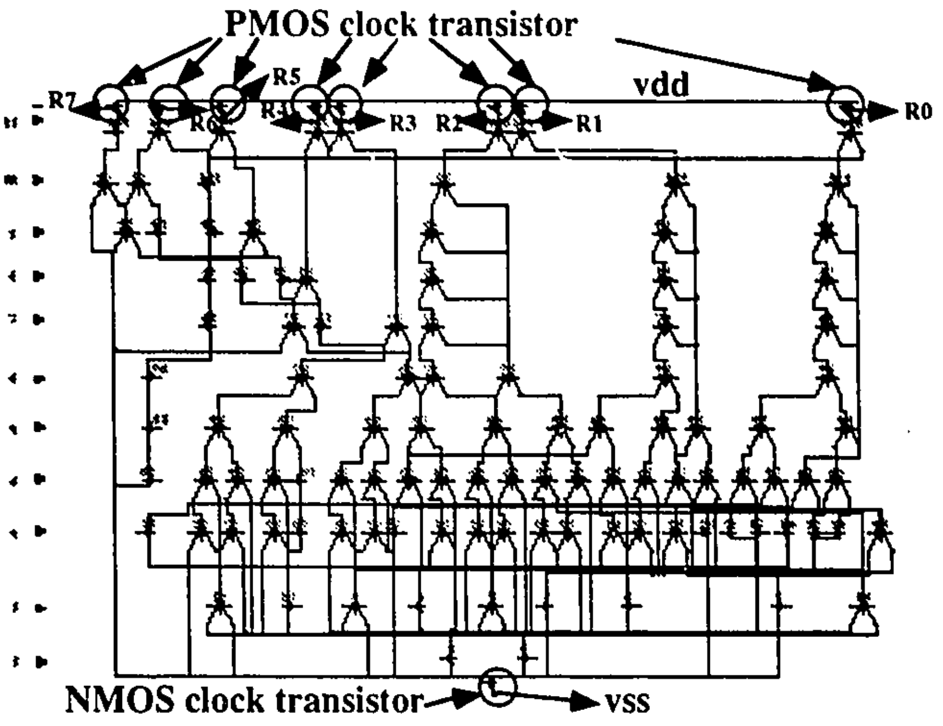
Figure 4.18 Schematic of the 7-bit Parallel Counter with Complement Outputs



The 8-3 subtractor for the 4-input pipeline neuron is not the same as the 8-3 subtractor, shown in Figure 4.19, for the 7-input pipeline neuron, since the former uses the advantage of the "don't care" optimization and the latter no longer has that advantage because the 3 input bits from the 7-bit parallel counter are already fully used. The layout of the 8-3 switching tree subtractor is shown in Figure C.16.

Each of the switching tree functional blocks in the 8-input switching tree pipeline neuron design and in the 7-input switching tree pipeline neuron design has been simulated for functionality with the HSPICE^{TM} simulator.

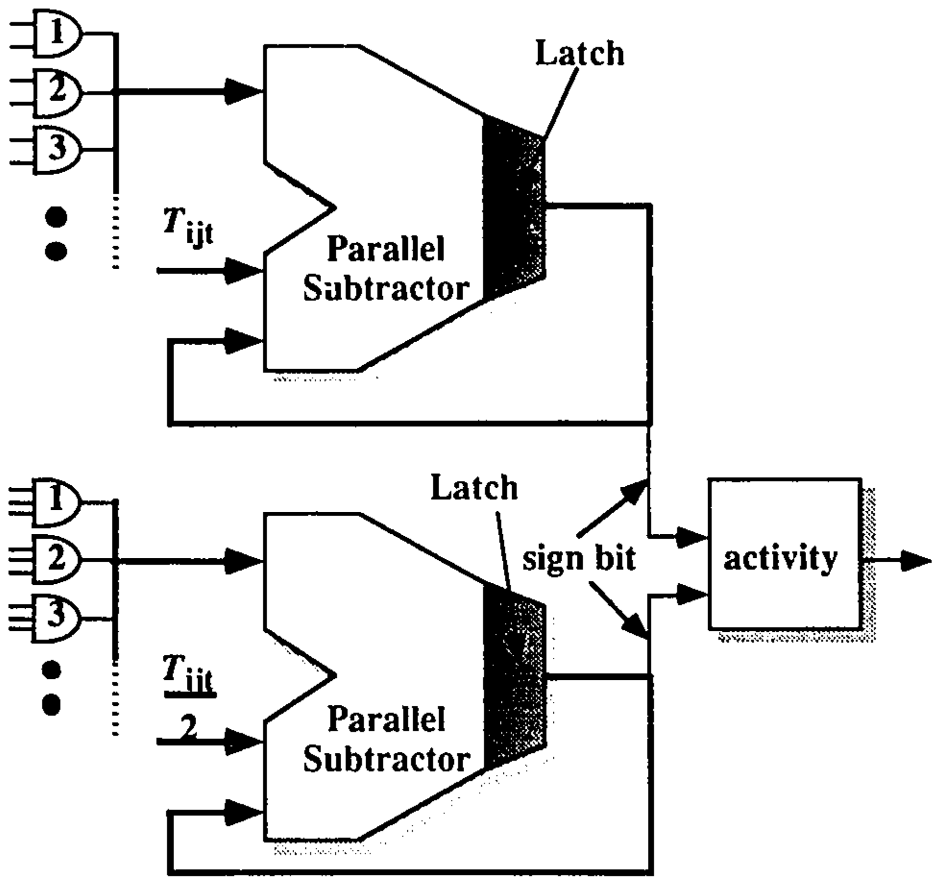
Figure 4.19 Schematic of the 8-3 Subtractor with Complemented Outputs



4.3 Single Block Neuron Design Approach

The single block neuron design is the other approach of neuron design. There are two dynamic logic neuron designs. The single block neuron design approach merges the parallel counter functional block, the subtractor functional block and the multiplexer together from the pipelined neuron design. It is shown in Figure 4.20. The single block neuron design has the advantage of using fewer latches in the design and it only requires the design of a single functional block, the parallel subtractor. The design of the activity still uses the logic gate.

Figure 4.20 Single Block Neuron Design



4.3.1 Reorganized Cell

Since the switching tree design is used in the neuron design, it is beneficial to take advantage of custom logic gates design because a lot of silicon area is wasted in the tcell logic gate in the BATMOS library. Although the reorganized custom logic gate designs still use the same transistor count and size as the tcells, their layouts are much more efficient.

Table 4.2. Comparison of Tcell and Reorganized Logic Gates

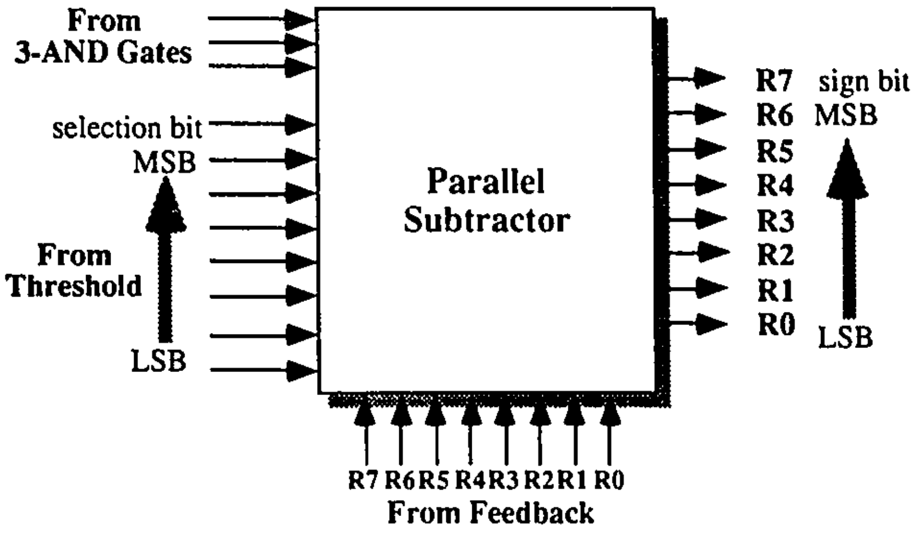
Gates	tcell (μm x μm)	reorganized (μm x μm)	percentage
AND-2	18 x 70.4 = 1267.2	28 x 21.9 = 613.2	48
AND-3	21.6 x 70.4 = 1520.64	35.2 x 21.9= 770.88	51
Buffer	14.4 x 70.4 = 1013.76	35.1 x 16.4 = 575.64	57
Inverter	10.8 x 70.4 = 760.32	12.8 x 21.9 = 280.32	39
OR-2	18 x 70.4 = 1267.20	29.1 x 21.9 =637.29	50

Table 4.2 shows the silicon area required for each of the logic gates needed in the neuron design. The area occupied by the reorganized logic gates is about 50% of the tcells. The layouts of the reorganized cell are shown in Figure C.8 to Figure C.12.

4.3.2 3-input Neuron Design

Two 3-input dynamic neuron designs have been finished. One uses the TSPC latch and the other one uses the UCDCS latch. The result of choosing the 3-input as the neuron design is due to the limitation of the memory storage that is required by the switching tree minimization program.

Figure 4.21 Diagram of the 3-bit Parallel Subtractor



The parallel subtractor requires 19 input bits: 3 from the AND gates, 7 from the threshold, 8 from the feedback of the output of the parallel subtractor and 1 selection bit that is the same bit that goes into the multiplexer in the pipeline neuron design.

Figure 4.22 Schematic of 3-bit Parallel Subtractor using TSPC Latch

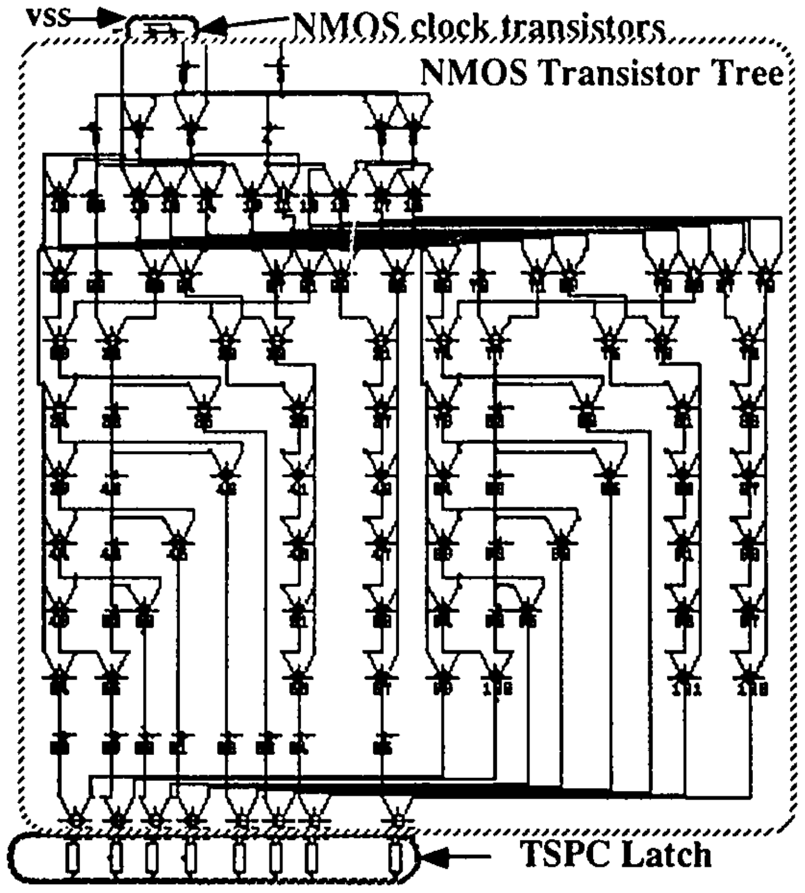
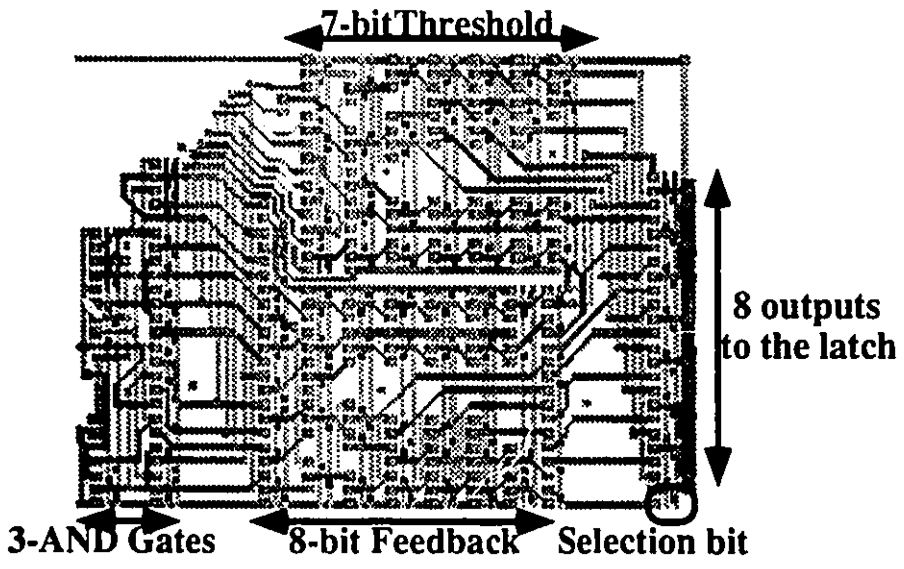


Figure 4.23 Layout of the NMOS Transistor Tree



The TSPC latch, used in figure 4.22, is the custom TSPC latch design as shown in Figure 3.18. The NMOS clock transistors are connected between the ground, vss, and the NMOS transistor tree. Figure 4.23 shows the layout of NMOS transistor tree of the 3-bit parallel subtractor and it shows the input and the output of the NMOS transistor tree. The dimension of the NMOS transistor tree is about 125μm x 155.8μm.

Figure 4.24 Schematic of 3-bit Parallel Subtractor Using the UCDCS Latch

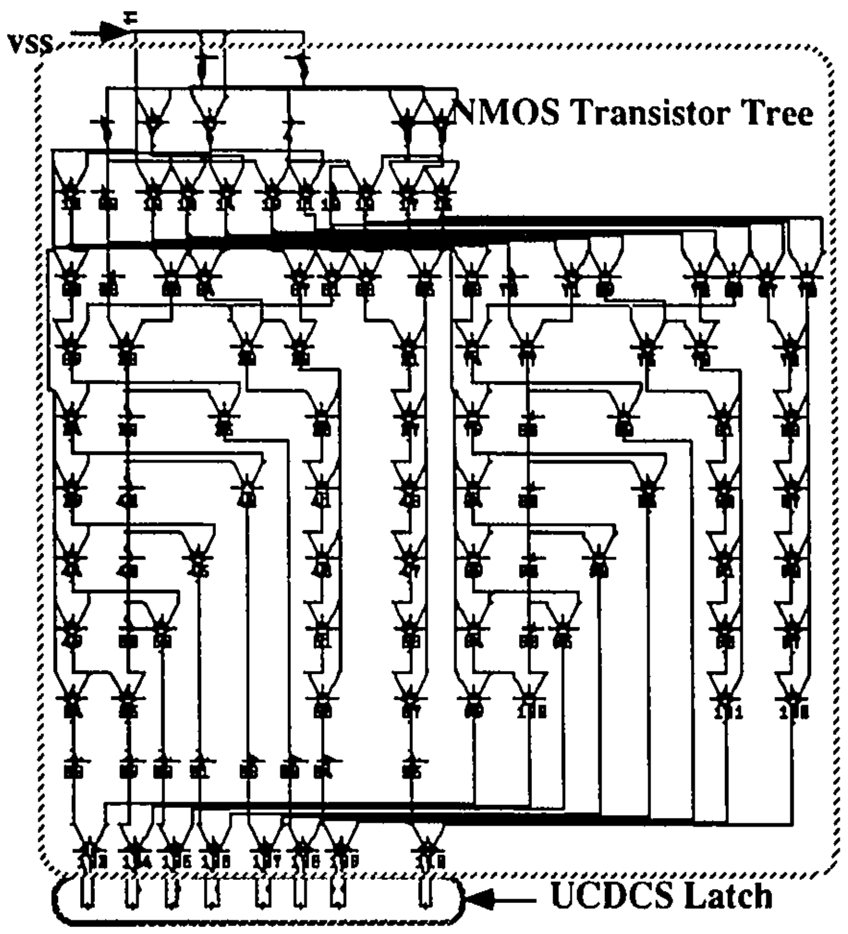
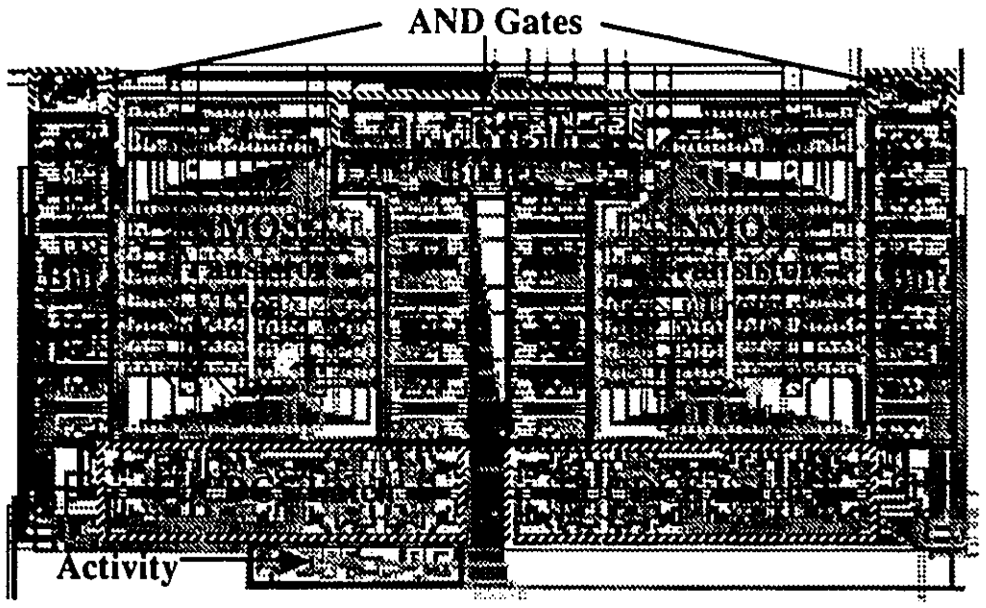


Figure 4.24 shows the 3-bit parallel subtractor that uses the UCDCS latch at the output of the NMOS transistor tree. Since the 3-bit parallel subtractor uses the UCDCS latch, it does not have the NMOS clock transistor connecting the transistor tree to the ground, vss. On the contrary, the NMOS transistor tree is connected to the power, vdd. The dimension of the NMOS transistors used in the transistor tree is 3μm x 0.8μm. Layout of the single block dynamic 3-input neuron design using the UCDCS latch is shown in Figure 4.25.

The single block dynamic neuron design using the TSPC latch has similar structure except that it uses the TSPC latch at the bottom. The dimensions of the 3-input neuron design using the UCDCS latch and using the TSPC latch are 424.5μm x 235.5μm and 428.6μm x 234.1μm, respectively.

Figure 4.25 Layout of the Single Block Neuron with UCDCS Latch



4.4 Discussion of Neuron Designs

Currently, there are two restrictions in the fabrication of the neuron design. One is the number of pads and the other is the core area. The BATMOS fabrication process provides only a 54 pads. 4 pads must be the power (vdd) pad and 4 pads must be the ground (vss) pad. Therefore, the number of pads available for the design is 46 pads. The pad used in the design has the dimension of *x*μm x 585.4μm. *x* is the x-dimension of the pads. e.g. the dimension of the vdd pad is 137μm x 585.4μm. The core dimension excluding the pads is 2344μm x 2172μm. Table 4.3 shows the approximate maximum number of neuron that can fit into the core. This is the maximum limitation of neuron count since the wiring is not included in the estimation.

Table 4.3. Capacity of Core in Fabrication

Design	max. of neuron (core area)
8-input logic gate neuron	1
4-input logic gate neuron	5
slow 8-input logic gate neuron	7
3-input dynamic neuron with TSPC latch	50
3-input dynamic neuron with UCDCS latch	50

Due to the restriction of the number of pads, the large number of input is not considered. For example, the 8-input neuron requires 24 input pads for input to the AND gates which feeds to the parallel counter, 8 input pads for the threshold, 1 input pad for the selection bit in the multiplexer, 1 input pad for the clock signal, 1 pad for the "beta regular" bit in the activity block and the 1 output pad for the output of the activity block. It needs 36 pads. It does not include the testing circuitry for output pads. Therefore, a design requiring a large number of pads is not practical to place into the core. The remaining designs are the 4-input logic gate neuron and the 3-input single block dynamic neurons. Since the 3-input single block dynamic neurons can have 10 times more neuron fitting into the core, it is much more reasonable to use them.

Table 4.4. Estimation of area per logic gate

Design	area/gate (μm^2)
8-input logic gate neuron	2930
4-input logic gate neuron	2107
slow 8-input logic gate neuron	~2479

From Table 4.4 the average of area per logic gate is about 2500μm^2. For the 3-input logic gate neuron after using ESPRESSO to minimize the number of logic gate is 601. The area for the 3-input logic gate neuron is 1.500mm^2. The area of the 3-input dynamic neuron is about 0.1mm^2 so the logic gate neuron is 15 times bigger than the dynamic neuron.

4.5 Simulation of Neuron Designs

The 3-input single block dynamic neurons are the final choice that will be put onto the test chip. Simulation of the functionality is done using the extracted layout of the 3-bit parallel subtractor and the extracted layout of the custom logic gates. Figure 4.26 shows the output of the SPICE simulation of the 3-bit parallel subtractor using a TSPC latch at 50MHz clock. The 3-bit parallel subtractor provides the complement output signals. 5 volt output represents "0" and 0 volt output represents "1".

Figure 4.26 SPICE Simulation of the TSPC Latch Dynamic Neuron

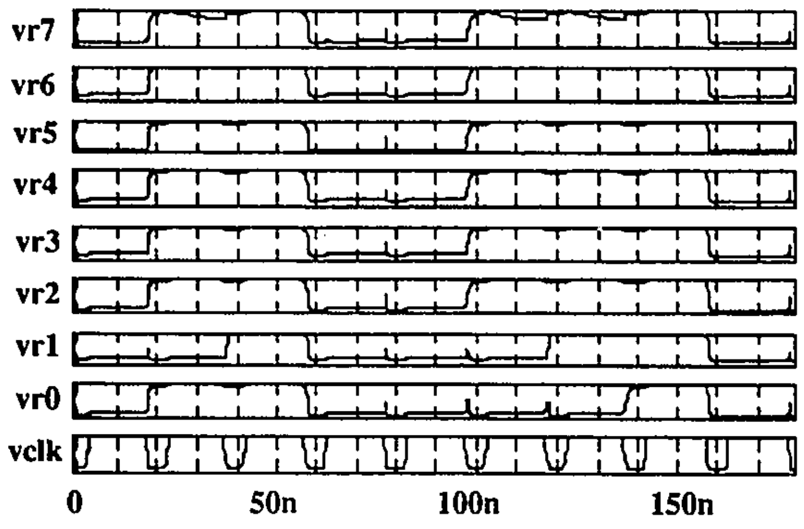


Figure 4.27 SPICE Simulation of the UCDCS Latch Dynamic Neuron

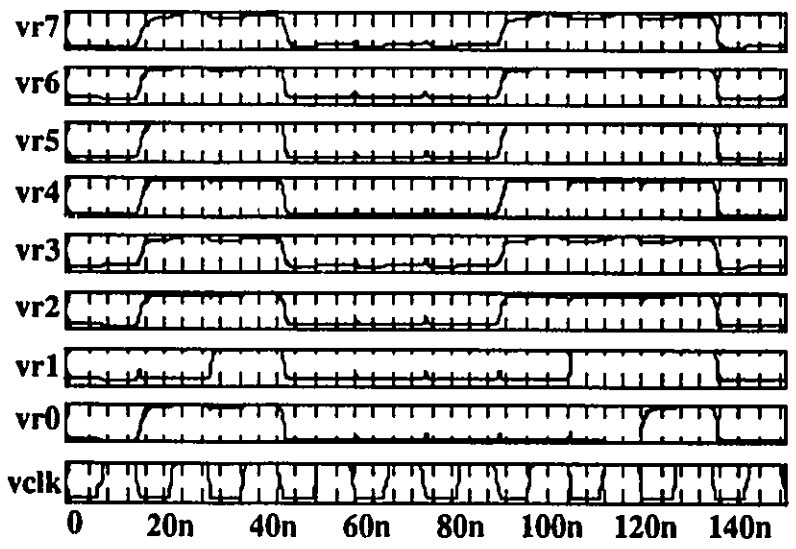
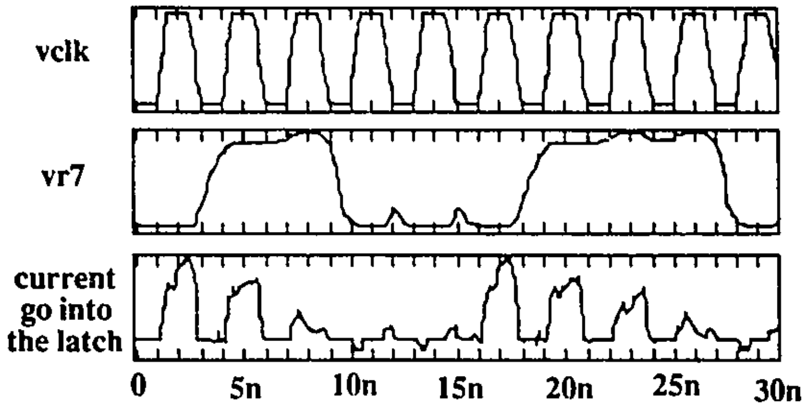


Figure 4.27 is the simulation result of the 3-input single block neuron using the UCDCS latch. The 66MHz clock is used in the simulation.

4.6 Additional Information on the Neuron Design

The reason to have a low clock speed in the simulation is that the current is fed into eight transistor trees. Without merging too many transistor trees, the operational clocking speed can be increased up to 5 times. This information is provided from the single block 3-input dynamic neuron design using the UCDCS latch.

Figure 4.28 Simulation of Single UCDCS Latch in 3-bit Parallel Subtractor

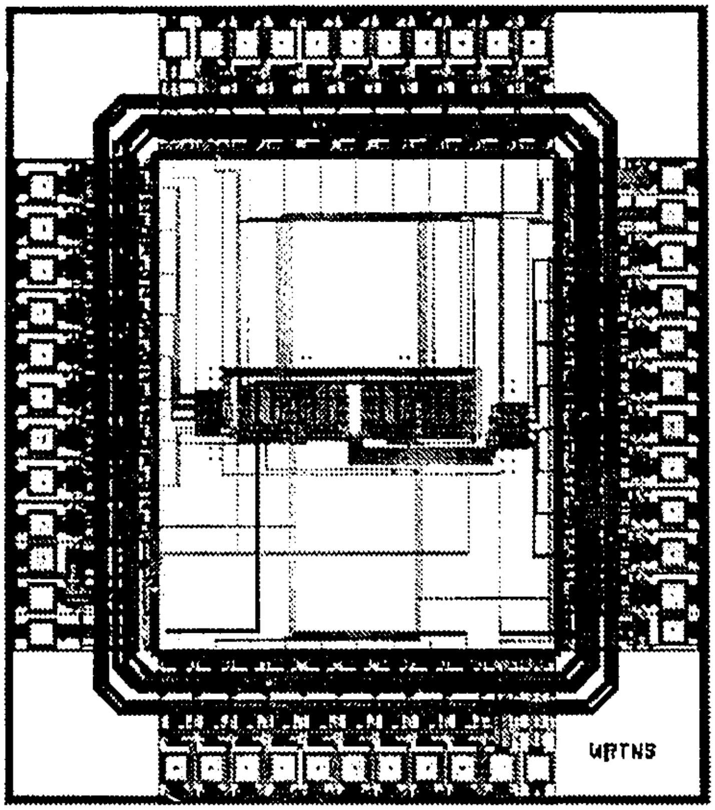


The SPICE simulation result in Figure 4.28 isolates the sign bit of the 3-bit parallel counter into the neuron and the input vectors use to test is the same as the 3-input single block dynamic neuron used in the UCDCS latch. The clock used in the simulation is 333MHz.

4.7 Test Cell for Fabrication

The test cell which has been sent for fabrication is shown in Figure 4.29. The test cell includes two 3-input single block dynamic neuron designs. One uses the TSPC latch and the other one uses the UCDCS latch.

Figure 4.29 The Mask Layout of the Test Chip



Pads included in the test chip are shown in Table 4.5

Table 4.5. Functionality of Pad

Functionality of Pad	no. of pad	name of pad
threshold	7	input
beta regular?	1	input
activity	2	output
selection bit of multiplexer	1	input
input to AND gate	9	input
output of 3-bit parallel subtractor	16	output
choice	1	input
power and ground	8	vdd and vss

Since there is a limitation on the number of pads, the functionality of both neurons needs to be tested one by one. The choice pad, input the signal to the choice block to allow one of the 3-bit parallel subtractor's output to be sent to the output pads. The choice block is a multiplexer block which includes sixteen 2-to-1 multiplexer.

4.8 Summary

This chapter considers two approaches of neuron design: the pipeline neuron design and the single block neuron design. Three logic gate neuron designs have been tried. These are the 8-input pipeline neuron design, the 4-input pipeline neuron design and the 8-input slow pipeline neuron design. Four dynamic neuron designs are investigated. The functional block of the 8-input pipeline neuron and the 7-input pipeline neuron are finished in either schematic or layout. The other two dynamic neuron designs are the single block 3-input neuron design. One of them uses the TSPC latch and the other one uses the UCDCS latch. Due to the limitation of the dimension of the core and the number of pad, the 3-input single block neuron designs are chosen. SPICE simulation result shows that the TSPC latch's neuron can function properly in 50MHz and the UCDCS latch's neuron can work with 66MHz. Additional information is that if the transistor trees are split in the 3-bit parallel subtractor, the operational clock speed can be improved up to 333MHz in the example in Section 4.6 The test chip has been sent for fabrication. It includes two functional neurons.

Chapter5
Conclusion

5.1 Contributions of the Thesis

In this thesis, two single block dynamic neurons were designed and implemented. They are used to improve the speed of the Hierarchy for Pattern Extraction (HyPE) artificial neuron network's training algorithm. The difference of the two neuron designs is that different latch designs are used. One design uses the true single phase clock (TSPC) latch and the other design uses the ultra fast dynamic current steering (UCDCS) latch. The designs are utilized in the hierarchical structure in the schematic and the layout. The NMOS transistor tree in the dynamic functional block is minimized by the minimization program provided in Section B.4 on page 95. Both neuron designs have been designed in the Cadence Edge™ environment using the 0.8 μ BATMOS technology.

The test chip has been fabricated. It includes both single block dynamic neuron designs.

Here is a summary of the main points of the thesis:

The background of the Hierarchy for Pattern Extraction (HyPE) artificial neuron network's architecture and the training algorithm have been reviewed. It shows the difference between the HyPE artificial neuron network and the other artificial neuron network.

The original algorithm was written in Smalltalk. The C program for the training is based on the algorithm that is extracted from it. It was a very rewarding experience learning object-oriented programming concepts in this process.

A review of the minimization algorithm's rules, based on the algorithm proposed by Bryant, has been presented. A switching tree minimization program written in the C programming language has been discussed as well. The structure and the functionality of the true single phase clocking (TSPC) latch and the ultra fast dynamic current steering (UCDCS) latch have been provided.

The pipeline neuron design and the single block neuron design have been presented. 8-input pipeline neuron, 4-input pipeline neuron and the slow 8-input pipeline neuron, using tcell library provided in the 0.8m BATMOS technology, have been presented. The 8-input pipeline dynamic neuron and the 7-input pipeline dynamic neuron have been introduced. Two single block neuron designs with 3-input have been introduced. Each of them uses a different kind of latch (the TSPC latch and the UCDCS latch).

The implementation of a test chip containing those two 3-input single block neuron design has been discussed. This can be used to test the functionality of the neuron design and the actual speed that the neuron design can provide in silicon. All the schematic and the layout designs are worked in the Cadence Edge^{TM} environment.

5.2 Future Work

In this architecture, there are a few things that need improvement. Reducing the number of levels will reduce the complexity of the artificial neuron network. Simplifying the regular beta neuron by removing the additional neuron's connectivity from the alpha virgin neuron to the beta regular neuron will not only simplify the training algorithm, it will also reduce the general neuron design to half in silicon.

The training algorithm will have to be modified in order to recognize more one group of patterns inside a single neural network.

Finally, a module generator [17] which will automatically generate the layout from logic specifications is required for designs with a large number of input variables. This module generator will have to take account node locality concerns.

REFERENCES

[1] L. Andrew Coward, "Pattern Thinking", Praeger Publishers, 1990.

[2] John Hertz, Anders Krogh, Richard G. Palmer, "Introduction to The Theory of Neural Computation", Addison-Wesley Publishing Company, 1991.

[3] H. Chan, H.M. Chan, G.A. Jullien, W.C. Miller, "An Artificial Neural Network VLSI Realization", Canadian Conference of Electrical and Computer Engineering, 1992.

[4] Neural Network Research Group, "Design and VLSI Implementation of An Intelligent Optical Sensor", Department of Electrical Engineering, University of Windsor, May, 1992.

[5] Ka-Wa Lei, "A 1.2μ Neural Network Design", M.A.Sc. Thesis, Faculty of Graduate Studies and REsearch, University of Windsor, 1994.

[6] Waleed Fakhr, M. Kamel and M.I. Elmasry, "Probability of Error, Mutual Information, and Size Minimization of Neural Networks", Proceedings of the International Joint Conference on Neural Networks (IJCNN'92), Baltimore, MD, USA, June 7-11, 1992.

[7] Waleed Fakhr and M.I. Elmasry, "Mutual Information Training and Size Minimization of Adaptive Probabilistic Neural Networks", International Symposium of Circuits and Systems (ISCAS'92), San Diego, 1992.

[8] Waleed Fakhr and M.I. Elmasry, "Minimum Description Length Pruning and Maximum Mutual Information Training of Adaptive Probabilistic Neural Networks", Proceedings of IEEE International Conference on Neural Network (ICNN'93), San Francisco, CA, USA, March 28 - April 1, 1993.

[9] Donald F. Specht, "Probabilistic Neural Networks and the Polynomial Adaline as Complementary Techniques for Classification", IEEE Transactions on Neural Networks, Vol. 1, no. 1, March 1990.

[10] Roger Grondin, "The synthesis and Modeling of High Speed Digital Switching Trees", M.A. Sc. Thesis, Faculty of Graduate Studies and Research, University of Windsor, 1991.

[11] Lino Del Pup, "The Development and Application of High-Speed Digital Switching Trees for Regular Arithmetic Arrays", M.A.Sc. Thesis, Faculty of Graduate Studies and Research, University of Windsor, 1991.

[12] James Christopher Czilli, "BiCMOS Technology and Some Applications in High Performance Arithmetic Structures", M.A.Sc. Thesis, Faculty of Graduate Studies and Research, University of Windsor, 1994.

[13] N. Weste, K. Eshraghian, "Principles of CMOS VLSI Design: A Systems Perspective", Addison-Wesley, 1985.

[14] R. E. Bryant, "Graph-based algorithms for Boolean function Manipulation", IEEE Transaction on Computer, vol. 35, pp.677-691, 1986.

[15] Yuan Ji-Ren, Ingemar Karlsson, and Christer Svensson "A True Single-Phase-Clock Dynamic CMOS Circuit Technique", IEEE Journal of Solid-State Circuit, Vol 22, No. 5, pp.899-901, October, 1987.

[16] M. Yasunaga *et al.*, "A Self-Learning Digital Neural Network Using Wafer-Scale LSI", IEEE Journal of Solid-State Circuits Vol 28, No. 2, pp.106-114, February 1993.

[17] Siddiq, Shakil Kaiser, "Module Generators from Topological Descriptions and Graph Theoretic Approach", M.A.Sc. Thesis, Faculty of Graduate Studies and Research, University of Windsor, 1994.

[18] "HSPICE User's Manual, Elements and Models", Volume 2, HSPICE Version H92, Meta-Software, Inc., 1992.

[19] H.M. Chan, "Dynamic Logic Synthesis with Application to Self-Timed Pipelines", M.A.Sc. Thesis, Faculty of Graduate Studies and Research, University of Windsor, 1992.

[20] R. Venkatesan, "FPGA Implementation of RMS Structures", M.A.Sc. Thesis, Faculty of Graduate Studies and Research, University of Windsor, 1994.

[21] Alagu Annaamalai, "Modeling and Synthesis of Special Parallel Architectures Using VHDL", M.A.Sc. Thesis, Faculty of Graduate Studies and Research, University of Windsor, 1991.

Appendix A

The Training Algorithm of HyPE

This Appendix consists of the training algorithm of the HyPE architecture and the equations that uses in the algorithm.

A.1 Equations

$$NAC = PF\frac{\sum_B \mathbf{R}_{jt}}{k1}\left(k2 + k3\frac{\sum_B \mathbf{R}_{jt} - \beta_{initial}}{\beta_{initial}}\right)$$

$$\mathrm{NV}_j = 15 \times \frac{\mathrm{NV}_j + \mathrm{SR}_j - \mathrm{S}_j}{10}$$

A.2 HyPE Algorithm

A.2.1 Overall Training Process

```
Initialize Brain Process
loop $l \leftarrow 1, 3$
{
    $NAM \leftarrow 1$
    $NA \leftarrow [l = 1]$
    loop $s \leftarrow 1, LP_l$
    {
        Wake Process
        if $(l = 1)$    $\beta_{initial} \leftarrow \sum_B [R_{i,2,t}] ; i \in S_{2,t}$
        Sleep Process
    }
}
```

A.2.2 Initialize Brain Process

```
  N[$T_{i,j,0}, R_{i,j,0}, \mathbf{C}_{i,j,0}, G_{i,j,0}$]; $1 \le j \le 4; i \in S_{j,0}$
  $R_{i,j,0} \leftarrow 0; 1 \le j \le 4; i \in S_{j,0}$
  $G_{i,j,0} \leftarrow 0; 1 \le j \le 4; i \in S_{j,0}$
  $\mathbf{C}_{i,j,0} \leftarrow \Psi_{n \in S_{j-1,0}} [0]; 1 \le j \le 4; i \in S_{j,0}$
$NA \leftarrow 1$
  $T_{i,j,0} \leftarrow \mathbf{T}[NA, j]; 1 \le j \le 3; i \in S_{j,0}$
$T_{1,4,0} \leftarrow 1$
loop $j \leftarrow 1, 3$
{
    loop $s \leftarrow 1, m_{ij0}$
    {
        $C_{\Re[1, N_{j-1}], i, j, 0} \leftarrow 1; i \in S_{j,t}$
    }
}
$IPF \leftarrow 1.0$
```

A.2.3 Wake Process

```
[$NAM = 1$]
{
    $NAM \leftarrow 0$
    loop $t \leftarrow 1, TP_t$
    {
        $\mathbf{F}_{0t} \Leftarrow \mathbf{P}_t$
         $\mathbf{E}_{0,t} \Leftarrow \Psi_{n \in S_{0t}} [1]$
        loop $j \leftarrow 1, 3$
        {
            loop $i \leftarrow 1, S_{jt}$
            }
                $G_{ijt} \leftarrow \left( \sum_B [C_{n,i,j,t-1} \wedge F_{n,j-1,t}] \ge T_{ijt} \right); n \in S_{j-1,t}$
                if $(j = 2)$
                {
                    $G_{ijt} \leftarrow \left( \left( 2 \sum_B [C_{n,j,t-1} \wedge F_{n,j-1,t} \wedge E_{n,j-1,t}] \ge T_{ijt} \right) \wedge R_{i,j,t-1} \wedge G_{ijt} \right); n \in S_{j-1,t}$
                }
                $G_{ijt} \leftarrow \left( \left( \sum_B [R_{i,j,t-1} \wedge G_{ijt}] \right) \le P_{jt} \right)$
            }
            $\mathbf{F}_{j,t} \Leftarrow \Psi_{i \in S_{jt}} [G_{ijt}]$
        }
        loop $s \leftarrow 1, 3$
        {
```

```
      if $\left(\left(\sum_B [F_{n,3,t}] < M\right) \wedge \left(\left(\sum_B [F_{n,2,t} \wedge R_{i,2,t-1}] < NAC\right) \vee [l = 1]\right)\right)$
      {
        $NA \leftarrow NA + 1$
        $\mathbf{F}_{0,t} \Leftarrow \mathbf{P}_t$
        $\mathbf{E}_{0,t} \Leftarrow \underset{i \in S_{0,t}}{\Psi} [1]$
        $T_{i,j,t} \leftarrow \mathrm{T}[NA, j]; (1 \le j \le 3); i \in S_{V_{j,t}}$
        loop $j \leftarrow 1, 3$
        {
          loop $i \leftarrow 1, S_{jt}$
          {
              $G_{ijt} \leftarrow \left(\sum_B [C_{n,i,j,t-1} \wedge F_{n,j-1,t}] \ge T_{ijt}\right); n \in S_{j-1,t}$
            if $(j = 2)$
            {
              $G_{ijt} \leftarrow \left(\left(2\sum_B [C_{n,j,t-1} \wedge F_{n,j-1,t} \wedge E_{n,j-1,t}] \ge T_{ijt}\right) \wedge R_{i,j,t-1} \wedge G_{ijt}\right); n \in S_{j-1,t}$
            }

            $G_{ijt} \leftarrow \left(\left(\sum_B [R_{i,j,t-1} \wedge G_{ijt}]\right) \le P_{jt}\right)$
          }
          $\mathbf{F}_{j,t} \Leftarrow \underset{i \in S_{jt}}{\Psi} [G_{ijt}]$
        }
      }
    }
    $\mathbf{C}_{ijt} \Leftarrow \underset{i \in S_{jt}}{\Psi} [C_{n,i,j,t-1} \wedge \left((R_{i,j,t-1} \vee F_{n,j-1,t} \vee E_{n,j-1,t}) \wedge \bar{G}_{ijt}\right) \vee ((R_{i,j,t-1} \vee F_{n,j-1,t}) \wedge G_{ijt})]$
    ..... $; (1 \le j \le 3); i \in S_{jt}$
    $R_{ijt} \leftarrow (R_{i,j,t-1} \vee G_{ijt}); (1 \le j \le 3); i \in S_{jt}$
     $\mathbf{FP}_{jt} \Leftarrow \underset{i \in S_{jt}}{\mathbf{VI}} [\mathbf{FP}_{i,j,t-1} + (R_{i,j,t-1} \wedge G_{ijt})]; 1 \le j \le 3$
      $\mathbf{IR}_{jt} \Leftarrow \underset{i \in S_{j-1,t}}{\mathbf{VI}} \left[\mathbf{IR}_{i,j,t-1} + \sum_B [G_{ijt} \wedge R_{i,j,t-1} \wedge F_{n,j-1,t} \wedge C_{nijt} \wedge R_{i,j-1,t-1}]\right]; 1 \le j \le 3; i \in S_{jt}$
    $\mathbf{C}_{1,4,t} \Leftarrow \underset{n \in S_{R_{3,t}}}{\Psi} [(C_{n,1,4,t-1} \wedge R_{n,3,t-1}) \vee \left(F_{n,3,t-1} \wedge R_{n,3,t-1}\right)]$
     $G_{1,4,t} \leftarrow \left(\sum_B [C_{n,1,4,t} \wedge F_{n,3,t}] > 1\right); n \in S_{R_{3,t}}$
    $PP = PP \leftarrow \left(G_{1,4,t} \wedge \overline{TNT}\right)$
    $\mathbf{C}_{1,4,t} \Leftarrow \underset{n \in S_{R_{3,t}}}{\Psi} [C_{n,1,4,t} \wedge (\bar{P}P \vee F_{n,3,t})$
    if $(PP = 1)$  $PF \leftarrow PF \times k$
  }
}
```

A.2.4 Sleep Process

```
loop $j \leftarrow 1, 3$
```

```
{
    $\mathbf{C}_{ijt} \Leftarrow \underset{i \in S_{V_{jt}}}{\Psi} [0]$
    $T_{ijt} \leftarrow \mathbf{T}[0, j]; i \in S_{V_{jt}}$
    $G_{ijt} \leftarrow 0; i \in S_{jt}$
    $R_{ijt} \leftarrow 0; i \in S_{V_{jt}}$
    loop $s \leftarrow 1, \left( NV_j - \sum_B E_{ijt} \right)$
    {
        $\mathbf{N}[T_{ijt}, R_{ijt}, \mathbf{C}_{ijt}, G_{ijt}]$
        $G_{ijt} \leftarrow 0; i \in S_{jt}$
        $R_{ijt} \leftarrow 0; i \in S_{V_{jt}}$
        $\mathbf{C}_{ijt} \Leftarrow \underset{i \in S_{V_{jt}}}{\Psi} [0]$
        $T_{ijt} \leftarrow \mathbf{T}[0, j]; i \in S_{V_{jt}}$
    }
    $IH_{njt} \leftarrow \left[ \frac{k_1}{\sum_{s \in S_{j-1,t}} IH_{s,j,t-1}} \times IH_{n,j,t-1} + \frac{k_1}{\sum_{s \in S_{j-1,t}} IR_{sjt}} \times IR_{njt} \right]; n \in S_{R_{j-1,t}}$
    $\mathbf{IH}_{jt} \Leftarrow \eta(\mathbf{IH}_{jt})$
    if $(j = 1)$    $IP_{njt} \leftarrow \left[ \sum_{s=1}^{n} IH_{sjt} + \frac{kk_2}{\sum_B E_{i,j-1,t}} \right]; n \in S_{R_{j-1,t}}$
    if $(j > 1)$
    {
        $IP_{njt} \leftarrow \left[ \sum_{s=1}^{n} IH_{sjt} + \frac{kk_3}{\sum_B E_{i,j-1,t}} \right]; n \in S_{R_{j-1,t}}$
        $IP_{njt} \leftarrow \left[ \sum_{s=1}^{n} IH_{sjt} + \frac{kk_4}{\sum_B E_{i,j-1,t}} \right]; n \in S_{V_{j-1,t}}$
    }
    loop $n \leftarrow 1, N_j$
    {
        if $\left( IP_{n-1,j,t} < \Re[1, \max(\mathbf{IP}_{jt})] \le IP_{n,j,t} \right)$ $C_{nijt} \leftarrow 1; 1 \le n \le m_{ijt}; i \in S_{V_{jt}}$
    }
    if $(j = 2)$
    {
        if $\left( C_{nijt} \wedge \bar{E}_{n,j-1,t} = 1 \right)$        $C_{nijt} \leftarrow 0; i \in S_{V_{jt}}$
    }
    if $(j = 2)$
    {
        $LL_{n,1,t} \leftarrow \sum_{s=1}^{n} E_{s,1,t}; n \in S_{V_{1,t}}$
        $LPF_{i,2,t} \leftarrow \sum_{s=1}^{i} FP_{s,2,t}; i \in S_{R_{1,t}}$
        loop $ss \leftarrow 1, \left( AVE \times \sum_B E_{i,2,t} \right)$
        {
            $a \leftarrow \Re[1, \max(\mathrm{LPF}_{2,t})]$
            $b \leftarrow \Re[1, \max(\mathrm{LL}_{1,t})]$
            loop $n \leftarrow 1, N_{j-1}$
            {
                if $(b \le LL_{s,1,t})$
                {
                    $d \leftarrow n$
                    $n \leftarrow N_{j-1}$
                }
            }
            loop $s \leftarrow 1, N_j$
            {
                if $(a \le LPF_{s,2,t})$
                {
                    $C_{d,s,2,t} \leftarrow 1$
                    $s \leftarrow N_j$
                }
            }
        }
    }
    if $(j = 2)$
    {
        $a \leftarrow$ (as integer) $\left[ \frac{\sum_B (C_{nijt} \wedge E_{n,j-1,t})}{BTF} \right. \quad ; i \in S_{R_{2,t}}$
        if $(a > T_{ijt})$     $T_{ijt} \leftarrow a$
    }
}
```

Appendix B

Source Code of All Programs

B.1 C Code for generating input patterns

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <malloc.h>
#include <sys/timeb.h>
#include <sys/types.h>

main(argc, argv)
int argc;
char *argv[];
{

 FILE *fp,*fi;
 int a,b,c,d,e,lo,input[54],prob[54],ele[54],count;
 int m;
 int fn;
srand(time ((time_t *)0));
if (argc != 3)
{
  printf("the format of input is as below:\n");
  printf("e.g. thingmake 'input_file' 'output_file'\n");
  exit(1);
}
for(b=0;b<54;b++)
{
  input[b]=0;
  prob[b]=0;
  ele[b]=0;
}

/* fp=fopen("inp.pat","r");*/
 fp=fopen(argv[1],"r");
  for(a=0;a<54;a++)
    {
     fscanf(fp,"%d\n",&input[a]);
    }
 fclose(fp);
 prob[0]=input[0];
for(a=1;a<54;a++)
{
 prob[a]=input[a]+prob[a-1];
}
count=prob[53]/20;
printf("count=%d\n",count);
 fi=fopen(argv[2],"w");
for(a=0;a<500;a++)/*generate 500 input patterns*/
 {
  for(c=0;c<54;c++)
  {
   ele[c]=0;
  }

/*for(b=0;b<count-5;b++)*/
for(b=0;b<21;b++)/* select 21 from 54 characteristic components from */
{
  m=(int)(((((int)rand()))/2147483647.00)*prob[53]);
  for(c=0;c<54;c++)
  {
   if(m<=prob[0])
   {
    ele[0]=1;
   }
   if((prob[c-1]<m)&&(m<=prob[c]))
   {
    ele[c]=1;
    c=54;
   }
  }
}
for(lo=0;lo<54;lo++)
{
 fprintf(fi,"%d ",ele[lo]);
}
fprintf(fi,"\n");
}
for (a=0; a<54; a++)
fprintf(fi,"%d ",input[a]);
fprintf(fi,"\n");
fclose(fi);
}
```

B.2 Training Program

```
#include <math.h>
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <time.h>
#include <sys/timeb.h>
#include <sys/types.h>
#define MAX 2000
```

```
struct NEURON
   {
     unsigned int T;
     unsigned int C[MAX];
     unsigned R:1;
     unsigned G:1;
   } N[4][MAX],BasN;

struct LEVEL
{
 unsigned int F[MAX];
 unsigned int E[MAX];
 unsigned int FP[MAX];
 unsigned int IR[MAX];
 unsigned int IH[MAX];
 unsigned int IRM;
 unsigned int IHM;
 unsigned int IP[MAX];
 unsigned int LL[MAX];
 unsigned int LPF[MAX];
 unsigned int S;
 unsigned int SR;
 unsigned int NV;
 unsigned int P[5];
 unsigned int NT[4];
 unsigned int m[2];
 }Level[4];

struct PROCESS
{
 unsigned int LP;
 unsigned int TP;
}Train[4];
int NA,NAM;
float PF;
int beta=0;

main(argc, argv)
int argc;
char *argv[];
{
FILE *fp;
int t=0,lb=0,tt=0,m=0,t1=0,TP=0,lt=0,lo=0,lp;
int count=0,dummy=0,*y,PP,TNT=1,tp=1,level;
int THforBETA=0;
int Gamma_Firing=0,l=0,lk=0,TT=0;
char *alp[10],*bet[10],*gam[10],*bas[10];
alp[0]="Alp0.rep"; bet[0]="Bet0.rep"; gam[0]="Gam0.rep"; bas[0]="Bas0.rep";
alp[1]="Alp1.rep"; bet[1]="Bet1.rep"; gam[1]="Gam1.rep"; bas[1]="Bas1.rep";
alp[2]="Alp2.rep"; bet[2]="Bet2.rep"; gam[2]="Gam2.rep"; bas[2]="Bas2.rep";
alp[3]="Alp3.rep"; bet[3]="Bet3.rep"; gam[3]="Gam3.rep"; bas[3]="Bas3.rep";
alp[4]="Alp4.rep"; bet[4]="Bet4.rep"; gam[4]="Gam4.rep"; bas[4]="Bas4.rep";
alp[5]="Alp5.rep"; bet[5]="Bet5.rep"; gam[5]="Gam5.rep"; bas[5]="Bas5.rep";
```

```
alp[6]="Alp6.rep"; bet[6]="Bet6.rep"; gam[6]="Gam6.rep"; bas[6]="Bas6.rep";
alp[7]="Alp7.rep"; bet[7]="Bet7.rep"; gam[7]="Gam7.rep"; bas[7]="Bas7.rep";
alp[8]="Alp8.rep"; bet[8]="Bet8.rep"; gam[8]="Gam8.rep"; bas[8]="Bas8.rep";
alp[9]="Alp9.rep"; bet[9]="Bet9.rep"; gam[9]="Gam9.rep"; bas[9]="Bas9.rep";

if (argc!=3)
{
 printf("You are missing some parameters:\n");
 printf("e.g. backup 'input_file1' 'input_file2'\n");
 exit(1);
}
for (lb=0;lb<10;lb++)
{
/****************INITIALIZE BRAIN*************************/
 fp=fopen("Train.rep","a");
 fprintf(fp,"%d\n",time ((time_t *)0));
 fclose(fp);
 init_brain (); /* initial brain */
 PF=1.0;
/*****************END INITIALISE BRAIN********************/
 for (tp=1;tp<4;tp++)
 {
  NAM=1;
  lp=0;
  while (lp<Train[tp].LP&&(NAM==1))
  {
   NAM=0;
   fp=fopen("Train.rep","a");
   fprintf(fp,"Training Process(%d) at loop(%d)\n",tp,lp);
   fprintf(fp,"%d\n",time ((time_t *)0));
   fclose(fp);
   if (tp==1||tp==2) Wake(tp,argv[1]);
   if (tp==3||tp==4) Wake(tp,argv[2]);
   if (tp==1) beta=Level[2].SR;
   Sleep();
   lp++;
  }
 }
 fp=fopen(alp[lb],"w");
 for (t=0;t<Level[1].S;t++)
 {
  count=0;
  fprintf(fp,"%3d %d %2d ",t,N[1][t].R,N[1][t].T);
  for(tt=0;tt<Level[0].S;tt++)
  {
   if (N[1][t].C[tt]==1) fprintf(fp,"%d ",tt);
   if (N[1][t].C[tt]==1&&Level[0].E[tt]==1) count++;
  }
  fprintf(fp," %d\n",count++);
 }
 fclose(fp);
 fp=fopen(bet[lb],"w");
 for (t=0;t<Level[2].S;t++)
 {
  count=0;
  fprintf(fp,"%3d %d %2d ",t,N[2][t].R,N[2][t].T);
  for(tt=0;tt<Level[1].S;tt++)
  {
   if (N[2][t].C[tt]==1) fprintf(fp,"%d ",tt);
   if (N[2][t].C[tt]==1&&Level[1].E[tt]==1) count++;
  }
  fprintf(fp," %d\n",count++);
 }
 fclose(fp);
 fp=fopen(gam[lb],"w");
 for (t=0;t<Level[3].S;t++)
 {
  count=0;
  fprintf(fp,"%3d %d %2d ",t,N[3][t].R,N[3][t].T);
  for(tt=0;tt<Level[2].S;tt++)
  {
   if (N[3][t].C[tt]==1) fprintf(fp,"%d ",tt);
   if (N[3][t].C[tt]==1&&Level[2].E[tt]==1) count++;
  }
  fprintf(fp," %d\n",count++);
 }
 fclose(fp);
 fp=fopen(bas[lb],"w");
 for (lo=0;lo<Level[3].S;lo++)
 {
  fprintf(fp,"%d ",BasN.C[lo]);
 }
 fprintf(fp,"\n");
 fclose(fp);
 for (level=1;level<4;level++)
  for (t=0;t<Level[level].S;t++)
  {
   N[level][t].T=50;
   N[level][t].R=0;
   for (tt=0;tt<Level[level-1].S;tt++)
     N[level][t].C[tt]=0;
  }
}
}
/**** END OF MAIN ****/

/***********************BEGIN WAKE***********************/
Wake(tp,argv)
int tp;
char *argv[];
{
FILE *fp,*f1;
int TT,lk,tt,*y,Gamma_Firing=0,Beta_Firing,lo,t,dummy=0,count=0,PP=0,TNT=1;
float NAC;
 y=(int *)calloc(55,sizeof(int));
 f1=fopen(argv,"r");
 for(TT=0;TT<Train[tp].TP;TT++)
 {
  NA=0;
  for(tt=0;tt<54;tt++)
   {
    fscanf(f1,"%d\n",&y[tt]);
    Level[0].F[tt]=y[tt];
   }
/*for (tt=0;tt<54;tt++) printf("%d",Level[0].F[tt]);
printf("\n");*/
  Gamma_Firing=0;
  if (tp==1) NA=1;
/**********Novelty Arousal Section***************/
  NAC=PF*(Level[2].SR/300.0)*(.25+1.25*(Level[2].SR-beta)/(float)(beta));
  while((Gamma_Firing<5)&&(NA<=3))
  {
   init_VNs_T_A(NA);
   check_activity(tp); /*check neuron activity*/
   Gamma_Firing=0;
   for(lo=0;lo<Level[3].S;lo++)
     Gamma_Firing=Gamma_Firing+N[3][lo].G;
   Beta_Firing=0;
   for(lo=0;lo<Level[2].S;lo++)
     Beta_Firing=Beta_Firing+(N[2][lo].G&&N[2][lo].R);
   NA++;
   if ((Beta_Firing<NAC)&&(tp>1)) break;
  }
  if ((NA!=1)&&(tp!=1)) NAM=1;
  update_connectivity();

/******* Update Basal Node Connectivity & Activity *******/
  BasN.G=0;
  for(t=0;t<Level[3].S;t++)
   BasN.C[t]=((BasN.C[t]&&N[3][t].R)||(N[3][t].G&&(!N[3][t].R)));
  dummy=0;
  for(t=0;t<Level[3].S;t++)
   dummy=dummy+((BasN.C[t])&&Level[3].F[t]);
  if (dummy>0) BasN.G=1;
  update_threshold_regular();/* Update Threshold & regular */

/***************the Pain and Pleasure***********************/
  if ((TT-(TT/3*3)==0)||tp==1||tp==2) TNT=1;
  else TNT=0;
  PP=(BasN.G&&(!TNT));
  for(t=0;t<Level[3].S;t++)
   BasN.C[t]=((BasN.C[t])&&((!PP)||(!Level[3].F[t])));
  if (PP==1) PF=PF*21/20.0;
fp=fopen("Train.rep","a");
  fprintf(fp,"Training Pattern(%2d) NA=%d TNT=%d NAM=%d PP=%d PF=%5.3f NAC=%7.3f Basal
Node Firing Static(%2d:%2d)->%d \n",TT,NA-1,TNT,NAM,PP,PF,NAC,Gamma_Firing,dummy,BasN.G);
fclose(fp);
  update_FP_IR();        /******* Update FP & IR *******/
 }
 fclose(f1);
/************Record to the rep file*****************/
/* fp=fopen("Alp.rep","w");
```

```
 for(tt=0;tt<Level[0].S;tt++)
   fprintf(fp,"tt=%3d IR=%3d IH=%3d IP=%3d\n",tt,Level[1].IR[tt],Level[1].IH[tt],Level[1].IP[tt]);
 fclose(fp);
 fp=fopen("Bet.rep","w");
 for(tt=0;tt<Level[1].S;tt++)
   fprintf(fp,"tt=%3d IR=%3d IH=%3d IP=%3d\n",tt,Level[2].IR[tt],Level[2].IH[tt],Level[2].IP[tt]);
 fclose(fp);
 fp=fopen("Gam.rep","w");
 for(tt=0;tt<Level[2].S;tt++)
   fprintf(fp,"tt=%3d IR=%3d IH=%3d IP=%3d\n",tt,Level[3].IR[tt],Level[3].IH[tt],Level[3].IP[tt]);
 fclose(fp);*/
}
/****************END WAKE***************/

/*****************initial brain function*****************/
init_brain ()
{
int level,t,tt,m,t1;
 srand(time ((time_t *)0));
Level[1].NT[0]=50; Level[1].NT[1]=7;  Level[1].NT[2]=6;  Level[1].NT[3]=5;
Level[2].NT[0]=50; Level[2].NT[1]=7;  Level[2].NT[2]=6;  Level[2].NT[3]=5;
Level[3].NT[0]=50; Level[3].NT[1]=6;  Level[3].NT[2]=6;  Level[3].NT[3]=6;
Level[1].P[1]=MAX; Level[1].P[2]=30;  Level[1].P[3]=30;  Level[1].P[4]=30;
Level[2].P[1]=50;  Level[2].P[2]=30;  Level[2].P[3]=30;  Level[2].P[4]=30;
Level[3].P[1]=4;   Level[3].P[2]=10;  Level[3].P[3]=10;  Level[3].P[4]=10;
Level[0].S=54;    Level[1].S=150;   Level[2].S=150;   Level[3].S=150;
Level[1].NV=150;   Level[2].NV=150;   Level[3].NV=150;
Level[1].SR=0;    Level[2].SR=0;    Level[3].SR=0;
Level[0].IRM=0;   Level[1].IRM=0;   Level[2].IRM=0;   Level[3].IRM=0;
Level[0].IHM=0;   Level[1].IHM=0;   Level[2].IHM=0;   Level[3].IHM=0;
Level[1].m[0]=15; Level[1].m[1]=20;
Level[2].m[0]=17; Level[2].m[1]=24;
Level[3].m[0]=14; Level[3].m[1]=26;
Train[1].LP=1;    Train[2].LP=20;   Train[3].LP=20;   Train[4].LP=1;
Train[1].TP=4;    Train[2].TP=40;   Train[3].TP=30;   Train[4].TP=30;
 for (tt=0;tt<54;tt++) Level[0].E[tt]=1;
 for (level=1;level<4;level++)
  {
   for(t=0;t<Level[level].S;t++)
   {
    N[level][t].T=Level[level].NT[1];
    N[level][t].R=0;
    N[level][t].G=0;
    Level[level].FP[t]=0;
    Level[level].IR[t]=0;
    Level[level].E[t]=0;
    for(tt=0;tt<Level[level-1].S;tt++)
    N[level][t].C[tt]=0;
    for(t1=0;t1<Level[level].m[0];t1++)
    {
     m=(int)((((int)rand()/2147483647.00)*(Level[level-1].S-1));
     N[level][t].C[m]=1;
    }
   }
  }
 }
/****************end of initial level function**************/

/****initial virgin neurons' T & reset the activity*********/
init_VNs_T_A(lk)
int lk;
{
int level,lo;
 for (level=1;level<4;level++)
   for (lo=0;lo<Level[level].S;lo++)
   {
    if (N[level][lo].R==0) N[level][lo].T=Level[level].NT[lk];
    N[level][lo].G=0;
    Level[level].F[lo]=0;
   }
}
/***end of initial virgin neurons' T & reset the activity***/

/****************check to activity of neuron in level*******/
check_activity(tp)
int tp;
{
int level,lt,l,t,count,lo,andy, THforBETA;
/*printf("\nR(active regular), r(non-active regular), N(active virgin), n(non-active virgin)");*/
for (level=1;level<4;level++)
{
 lt=0;
 for(lo=0;lo<Level[level].S;lo++)
 {
   count=0;
   THforBETA=0;
   for(l=0;l<Level[level-1].S;l++)
   {
        if (((N[level][lo].C[l])&&(Level[level-1].F[l]))==1) count++;
        if (level==2)
        {
          if ((N[level][lo].C[l])&&(Level[level-1].F[l])&&(Level[level-1].E[l])==1)
           THforBETA=THforBETA+2;
        }
   }
   if (level==2)
   {
    if (((count>=N[level][lo].T)&&(((THforBETA>=N[level][lo].T)
          &&(N[level][lo].R))||(!N[level][lo].R)))==1)
         N[level][lo].G=1;
   }
   else
    if (count>=N[level][lo].T) N[level][lo].G=1;
   if (((!(N[level][lo].R))&&(N[level][lo].G))==1) lt++;
   if ((lt>Level[level].P[tp])&&(!(N[level][lo].R))) N[level][lo].G=0;
 }
 andy=0;
 for(lo=0;lo<Level[level].S;lo++)
 {
   Level[level].F[lo]=N[level][lo].G;
   if (N[level][lo].G==1) andy++;
 }
/* count=0;
 printf("Neuron in level (%d).\n",level);
 for (lo=0;lo<(Level[level].S/150);lo++)
 {
  for (l=count;l<count+150;l++)
  {
   if ((N[level][l].R&&N[level][l].G)==1) printf("R");
   if ((N[level][l].R&&(!N[level][l].G))==1) printf("r");
   if (((!N[level][l].R)&&N[level][l].G)==1) printf("N");
   if (((!N[level][l].R)&&(!N[level][l].G))==1) printf("n");
  }
  printf("\n");
  count=count+150;
 }
 for (l=count;l<Level[level].S;l++)
 {
  if ((N[level][l].R&&N[level][l].G)==1) printf("R");
  if ((N[level][l].R&&(!N[level][l].G))==1) printf("r");
  if (((!N[level][l].R)&&N[level][l].G)==1) printf("N");
  if (((!N[level][l].R)&&(!N[level][l].G))==1) printf("n");
 }
  printf("\n");

 count=0;
 printf("\nNeuron threshold in level (%d).\n",level);
 for (lo=0;lo<(Level[level].S/50);lo++)
 {
  for (l=count;l<count+50;l++)
  {
printf("%2d ",N[level][l].T);
  }
  printf("\n");
  count=count+50;
 }
 for (l=count;l<Level[level].S;l++)
 {
  printf("%2d ",N[level][l].T);
 }
 printf("\n");*/

}
/* count=0;
 printf("Neuron in level (3).\n");
 for (lo=0;lo<(Level[3].S/100);lo++)
 {
  for (l=count;l<count+100;l++)
  {
   if ((N[3][l].R&&N[3][l].G)==1) printf("R");
   if ((N[3][l].R&&(!N[3][l].G))==1) printf("r");
   if (((!N[3][l].R)&&N[3][l].G)==1) printf("N");
```

```
    if (((!N[3][l].R)&&(!N[3][l].G))==1) printf("n");
   }
   printf("\n");
   count=count+100;
  }
 for (l=count;l<Level[3].S;l++)
 {
  if ((N[3][l].R&&N[3][l].G)==1) printf("R");
  if ((N[3][l].R&&(!N[3][l].G))==1) printf("r");
  if (((!N[3][l].R)&&N[3][l].G)==1) printf("N");
  if (((!N[3][l].R)&&(!N[3][l].G))==1) printf("n");
 }
  printf("\n");*/

}
/**********end of check to activity of neuron in level*******/

/*****************update connectivity***********************/
update_connectivity()
{
int level,t,tt,a,b,c,d,e;
 for(level=1;level<4;level++)
 {
  for(t=0;t<Level[level].S;t++)
  {
   for(tt=0;tt<Level[level-1].S;tt++)
   {
    a=((!(N[level][t].R))&&(!(N[level][t].G)));
    b=((Level[level-1].F[tt])&&(!(N[level][t].G)));
    c=((Level[level-1].E)&&(!(N[level][t].G)));
    d=((N[level][t].R)&&(N[level][t].G));
    e=((Level[level-1].F[tt])&&(N[level][t].G));
    N[level][t].C[tt]=N[level][t].C[tt]&&(a||b||c||d||e);
   }
  }
 }
}
/****************end of update connectivity****************/

/***********update Firing Population & Input record**********/
update_FP_IR()
{
int level,t,tt,dummy;
 for(level=1;level<4;level++)
 {
  for(tt=0;tt<Level[level-1].S;tt++)
  {
   Level[level].FP[tt]=Level[level].FP[tt]+((N[level][tt].R)
     &&(N[level][tt].G));
   dummy=0;
   for(t=0;t<Level[level].S;t++)
   {
    dummy=dummy+(N[level][t].G&&N[level][t].R&&Level[level-1].F[tt]
            &&N[level][t].C[tt]&&Level[level-1].E[tt]);
    }
   Level[level].IR[tt]=Level[level].IR[tt]+dummy;
  }
 }

/*************end of update Input record*******************/

/*************Update the threshold and regular***************/
update_threshold_regular()
{
int level,t,tt,count,count1;
 for (level=1;level<4;level++)
 {
  count1=0;
  for (t=0;t<Level[level].S;t++)
  {
   count=0;
   if (((!N[level][t].R)&&(N[level][t].G))==1)
   {
    for (tt=0;tt<Level[level-1].S;tt++)
    {
      if ((N[level][t].C[tt]&&Level[level-1].F[tt])==1) count++;
    }
    N[level][t].T=count-1;
   }
   N[level][t].R=((N[level][t].R)||(N[level][t].G));
   Level[level].E[t]=N[level][t].R;
   if (N[level][t].R==1) count1++;
  }
  Level[level].SR=count1;
 }
}
/*********End of Update the threshold and regular************/

/************************SLEEP*********************/
Sleep()
{
FILE *fp;
int level, t=0,tt=0,ttt=0,m=0,a=0,b=0,d=0,f=0,ss=0,s=0;
int count,test[MAX];
float z;
 srand(time ((time_t *)0));
 for (level=1;level<4;level++)
 {
  Level[level].IRM=0;
  for(t=0;t<Level[level-1].S;t++)
   Level[level].IRM=Level[level].IRM+Level[level].IR[t];
 }
/* for (t=0;t<54;t++) printf("%d ",Level[1].IH[t]);
 printf("\n");
 printf("IHM(Alpha, Beta, Gamma(%d %d %d)\n",Level[1].IHM,Level[2].IHM,Level[3].IHM);*/

/**************Reset All Virgins Not Imprinted *************/
 for (level=1;level<4;level++)
 {
  for(t=0;t<Level[level].S;t++)
  {
   if((!N[level][t].R))
   {
    N[level][t].T=Level[level].NT[0];
    N[level][t].G=0;
    Level[level].F[t]=0;
    for(tt=0;tt<Level[level-1].S;tt++)
     N[level][t].C[tt]=0;
   }
  }
 }
/******Remove Old Virgin Connections to Beta & Gamma*******/
 for(t=0;t<Level[2].S;t++)
 {
  if(N[2][t].R)
  {
   for(tt=0;tt<Level[1].S;tt++)
   {
    N[2][t].C[tt]=(N[2][t].C[tt])&&(N[1][tt].R);
   }
  }
 }
 for(t=0;t<Level[3].S;t++)
 {
  if(N[3][t].R)
  {
   for(tt=0;tt<Level[2].S;tt++)
   {
    N[3][t].C[tt]=(N[3][t].C[tt])&&(N[2][tt].R);
   }
  }
 }

/**********************Generate New Virgins**********************/
 for (level=1;level<4;level++)
 {
  Level[level].NV=(int)((float)(Level[level].NV+Level[level].SR-Level[level].S)*15.0/10.0);
  if(Level[level].NV<30)
   Level[level].NV=40;
  else if(Level[level].NV>80)
   Level[level].NV=80;
  if(Level[level].NV+Level[level].SR<150)
   Level[level].NV=200-Level[level].SR;
  Level[level].S=Level[level].NV+Level[level].SR;
 }
  fp=fopen("Train.rep","a");

  fprintf(fp,"No. of virgin neruon (%3d:%3d:%3d)\n",Level[1].NV,Level[2].NV,Level[3].NV);
  fprintf(fp,"No. of regular neruon(%3d:%3d:%3d)\n",Level[1].SR,Level[2].SR,Level[3].SR);
  fprintf(fp,"No. of neruon       (%3d:%3d:%3d)\n",Level[1].S,Level[2].S,Level[3].S);

/************Update Input History and Normailize************/
```

```
 for (level=1;level<4;level++)
 {
  for(t=0;t<Level[level-1].S;t++)
  {
   if ((Level[level].IHM!=0)&&(Level[level].IRM!=0))
   {
    Level[level].IH[t]=(int)((5000.0/Level[level].IHM)*Level[level].IH[t])
        +(int)((5000.0/Level[level].IRM)*Level[level].IR[t]);
   }
   else if ((Level[level].IHM==0)&&(Level[level].IRM!=0))
    Level[level].IH[t]=(int)((float)(10000.0/Level[level].IRM)*Level[level].IR[t]);
   else if ((Level[level].IHM!=0)&&(Level[level].IRM==0))
    Level[level].IH[t]=(int)((float)(10000.0/Level[level].IHM)*Level[level].IH[t]);
   else
   {
    if (N[level-1][t].R==1) Level[level].IH[t]=(int)(10000.0/Level[level-1].SR);
    else Level[level].IH[t]=0;
   }
  }
 }
/* for (t=0;t<Level[1].S;t++) printf("%d ",Level[2].IH[t]);
 printf("\n");
 for (t=0;t<Level[2].S;t++) printf("%d ",Level[2].IR[t]);
 printf("\n");*/

 for (level=1;level<4;level++)
 {
  Level[level].IHM=0;
  for(t=0;t<Level[level-1].S;t++)
  {
   Level[level].IHM=Level[level].IHM+Level[level].IH[t];
   Level[level].IR[t]=0;
  }
 }

/*******************Input Population Generation*******************/
 fprintf(fp,"after normalise\n");
 fprintf(fp,"IRM(Alpha, Beta, Gamma)(%5d %5d %5d)\n",Level[1].IRM,Level[2].IRM,Level[3].IRM);
 fprintf(fp,"IHM(Alpha, Beta, Gamma)(%5d %5d %5d)\n",Level[1].IHM,Level[2].IHM,Level[3].IHM);
 Level[1].IP[0]=Level[1].IH[0]+(int)(10000.0/(float)(Level[0].S));
 for(t=1;t<Level[0].S;t++)
 {
  Level[1].IP[t]=Level[1].IP[t-1]+Level[1].IH[t]+(int)(5000.0/(float)(Level[0].S));
 }
 for (level=2;level<4;level++)
 {

  if(N[level-1][t].R==1)
   Level[level].IP[0]=Level[level].IH[0]+(int)(1000.0/(float)(Level[level-1].SR));
  else Level[level].IP[0]=Level[level].IH[0]
      +(int)(10000.0/(float)(Level[level-1].NV));
  for(t=1;t<Level[level-1].S;t++)
  {
   if(N[level-1][t].R==1)
```

```
      Level[level].IP[t]=Level[level].IP[t-1]+Level[level].IH[t]
         +(int)(1000.0/(float)(Level[level-1].SR));
    else Level[level].IP[t]=Level[level].IP[t-1]+Level[level].IH[t]
         +(int)(10000.0/(float)(Level[level-1].NV));
   }
  }
/* for (t=0;t<Level[1].S;t++) printf("%d ",Level[2].IP[t]);
 printf("\n");*/
 fprintf(fp,"IP (Alpha, Beta, Gamma)(%5d %5d %5d)\n",Level[1].IP[Level[0].S-1],Level[2].IP[Level[1].S-
1],Level[3].IP[Level[2].S-1]);
fclose(fp);
/************************************************/
 for (level=1;level<4;level++)
 {
  for(t=0;t<Level[level].S;t++)
  {
   if(!N[level][t].R)
   {
    for(ttt=0;ttt<Level[level].m[1];ttt++)
    {
     m=(int)((((int)rand()/2147483647.00)*(Level[level].IP[Level[level-1].S-1]));
     for(tt=0;tt<Level[level-1].S;tt++)
     {
/*        if(m<=Level[level].IP[0])
       {
        N[level][t].C[tt]=1;
        tt=Level[level-1].S;
       }
       else if((Level[level].IP[tt-1]<m)&&(Level[level].IP[tt]>=m))
       */
       if(m<=Level[level].IP[tt])
       {
        N[level][t].C[tt]=1;
        tt=Level[level-1].S;
       }
      }
     }
    }
   }
  }

/****Create Populations For Adding New I/P's to Reg Beta*******/
 Level[2].LPF[0]=Level[2].FP[0];
 for(t=1;t<Level[2].S;t++)
   Level[2].LPF[t]=Level[2].FP[t]+Level[2].LPF[t-1];
 Level[2].LL[0]=!N[1][0].R;
 for(t=1;t<Level[1].S;t++)
   Level[2].LL[t]=Level[2].LL[t-1]+!N[1][t].R;
/************Add New Connection to Beta Regular***********/
 for(t=0;t<5*Level[2].SR;t++)
 {
   a=(int)((((int)rand()/2147483647.00)*(Level[2].LPF[Level[2].S-1]));
   b=(int)((((int)rand()/2147483647.00)*(Level[2].LL[Level[1].S-1]));
   for(ss=0;ss<Level[1].S;ss++)
   {
    if(b<=Level[2].LL[ss])
    {
     d=ss;
     ss=Level[1].S;
    }
   }
  for(f=0;f<Level[2].S;f++)
  {
   if(a<=Level[2].LPF[f])
   {
    N[2][f].C[d]=1;
/* printf("a=%4d b=%4d upper=%3d regular=%3d\n",a,b,d,f);*/
    f=Level[2].S;
   }
  }
 }
 for (level=1;level<4;level++)
  for (t=0;t<Level[level].S;t++)
   Level[level].FP[t]=0;
/***************Increase the Beta Regular Threshold*************/
 for(t=0;t<Level[2].S;t++)
 {
  s=0;
  if(N[2][t].R)
  {
   for(tt=0;tt<Level[1].S;tt++)
    s=s+(N[2][t].C[tt]&&N[1][tt].R);
   a=(int)(s/25.0);
   if(a>=N[2][t].T)
    N[2][t].T=a;
  }
 }
 d=Level[1].S/10;
 for (t=0;t<Level[2].S;t++)
  for (tt=0;tt<Level[1].S;tt++)
   test[t]=test[t]+N[2][t].C[tt]&&N[1][tt].R;
 for (t=0;t<Level[3].S;t++)
 {
  count=0;
  for (tt=0;tt<Level[2].S;tt++)
   if (test[tt]<d&&N[3][t].C[tt]==1) count++;
  if (count<2) N[3][t].T=50;
 }
}
/******************************END SLEEP******************************/
```

B.3 recall program

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

```
#include <sys/timeb.h>
#include <sys/types.h>
#define MAX 2000


struct NEURON
   {
    unsigned int T;
    unsigned int C[MAX];
    unsigned int R;
    unsigned int G;
   } N[5][MAX];
struct LEVEL
   {
    unsigned int S;
    unsigned int use[MAX];
   } Level[5];
main(argc, argv)
int argc;
char *argv[];
{
FILE *fp;
int level, tt,it,t,p,pp,n,s,count,lo,st,act;
int y[54];
Level[0].S=54;Level[4].S=1;
for (level=1;level<4;level++)
{
 fp=fopen(argv[level],"r");
 while (!feof(fp))
 {
  fscanf (fp,"%d",&t);
  fscanf (fp,"%d %d",&N[level][t].R,&N[level][t].T);
  pp=-1;
  while (1)
  {
   fscanf(fp,"%d ",&p);
   if (p>pp)
   {
    N[level][t].C[p]=1;
   }
   else break;
   pp=p;
  }
  fscanf(fp,"\n");
 }
 Level[level].S=t;
 fclose(fp);
}
fclose(fp);
fp=fopen(argv[4],"r");
for (t=0;t<Level[3].S;t++)
{
 fscanf(fp,"%d",&N[4][0].C[t]);
}
```

```
fclose(fp);
/***********remove the non-use neuron *******/
for (t=0;t<Level[3].S;t++)
{
 Level[3].use[t]=N[4][0].C[t];
 if (N[3][t].T==50) Level[3].use[t]=0;
 N[3][t].R=Level[3].use[t];
}

for (t=0;t<Level[3].S;t++)
{
 for (tt=0;tt<Level[2].S;tt++)
   if (N[3][t].C[tt]&&N[3][t].R==1) Level[2].use[tt]=1;
}
for (t=0;t<Level[2].S;t++)
{
 N[2][t].R=Level[2].use[t];
}
for (t=0;t<Level[2].S;t++)
{
 for (tt=0;tt<Level[1].S;tt++)
   if (N[2][t].C[tt]&&N[2][t].R==1) Level[1].use[tt]=1;
}
for (t=0;t<Level[1].S;t++)
{
 N[1][t].R=Level[1].use[t]&&N[1][t].R;
 if (N[1][t].R==0) N[1][t].T=999;
}
/*for (t=0;t<Level[1].S;t++) printf("%d",Level[1].use[t]);
printf("\n");
for (t=0;t<Level[1].S;t++) printf("%d",N[1][t].R);
printf("\n");*/
/***************** re-organize the regular & virgin ********/
for (level=1;level<4;level++)
{
 st=Level[level].S;
 for (t=0;t<Level[level].S;t++)
 {
  if (N[level][t].R==0)
  {
   for (it=st;it>=t;it--)
   {
    if (N[level][it].R==1)
    {
     N[level][t].R=N[level][it].R;
     N[level][it].R=0;
     N[level][t].T=N[level][it].T;
     for (tt=0;tt<Level[level-1].S;tt++)
       N[level][t].C[tt]=N[level][it].C[tt];
     for (tt=0;tt<Level[level+1].S;tt++)
     {
      if (N[level+1][tt].C[it]==1) N[level+1][tt].C[t]=1;
      N[level+1][tt].C[it]=0;
     }
```

```
       st=it;
       it=t;
      }
     }
    }
   if (t>=st) t=Level[level].S;
  }
  count=0;
  for (t=0;t<Level[level].S;t++)
  {
   if (N[level][t].R==1) count++;
   /*printf("%d",N[level][t].R);*/
  }
/* printf("\n");*/
  printf("level(%d) finish (%d:%d)\n",level,Level[level].S,count);
  Level[level].S=count;
 }

/********* recall*************/
fp=fopen(argv[5],"r");
act=0;
printf("%s\n",argv[5]);
for (lo=0;lo<50;lo++)
{
  for (t=0;t<54;t++)
  {
   fscanf(fp,"%d",&N[0][t].G);
  }
  for (level=1;level<5;level++)
  {
    for (t=0;t<Level[level].S;t++)
    {
     count=0;
     N[level][t].G=0;
     for (tt=0;tt<Level[level-1].S;tt++)
     {
      if (N[level-1][tt].G&&N[level][t].C[tt])
      {
       count++;
       if (count>=N[level][t].T)
       {
        N[level][t].G=1;
        tt=Level[level-1].S;
       }
      }
     }
    }
   }
/*   for (t=0;t<Level[1].S;t++)
      printf("%d",N[1][t].G);
    printf("\n");*/
  printf("Basal neuron(%d)->%d\n",lo,N[4][0].G);
  if (N[4][0].G==1) act++;
 }
```

```
printf("Number of pattern be recalled=%d\n",act);
}
```

B.4 Switching Tree Minimization Program

```
#include "stdio.h"

main()
{
FILE *ptr,*ptr1;
int nn,c,a[2][8],array[3][5000],i,m,j,count[5000],tmp,tmp1;
char *inp[5],*outp[5];
tmp1=0;
inp[0]="a.out";
outp[0]="b.out";
/* Must have a "a.out" file include no. of output and the data list */
ptr=fopen("array.dat","w");
fprintf(ptr,"%d %d ",0,0);
fclose(ptr);
while (c!=1)
{
 ptr=fopen("array.dat","r");
 fscanf(ptr,"%d %d ",&m,&tmp);
 if (m!=0)
 {
  for (i=0;i<m;i++) fscanf(ptr,"%d %d %d %d ",&count[i],&array[0][i],&array[1][i],&array[2][i]);
 }
 fclose(ptr);
 printf("%d\n",m);
 printf("loop %d.... \n",tmp);
 if (tmp1==0)
 {
  ptr=fopen(inp[0],"r");
  ptr1=fopen(outp[0],"w");
 }
 else
 {
  ptr=fopen(outp[0],"r");
  ptr1=fopen(inp[0],"w");
 }
 fscanf(ptr,"%d",&nn);
 fprintf(ptr1,"%d\n",nn);
 c=0;
 while (!feof(ptr))
 {
  for (i=0;i<nn;i++)
   fscanf(ptr,"%d ",&a[0][i]);
  for (i=0;i<nn;i++)
   fscanf(ptr,"%d ",&a[1][i]);
  c++;
  for (i=0;i<nn;i++)
```

```
   {
    j=0;
    if (a[0][i]==a[1][i])
    {
     fprintf(ptr1,"%d ",a[0][i]);
     j=5000;
    }
    else
     if (m!=0)
     {
      for (j=0;j<m;j++)
      {
       if (a[0][i]==array[0][j]&a[1][i]==array[1][j]&count[j]==tmp+1&(array[2][j]==i-
1|array[2][j]==i+1|array[2][j]==i))
            {
             fprintf(ptr1,"%d ",j+2);
             j=5000;
       }
          }
     }
    if (j<5000)
    {
     count[m]=tmp+1;
     array[0][m]=a[0][i];
     array[1][m]=a[1][i];
         array[2][m]=i;
     m++;
     printf("%3d %3d %3d\n",array[0][m-1],array[1][m-1],array[2][m-1]);
     fprintf(ptr1,"%d ",m+1);
    }
   }
   fprintf(ptr1,"\n");
  }
  fclose(ptr);
  fclose(ptr1);
  ptr=fopen("array.dat","w+");
  fprintf(ptr,"%d %d\n",m,tmp+1);
  for (i=0;i<m;i++)
  {
   fprintf(ptr,"%2d %3d %3d %3d\n",count[i],array[0][i],array[1][i],array[2][i]);
  }
  fclose(ptr);
  if (tmp1==0) tmp1=1;
  else
   tmp1=0;
 }
 return(0);
}
```

Appendix C

Layout of the Design

This Appendix contains layouts and schematics that involve in this thesis research.

Figure C.1 Schematic of the 8-bit Counter

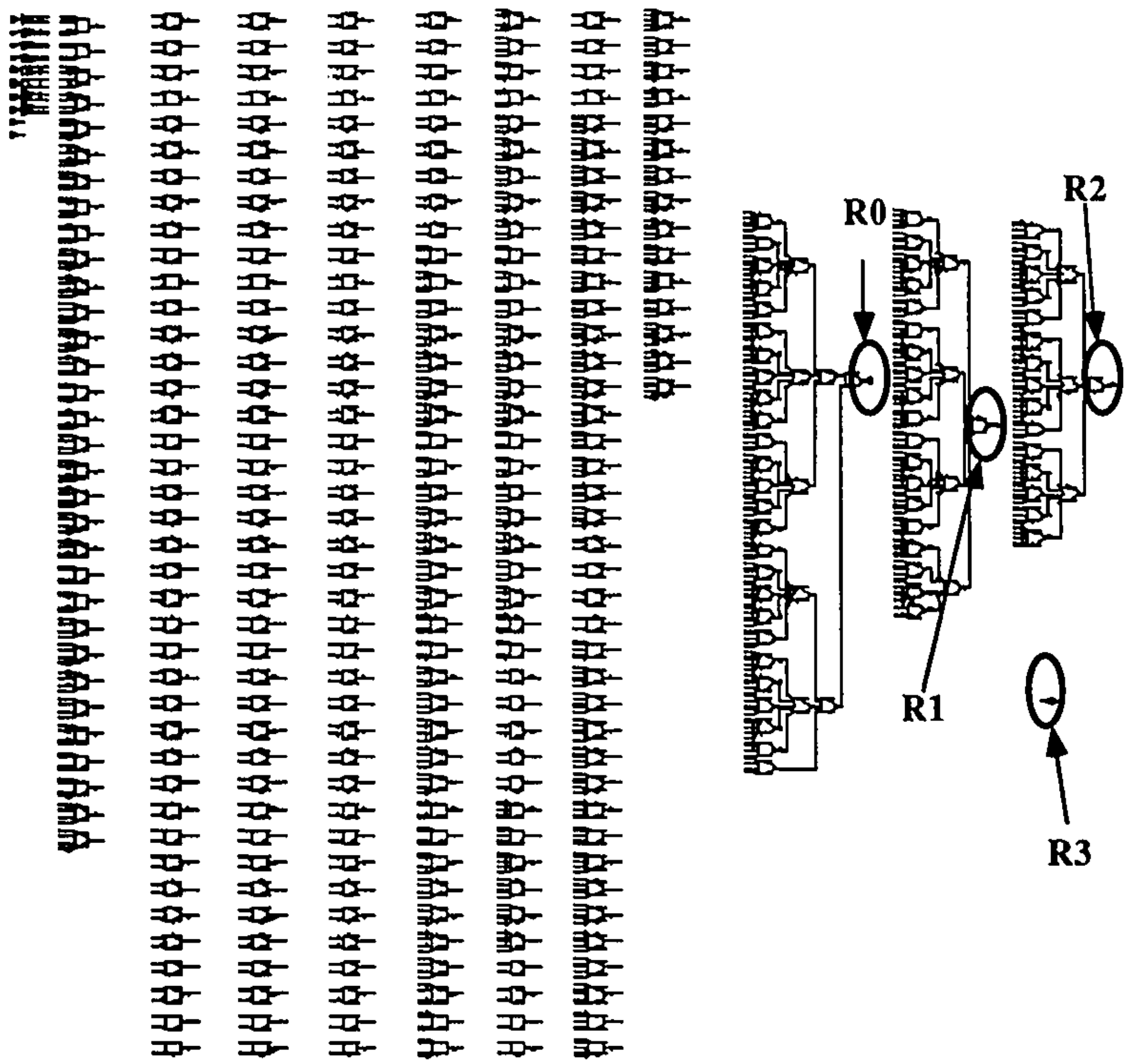


Figure C.2 Schematic of the 8-4 Subtractor

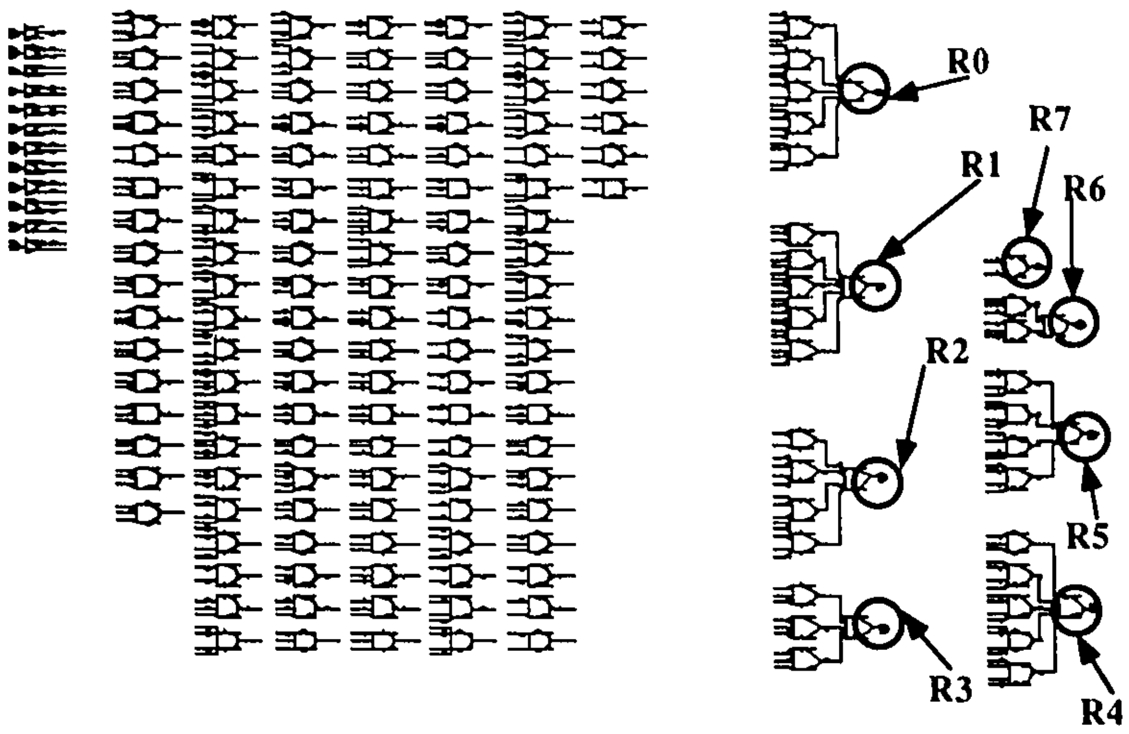


Figure C.3 Layout of 8-input Pipeline Logic Gate Neuron

Figure C.4 Schematic of the 4-bit Counter

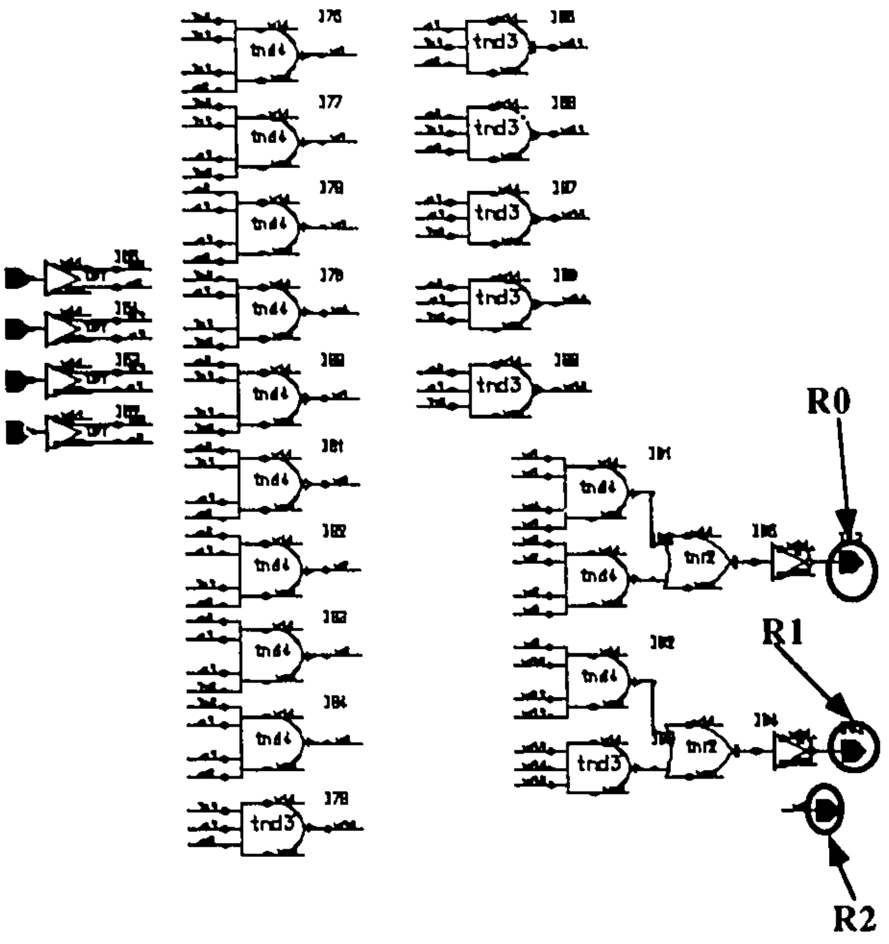


Figure C.5 Schematic of the 8-3 Subtractor

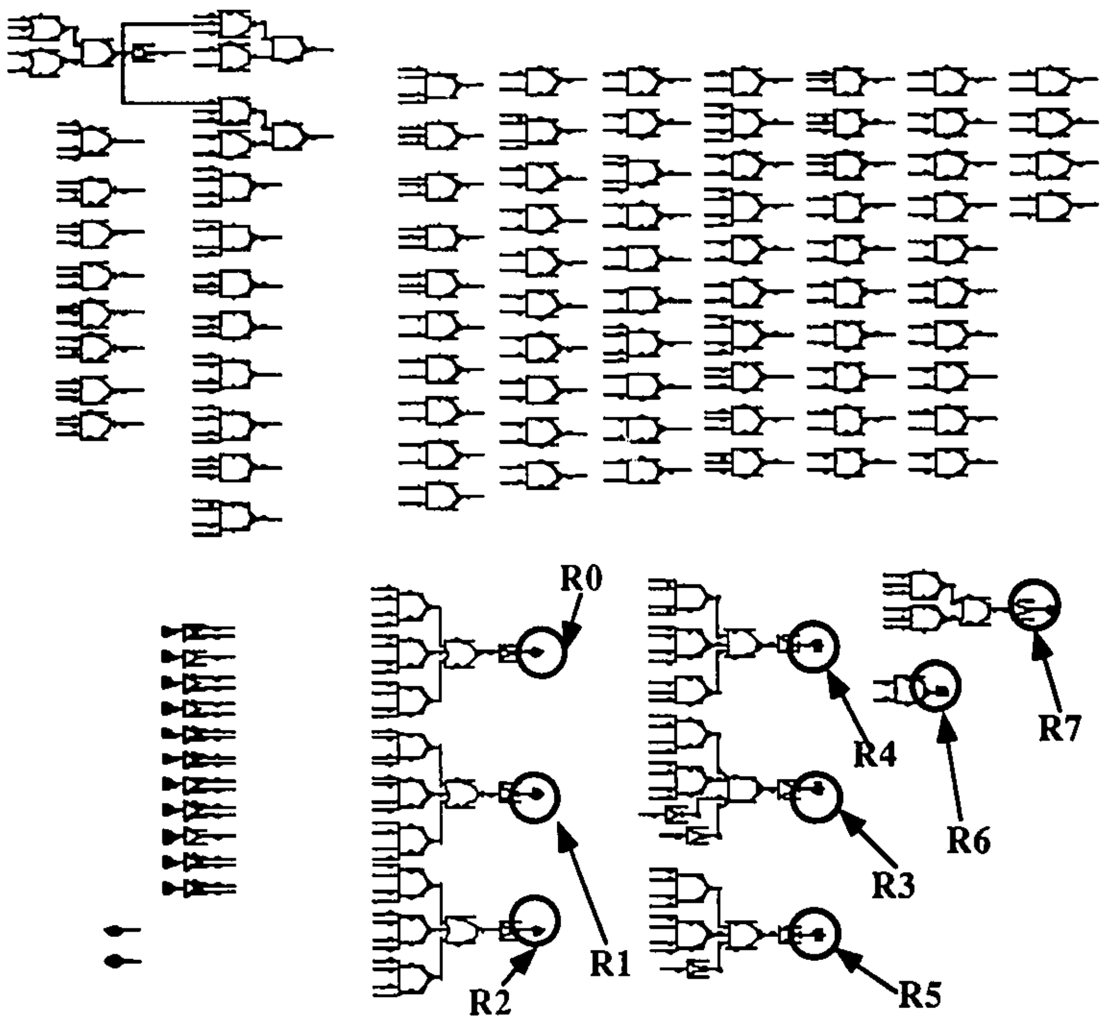


Figure C.6 Layout of 4-input Pipeline Logic Gate Neuron

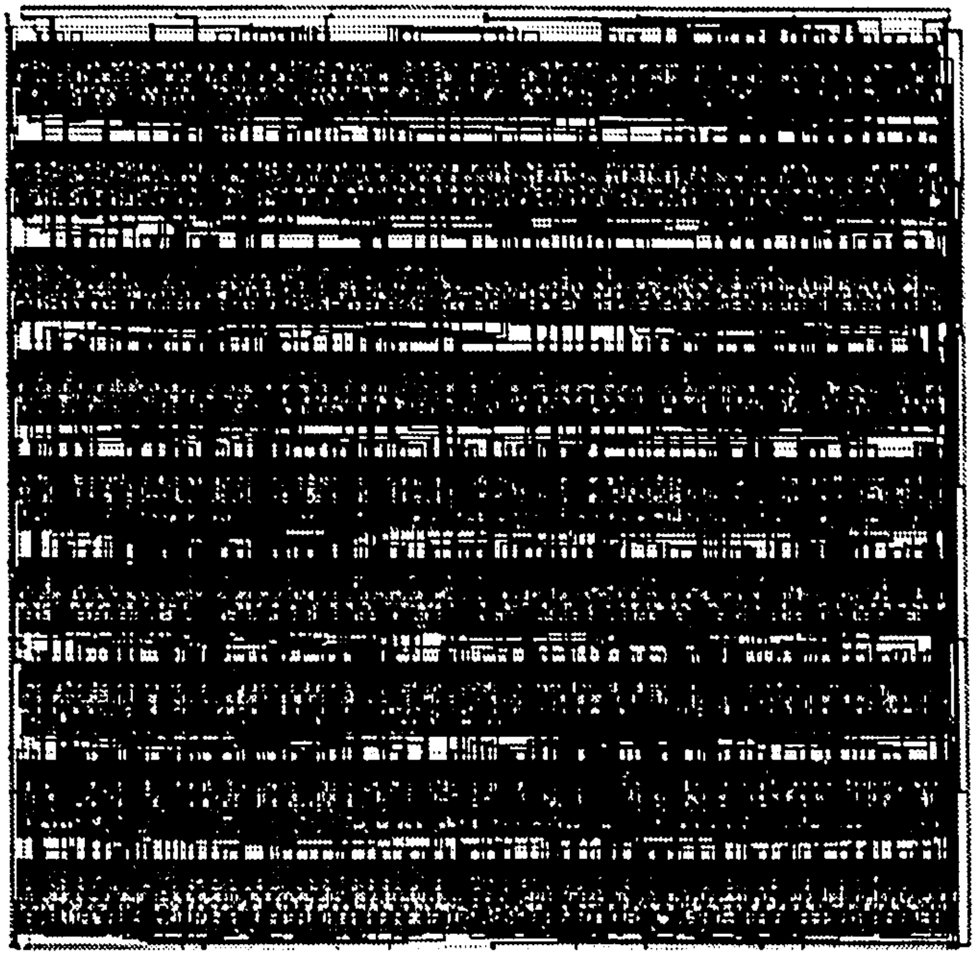


Figure C.7 Layout of the Resettable D-type Negative-edge Flip-Flop

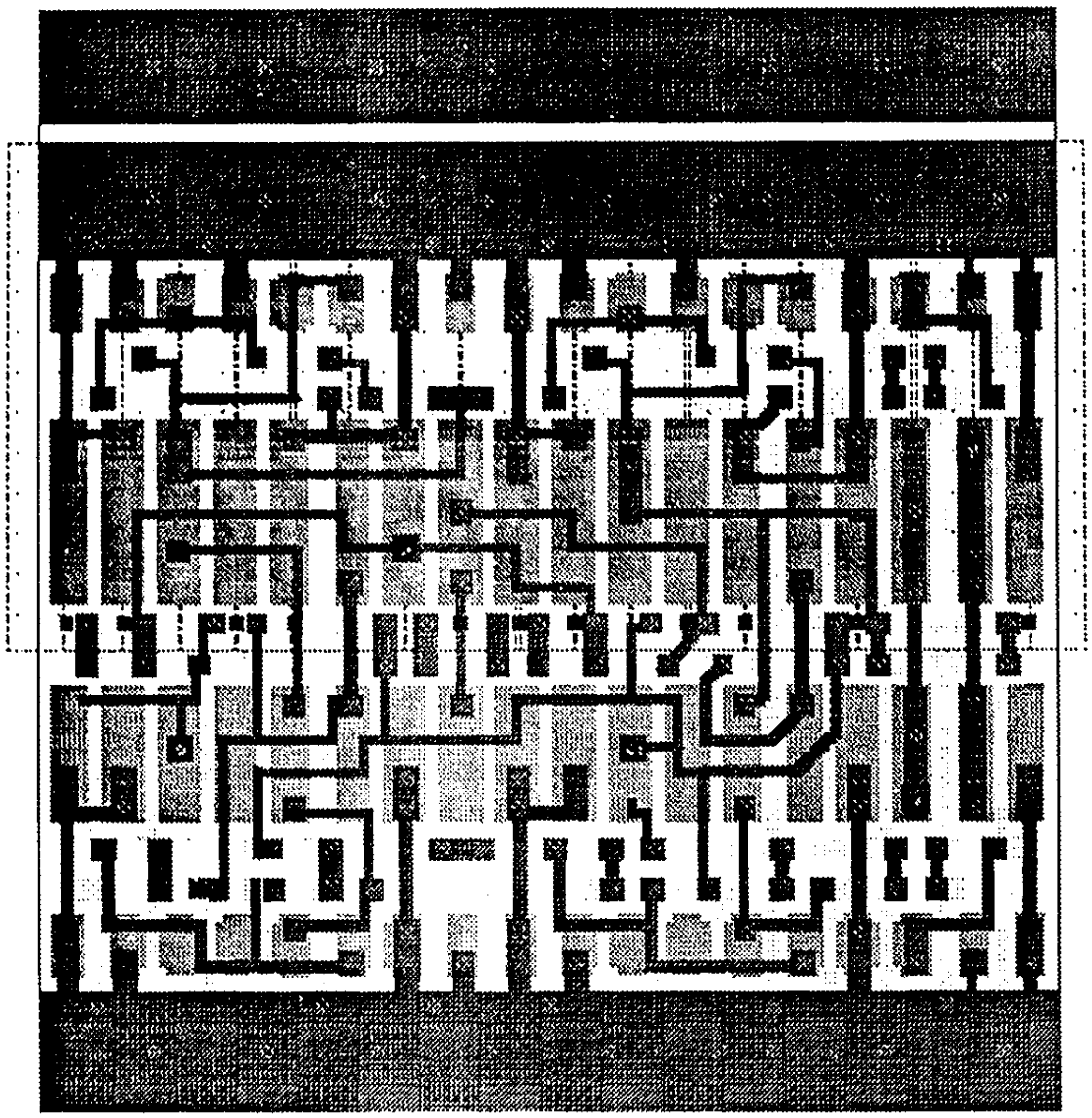
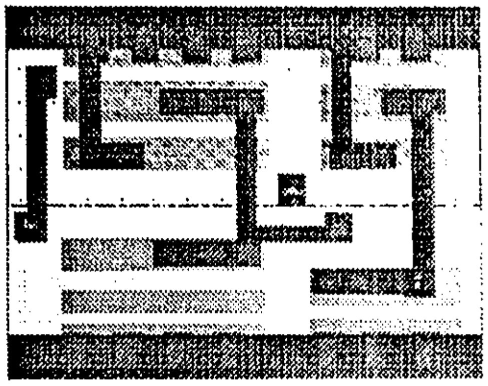


Figure C.8 AND-2 Layout in Custom (left) and Tcell (right)Design



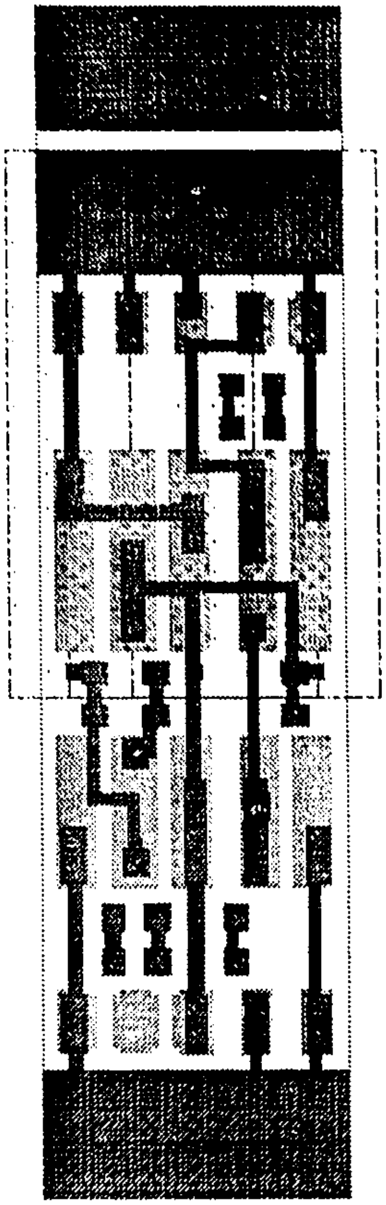
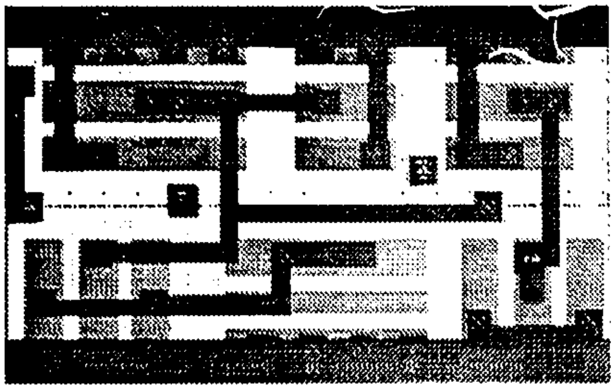


Figure C.9 AND-3 Layout in Custom (left) and Tcell (right) Design



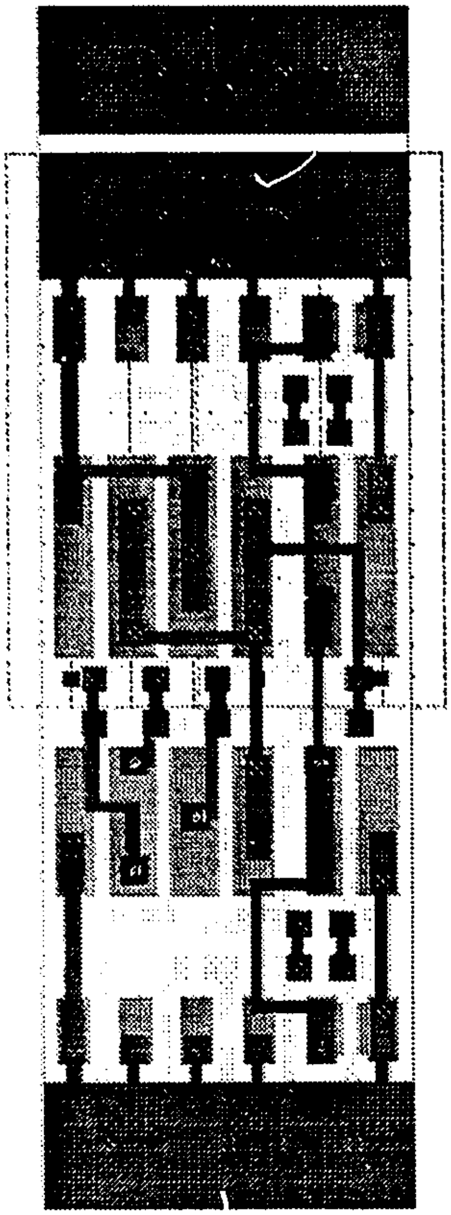


Figure C.10 Buffer Layout in Custom (left) and Tcell (right) Design

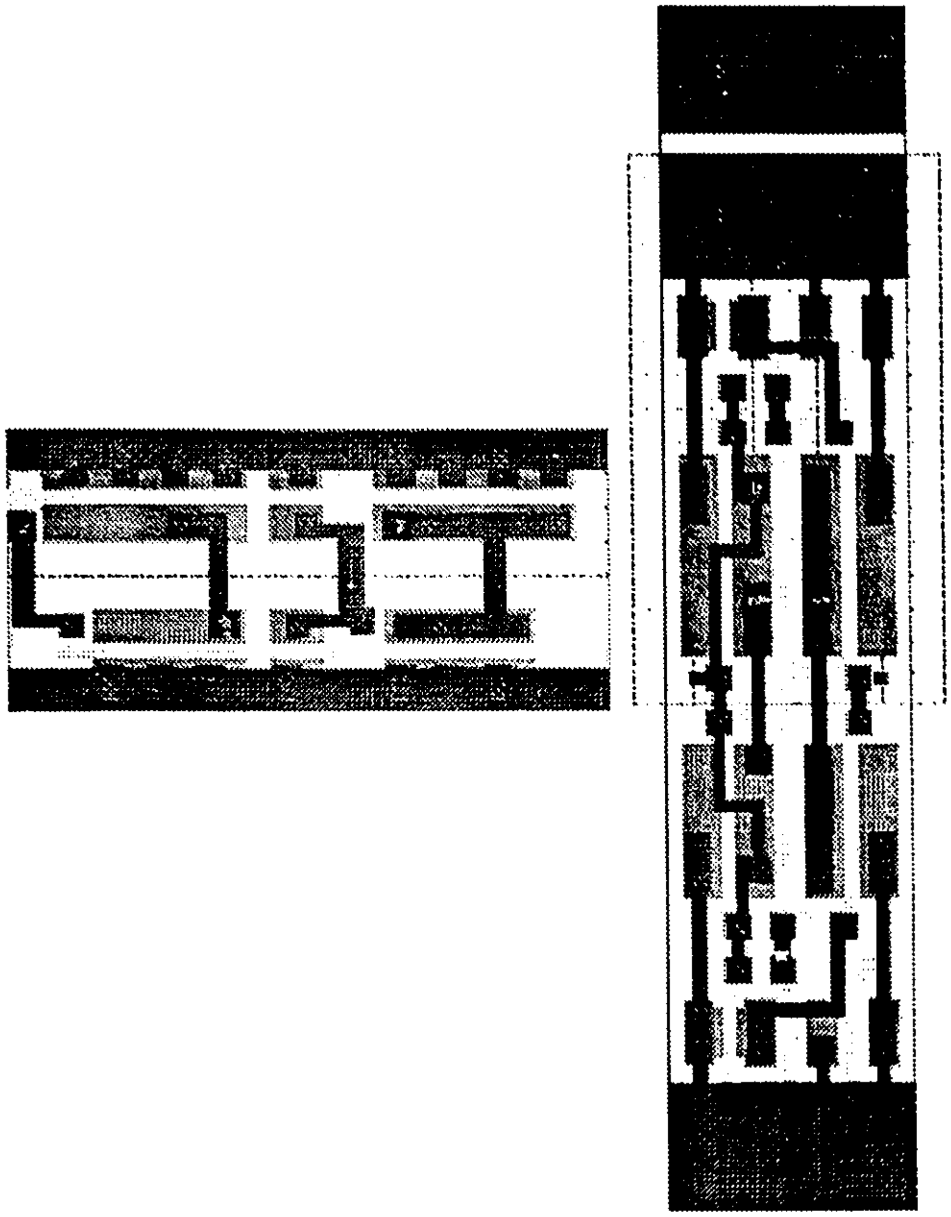
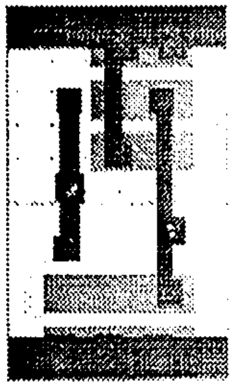


Figure C.11 Inverter Layout in Custom (left) and Tcell (right) Design



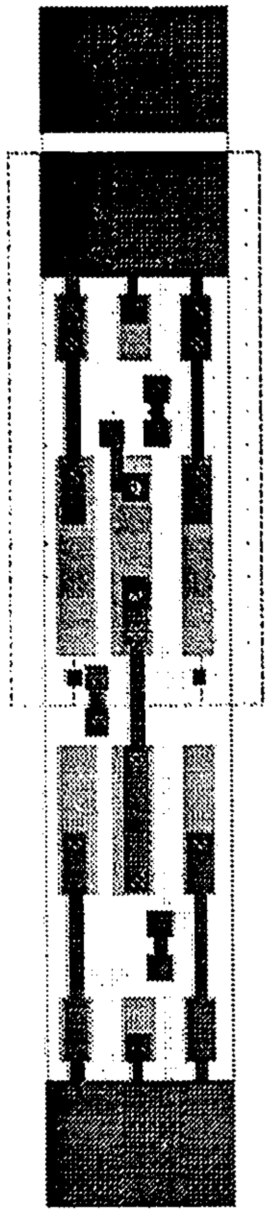
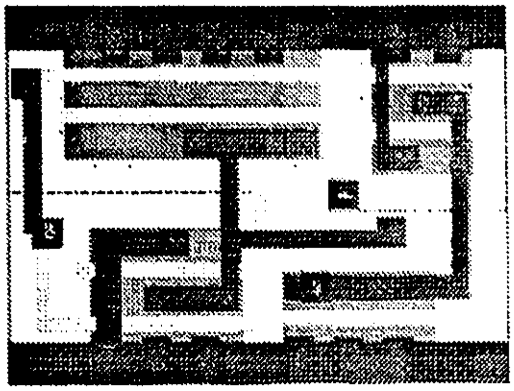


Figure C.12 OR-2 Layout in Custom (left) and Tcell (right) Design



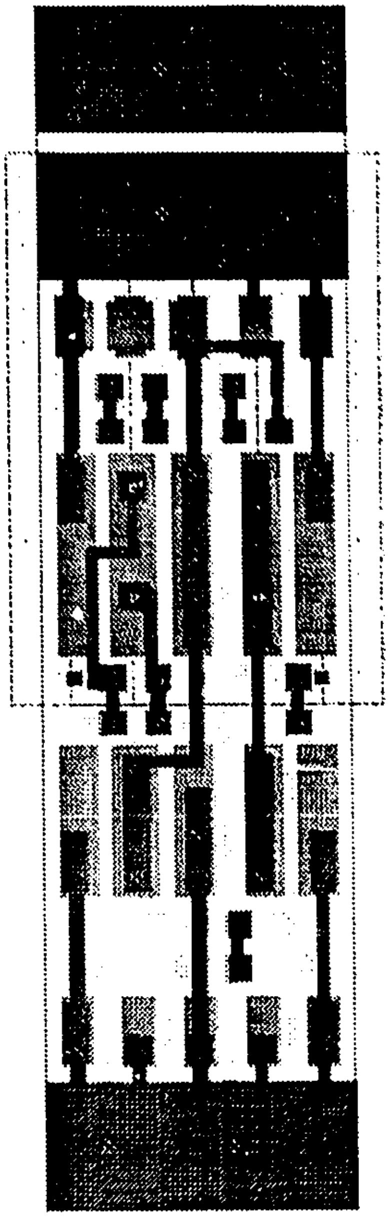


Figure C.13 Layout of the 8-bit Parallel Counter with Complement Outputs

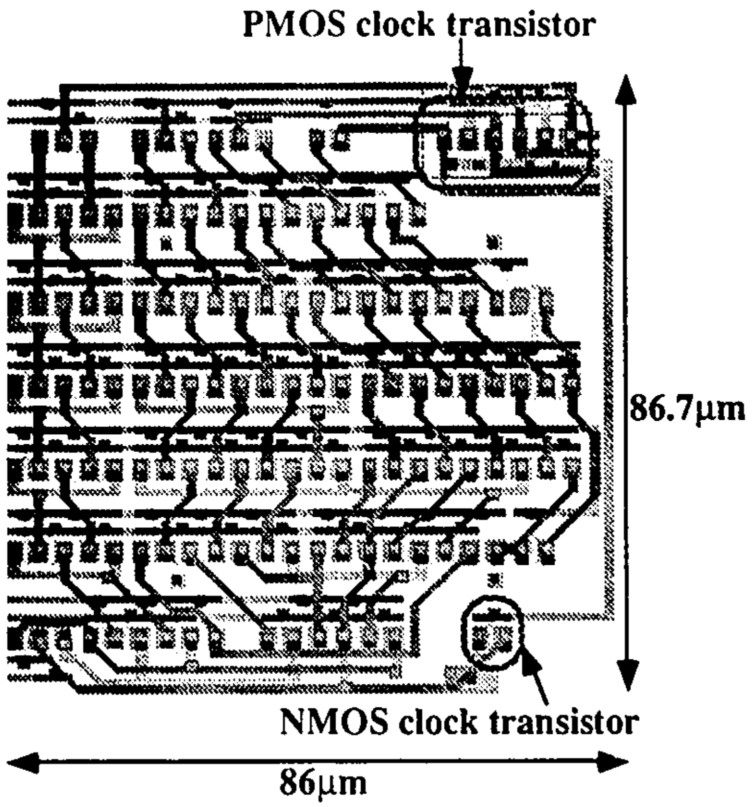


Figure C.14 Layout of the 8-4 Subtractor with Complement Outputs

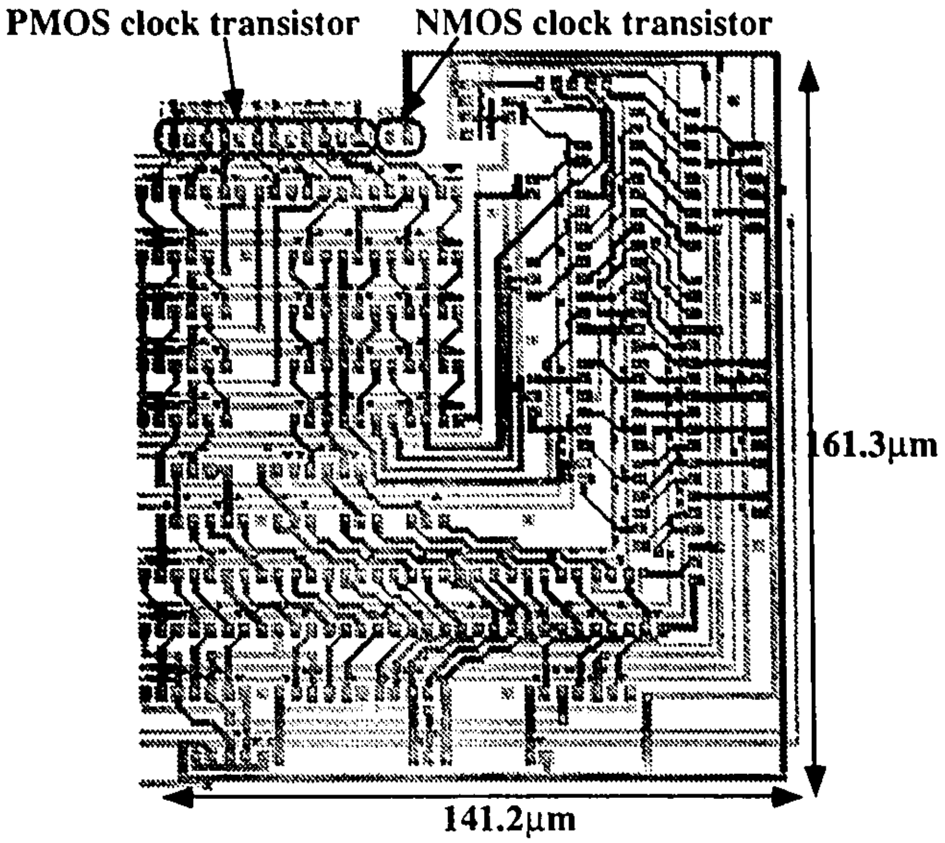


Figure C.15 Layout of 7-bit Parallel Counter with Complement Outputs

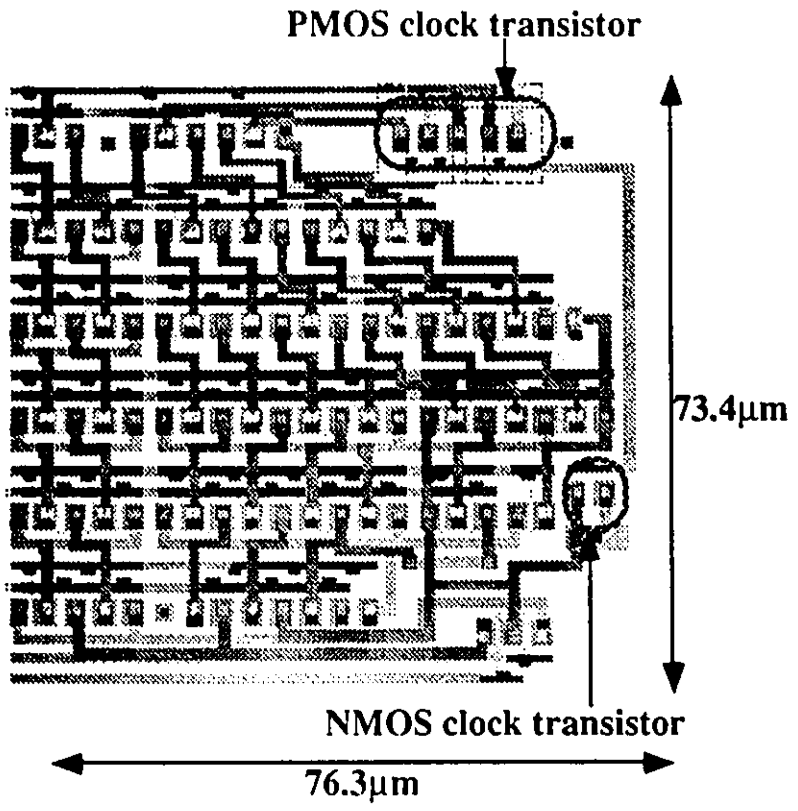
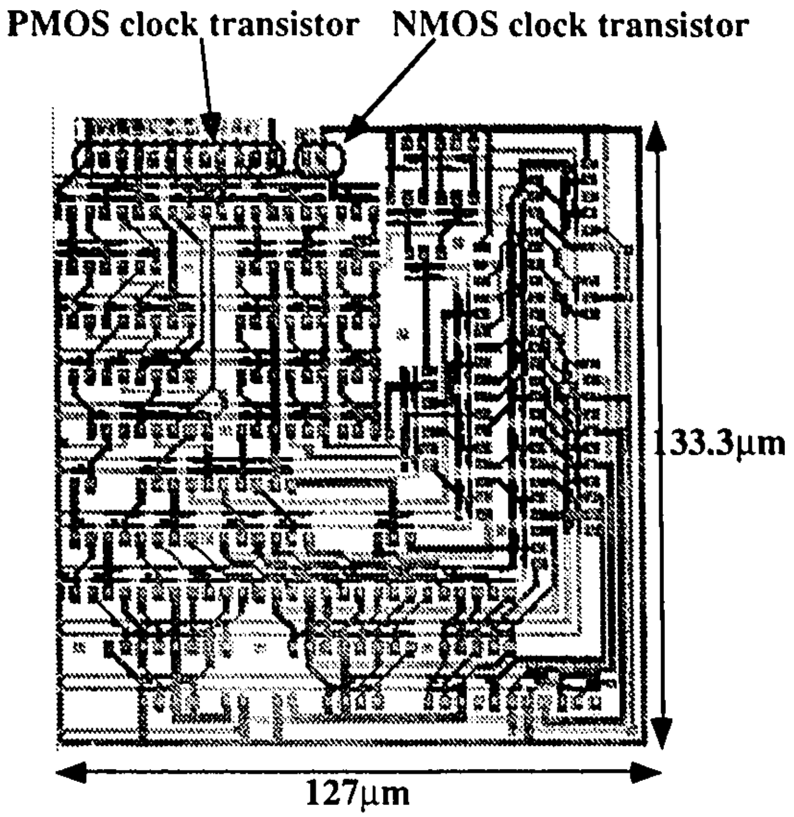


Figure C.16 Layout of the 8-3 Subtractor With Complement Outputs



Appendix D

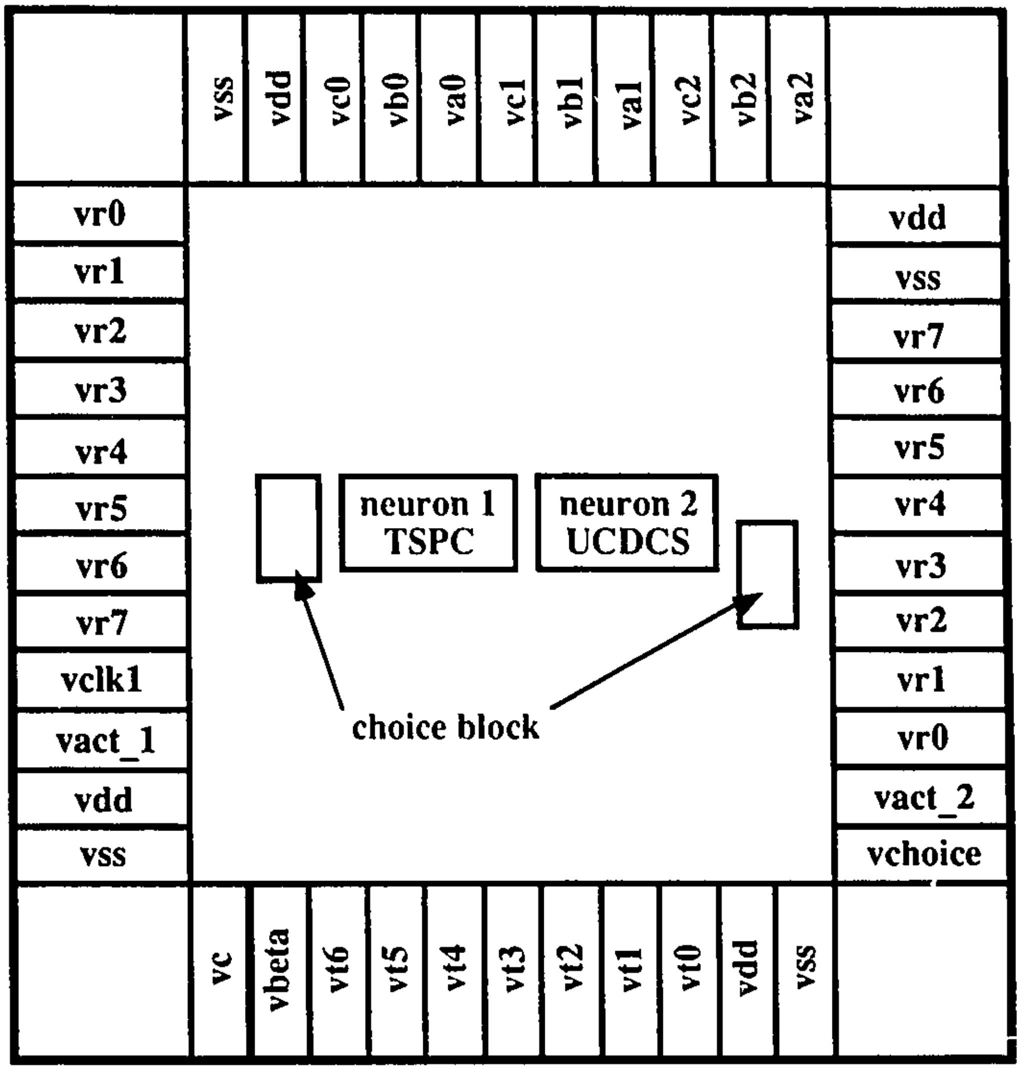
Pads Setting in the Test Cell

This Appendix includes the diagram of the pad setting in the test cell. The configuration of the pad is as shown in Table D.1.

Table D.1 Configuration of the Pad

pad name	comment
vss	ground pad
vdd	power pad
va0, va1, va2	connectivity, C_{nijt}
vb0, vb1, vb2	firing status, $F_{i,j-1,t}$
vc0, vc1, vc2	regular?, $R_{i,j-1,t}$
vclk1	the clock signal for the latch
vt0...vt6	the threshold, T_{ijt}
vbeta	regular?, R_{ijt}
vc	selection bit of the 3-bit parallel subtractor
vact_1	activity of neuron 1 using the TSPC
vact_2	activity of neuorn 2 using the UCDCS
vchoice	selection bit of choice block (testing outputs)
vr0..vr7 (left)	testing outputs of parallel counter with 3-input AND gate
vr0..vr7 (right)	testing outputs of parallel counter with 2-input AND gate

Figure D.1 Diagram of Pads Setting



Vita Auctoris

Hung, Kai Yiu was born in Hong Kong on September 10, 1966. He completed the secondary school education in Hong Kong in 1984. In 1987, he came to Canada as a visa student in Columbia Secondary School in Hamilton, Ontario. In September of 1987, he enrolled at the Electrical Engineering program at the University of Windsor. He graduated with a bachelor of Applied Science in 1991. At the same year, he was admitted into the Master's program in the Department of Electrical Engineering, University of Windsor, and has been a member of the VLSI research group since 1991.