USC-SIPI REPORT #180

VLSI Neurocomputers: EE 599 Term Projects / 1990-1991

edited by

Bing J. Sheu and Chia-Fen Chang

May 1991

Signal and Image Processing Institute
UNIVERSITY OF SOUTHERN CALIFORNIA
Department of Electrical Engineering-Systems
Powell Hall of Engineering
University Park/MC-0272
Los Angeles, CA 90089 U.S.A.

VLSI Neurocomputers

- EE599 Term Projects / 1990-1991 -

Edited by Bing. J. Sheu, Ph.D. and Chia-Fen Chang

Table of Contents

Part I. Digital Neurocomputer Architectures and Design

Part II. Applications

Part III. Structured Network Solvers for Matrix Algebra

SYSTOLIC NEURAL NETWORK

Auther: Te-Ho Chen

EE 599 term project

partner: Juinn-yan Chen

professor: B. J. Sheu

Date: November 28, 1990

A.Introduction:

In this project, we tried to implement systolic neural array to take advantage of benefits of neural network, that massively parallel computation and learning algorithm available for neural networks, and benefits of systolic array , that compressing multi-dimensionnal PE into one-dimensional PEs. And this system can satisfy the requirements of digital signal / image processing.

B. work partition:

I take care of system level and control unit, and Mr. Juinn-yan Chen works on processing element.

C. system architecture: (Fig 1)

.use interface chip as one I/O port, host issue macroinstruction and dimension, then download data via DMA.

. control unit includes 4 major parts:

a.memory access part: for download data to each local memory.

b. fault detection and recovery parts: detect faulty PE , correct error and recover by spare PE.

c. I/O port: recept data , macrocode and dimension from fost, upload data, faulty signal to host.

d. contol unit: to control operation of each PE , and to broadcast and collect data.

D. implementation:

1. fault detection and recovery part:

.use microprogramming

.advantage: easy to design, maintain, and expand; great portability and compatibility with different performance and different architecture of host and PEs.

.disadvantage: slower than dedicated control finite state machine.

(could use high speed memory to improve)

microprogram and supporting hardware isin Fig 2.

.use 2 level microprogram to speedup the accessing speed for most operations are repetious and commonly used.

.use program table on top of microprogram for portability with host.

.4 major operations in microprogram:

a. next address operation: next operation is at next address. opcode=00. ✓

b.conditionnal branch operation: next operation's address depends on condition in the microcode. opcode=01.

c.while do operation: could be done by combination of next address and conditional branch operations, and use a dedicated counter to count the iteration number. opcode=11.

d.stop operation: for the last operation in nanoprogram, test if repetition enough (content of dedicated counter for dimension of macroinstruction of host). If yes, microprogram go to next operation, else nanoprogram go to first nanooperation and trigger dimension counter to count down. opcode=10.

refer to Fig 3,4

.fault tolerant algorithm use weighted check sum to detect and correct error. (ref 1) Fig 5. Flowchart and corresponding microoperation is in Fig 6. For multiple faulty PEs, the algorithm need a little modification. (ref 1). ✓

. Error detection in microprogram and counter:

.use parity bits to detect one or two adjacent faulty bits.(ref 2) Fig 7, Fig 8.

. required ALU , register and bus even some nanoprogram are the same with PE's. (my partner took care of that part)

.after completion of error detection and correction, recovery operation can be accomplished by dedicated microprogram (not combinational ckt, in order to facilitate the scaling (increase PE number). Fig 9.

✓

2.control unit:

.to control and monitor each PE's operation .
.to broadcast and collect data to and from each PE.
.accomplished by microprogramming.
.macroprogram (for PEs) format is composed of opcode and iteration number.
.Fig 10.

3 I/ O port and memory access parts : future work.

E. discussion

.could do RBP and HMM algorithm .(ref 3) ✓
.also can do normal algorithm by ring architecture. eg. vector quantization. Fig 12.
. most operations in PE can be done in pipeline by dedicated buses and registers, and so does parallel processing among PEs. High performance is expected. For 4 PEs, 256*256 image, 2*2 window, 1024 codewords, the throuput for each PE is 1/ 15 nsec, then throughput of whole system is 2 frames/ sec; copression rate is 3.2:1.
.This system could be general-purpose scalable digital signal processing array (extra application can be done by adding or modifying microprogram). ✓
.also can be special purpose accelerator by using dedicated microprogram in small ROM to set up the configuration .
.suitable to apply to HDTV for image compression and store.
.also suitable for PC for image processing: this system doesn't need high speed host because of macroinstruction. (more image and less text is the future trend).

.could be self - organized system by expanding I / O part and memory management.

.also could be emulator for neural network and multiprocessors system.

.This project isn't finished yet, and needs lots of work.

✓

Reference:

1."VLSI array processors " by S. Y. Kung
Prentice - Hall , 1988, pp 402—408.

2. " Microprogrammed control and reliable design of small computers " ✓
by George D. Kraft / Wing N. Toy.
Prentice - Hall inc., 1981, P 297 and 277.

3. " A systolic neural network architecture for hidden Markov Models "
by Jenq - Neng Hwang, John A. Vlontzos, and Sun-Yuan Kung. ✓
IEEE Trans. on ASSP, vol. 37, NO. 12, Dec 1989, pp 1967 —1979.

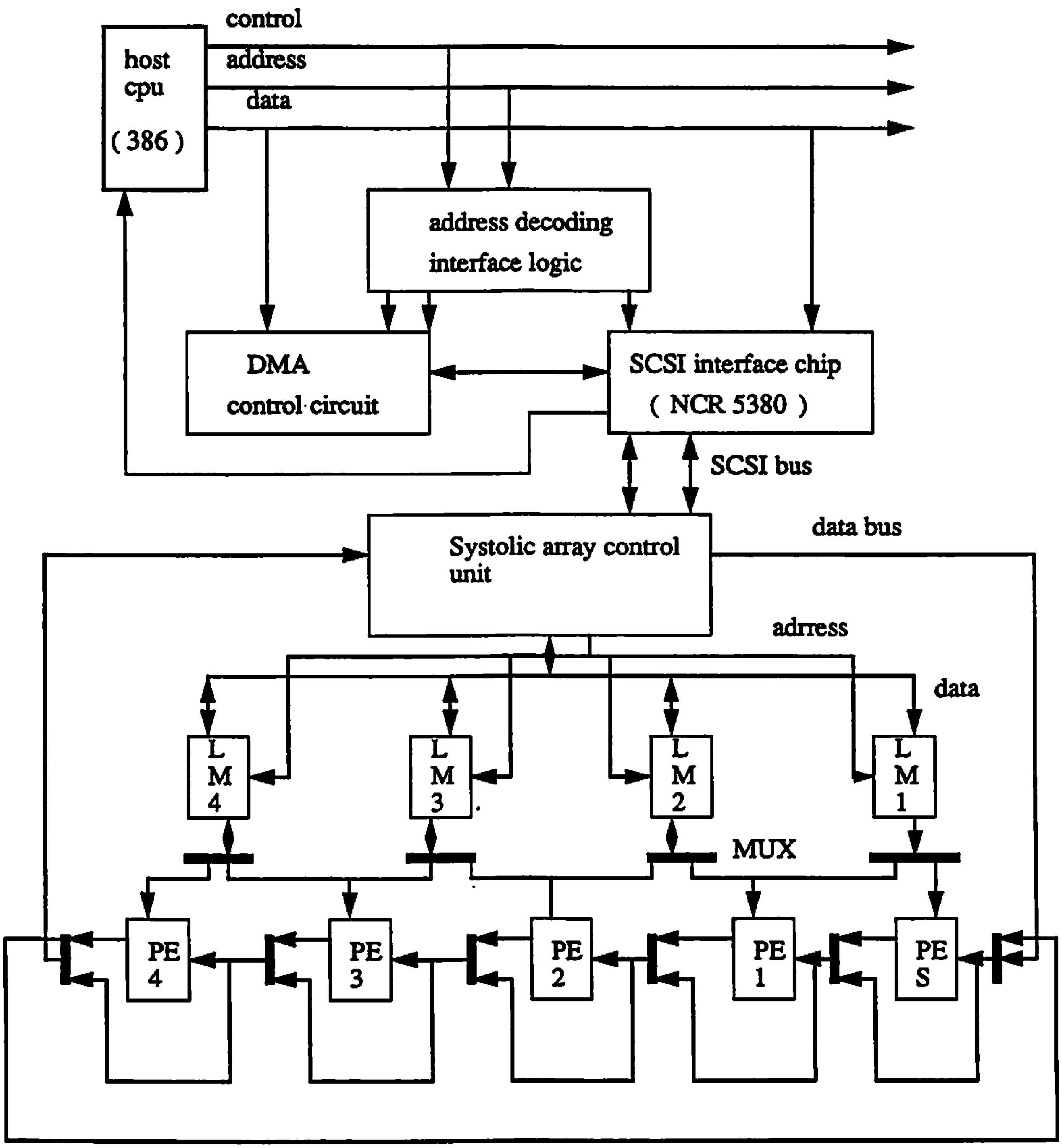


Fig 1. Systolic array neural network architecture. (4 PE's are shown , more PE's and LM's can be added with modification of program memory content in control unit.)

mware/hardware implementation:

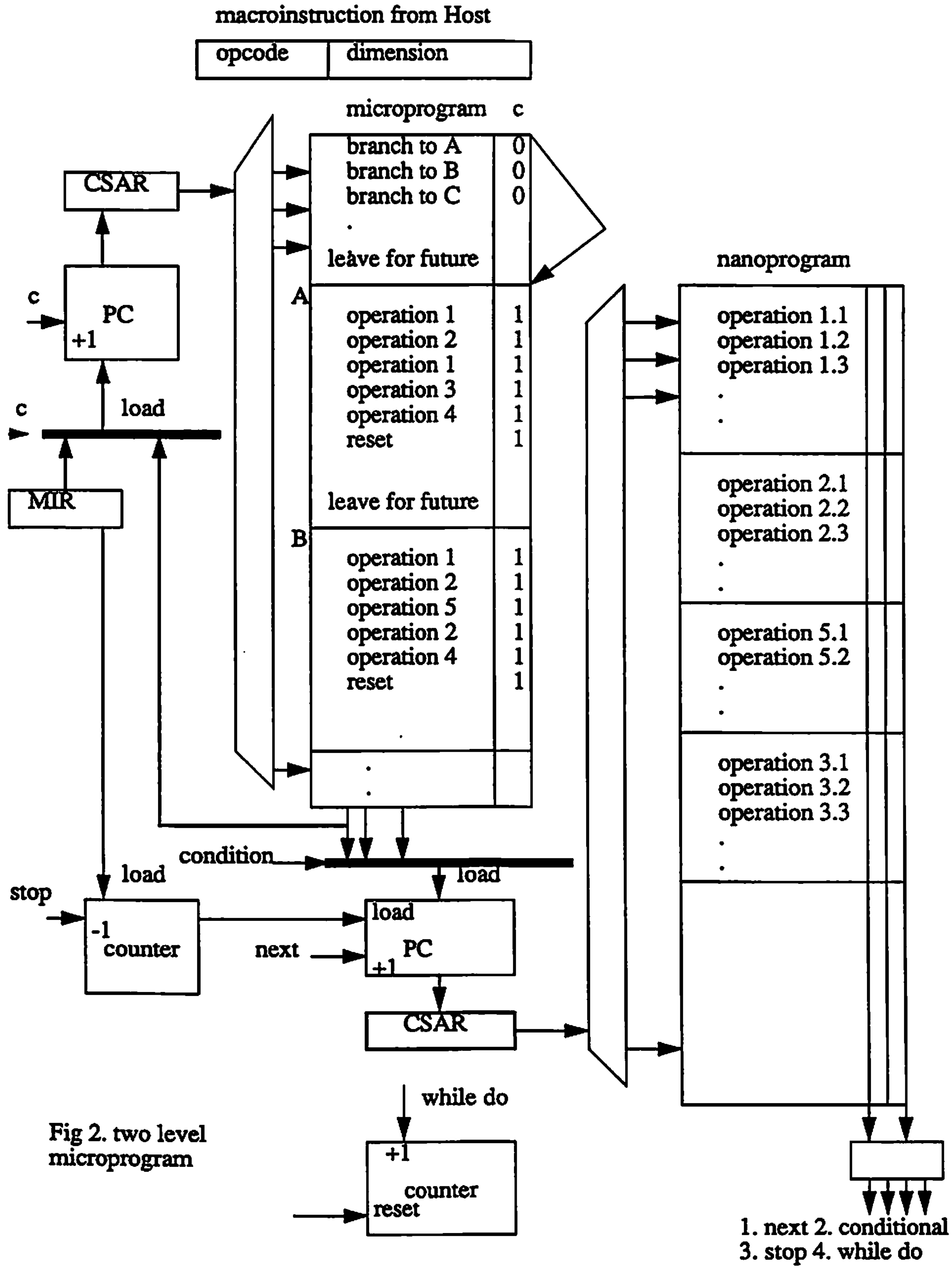
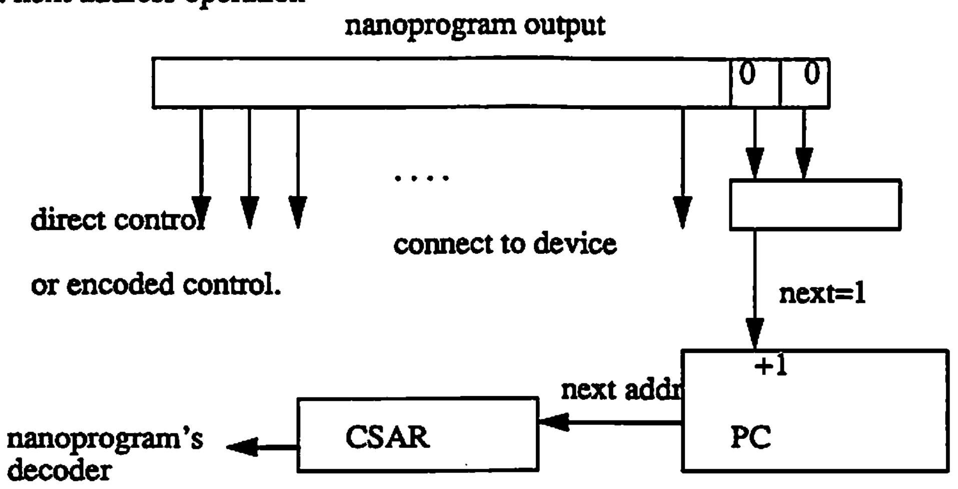


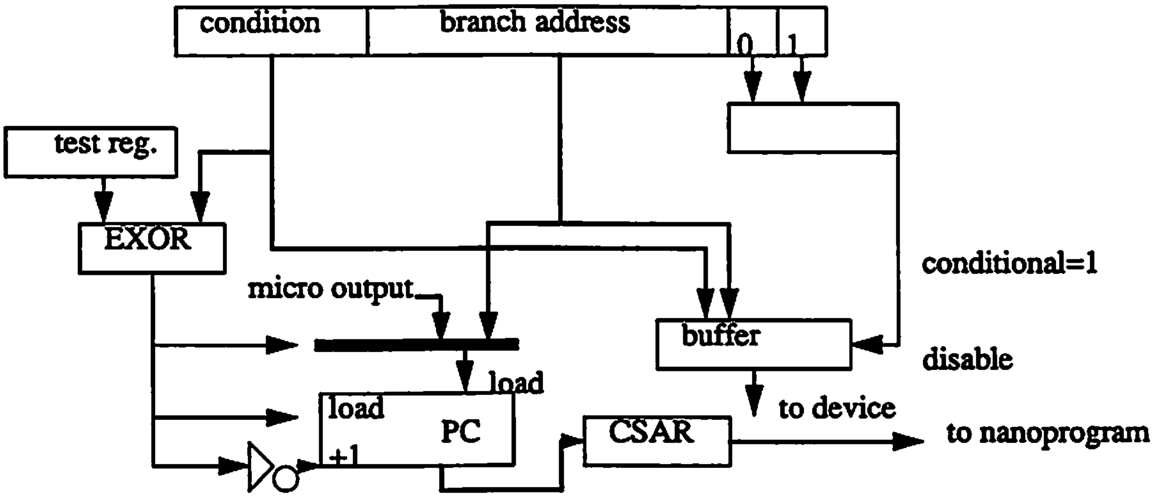
Fig 2. two level microprogram

architecture for 4 kinds of operation: (in nanoprogram)

1. next address operation



2. conditional branch operation:



3. while do operation:

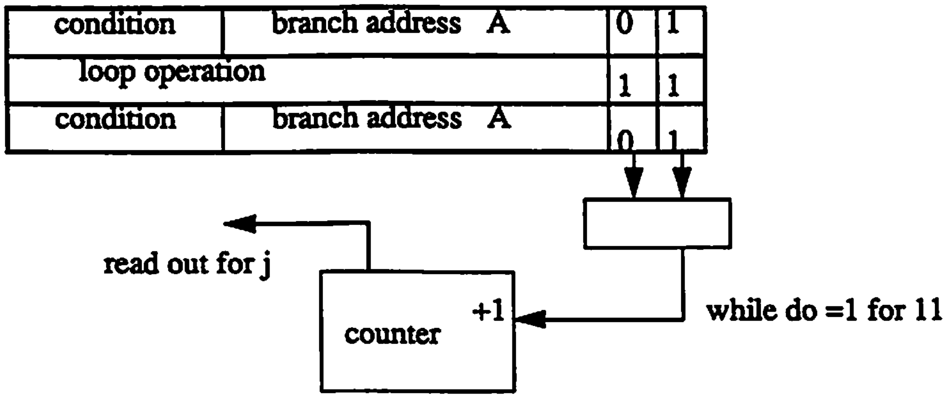


Fig 3. microoperation format and supporting hardware.

4. stop operation: the last operation in nanooperation sequence.

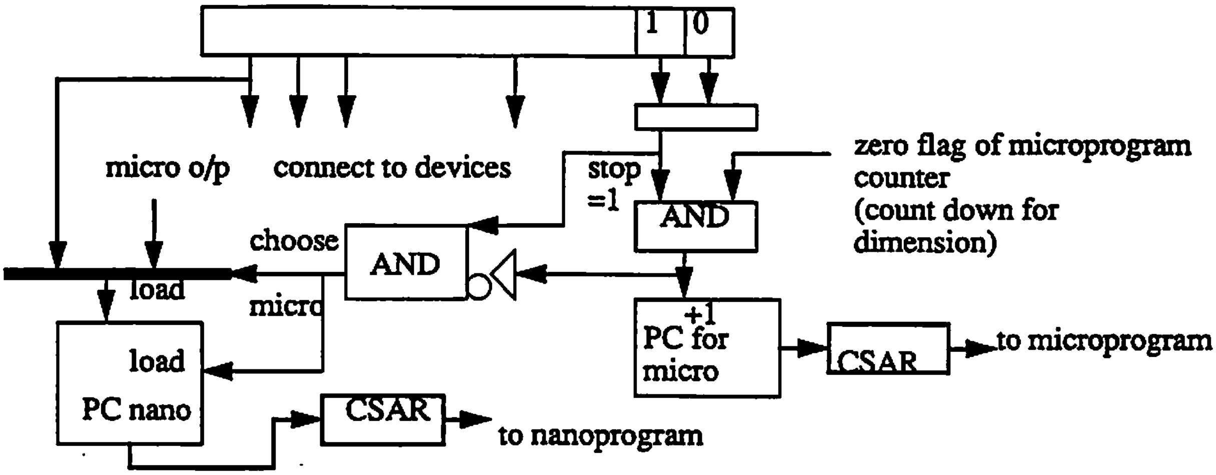


Fig 4. stop oeration

Fault tolerant for microprogram and counter:

1. fault tolerant for microprogram:

   . assume only one bit error or 2 bit adjacent errors considered.

   .use one parity bit for ease (m -out-of-2m could detect multiple adjacent errors, but complicated, and un-systematic)

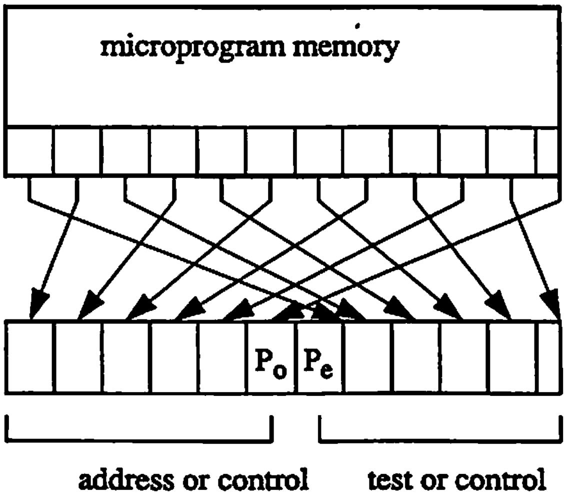


Fig 7 interleaved coding fault detection for microprogram

Weighted checksum error correcting code:

1. MVM:

$$\begin{pmatrix} A \\ ws1 \\ ws2 \end{pmatrix} \begin{bmatrix} x \end{bmatrix} = \begin{bmatrix} b \\ s1 \\ s2 \end{bmatrix}$$

ws1=w1*A, ws2=w2*A
S1=ws1*x, S2=ws2*x

2. OPU:

$$\begin{pmatrix} G_{N*N} \\ WS1_{1*N} \\ WS2_{1*N} \end{pmatrix} \begin{pmatrix} H_{N*N} \end{pmatrix} = \begin{pmatrix} \triangle W_{N*N} \\ S1_{1*N} \\ S2_{1*N} \end{pmatrix}$$

ws1=w1*G, ws2=w2*G
S1=ws1*H, S2=ws2*H

3.VMM:

$$\begin{pmatrix} X_{1*N} \end{pmatrix} \begin{pmatrix} A_{N*N} & WS1_{N*1} & WS2_{N*1} \end{pmatrix} = \begin{pmatrix} b_{1*N} & S1 & S2 \end{pmatrix}$$

ws1=A*w1, ws2=A* w2, s1= X*ws1, s2 = X*ws2
,

w1=[1,1,1....1], w2=[1, 2, 2^2, 2^3,..., 2^{N-1}],

R1=w1*X-S1, R2=w2* X-S2(depend on format of ws1,2)

IF R1 ≠ 0 and R2 ≠ 0 then R2/R1=2^{j-1} ,b_j error , and correct $b_j = b_j$ - R1.

Implementation in neural PE for case of R1:

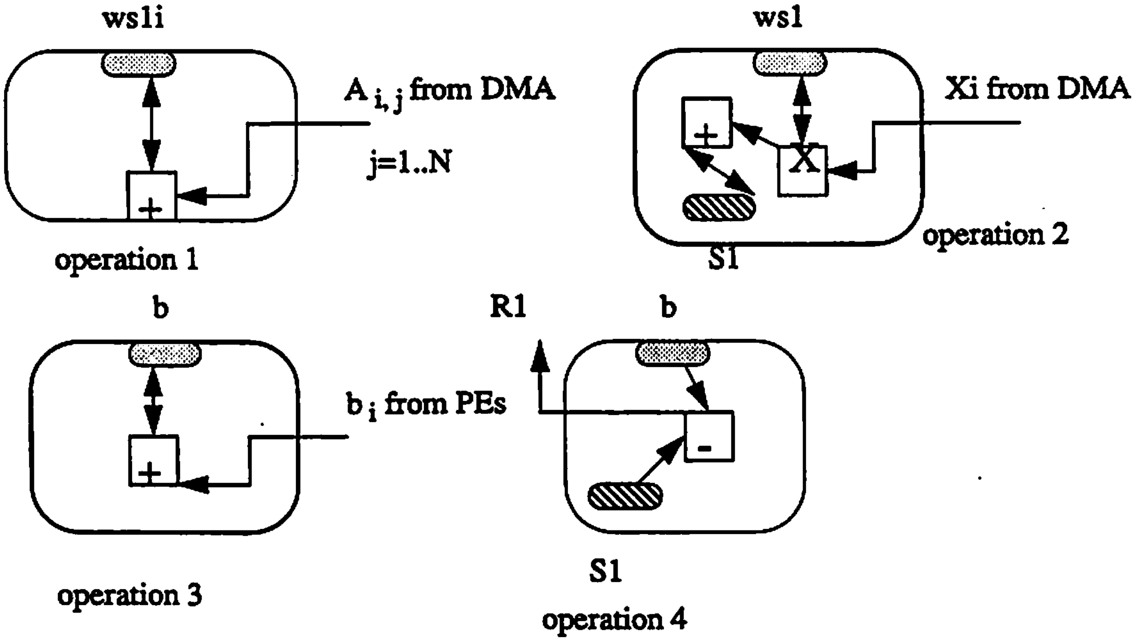


Fig 5. weighted check sum coding

Microprogramming for weighted check sum coding and fault tolerant:
take MVM as example:

flowchart

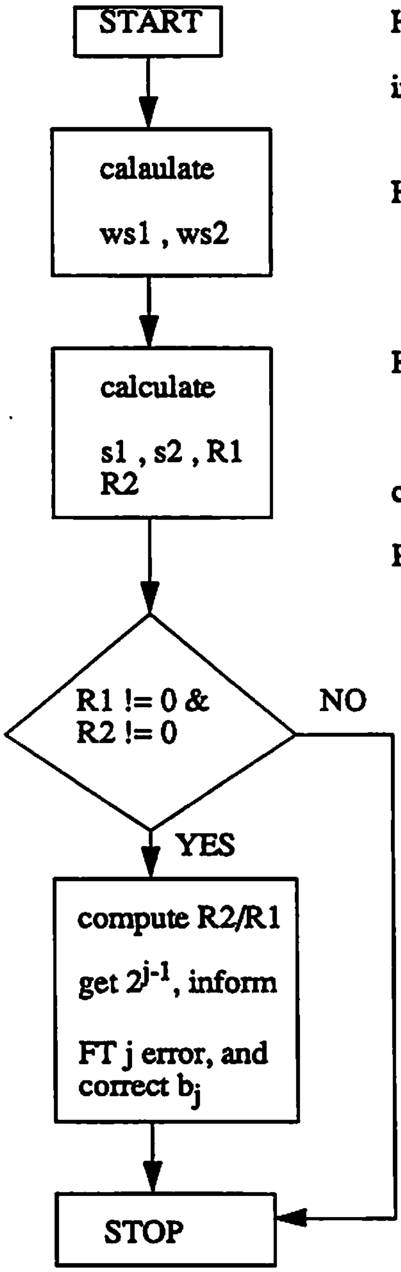


Fig 6. flowchart of weighted check sum code

processing time	microprogram
Host download the instruction.	
Host download A	1. operation 1
Host download X	2. opewration 2 for S1,2 3. operation 1 for w1*x , w2*x
collecting results from PEs	4. operation 3 for R1 , R2
after collection.	5. operation 4 for condition.

Analyze all the microoperation in the specific microprogram, there are 3 major operatios:

1: next address operation : next operation is placed at next address.

2: conditional branch operation: next operation's address depends on condition.

3: While do command: like loop operation, except testing at beginning of each loop to see continue or not.

2. fault tolerant for counter: also use single parity bit.

present	next	parity change
------- 0	------- 1	yes
----- 0 1	------ 1 0	no
---- 0 1 1	----- 1 0 0	yes
-- 0 1 1 1	--- 1 0 0 0	no

From the table, we find that

1). If the rightmost 0 occurs in an even-number bit, then parity change.

2). otherwise no.

parity generation implementation:

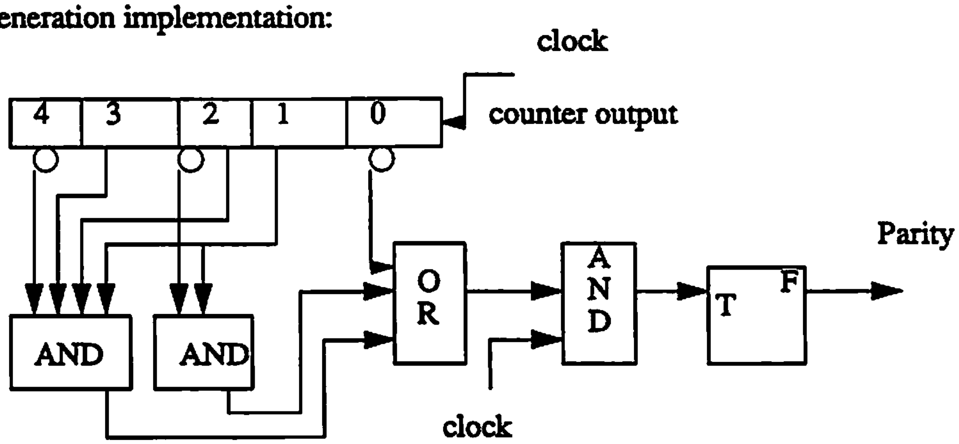
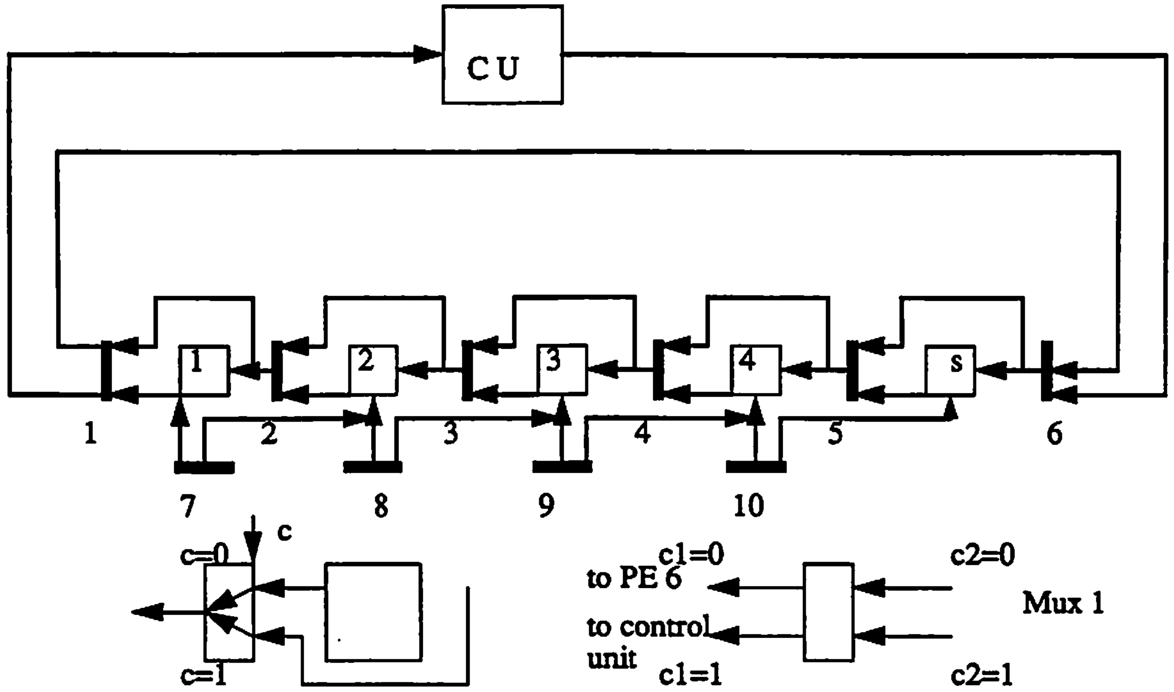


Fig 8. counter parity bit generation.

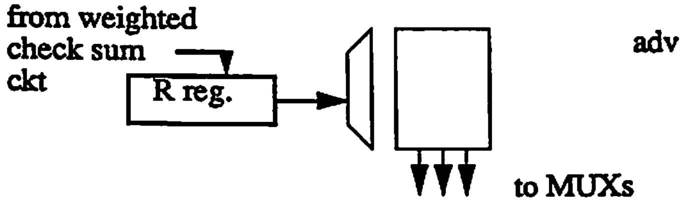
Recover circuit after knowing falty PE:



condition	1 c2 c1	2	3	4	5	6	7	8	9	10
normal	0 T	0	0	0	1	T	0	0	0	0
PE4 fault	0 T	0	0	1	0	T	0	0	0	1
PE3 fault	0 T	0	1	0	0	T	0	0	1	1
PE2 fault	0 T	1	0	0	0	T	0	1	1	1
PE1 fault	1 T	0	0	0	0	T	1	1	1	1

T=1: when vector is partitioned
T=0: otherwise

The above words can be directly stored as microprogram. (c1 , MUX 6 excluded).

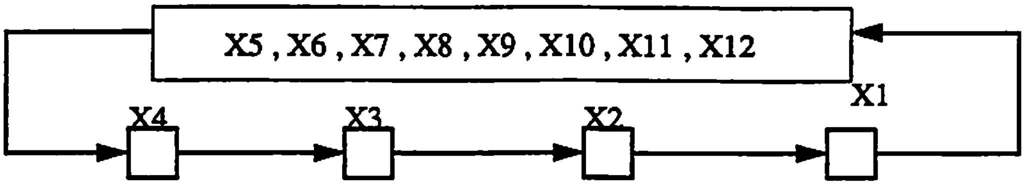


adv: easy to increase PE number.

Fig 9. recover y microprogram.

. Control unit:

MVM (matrix - vector multiplication) : [A]$_{12*12}$[X]$_{12*1}$ partition method:



W_{11}	W21	W3,1	W4,1
W_{12}	W22	W3,2	W4,2
.		.	.
$W_{1,12}$	W2,12	W3,12	W4,12
	.		
W_{51}	W61	W7,1	W8,1
.	.	.	.
.	.	.	.
$W_{5,12}$	W6, 12	W7,12	W8,12
$W_{9,1}$	W10,1	W11,1	W12,1
.	.	.	.
$W_{9,12}$	W10,12	W11,12	W12, 12

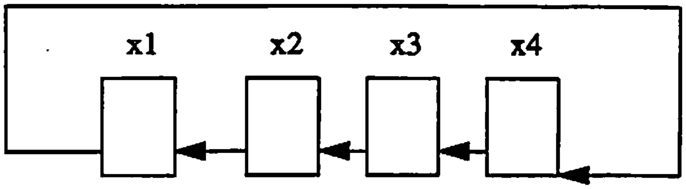
weght arrangement is different from original because serially not parallely input vector.

step	instruction from control	PE1	PE2	PE3	PE4
1	36 mvma to PE 1	mvm a	——	——	——
2	36 mvm a to PE 2	mvm a	mvm a	——	——
3	mvm a to PE 3 36	mvm a	mvm a	mvm a	——
4	36 mvm a to PE 4	mvm a	mvm a	mvm a	mvm a
....	——	mvm a	mvm a	mvm a	mvm a
37	mvmb 3 to PE1	mvm b	mvm a	mvm a	mvm a
38	mvmb 3 to PE 2	mvm b	mvm b	mvma	mvm a
39	mvmb 3 to PE 3	mvm b	mvm b	mvm b	mvm a
40	mvmb 3 to PE 4	——	mvm b	mvm b	mvm b

mvm a: weighted sum processing, mvm b : threshold activation. 36, 1 means dimension.

could use microprogram to do control .

Fig 10. control unit microprogram.



C_{11}	C_{22}	C_{33}	C_{44}
C_{12}	C_{23}	C_{34}	C_{41}
C_{13}	C_{24}	C_{31}	C_{42}
C_{14}	C_{21}	C_{32}	C_{43}
C_{51}	C_{62}	C_{73}	C_{84}
C_{52}	C_{63}	C_{74}	$C_{8\ 1}$
.	.	.	.
C_{91}	$C_{10\,2}$	$C_{11\,3}$	$C_{12\,4}$
.	.	.	.
.	.	.	.
C_{N4}	0	0	0

image(i)=(x1 x2 x3 x4')^t

codewords = ($C_{11}\ C_{12}\ C_{13} \dots C_{1N}$

$C_{21}\ C_{22}\ C_{23} \dots C_{2N}$

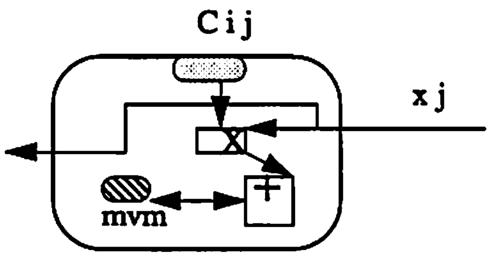
.

.

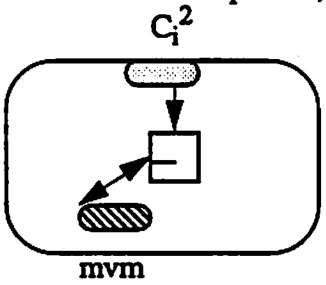
$C_{N1}\ C_{N2}\ C_{N3} \dots C_{NN}$)

min [(image-C_i)^2]
=Max [2C_i image - C_i^2]

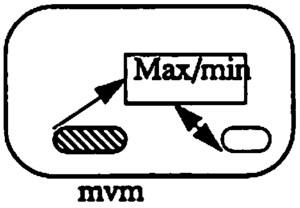
assume :$C_i^2$is known
(If unknown, use original equation)



operation in PE i



operation in PE i



operation in PE i

after N iterations o/p M/m

Fig 12. one of the application except RBP, HMM of systolic neural circuit.

Term Project for EE599

Report writen by

Juinn-Yan Chen

VLSI Implementation for Systolic Neurial Network

Introduction

The original idea of this report comes from the publication of S.Y. Kung, " A systolic Neurial Network Architecture for Hidden Markov Models", IEEE Transactions on Acoustics, Speech and Signal Processing. vol 37, No 12. DEc 1989.

In S.Y. Kung's paper , he advocates a systolic neurial network architecture for implementating the hidden Marcov Models (HMM's). He also gives a unified algorithm formulation for recurrent back-propagation (RBP) network and HMM's for architectural design.

However, we find this architecture is very powerful and not just only suitable for Artificial Neurial Network (ANN) but also signal processing.We can find the data processing portion between ANN and Signal Processing is very similar. Therefore, we want to modify this structure to adapt more applicastions.

Work Partition

This work will be acomplished by two-people team work. One is for the overall system architecture design. One is for the design of Processor Element (PE).

Teho Chen will be responsible for the overall system architecture which includes

1 . Host computer.

2 . Ring Controller.

3. Fault Tolerance Architecture.

His work will concentrate on finding algorithms for a specific application , and implement these algorithms in a systolic procedure, then send macro commands to each PE and monitor its opeartion.

I will be responsible for the VLSI Implemetation of the PEwhich includes

1 . Control Portion.

2 . Data Communication Portion.

3 . Data Processing Portion.

My work will concentrate on decompose the mavro command received from Host Computer into several micro commands , find algorithms for these micro commands then implement them with a systolic procedure.

Algorithm

It can be shown that operations in both the retrieving and learning phase of RPB'sand HMM's can be formulated as consecutive matrix-vector multiplication (MVM) , consecutive outer-product updating (OPU) and consecutive vector-matrix multiplication (VMM) problems. In terms of the array structure all these formulations lead to a universal ring systolic array architecture. We can find all these operations call for an MAC (multiplication and accumulation) processor. So, we can integrate the three architectures for these operation into one processor element.

Below are the algorithm for MVM, OPU and VMM

MVM :

$$u_i(l+1) = \Sigma\, w_{ij} a_j(l+1)$$

$$a_i(l+1) = f_i(u_j(l+1), \theta_j(l+1))$$

where u_i is the net inputs, θ_j is the external inputs, fi is the nonlinear activation function, a_i is the activation function, w_{ij} is the weighting function.

OPU :

$$\Delta w_{ij} \Leftarrow \Delta w_{ij} + g_i(l+1)h_j(l+1)$$

$$w_{ij} \Leftarrow w_{ij} \oplus \eta\Delta w_{ij}$$

where ⊕ is a minus opoeration in RBP's and a multiplication operation in HMM's

and $g_i(l+1)=\delta_i(l+1)f'(l+1)$, $h_j(l+1)=a_j(l)$ for RBP's

$g_i(l+1)=\delta_i(l+1)f_i(\theta(l+1))$, $h_j(l+1)=a_j(l)$ for HMM's

VMM :

$$\delta_i(l) = \Sigma\, g_j(l+1) w_{ij}$$

where δ_i is the back-propagated corrective signal.

* Overlook for the overall system

It's clear that we can have more operations except these three operation. However, for a ANN application, these will be enough. So, we will concentrate on integrating these three operations into a single PE. Other algorithms such as FFT, Convolution and Viterbi are mainly composed of MVM, OPU and VMM operation. Therefore, it is desirable to design a programmable PE that will be used not only in ANN.

** General Operatio

For this purpose , we devided the overall system into Host Computer, Ring Controller, PE and connection. The Host Computer will analysis whole problem and choose a specific program to work. This specific program will be decomposed into consecutive MVM, OPU and VMM operations. And these operatioins will be recognized as macro commands.

* First Version of micro instruction

The PE will receieve macro command and decompose this command into several micro instructions. These micro instructions will control the switch of bus to ensure accurate data flow and processing.

* RISC approach of micro instruction

After we got the original micro instruction steps , we found there are many overlap ther.

Therefore , we can redece these instruction steps into compact instructions. We should notice the data cannot be the internal bus simutaneously. Besides, the multiplication and the addition should be seperate apart at least one step.

* Pipeline apoproach

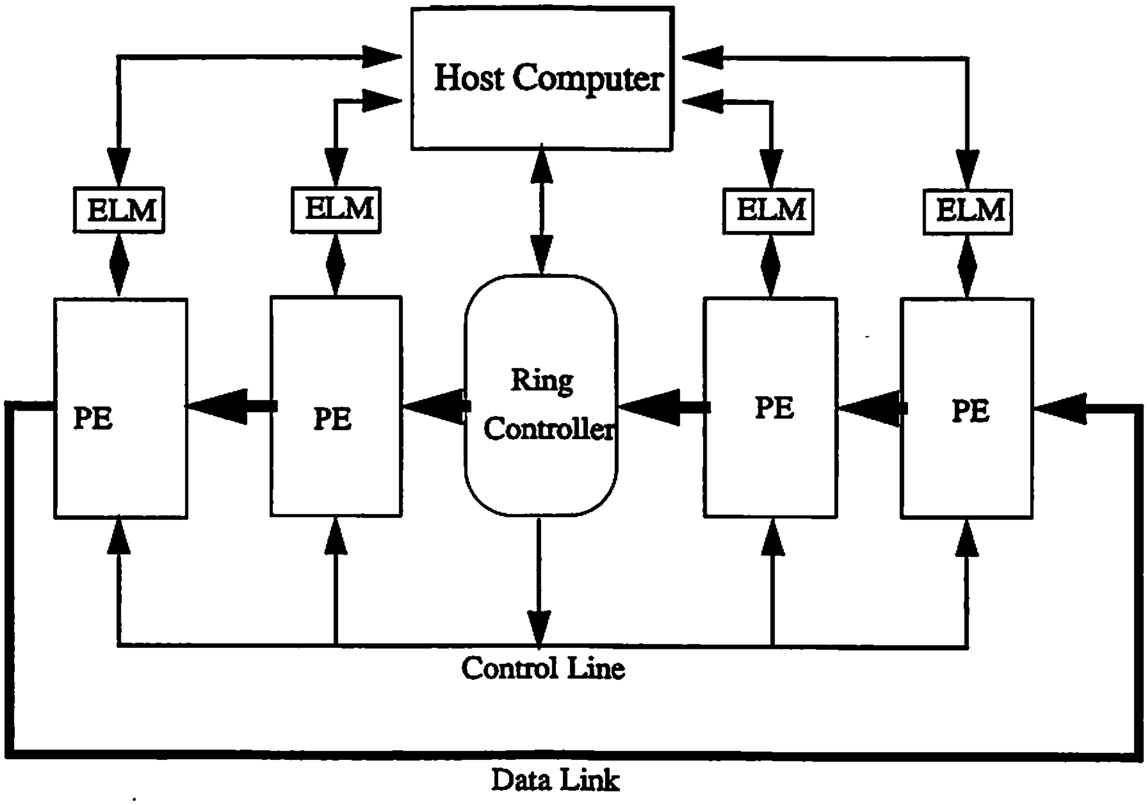
In order to increase the speed of the processor, we can break the internal bus into more bus line. Then we can overlap more steps together because they do not occupy the same bus.

* Circuit Design Consideration

* The speed of the PE will be limited by multiplier. We choose wallice multiplier . Because wallice structure can be seprated into 4*4 muiplier block and wallice tree block, we can add a pipeline procedure here and increase the speed of the PE. Actually , in this report ,we do not do pipeline here.

*Adder will be Carry Look Ahead Adder

* Register and memory will use standard cell design.

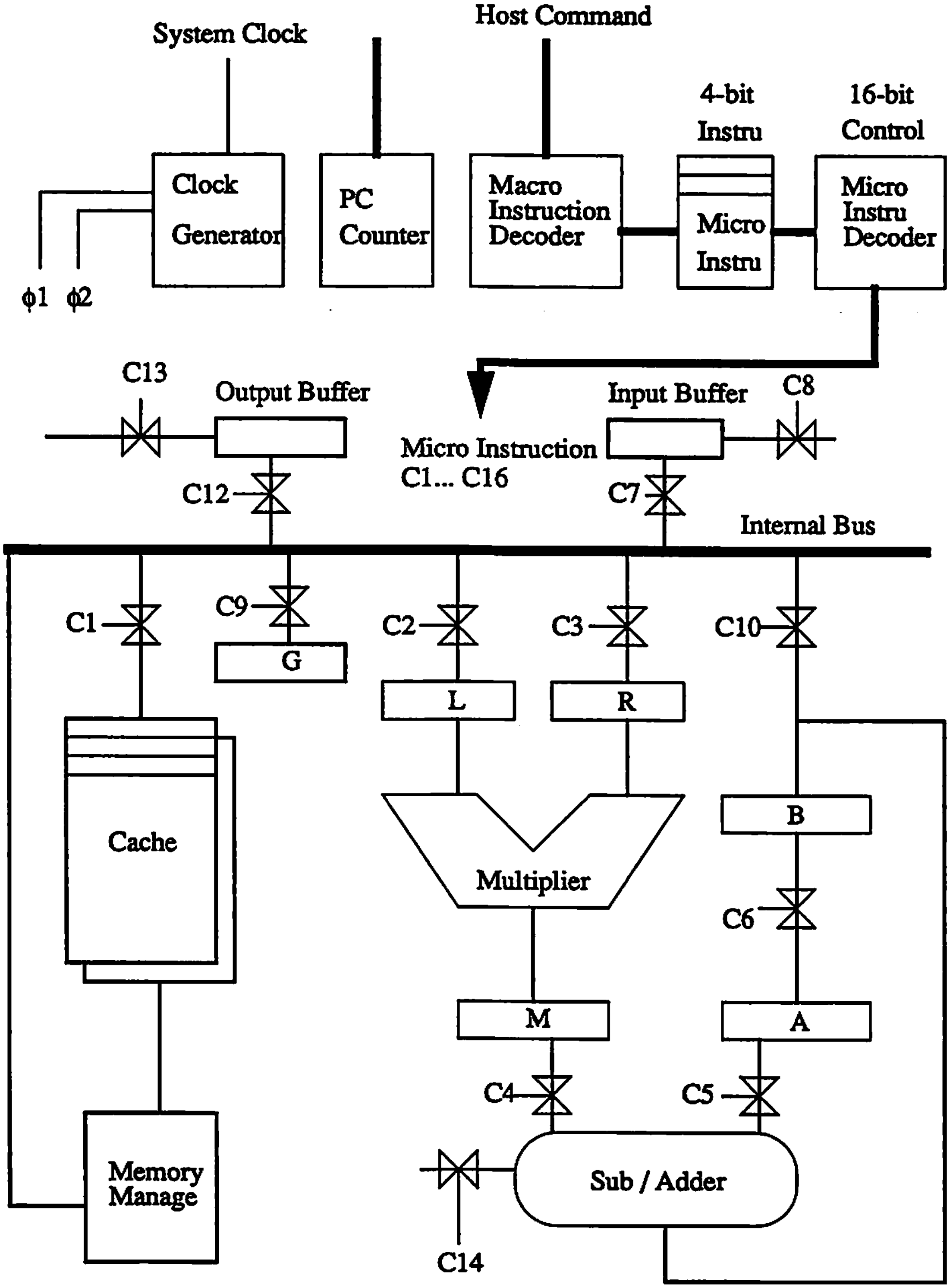


Overall System Architecture for Ring Systolic Array

* Host Computer load data and commands into PE through Ring Controller.

* For a specific algorithm, Host Computer will chose a specific program composed of macro command sequences.

* Data such as w_{ij} , Δw_{ij} , g_i , h_j , δ_j and η...will be loaded into External Local Memory from Host Computer directly.



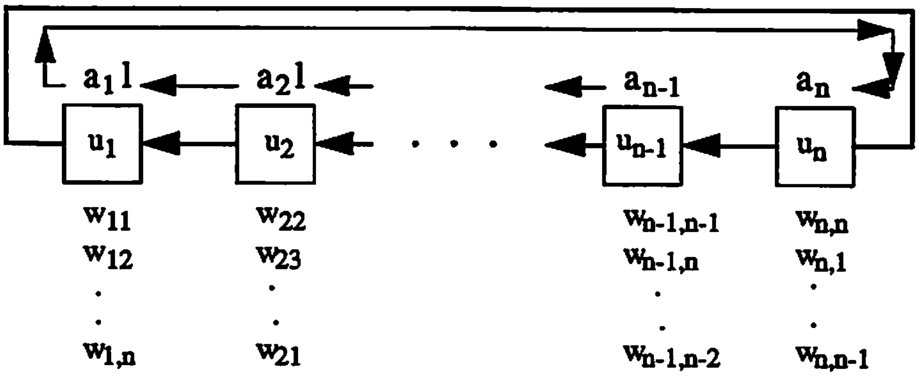
Processing Element Building Blocks for Ring Systolic Array

MVM OPERATION

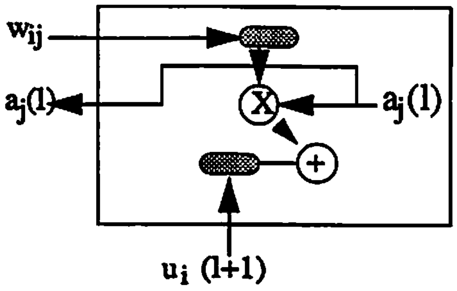
* Algorithm

$$u_i(l+1) = w_{ij}a_j(l)$$

* Associative Ring Structure



* Associative Data Processing Structure



* Associative Original Micro Instruction Bus Control

0 initialize zero A=0 B=0

Step	Micro instru description	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	c11	c12	c13	c14
1	load $a_j(1)$								1						
2	a_j (l) ⟶ L ,output		1					1					1		
3	w_{ij} ⟶ R	1		1											
	Do multiply														
4	$w_{ij}\, a_j(1) + u_i\,(l+1)$ ⟶ B				1	1									
	Do addition														
5	B ⟶ A ($u_i(l+1)$)					1									

* Associative Reduced approach of Micro Instruction

Step	Micro instru description	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	c11	c12	c13	c14
1’	Group 1 , 2 , 4		1		1	1		1	1				1		
2’	Group 3 , 5	1		1			1								

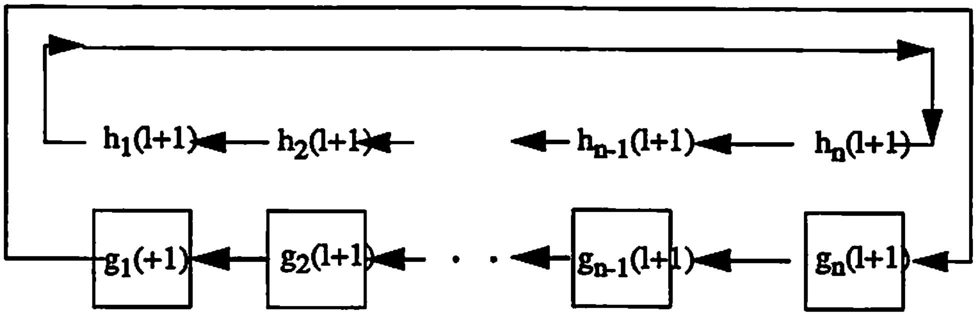
* Time comsumiing 1x addition + 1xmultiplication

OPU OPERATION

* Algorithm :

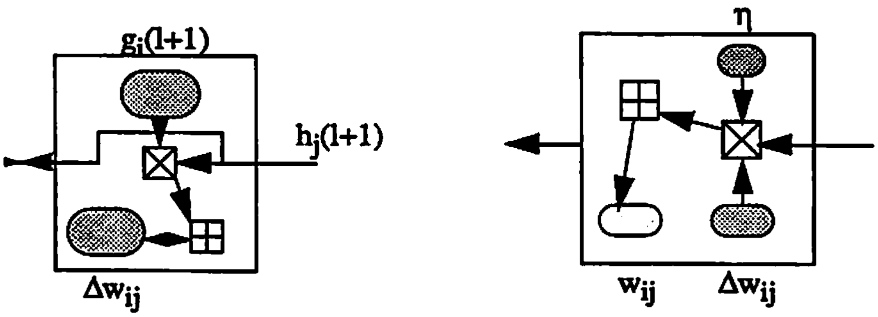
$$\Delta w_{ij} \Leftarrow \Delta w_{ij} + g_i(l+1)h_j(l+1)$$
$$w_{ij} \Leftarrow w_{ij} \oplus \eta\Delta w_{ij}$$

* Associatative Ring Structure :



w_{11}	Δw_{11}	w_{22}	Δw_{22}	$w_{n-1,n}$	$\Delta w_{n-1,n-1}$	$w_{n,n}$	$\Delta w_{n,n}$
w_{12}	Δw_{12}	w_{23}	Δw_{23}	$w_{n-1,n}$	$\Delta w_{n-1,n}$	$w_{n,1}$	$\Delta w_{n,1}$
$w_{1,n}$	$\Delta w_{1,n}$	w_{21}	Δw_{21}	$w_{n-1,n-2}$	$\Delta w_{n-1,n-2}$	$w_{n,n-1}$	$\Delta w_{n,n-1}$

* Associatative Data Processing Structure



* Associative Original Micro Instruction Bus Control

Step	Micro instru description	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	c11	c12	c13	c14
1	load g_i(l+1) to L from G		1							1					
2	read h_j(l+1) to buffer								1						
3	h_j(l+1) → R,buffer			1				1					1		
	Do multiplication														
4	read Δw_{ij} from cache	1									1				
5	Δw_{ij} → A						1								
6	Do addition → B				1	1									
7	restore Δw_{ij} to cache,R	1		1							1				
8	load η from G		1							1					
	Do multiplication														
9	load w_{ij} from cache	1									1				
10	w_{ij} → A						1								
11	Do addition				1	1									

restore w_{ij}

* Associative Reduced Micro Instruction

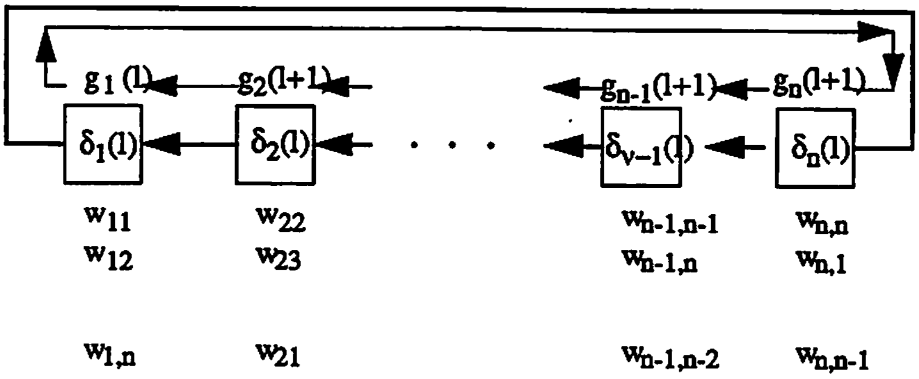
Step	Micro instru description	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	c11	c12	c13	c14
1'	Group 1 , 5		1				1			1					
2'	Group 2 , 3 , 10			1				1	1				1		
3'	Group 4 , 6	1			1	1					1				
4'	Group 7 , 11	1		1							1				
5'	Step 8		1							1					
6'	Step 9	1									1				

VMM OPERATION

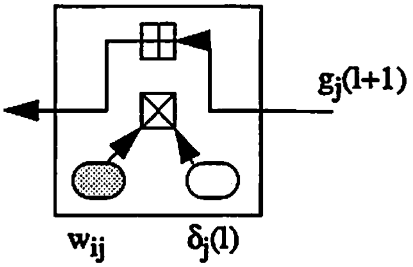
* Algorithm:

$$\delta_i(l) = \sum_{j=1}^{n} g_i(l)\, w_{ij}$$

* Associative Ring Structure



* Associative Data Processing Structure



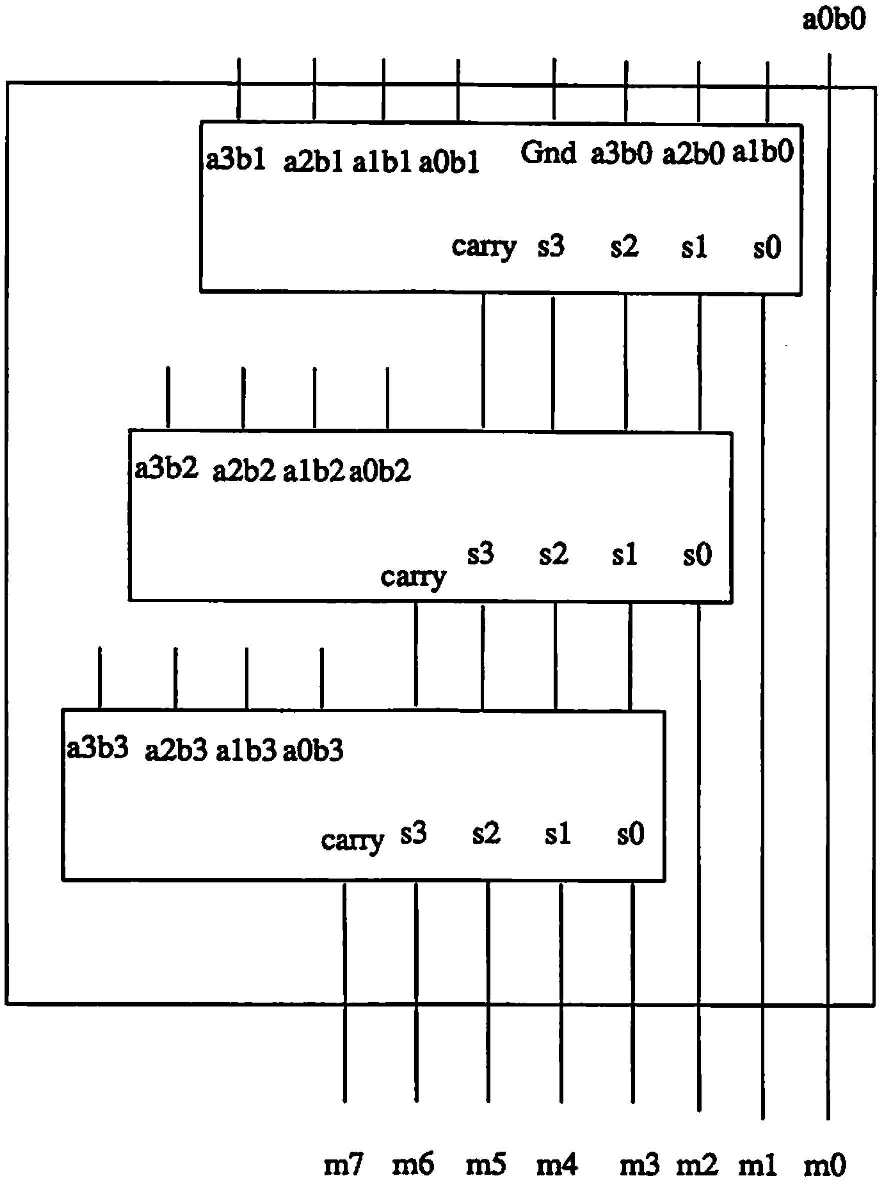
* Associative Original Micro Instruction Bus Control

Step	Micro instru description	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	c11	c12	c13	c14
1	read g_j (l+1) to buffer								1						
2	g_j (l+1) → L,output		1					1					1		
3	feach w_{ij} from cache to R	1		1											
	Do multiplication														
4	Do addition				1	1									
5	B → A						1								

* Associative Reduced approach of Micro Instruction

Step	Micro instru description	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	c11	c12	c13	c14
1'	Group 1 , 2 , 4		1		1	1		1	1				1		
2'	Group 3 , 5	1		1			1								

* Time comsumiing 1x addition + 1xmultiplication

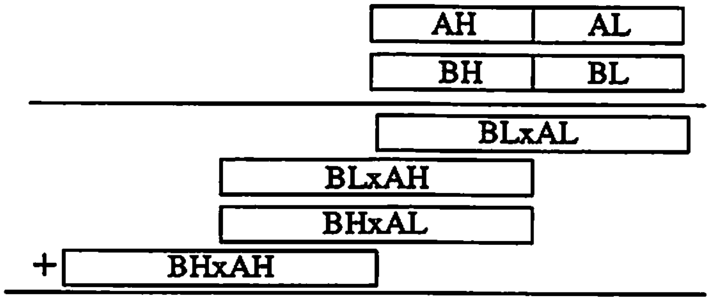


Building Block of 4x4 multiplier

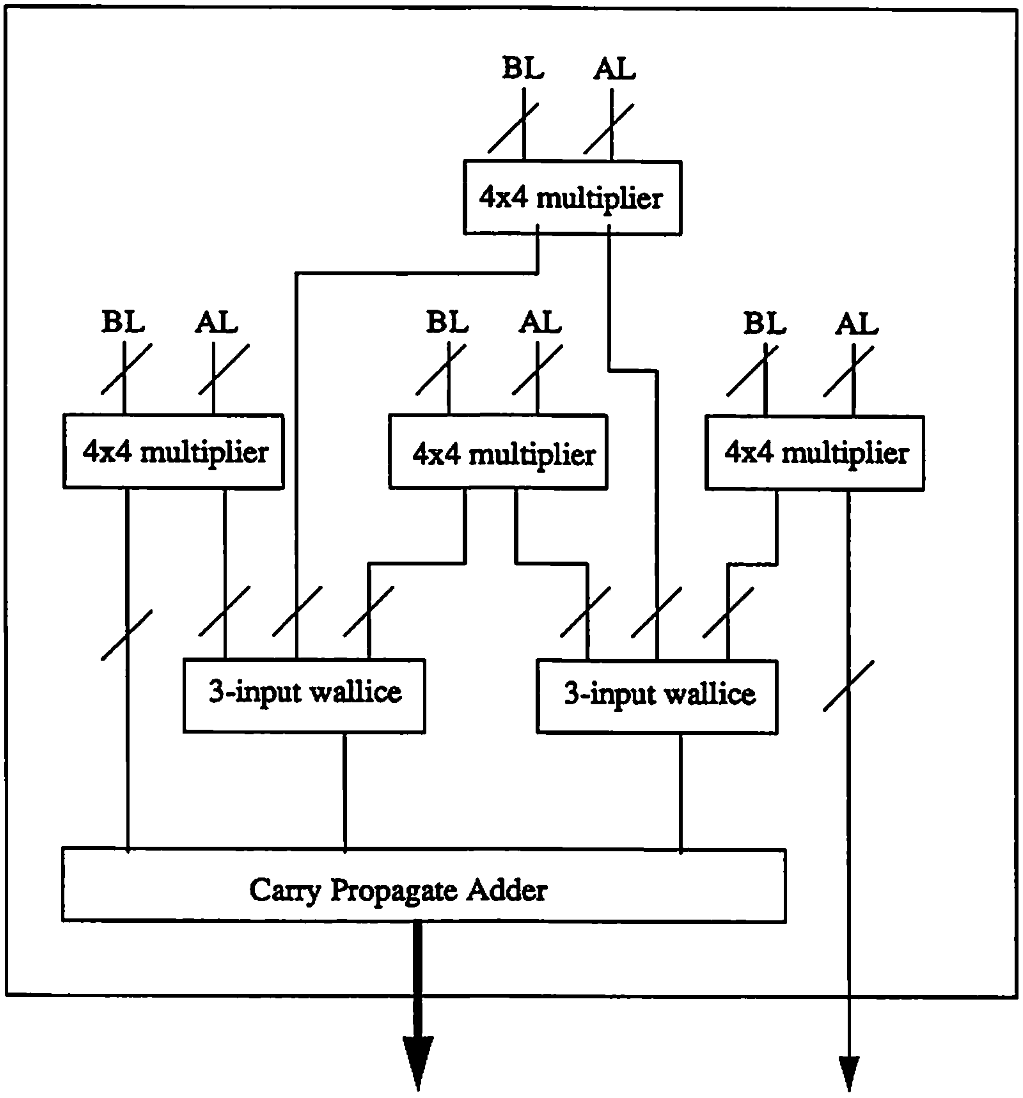
* the three small building blocks are 4-bit carry look ahead adder

* time comsuming is (1xAND gate delay + 3x Carry-Look -Ahead-Adder delay)

* Algorithm for an 8x8 Wallice multiplier



* For an 8x8 multiplier, we need 4x4 multiplier, 3-input wallice tree, and a carry propagate adder.



EE599
Digital VLSI Neurocomputer Design
Spring 1991

Mike Wang
573-75-0171

ABSTRACT

In this project, a general purpose 4-input, 4-output, 3-layered, fully-connected Back Error Propagation network is realized by using array processors. This chip is designed to be scalable and reconfigurable meaning that a network of any number of layers, inputs, outputs and connections can be realized using multi-chip configuration. Fast on-chip learning is also provided with each layer's weights updated simultaneously. Some measure of fault tolerance is also incorporated for this chip to prevent process time error from rendering the chip useless.

Table of Contents

In this project, a general purpose 3-layer, 4-input, 4-output Back-Error Propagation network is realized by using array processor design. The choice is made due to its following advantages:

(1) fast forward propagation time: the I/O can be pipelined in and out of the chip with pipeline period $\alpha = 1$.

(2) it offers on-chip, local, simultaneous weight updates. This enhances the learning speed of the algorithm.

(3) weights can be both loaded and unloaded before and after training process, so one can store the trained weights for different applications and just load the appropriate weight among different learned applications.

Two 4x4 arrays of PE's are used to realize the network. As S.Y. Kung suggested, each array can be further projected into a 1X4 systolic array. However, as pointed out in the appendix, this method suffers the drawback that the I/O can not be pipelined, and the I/O rate is effective 1/4 of that of the 4x4 design. There is thus a trade-off between speed and area used. Moreover, the systolic array design will be unsuitable for training as it has longer training time and will have to use either additional dedicated circuitry or off-chip learning. The 4x4 array is very suitable for the learning process as the data for backward propagation actually travel along the transpose of the array. It is also shown in the appendix that as the problem size grows, the time required for learning actually grows almost linearly . This is again a very good reason for choosing this design method.

As compared with some commercial products, this chip definitely has less number of neurons and synapses. However, it can be easily explained as follows:

(1) this design tries to maximize the speed of operation at a cost of area used. Most of the commercial chips use memory to store information of synapse weights and neuron outputs, and the PE's are actually shared among neurons. However, that will result in a slower chip, especially for learning.

(2) because actual layout of chip is not done, so only an estimation of the numbers of PE's is given. From the EE599 VLSI for DSP class project, I found out that a PE with an 8 bit multiplier, a 16 bit adder, and many latches and registers will occupy approximately 1020λ $x1459\lambda$ of area. If 1um technology is used, a PE will results in 0.015 cm^2 of area.

So two 4x4 array will result in about 0.5 cm^2 of area. Therefore, conservative estimation was given for the array size thus the number of neurons and synapse weights.

(3) since this chip is scalable and reconfigurable, multi-chip configuration can be used to increase the network to any size the problem requires.

Moreover, from the VLSI DSP class project, it was found out that the PE can finish all required operation within 20ns. Thus, a 50 Mhz system clock is used for this chip. However, because of this chip is designed to be scalable, the number of pins on the chip is large. In order to constrain the pin number to be within 300, the I/O is forced to multiplex its higher byte with its lower byte thus results in an I/O rate of 25 Mhz. This is still in reasonable range of speed. This is a trade-off between the number of pins on IC (the scalability) and the speed of I/O operation.

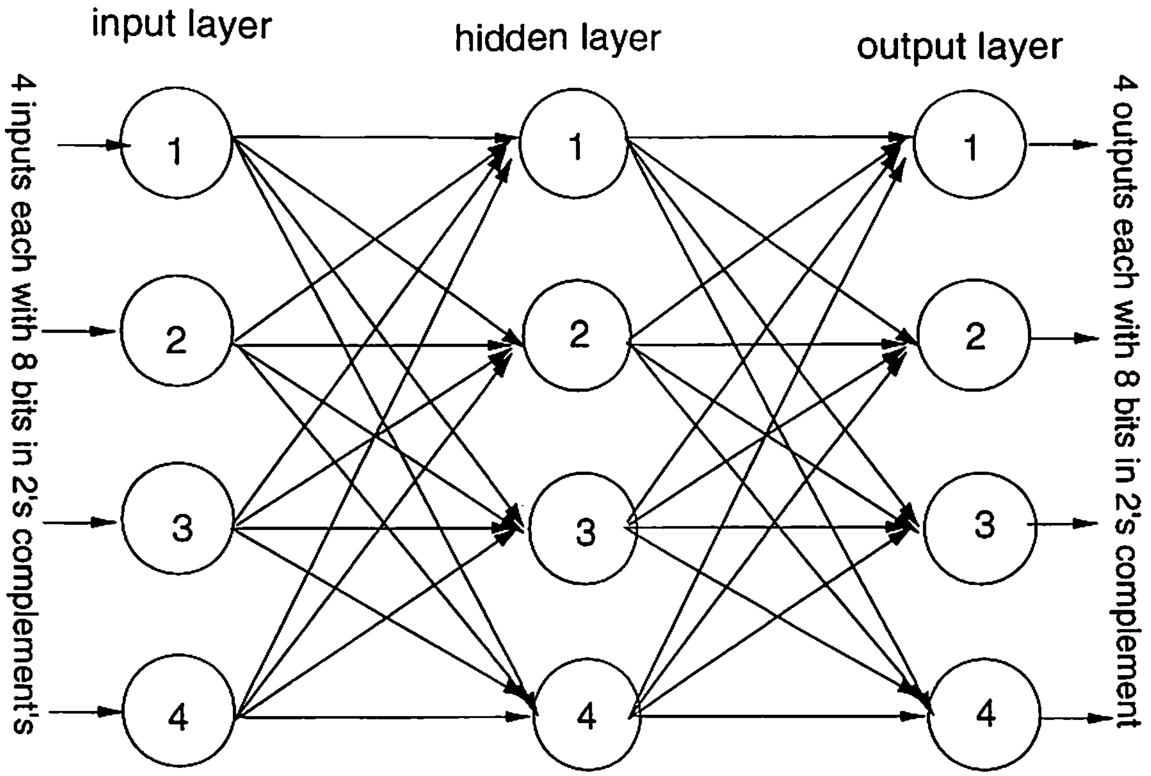
Since each PE must be able to take data for both the forward and backward operations, it must have 16-bit data bus for all its 4 sides. Some kind of routing is required within each PE, and special design consideration is also given as some 16 bits and 8 bits data lines have to connect. Because each of the 4 sides of a PE has a 16-bit data bus, the number of pins on the chip is therefore large as seen from the chip floor plan. This can be improved if the weights within the PE arrays can be switched to realize the transpose of original array. However, additional connection and logic circuit will be required. So for simplicity, it is not implemented this way.

Each of the sigmoid function and its prime is realized by using table look-up from a 16x8 ROM circuit on the chip. I chose 16 points because I want to use as small of an area as possible to realize these functions. The results are shown in appendix with quantization error also. One can increase the precision easily by following the analysis in the appendix.

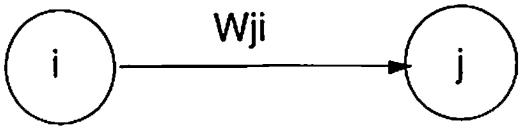
Even though the array processor approach looks straightforward on the surface, beneath it are many design choices, trade-offs and difficulties. There remain many optimizations to be done in both the system level and physical layout level. Nevertheless, the proposed architecture provides a very good starting point for a full IC design process.

Appendix 1

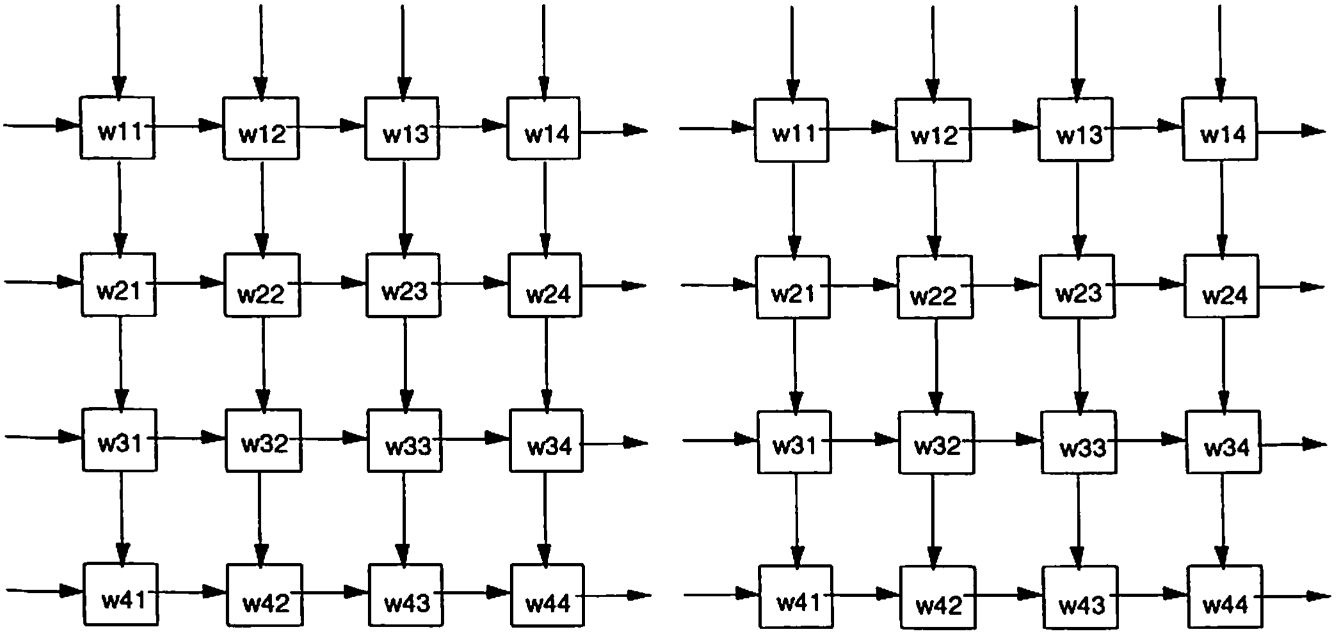
Network topology of a Back-Error Propagation Network


input layer
hidden layer
output layer
4 inputs each with 8 bits in 2's complement's
1
2
3
4
1
2
3
4
1
2
3
4
4 outputs each with 8 bits in 2's complement

THE WEIGHTS BETWEEN 2 NEURONS ARE DEFINED AS THIS



the 16 weights between each 2 layers are embedded in the array of PEs

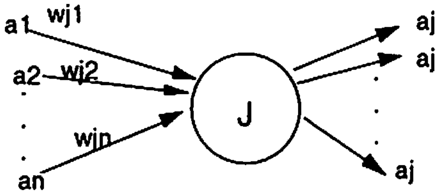


NETWORK TOPOLOGY

For Backpropagation network

Forward Operation:

the data propagate from input to output layer.



for input unit, output of neuron i= input at neuron i =ai
for hidden and output layers, the neuron output is calculated as follows:

$$S_j = \sum_i a_i w_{ji}$$

$$a_J = f(S_j) = \frac{1}{1+\exp(S_j)}$$

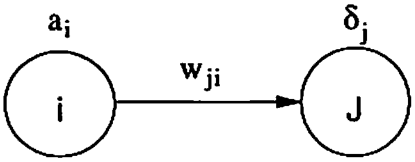
Backward Operation:

the data propagate from output layer back to input layer.

for output layer:

$\delta_j = (t_j - a) \quad f'(S_j)$ where t_j is the trained pattern

for all hidden layers, it is necessary to change weights between it and the previous layer first , then one can calculate the delta for the hidden layer.



$\Delta w_{ji} = \eta\ \delta_j a_i$ the Delta rule
η is the learning rate

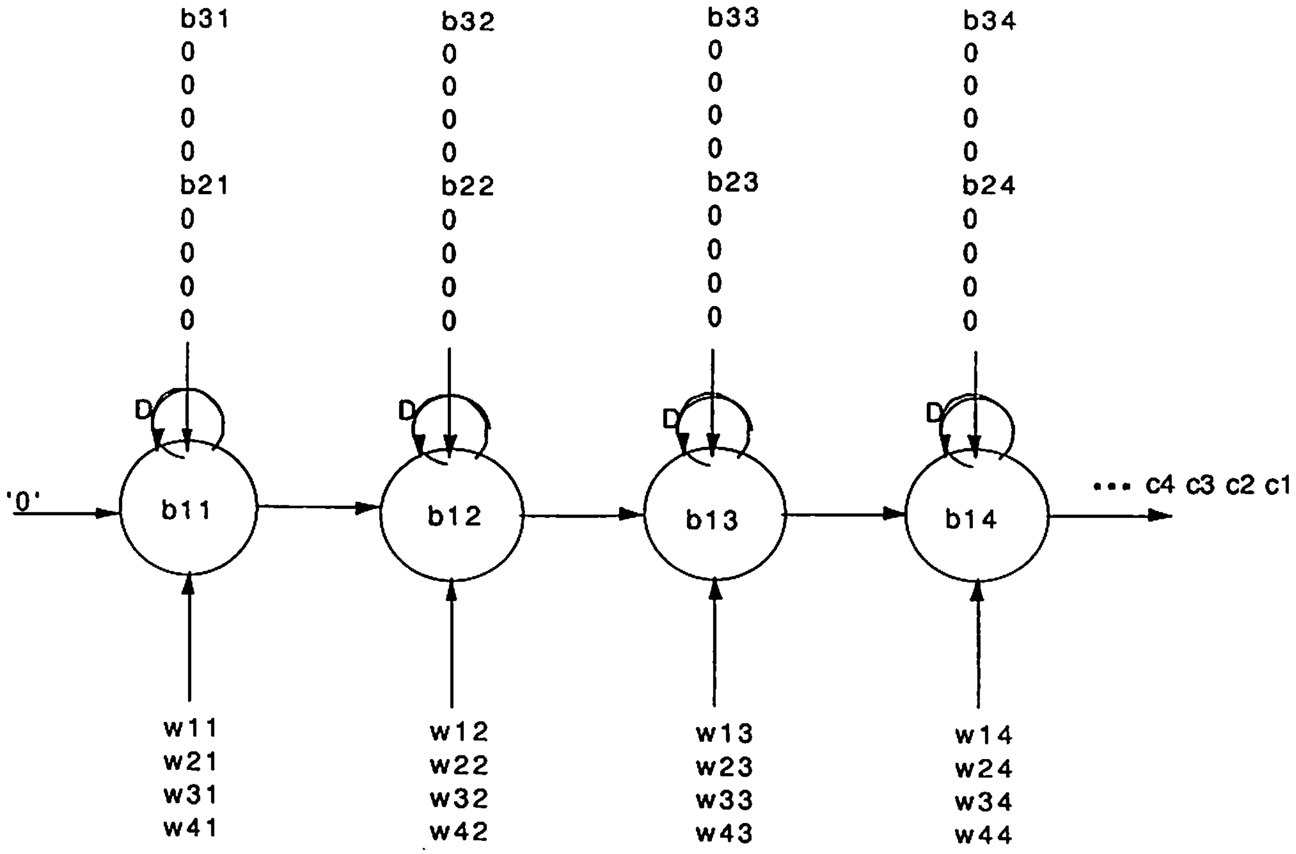
for the hidden layer:

$$\delta_j = \left[\sum_k \delta_k w_{kj}\right] \quad f'(S_j)$$

NOTE: for backward propagation, the transpose of the W matrix is used while for forward operation it is W that is used

Appendix 2

Why Not a Systolic Array



[W] x [b] = [c]

where bi1 - bi4 represent the ith set of data

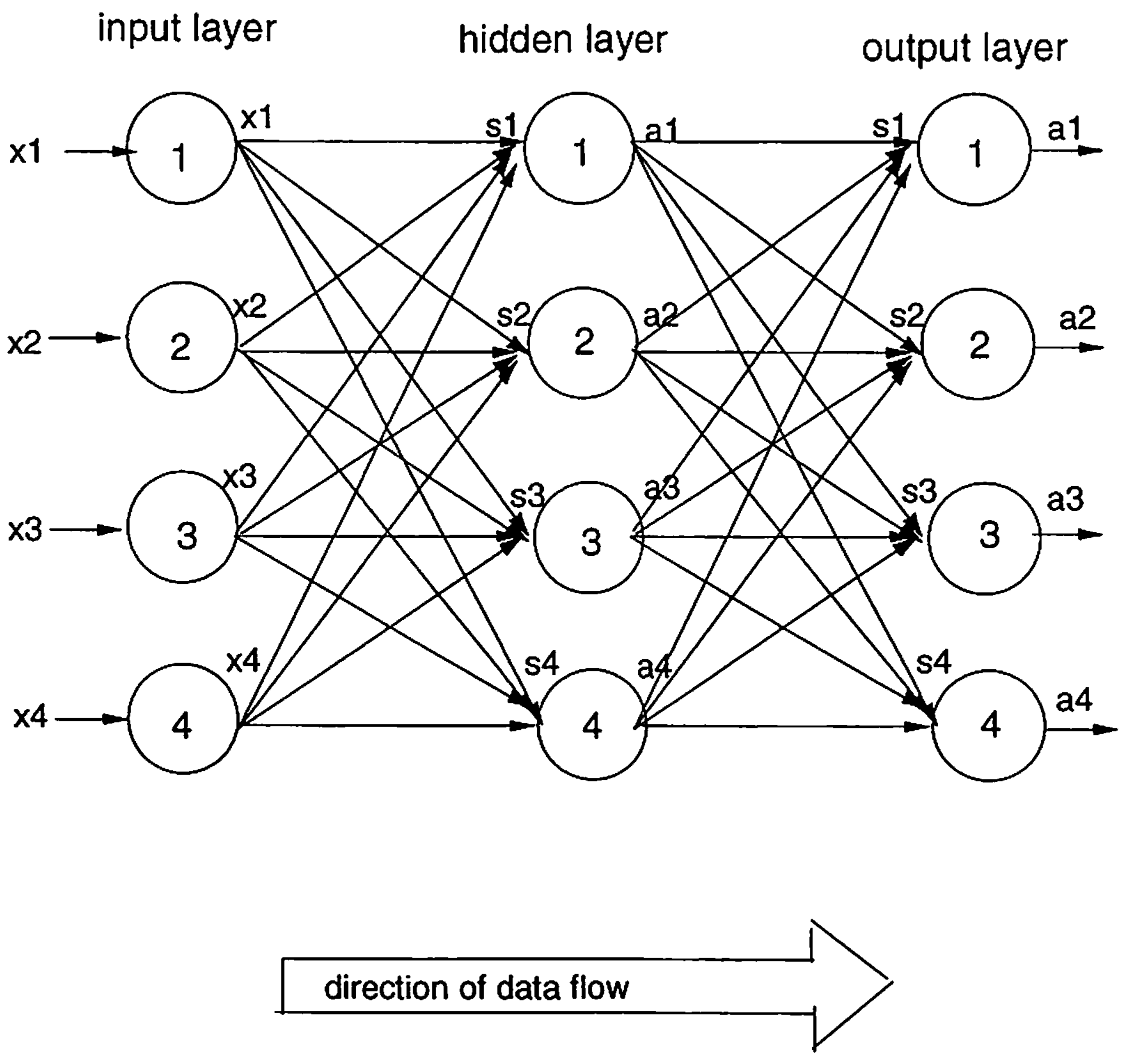
As can be seen from above, even though the output is one number each clk cycle, the input can only be applied every 4 clk cycles-- it is not pipelined. So the effective rate for input and output is 1/4 of that using a 4x4 array. This may render the chip unsuitable for real-time processing. This is a trade off between hardware and speed.

Moreover, the systolic array is not suitable for calculating weight changes, and dedicated hardware or off-chip learning may become necessary. So the 4x4 array actually makes design simpler and learning faster than what the systolic array can offer. There is again some trade off of hardware vs.simplicity and speed.

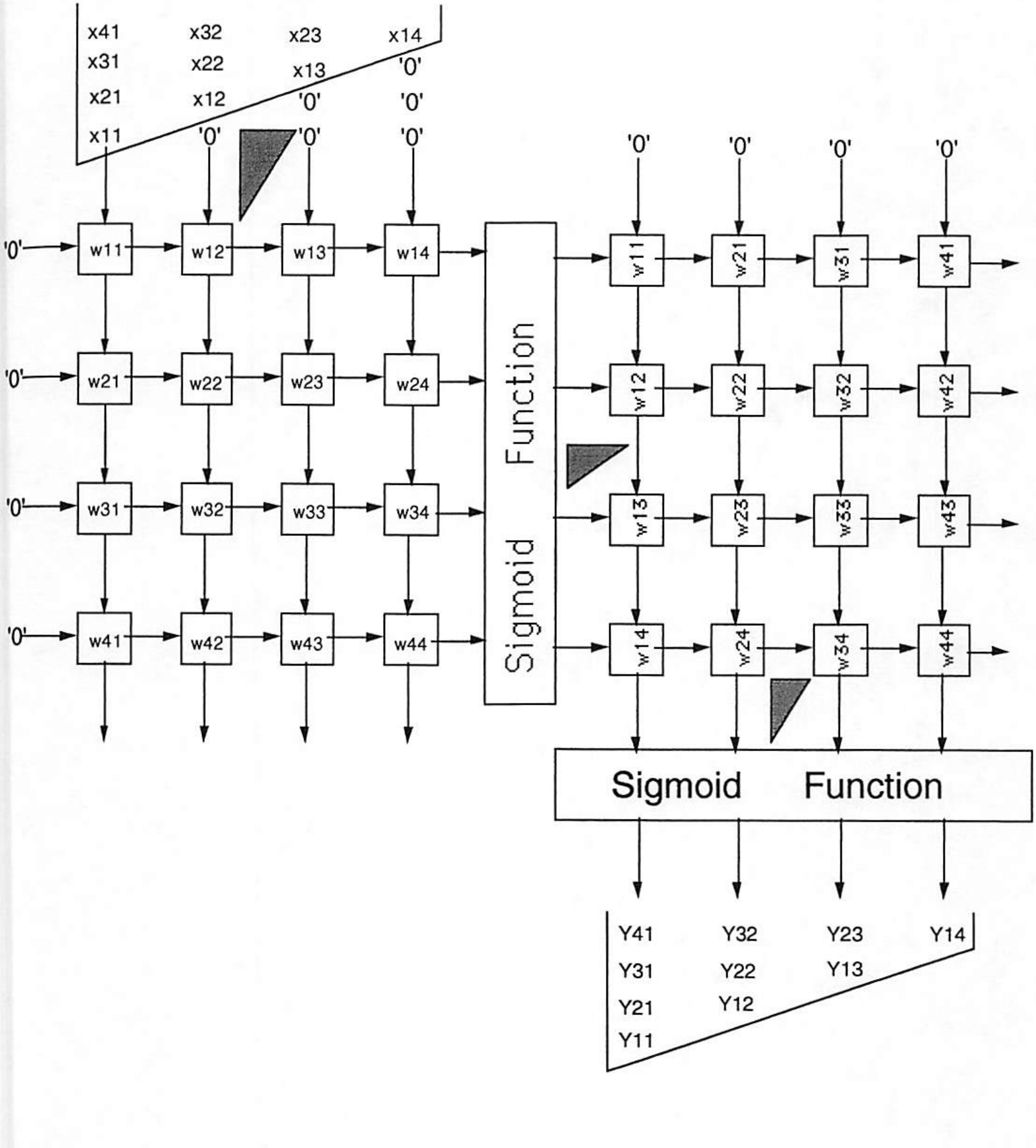
Appendix 3

Forward / backward operations and their data flow path in PE arrays

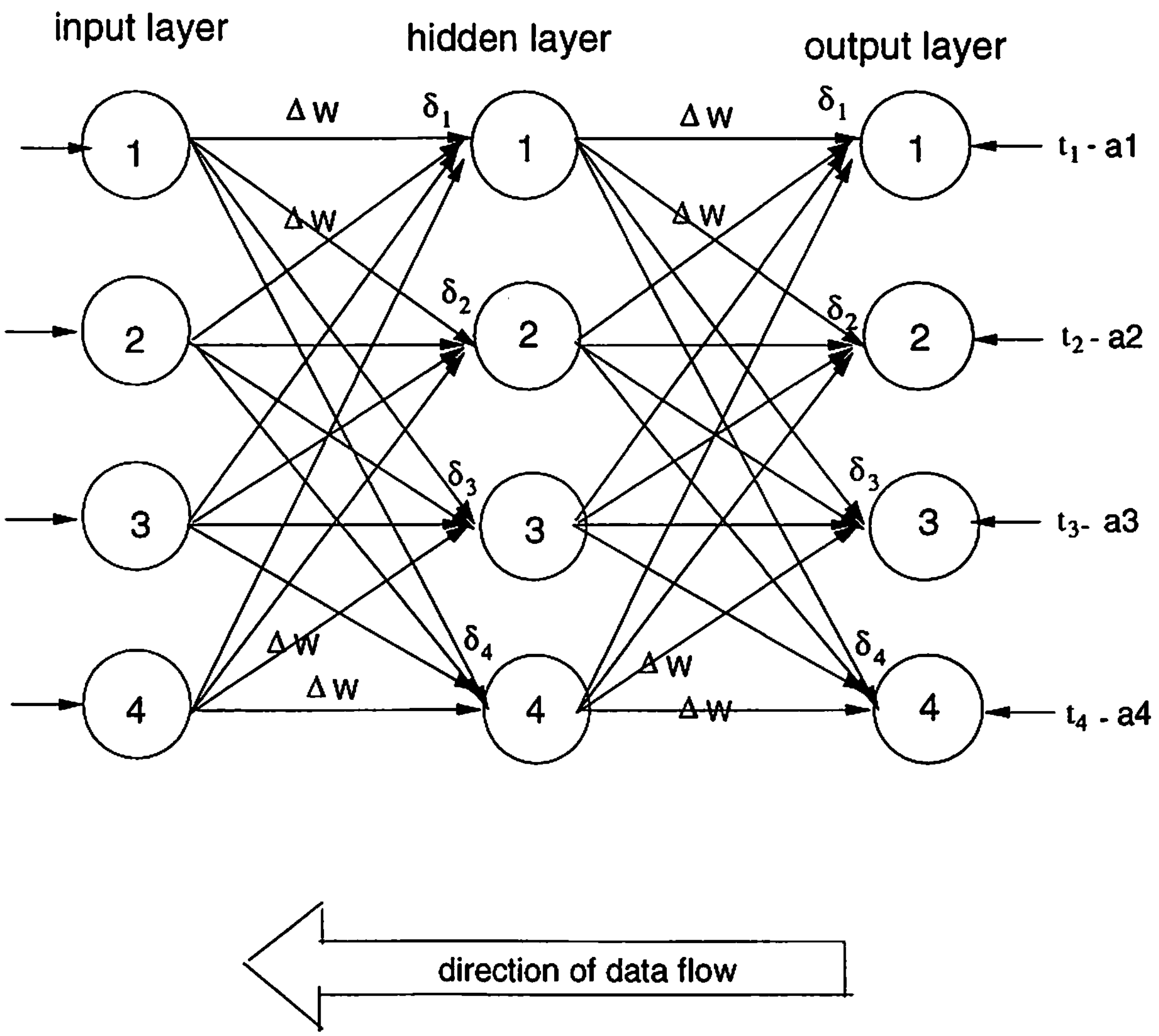
Forward Operation



DATA FLOW DIAGRAM OF FORWARD PATH

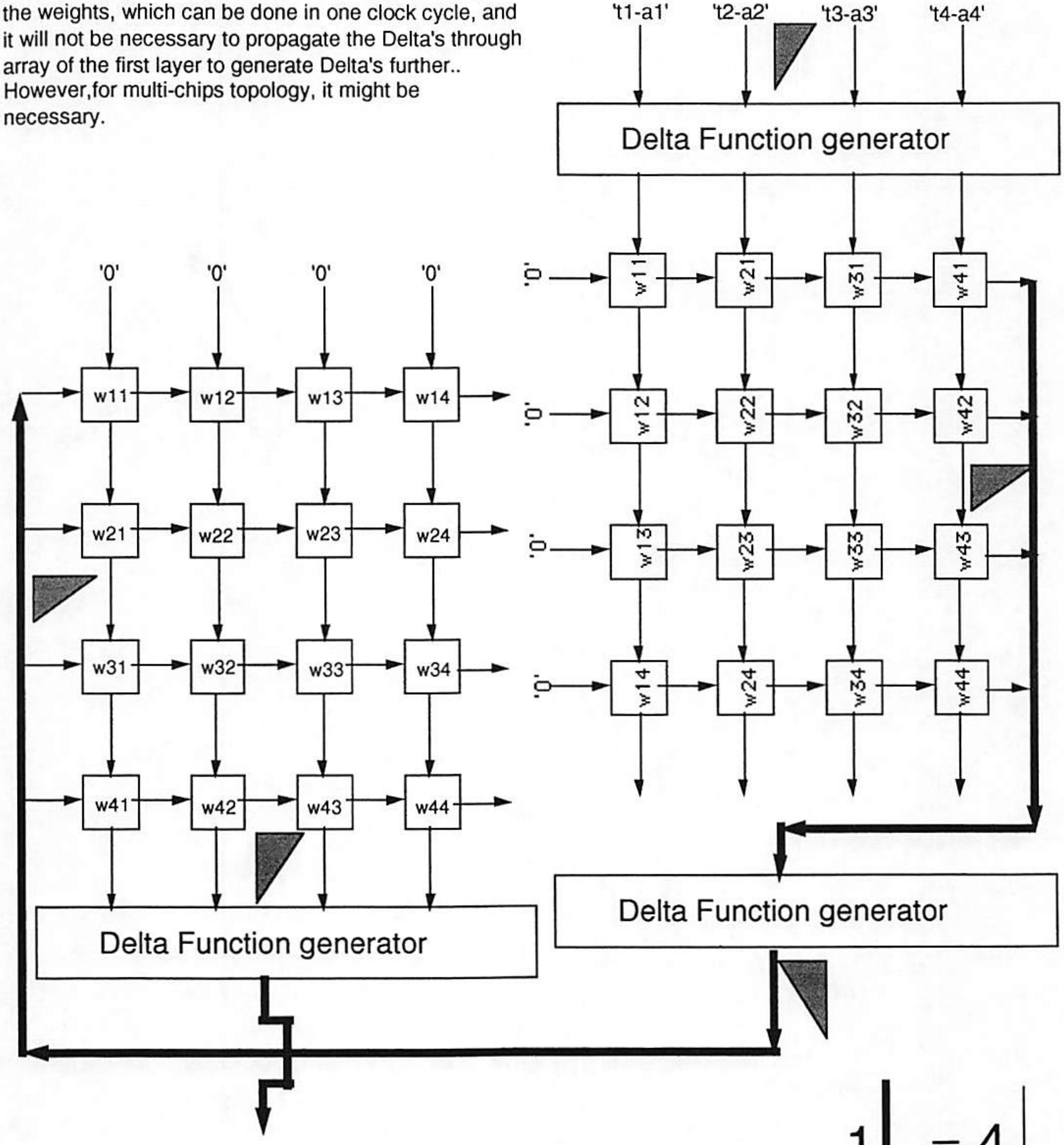


Backward Operation



DATA FLOW DIADRAM OF BACKWARD PATH

Note: for one chip operation, it is only necessary to change the weights, which can be done in one clock cycle, and it will not be necessary to propagate the Delta's through array of the first layer to generate Delta's further.. However,for multi-chips topology, it might be necessary.



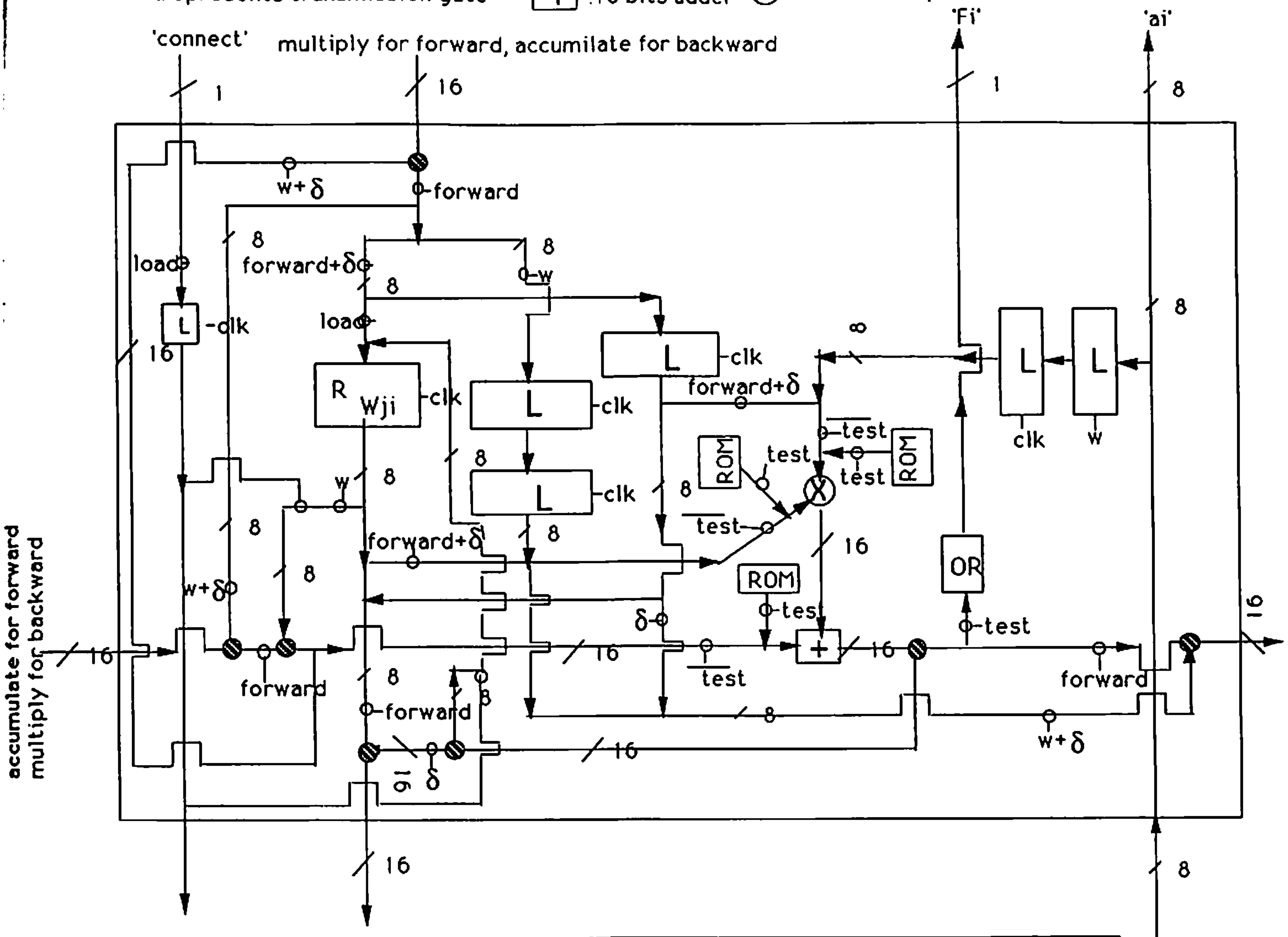
NOTE: before each layer's δ_j's are propagated back to the previous layer ,
the weights between the layers are first modified by the Delta rule.

1| = 4|

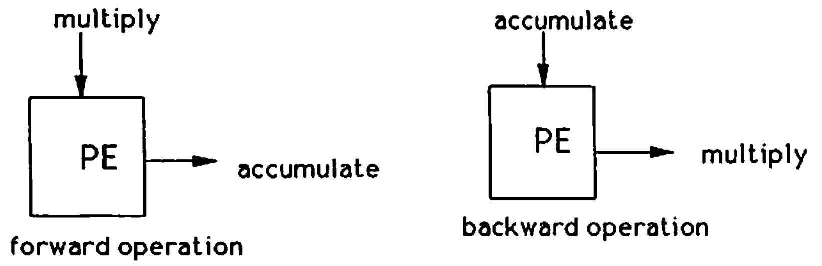
Appendix 4 PE / chip floor plan

L :latch R :register OR 8 input bits OR gate ● :represent a junction where only 8 out the 16 lines are connrcted.

○:represents transmission gate + :16 bits adder ⊗ : 8 bits multiplier



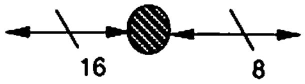
Note: ROM is actually an 8 or 16 bit fixed number in 2's complement format used to check the integrity of the multiplier and adder,thus the integrity of the PE. The product of 2 number add the negative of the correct result should give a 0,if not error occurs.



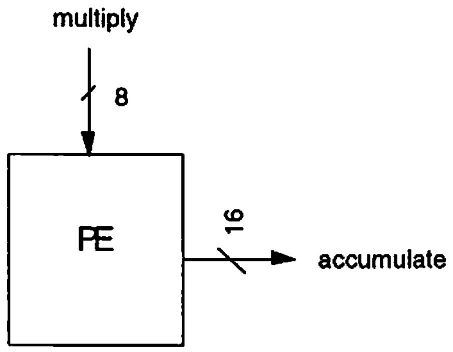
Note :because ai is a broadcasted signal, the weight update for the whole array can be done locally and simultaneously in one clock cycle

PE Wji Floor Plan

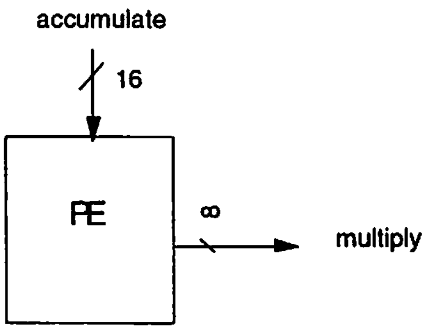
Note: becaust the same PE is used for both forward and backward operation with one side being able to take either 8 or 16 bits of inputs ,it is necessary to make them all 16 bits with some bits not necessaryly connected for different operations. This cause the need for the connector ▨ shown in the flow plan.



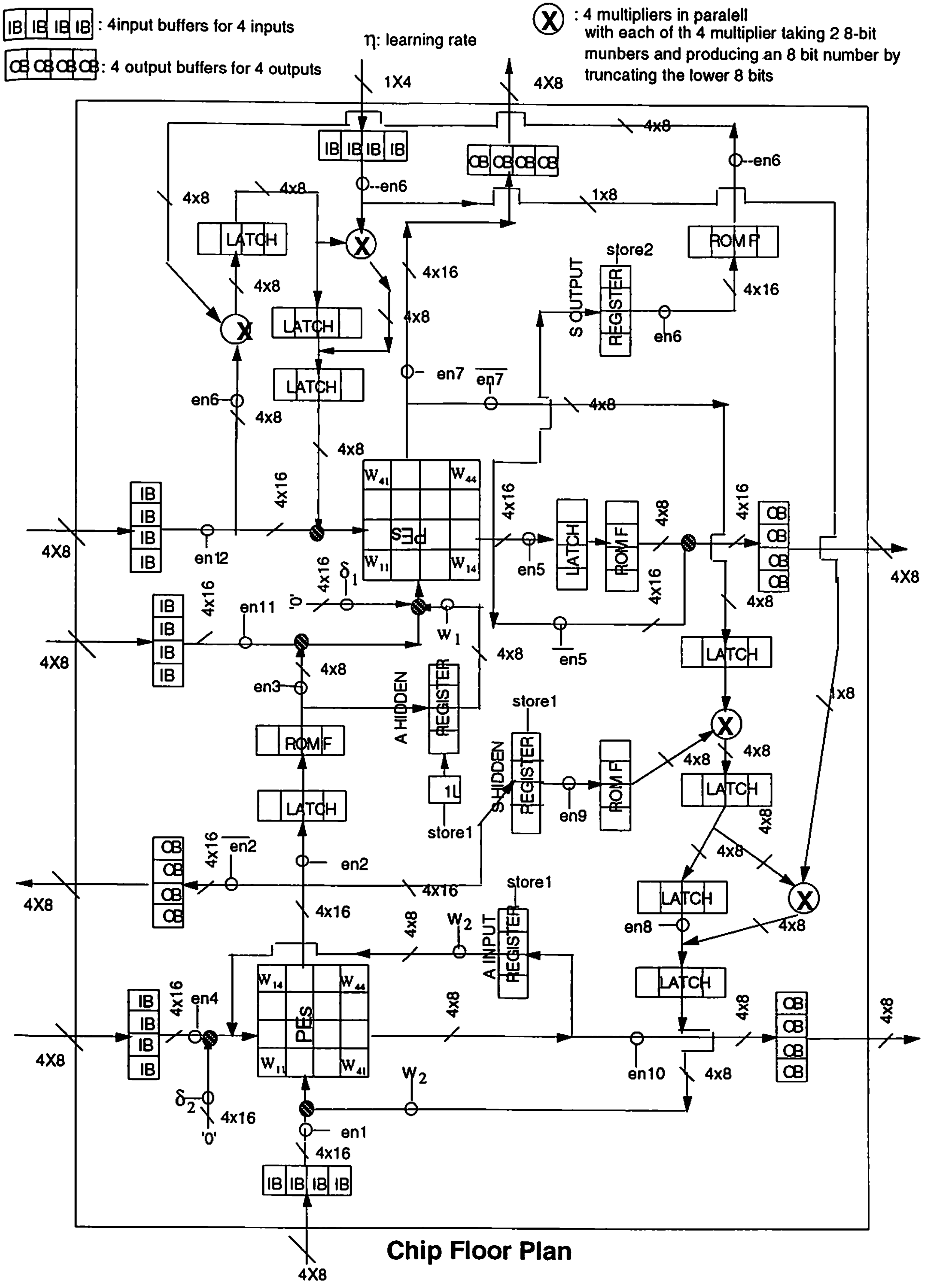
this connector simply outputs the higher 8 bits of the 16 bits input signal or puts the 8 bits input to the higher byte of the 16bits number



forward operation

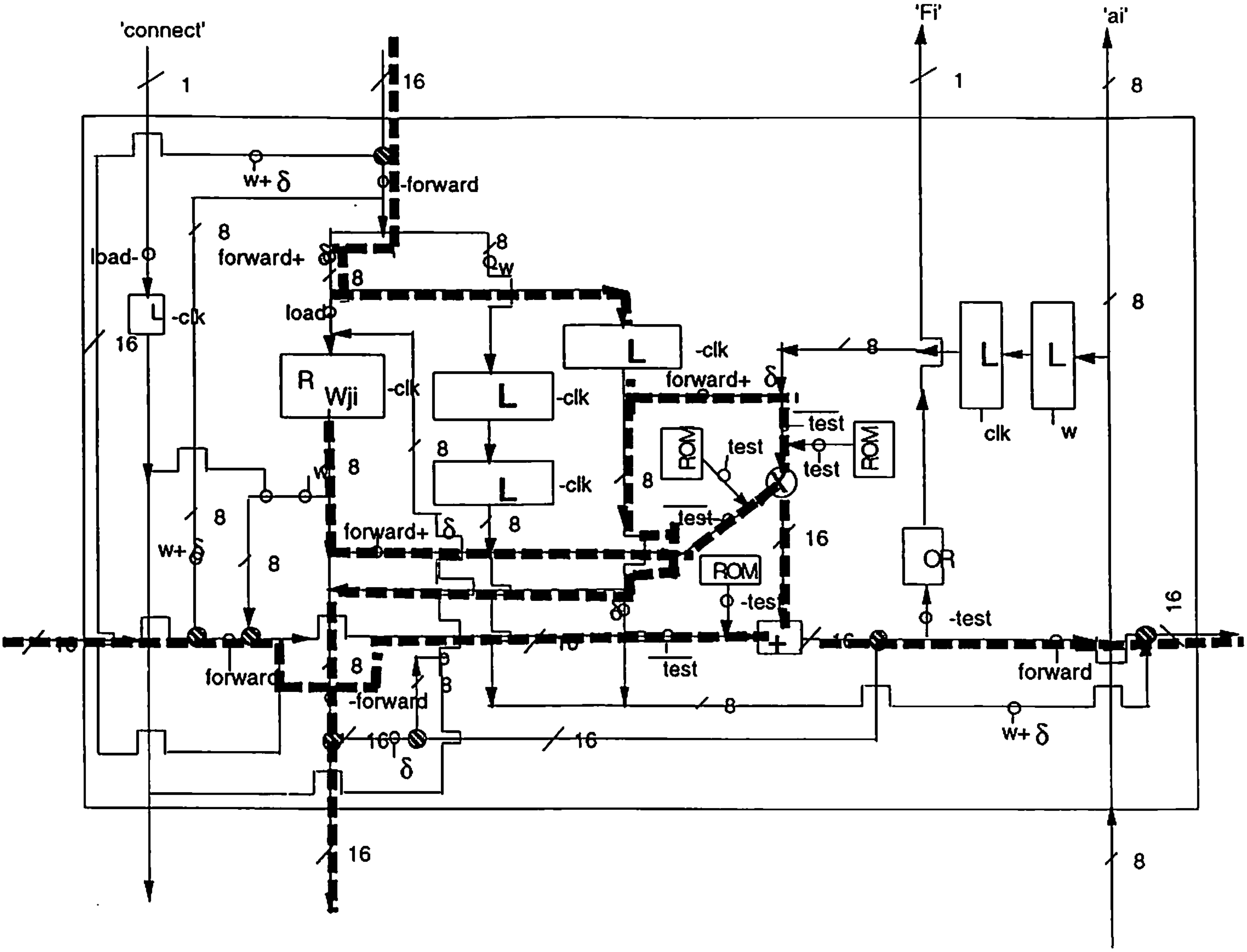


backward operation

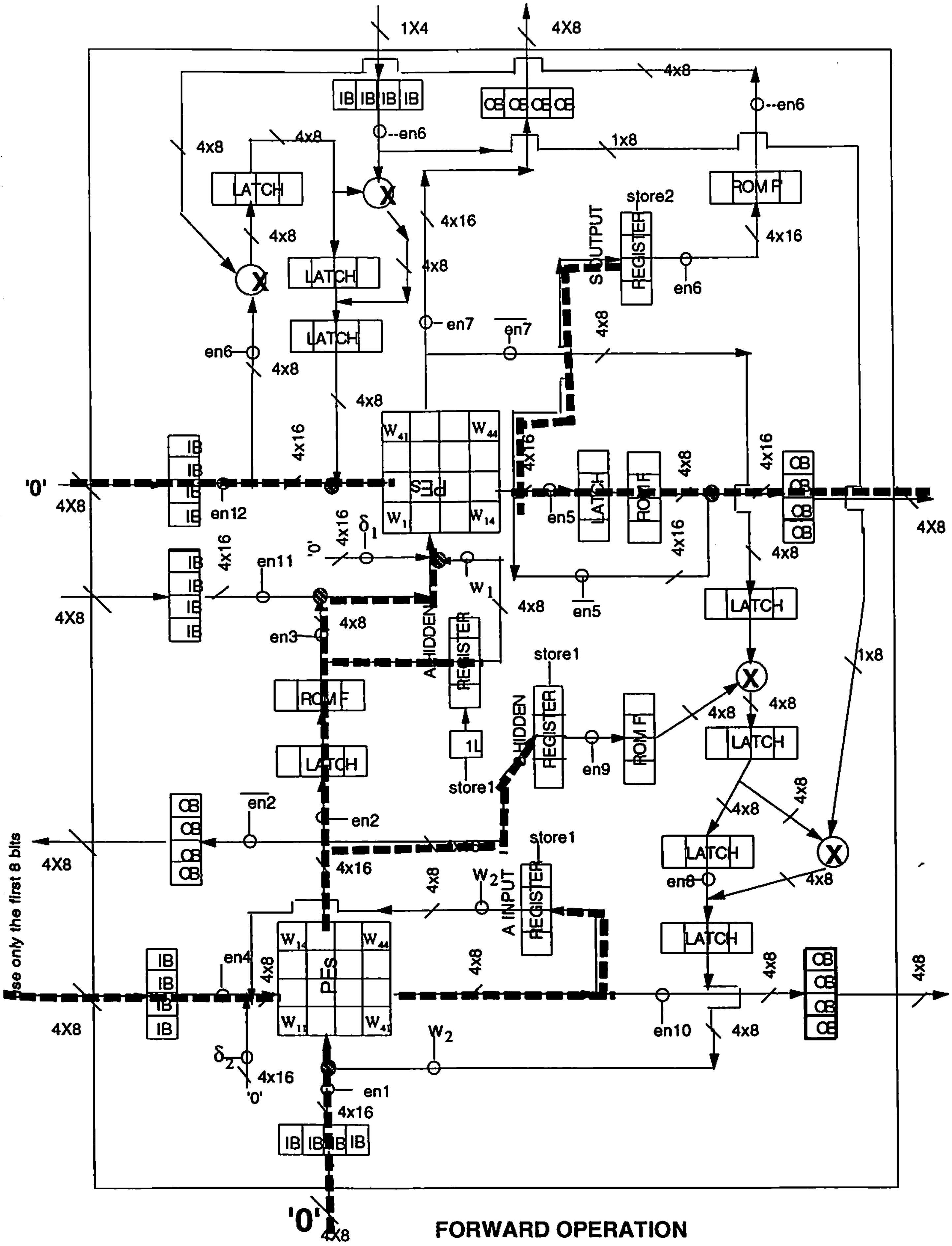

IB IB IB IB : 4input buffers for 4 inputs
OB OB OB OB : 4 output buffers for 4 outputs
η: learning rate
X : 4 multipliers in paralell
with each of th 4 multiplier taking 2 8-bit
munbers and producing an 8 bit number by
truncating the lower 8 bits
1X4
4X8
4x8
1x8
en6
LATCH
ROM F
store2
S OUTPUT
REGISTER
4x16
4x8
en7
en7
W41
W44
W11
W14
PES
en12
en11
'0'
δ1
W1
en5
en3
A HIDDEN
store1
1L
S HIDDEN
en9
en2
A INPUT
W2
en4
δ2
en8
en10
en1
4X8

Chip Floor Plan

Appendix 5 data path in PE / chip for forward operation

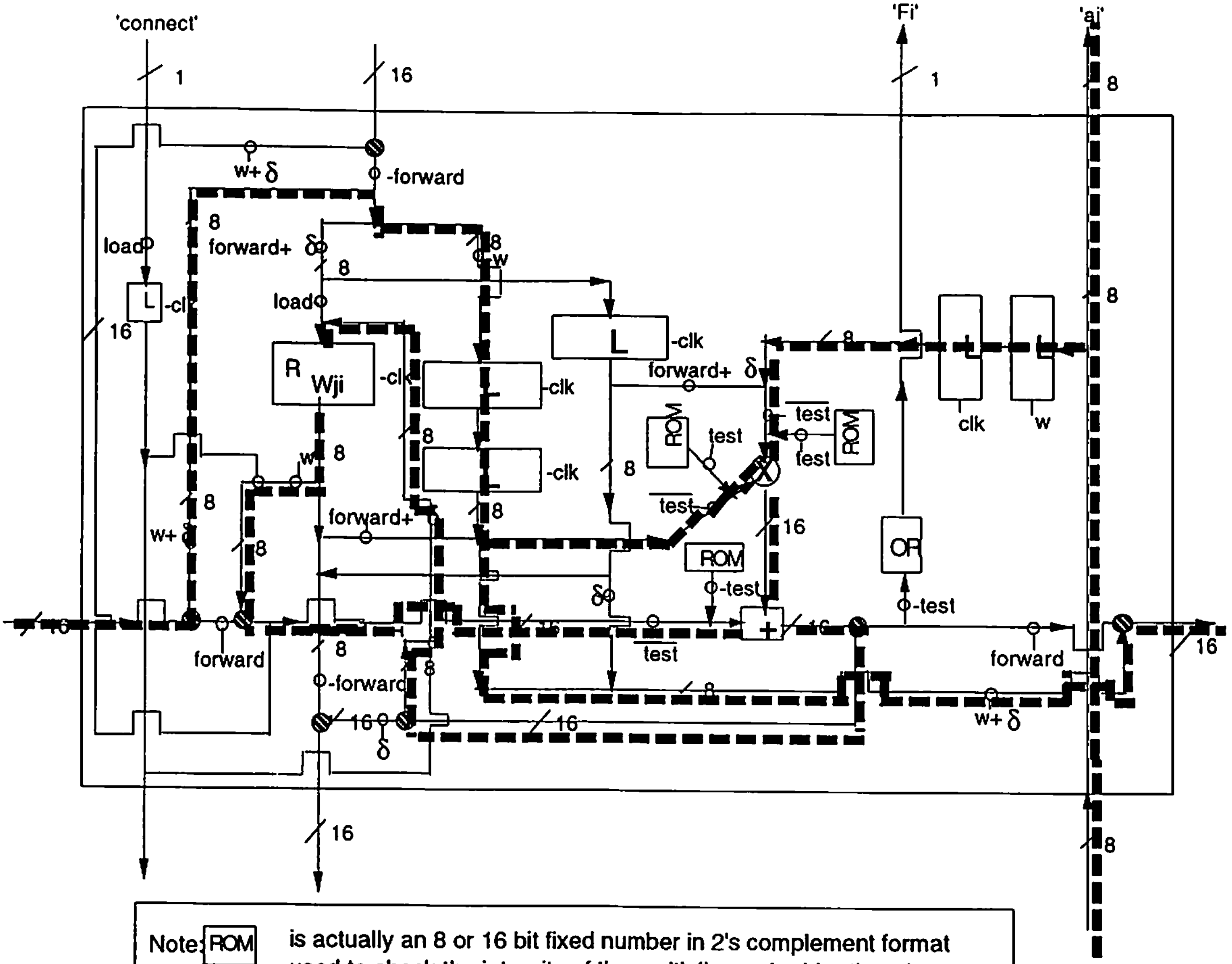


FORWARD



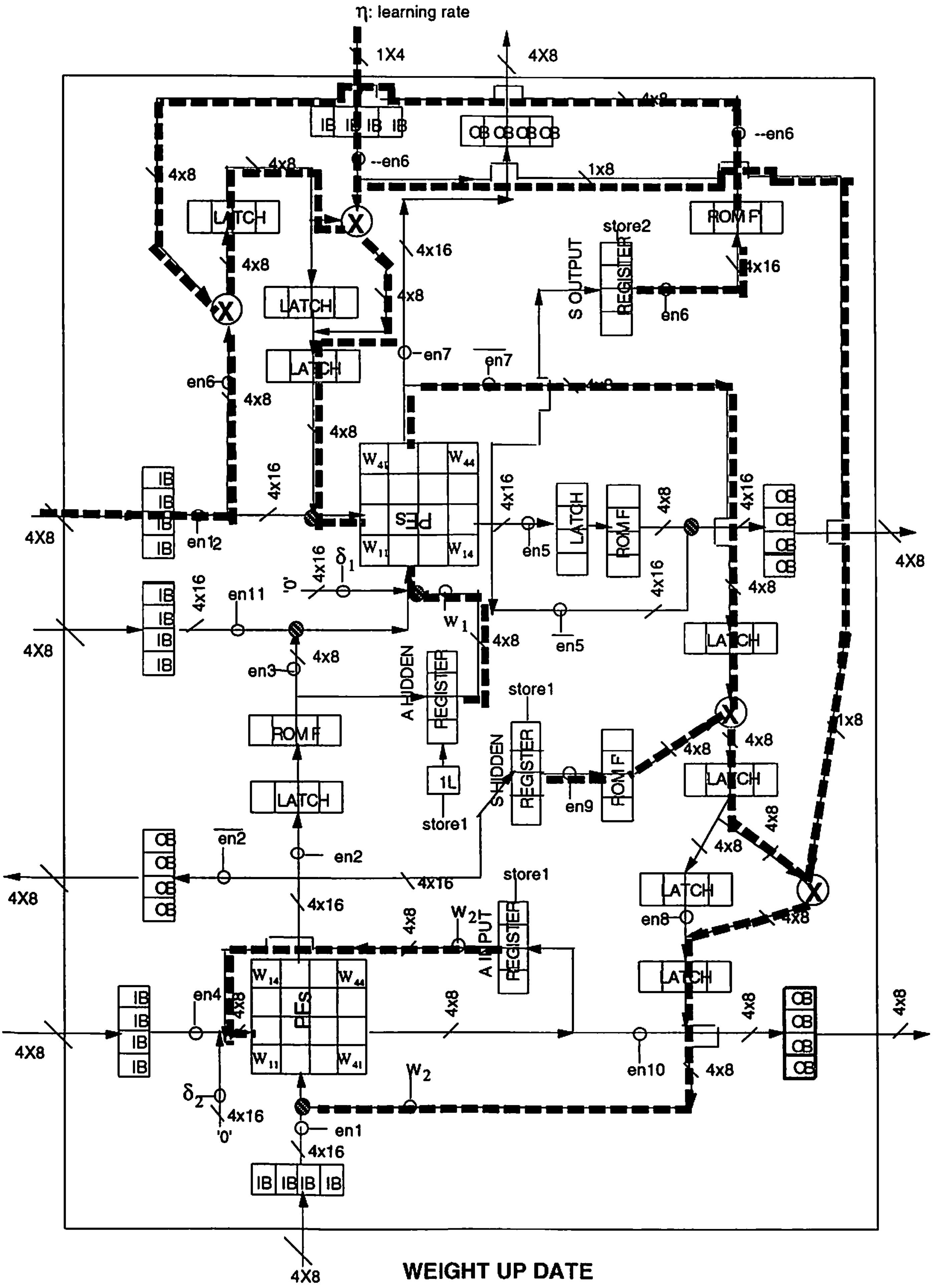
FORWARD OPERATION

Appendix 6 data path in PE / chip for weight update



Note: ROM is actually an 8 or 16 bit fixed number in 2's complement format used to check the integrity of the multiplier and adder,thus the integrity of the PE. The product of 2 number add the negative of the correct result should give a 0,if not error occurs.

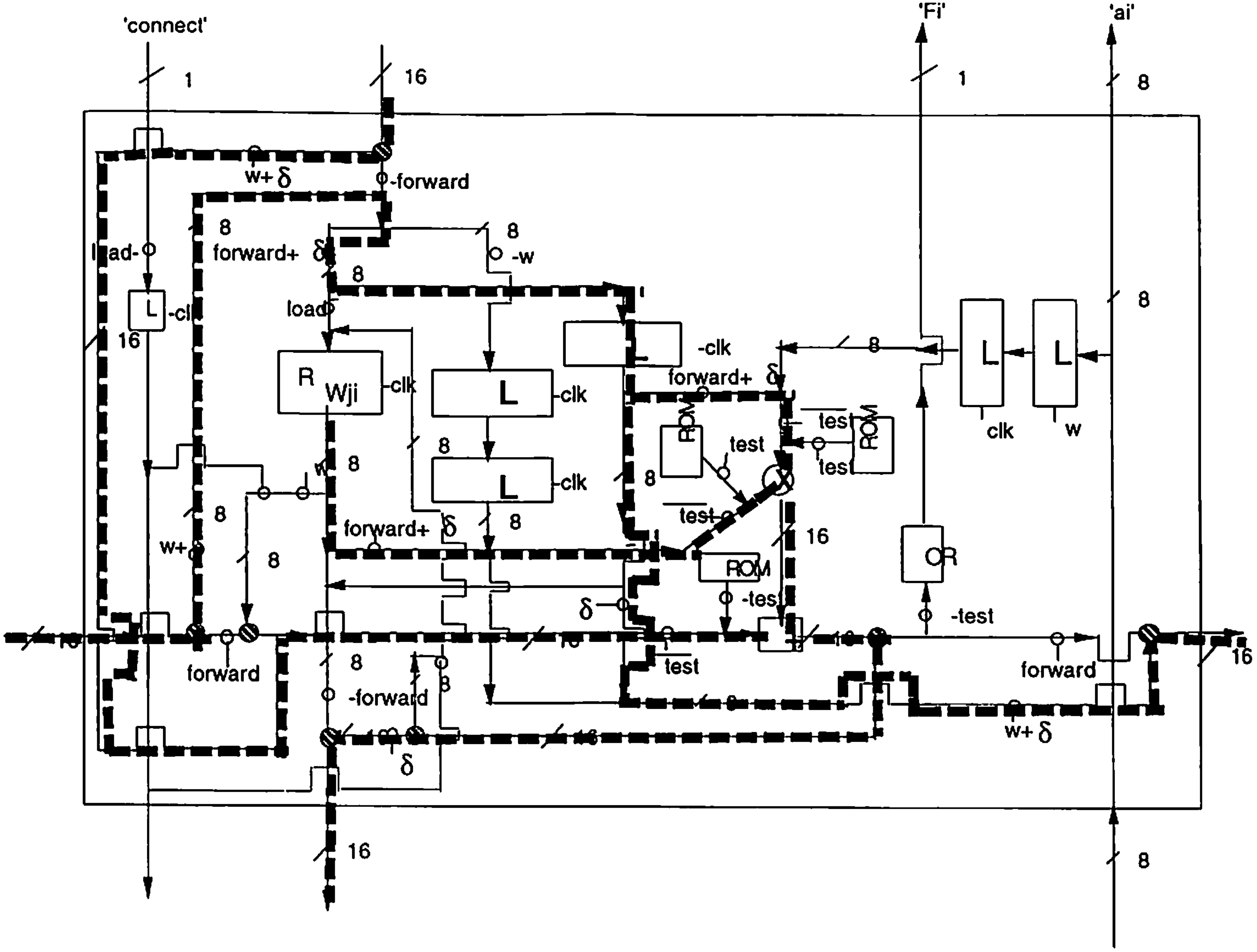
WEIGHT UPDATE



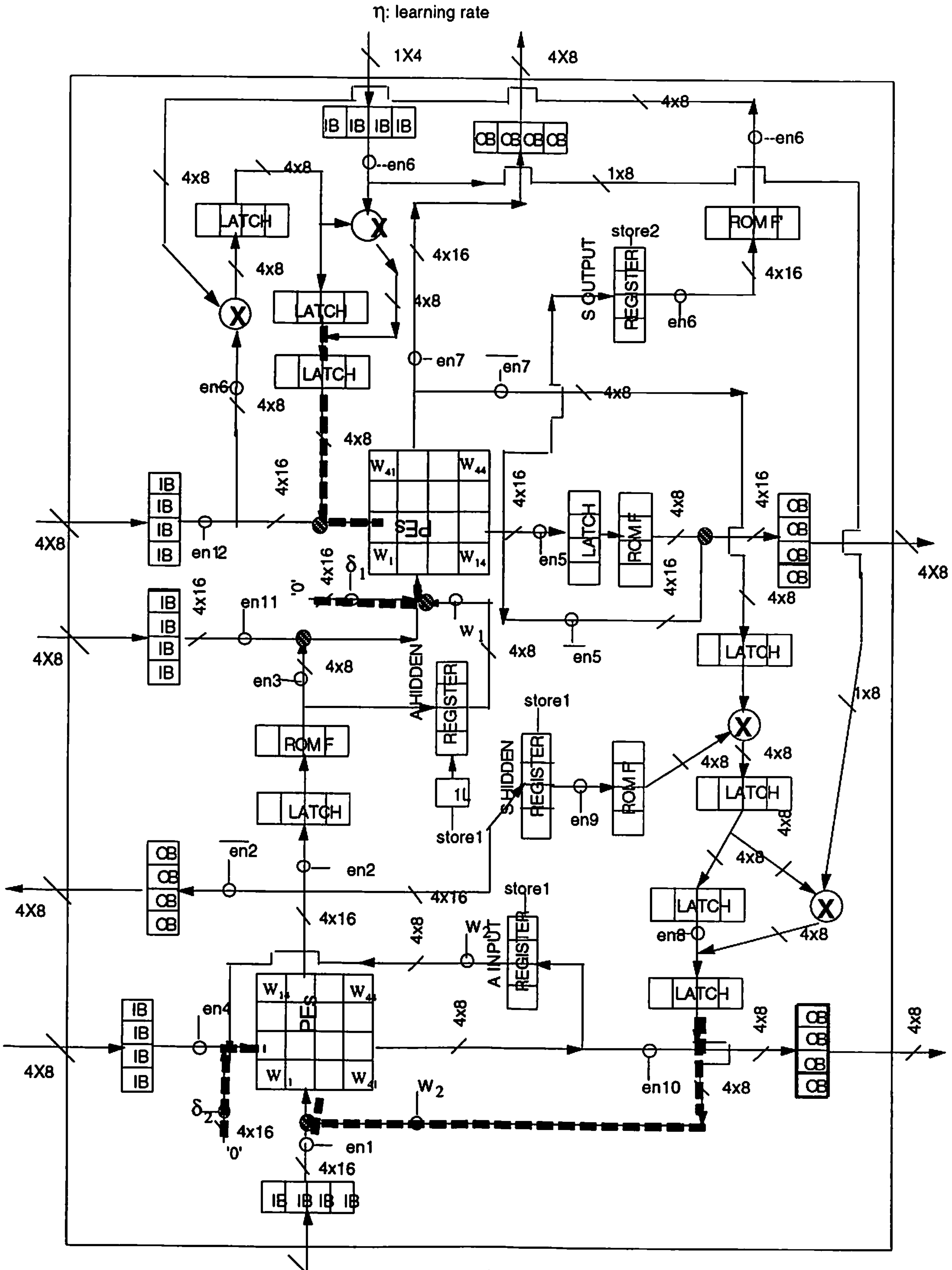
WEIGHT UP DATE

Appendix 7

data path in PE / chip for delta calculation



Delta calculation



delta calculation

Appendix 8 signals and operation

Signals and Operation

On the chip there are the following signals that need to be fed by the host

test	self test for fault tolerant connection path set up
load	load wji's
forward	help each PE to make the right connection for forward operation
en1-en12	set up the topology for either multi- or uni- chip operation
connect1-8	to make the network not fully connected
store1	store first layer's ai and Si
store2	store 2nd layer's ai and Si
clk	system clock which is about 1/20ns =50Mhz
Vdd	power supply
GND	power supply
w1	change 2nd layer weights
w2	change the first layer weights
δ_1	calculate the 2nd layer delta's
δ_2	calculate the ast layer delta's

there are 32 pins required for the above 32 signls
with the (4x8)x8+4=260 pins on the chip floor plan,
there are a total of 292 pins on this chip .
this rather large number of pins id due to the fact that
this chip is fully scalable-- any number of neurons and
layers are possible.

For one chip operation, there are 4 phases to consider
(1) test phase
(2) load phase
(3) forward phase
(4) backward phase

TEST PHASE
the test signal nees to be on for one clock cycle , and at the end of next cycle the connection logic circuit should have the topology set up since it only employs simple 2 level logic. If more then 2 PE's in an array are detected faulty, the chip shoule be rejected at the production stage by using testing probes.

LOAD PHASE

wi1 wi2 wi3 wi4 (weights in each row) are pipelined down when the signal load is On. It takes 1 clock cycle to load one row, so for 4 rows it takes 4 clock cycles to finish the loading. Note for the 2nd array of weights,because its topology is the transpose of the first array, so we have to enable the δ_1 signal as well. This is generally true for any hidden layer weights

it is also necessary to load the connect bits to determine if the synapse connection exist. If a connection between neurons does not exist, the corresponding weight is loaded as '0' and also the corresponding connection bit is set to '0' to prohibit future weight update, so that weight remains at '0'

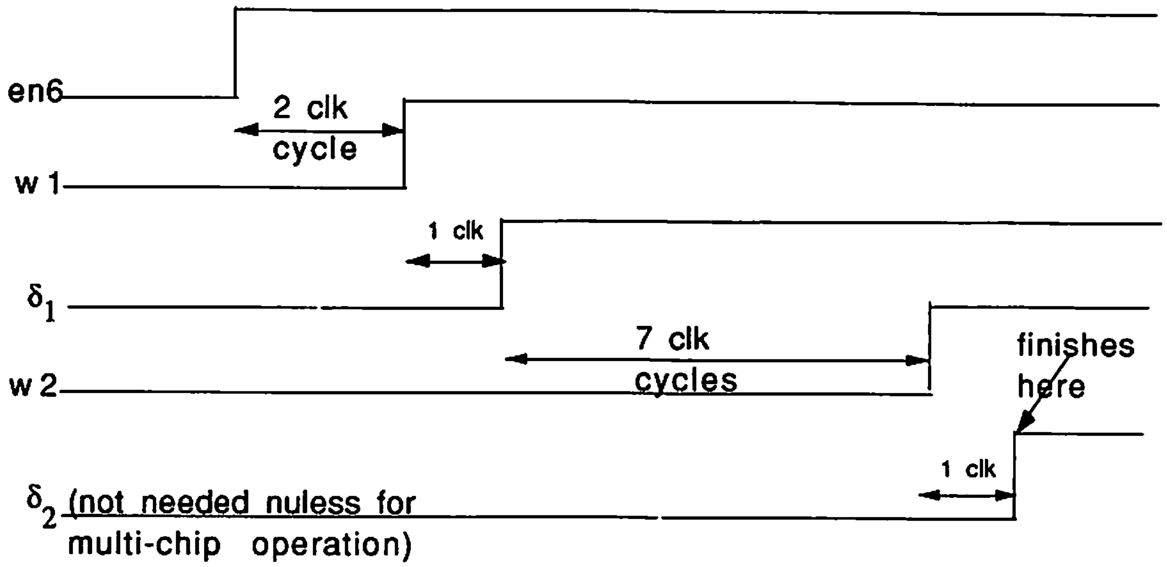
FORWARD PHASE

after the weights have been loaded, we can apply the following signaltoestablish the right path for forward phase
en1 en2 en3 en4 en5 en12
all these signals must be '1' during the period of forward training. All the other eni's are set to '0' for 1 chip operation.
Note : if the net work has been properly trained, it will not neeed to apply the backward phase, and the input can be pipelined into the system , and the output will be pipelined out at the same rate. Because the number of pins is very large, it was necessary to multiplex the higher byte with the lower byte for both the input and output. Thus, the rate data does in and comes out is half of that of the system clock, so the speed is about 25 Mhz which is still O.K. for most applications.

If backward operation is to be expected, store1 and store2 should go to '1' to enable the weighted sum Sj and neuron output ai to be stored. For one chip ooperation, it 2 latches and 4 PE's (6 clk cycles) to finish the first signal propergation, and it should be kept on for 4 clk cycles for all the pipelined signal to be captured. Similar comments can be applied to store2 signal.

BACKWARD PHASE

this is necessary for the network to learn to adjust the weights
for proper operation, after one forward operation has completed for one input set, thehost computer compares the answer with the desired answer, and the error ti-ai are fed back. the following signals have to be set to ;'1' for proper operation topology en6 en9 en8, and en7 is set to '0'
there are also other signals that can go on only at the right time and they are shown as below

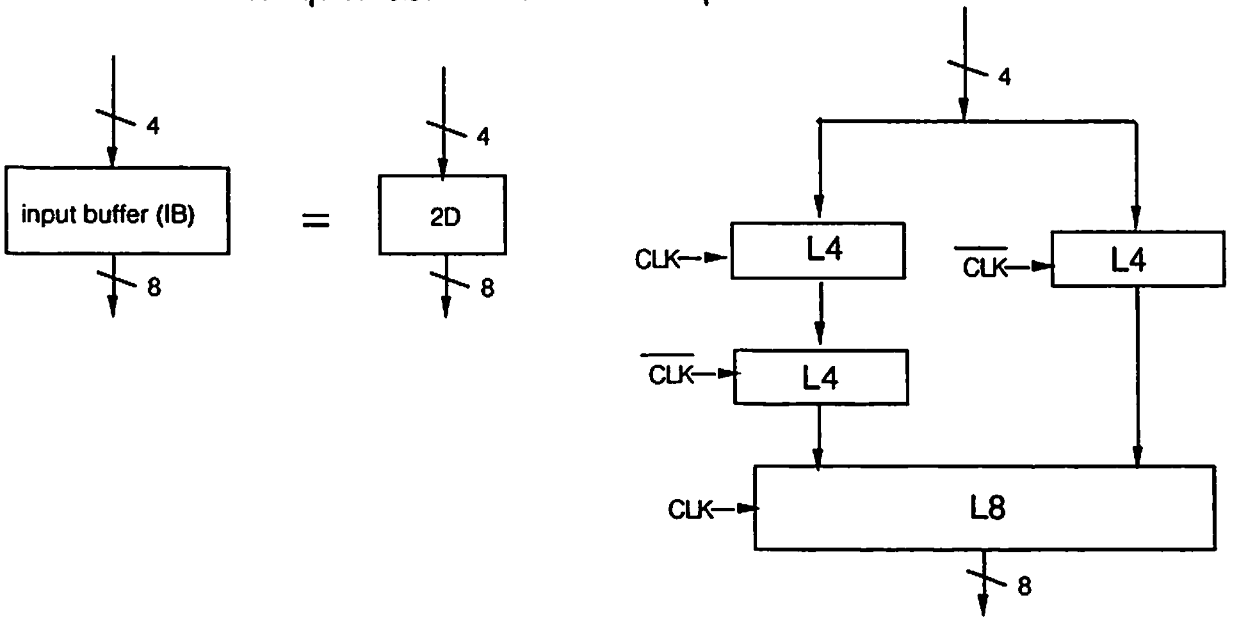


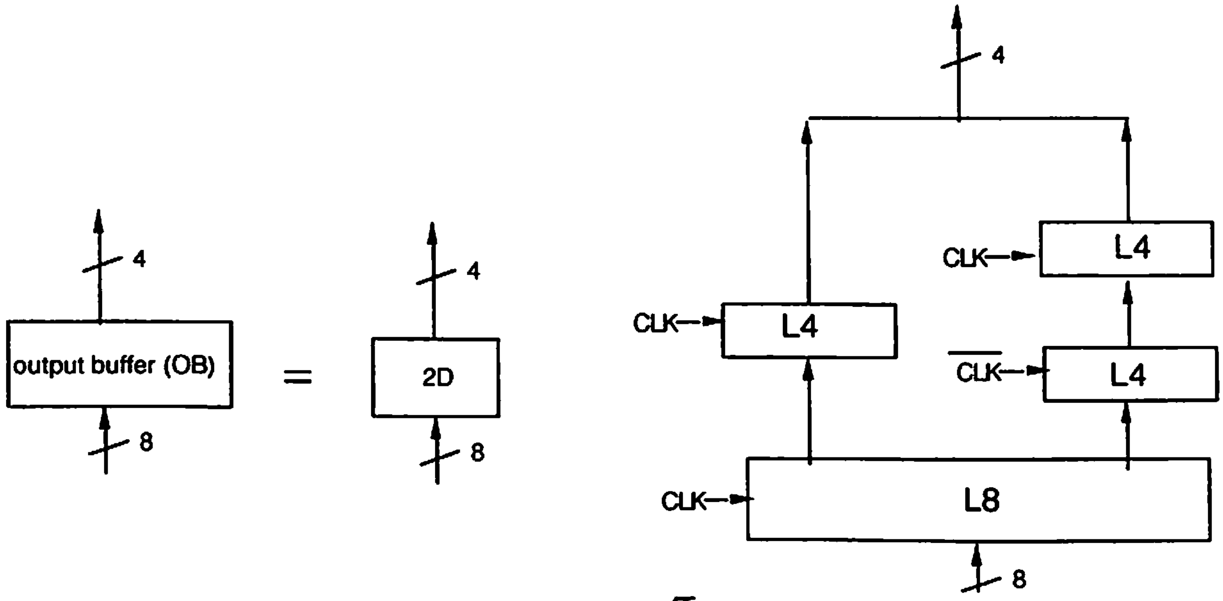
note: it takes 11 clk cycles to finish 1 chip backward phase, and out of which 7 is the big factor. But the 7 cycles consists of 3 latches used for 2 multiplications and 4 cycles for signals to propagate down the array. So it is obvious that as the array size increases using many chips, the learning time will increase almost linearly with the arrsy size or problem size. This is actual an indication of pretty fast learning time as in many case the learning time will vary expotentially with the problem size. There are some unused signals which will be necessary for multi-chip topology.

Appendix 9

I/O BUFFER DESIGN

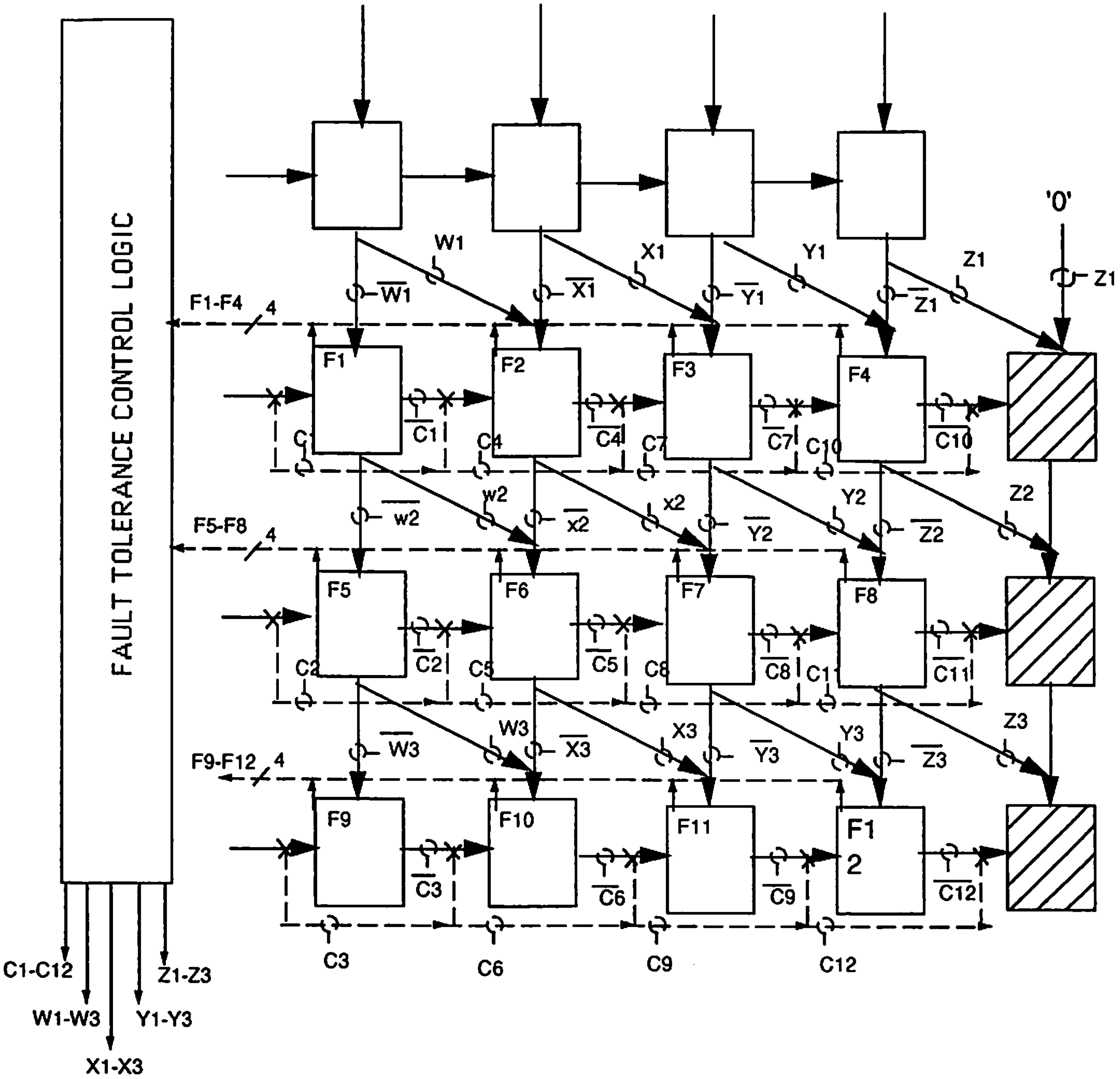
Since this chip is design to be scalable - being able to expand the number of neurons and layers, it will be inevitable to have many pins on the chip for this off-chip interconnection purpose.
Therefore, the I/O has to be multiplexed so the first 4 bits of data is fed in/out first, then at the next clock cycle the last 4 bits. So the speed is effectively reduced by half.
This is a trade off between speed and flexibility and expandability
But since the clock is about 50Mhz, the data I/O is then about 25Mhz. It is still quite fast for most real time problems.





Appendix 10 Fault tolerance design with example of data flow in case of PE failure

THE REAL ARRAY OF PE'S WITH FAULT TOLERANCE FEATURE



The control logic generate output signals to establish the right connection among PE's so we will have a virtual 4x4 standard array.
For simplicity, all the rest diagram will employ only the virtual PE array, and it should be understood that the real array actually looks like this diagram.

Chip fails if for each PE arrays more than one Fi are '1'

The Fi in each chip is generated in the TEST phrase of this chip operation. It uses the stored numbers in the PE and multiplt 2 of then then subtract the other. If the subtracted number is the correct number, 16 bits of '0' will result. These correctness of the operation can then be checked by ORing these 16 bits. If the result is wrong , the OR will output a '1' for the Fi. Otherwise, it will be '0' for correct operation.

If any one of the 12 lower 3 rows of PE's fails (any one of the signals F1-F12 goes to '1'), the 3 shaded duplicated PE's will operate. But no more than 1 PE can fail in the array -- if so, chip fails.

The cortrol logic inputs F1 -F12 and outputs the Following signals W1-W3, X1- X3, Y1-Y3,Z1-Z3, C1=C12, and they are as follow

W1=F1
W2=F5
W3=F9

X1=F1+F2
X2=F5+F6
X3=F9+F10

Y1=F1+F2+F3
Y2=F5+F6+F7
Y3=F9+F10+F11

C1=W1
C2=W1+W2
C3=W1+W2+W3

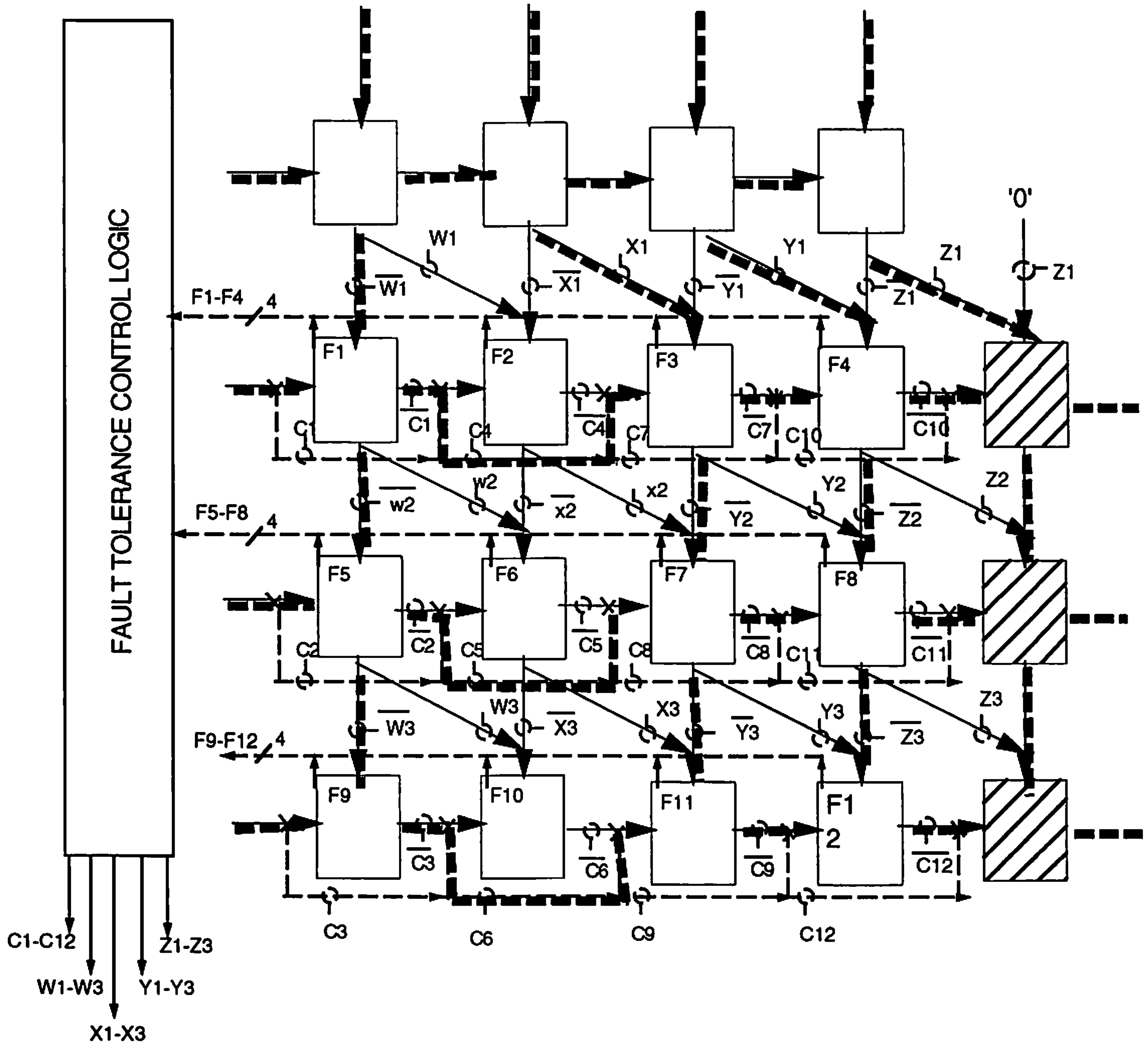
C4=X1
C5=X1+X2
C6=X3+X2+X1

C7=Y1
C8=Y1+Y2
C9=Y1+Y2+Y3

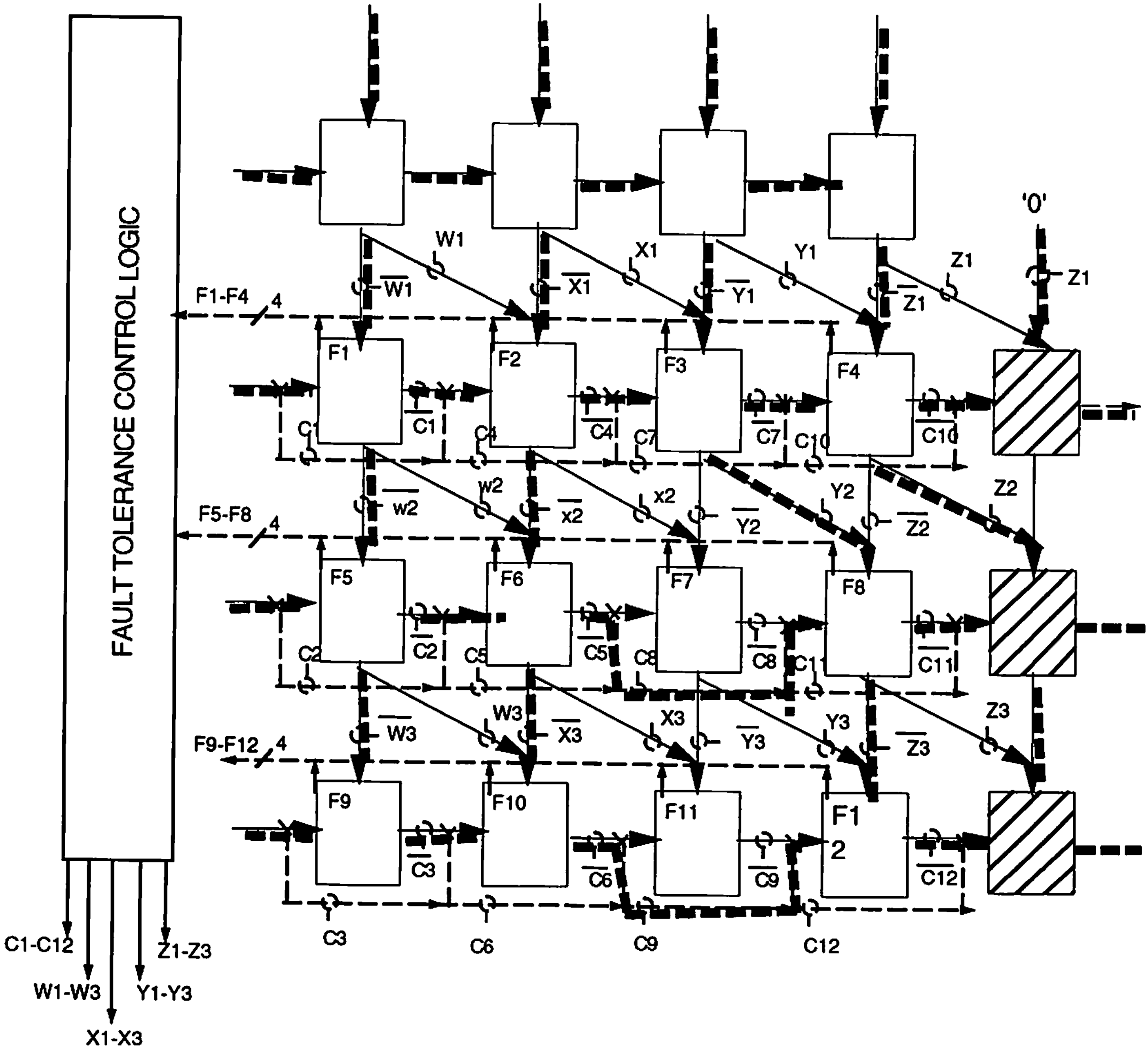
C10=Z1
C11=Z1+Z2
C12=Z1+Z2+Z3

And here are some example of how it works

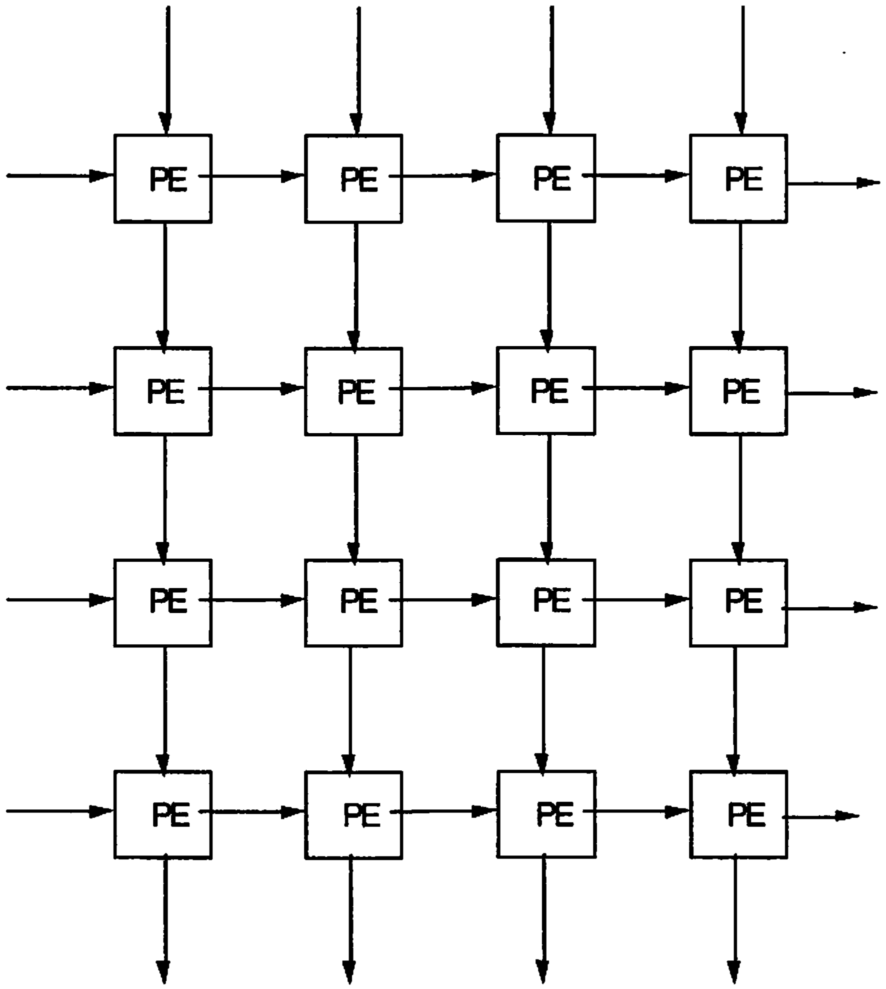
Example of array data flow in case of a PE failure
Let F2=1



Example of array data flow in case of a PE failure
Let F7=1



THE VIRTUAL ARRAY OF PE'S



Appendix 11 Sigmoid and Sigmoid prime realization with quantization error

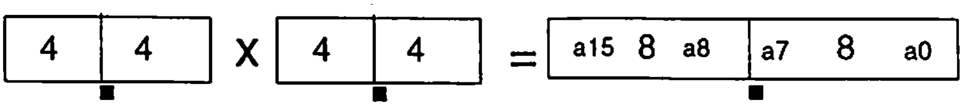
Realization of Sigmoid and Sigmoid Prime Functions

Sigmoid function f(x)=1/(1+exp(-x)) and its first derivative Sigmoid Prime f'(x)=exp(-x)/(1+exp(-x))^{2}can be generated by basically 2 methods
(1) using analog circuits, but this requires A/D and D/A operations , which may complicate the design and the circuits are not reliable.
(2) using table look-up which is a lot simpler and straight forward.

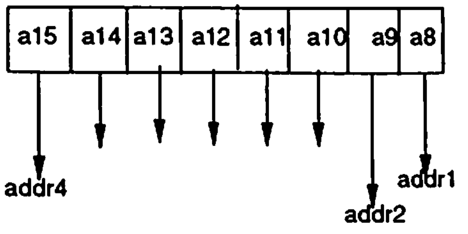
However, since both operands for multiplication are 8 bits 2's complement numbers, the resulting number must be 16 bits long. Therefore, ideally we need a RAM or ROM part that converts an16 bits input address number to a 8 bits data output. This is a 64kx8 memory capacity which is definitely too large for this application, especially many of them have to be used by this chip to achieve the pipeline operation.

Nevertheless, since both f(x) and f'(x) pretty much saturate beyond x=+5 and x=-5, with a little logic we can closely approximate these functions by using only a few of the 16 address bits.

As an example, if I only want to choose 16 points to approximate these functions, and the inputs and outputs of multiplication are as below:



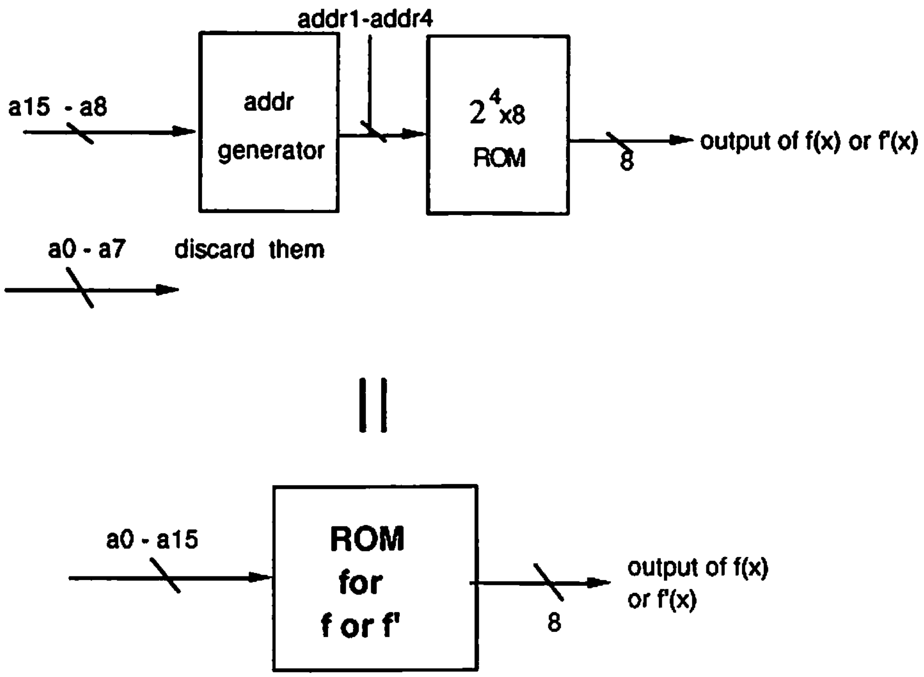
We can just choose the higher 8 bits for address generation of the 16 numbers as follows



if a15 =0 (positive number),addr3=1 if any of a10 to a14 =1.
if a15 =1 (negative), addr3=1 if all of a10 to a14 are 1.

This selection makes any number less than -8 equal -8 and any number greater than 7 equal 7.

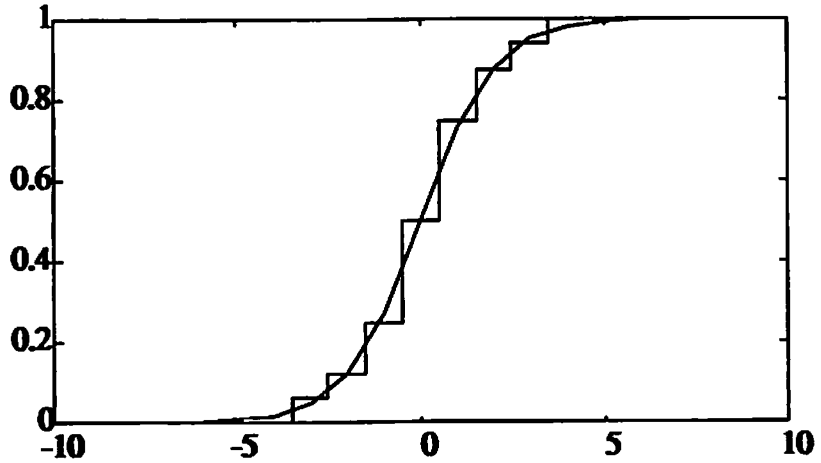
addr3= a15 (a10+a11+a12+a13+a14)+a15 (a10 a11 a12 a13 a14) and conceptually it looks like the following



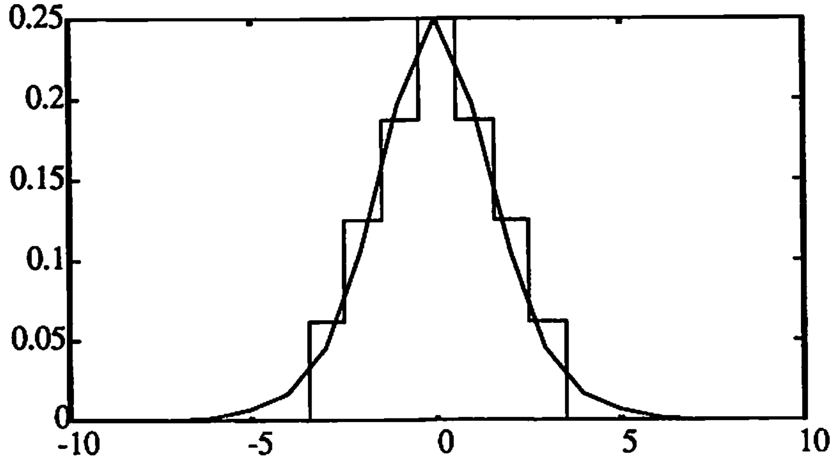
and here are some data for this implementation

input number	addr4-addr1	output for f	outputfor f'
7 and up	0111	0001.0000= 1	0000.0000= 0
6 to 7	0110	0001.0000= 1	0000.0000= 0
5 to 6	0101	0001.0000= 1	0000.0000= 0
4 to 5	0100	0001.0000= 1	0000.0000= 0
3 to 4	0011	0000.1111= 0.9375	0000.0001= 0.0625
2 to 3	0010	0000.1110= 0.875	0000.0010= 0.125
1 to 2	0001	0000.1100= 0.75	0000.0011= 0.1875
0 to 1 and -1	0000	0000.1000= 0.5	0000.0100= 0.25
-1 to -2	1111	0000.0100= 0.25	0000.0011= 0.1875
-2 to -3	1110	0000.0010= 0.125	0000.0010= 0.125
-3 to -4	1101	0000.0001= 0.0625	0000.0001= 0.0625
-4 to -5	1100	0000.0000= 0	0000.0000= 0
-5 to -6	1011	0000.0000= 0	0000.0000= 0
-6 to -7	1010	0000.0000= 0	0000.0000= 0
-7 to -8	1001	0000.0000= 0	0000.0000= 0
-8 and down	1000	0000.0000= 0	0000.0000= 0

Graphs of f(x) and f'(x) showing quantization error



Sigmoid function

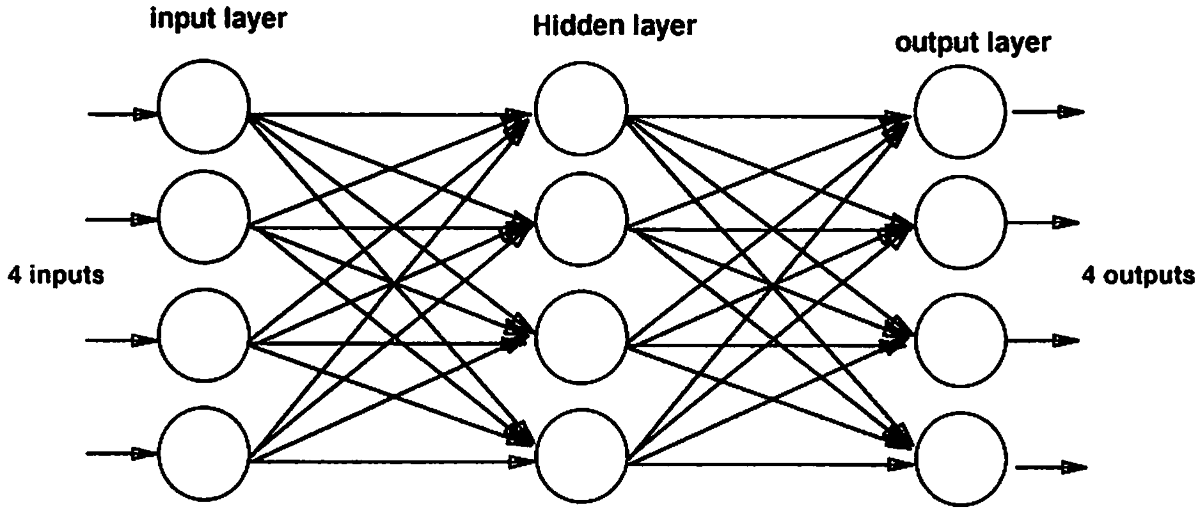


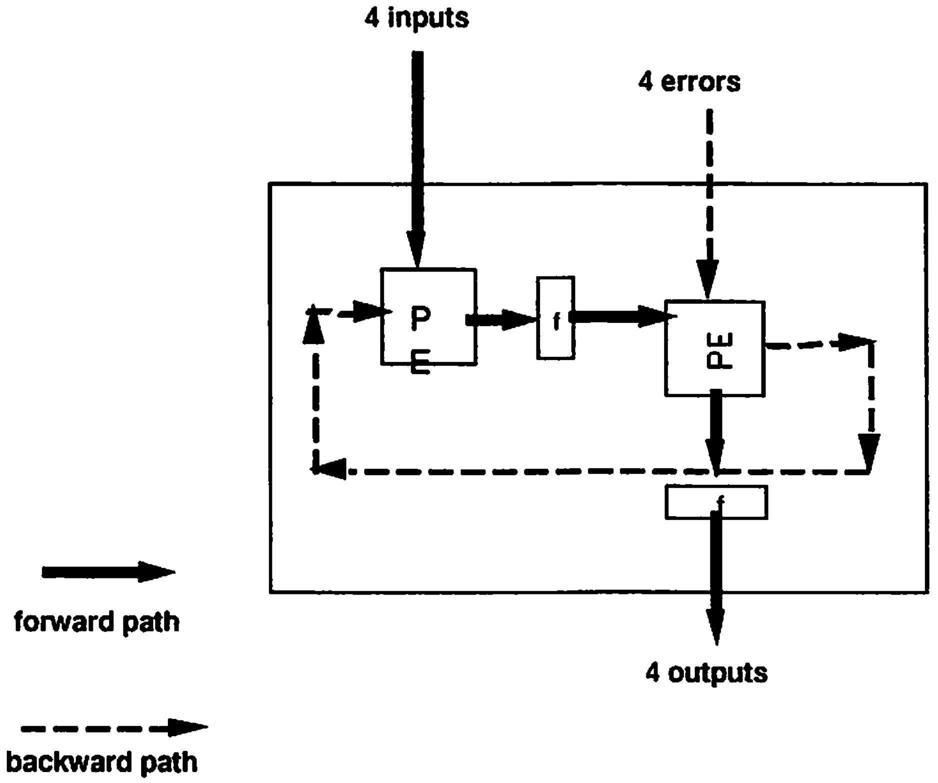
Sigmoid function prime

Appendix 12

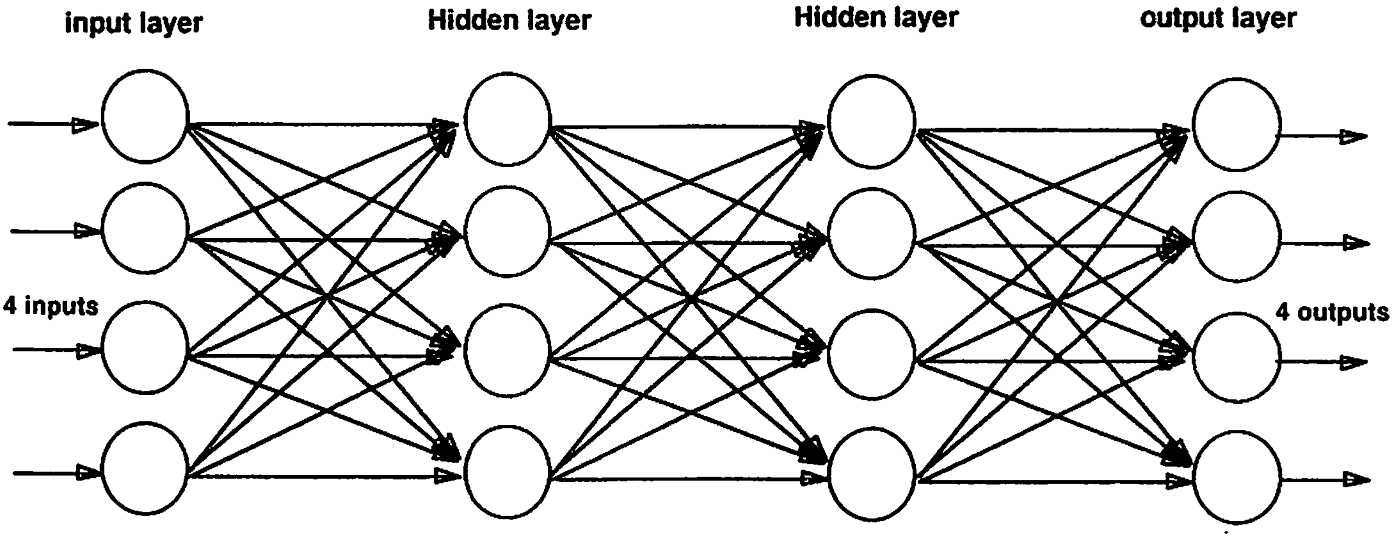
multichip configuration

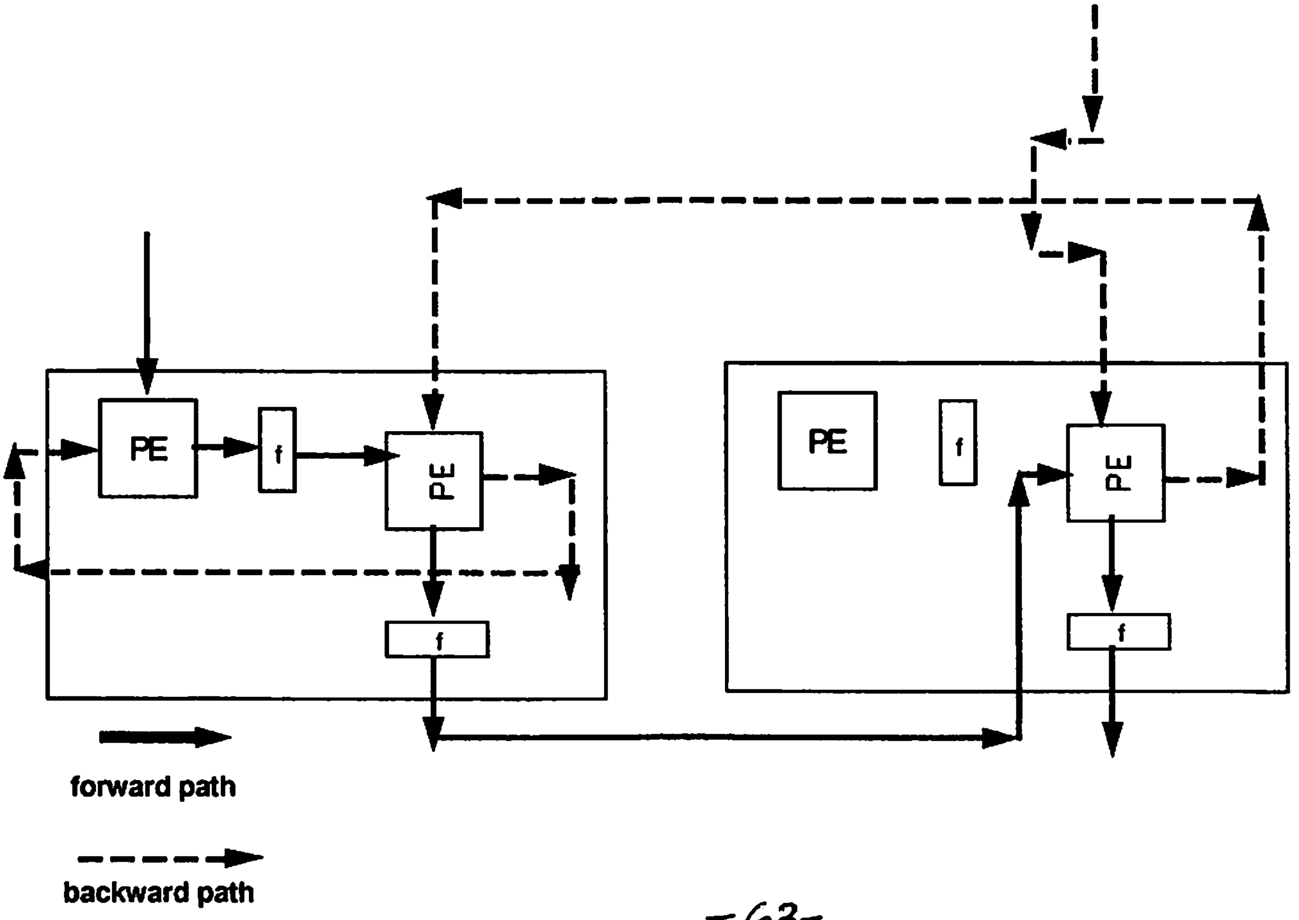
One Chip Operation With Simplified Data Flow



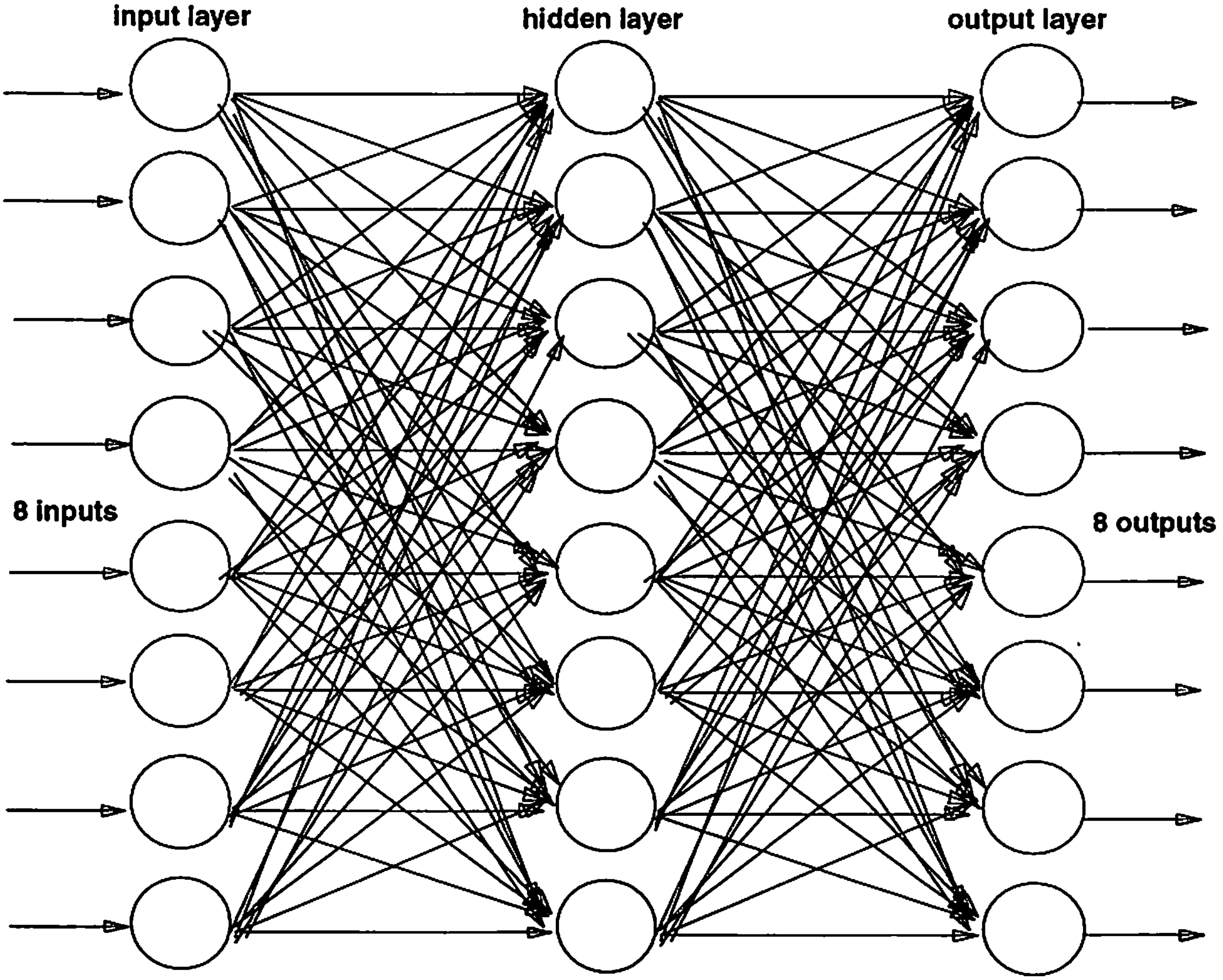


2 chips can be used to realize a 2 hidden layers network

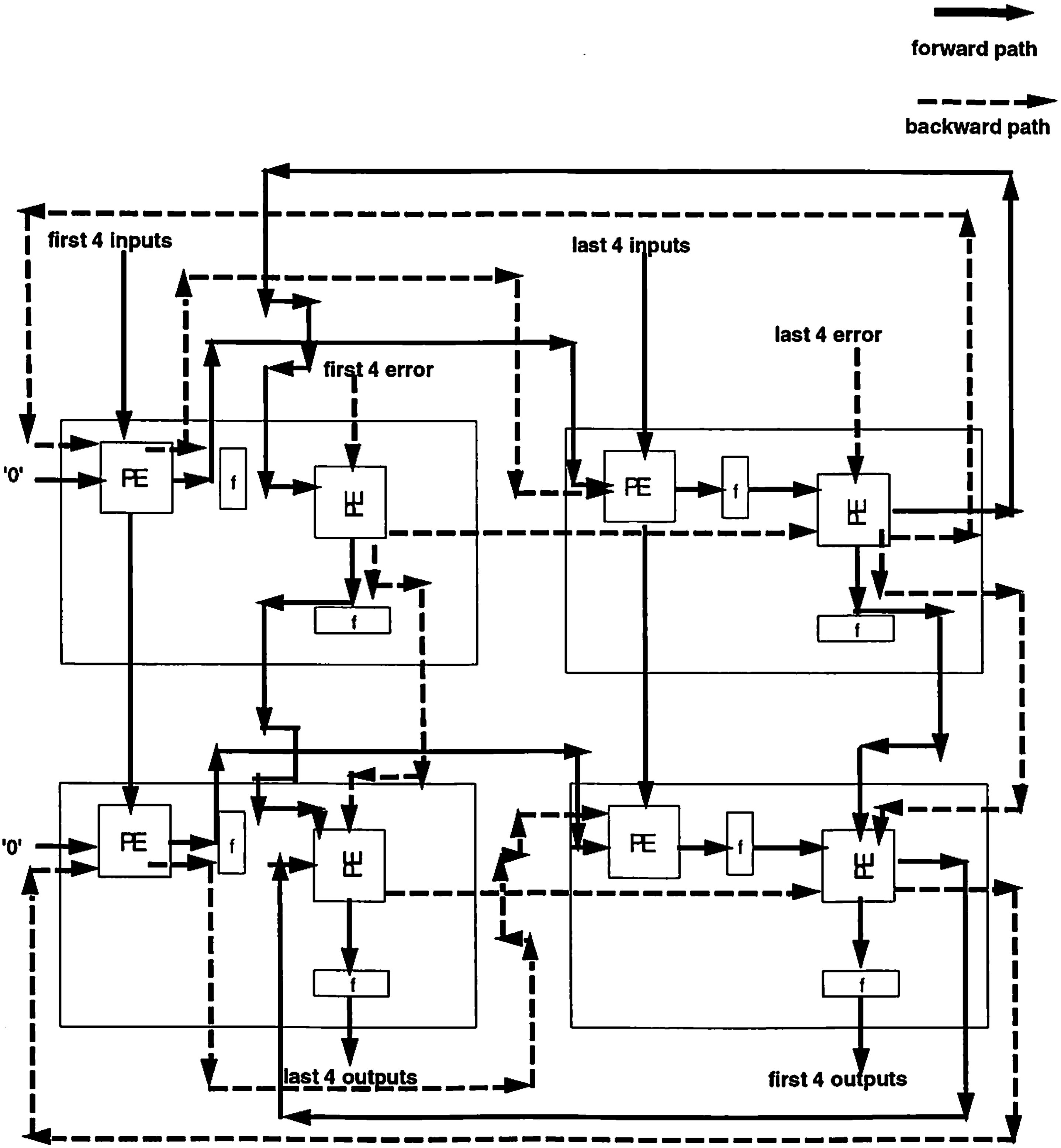




A 3-layer, 8-input, 8-output net work can be implemented by using 4 chips shown on the next page



A 3-layer, 8-input, 8-output net work can be implemented by using 4 chips shown below



EE599 VLSI Neurocomputing Term Project

04/29/91

Name: OKADA, Hiroto

SS.# 602-38-7312

Title: A Study of an Application of the Back-propagation Network

for the Surface Reconstruction.

1. Abstract

In this project, the software simulation of surface reconstruction using the back-propagation network has been reported. Surface reconstruction from sparse data has a lot of application because it can recover the 3 -D shapes of objects. For example, the inspection of products in a factory to distinguish no-good items from good items (*), the robotic eyes to find obstacle shapes from sensory data. The Neural Network is considered to demonstrate its strong features for surface reconstruction, because the relationship between the output (surface) and the input data (ex. sensory data) can not been known clearly. The neural network can construct the relationship in its system by learning. (actually, the surface reconstruction using the neural network is getting popular.)(1) We have chosen the back-propagation network, because there is the theorem proven which guaranty that the back propagation network can appropriate any functions in real world. For practical implementation, the initial conditions (the initial weights, the number of the hidden layers, and hidden layer neurons) should be defined appropriately to get the global minimum. Through the simulation using Matlab, the network for XOR,AND, and OR functions has been discussed. An approach to general functions has also been discussed.

2. The background theorem

Surface reconstruction using the backpropagation network is supported by the following theorem.(2)

Theorem: Given any $\varepsilon > 0$ and any L_2 function f: $[0,1]^n$-->R^m, there exists a three-layer backpropagation neural network that can approximate f to within ε mean squared error accuracy.

Although "three layers are always enough" in solving real-world problems it is often essential to have four, five, or even more layers. We also realize that although the theorem guarantees the ability of a multilayer network with the correct weights to accurately implement an arbitrary L_2 function, it does not comment on whether or not these weights can be learned using any existing learning law.

3. The back-propagation network

The back-propagation network, and algorithm shown in Fig. 1 (the one discussed in the class.)has been used.

The input and output ranges are normalized to [0,1], because the output is limited by the output range of the sigmoid function, [0,1]. (Fig.2)

4. The simulation of XOR, AND, OR functions

The simulation of XOR function (the one in Home work #4), AND, and OR functions are discussed as ones of simple cases. For each function, there are four input patterns. (Fig.3) The network used consists of the input layer of 2 input nodes and 1 bias node, 1 hidden layer which has 2 hidden nodes and 1bias node, and 1 output node in a output layer. (Fig.4)

The initial weights have been chosen such that the outputs for all input patterns are the midpoint (0.5)of the expected output (0 or 1). (Fig.5)

The training sequence: We tried the network for the four patterns, one by one, and then repeated the training, where the initial weights for a new pattern equal the final converged weights for the previous pattern. If the sum of absolute errors for a pattern is less than a threshold(ε_1), the training is changed to the next pattern, and if the sum of absolute errors for all the four patterns is less than another threshold (ε_2), all training stopped. The pattern sequence to be trained is; x1,x2,x3,x4,x1, (3) (Fig.6)

(1) XOR function: We examined four patter sequence shown in Fig.7. For all the sequence, the network was trained to perform XOR function, but with different iteration times, where 10% accuracy, i.e. $_1\varepsilon = 0.1, \varepsilon_2 = 0.4$ (Fig.8) We can find the pattern sequence (1), (2),in which the patterns of the expected output of 0 are fed into the network before the patterns of expected output 1, can get convergence with smaller iteration times. The simulation with 1% accuracy, i.e. $\varepsilon_1 = 0.01, \varepsilon_2 = 0.04$, was done. For the both cases, the final weights and outputs are shown in Fig.9 Using the final weight, the surface of XOR function are constructed for $(x, y) = (0.1 \times i, 0.1 \times j)$, where i,j = 0,1,2,...,10. The result graphs are shown in Fig.10. From the results, we can find the surface for 1% accuracy is more symmetric and smooth than that for 10% accuracy. The Matlab simulation program for XOR function is listed in list. 1

(2) AND, OR functions: With 10% accuracy, the training for AND, OR functions are done successfully. The results of the final outputs and iteration time are listed in Fig. 11, and the reconstructed surfaces are shown in Fig.12.

5. A Consideration for general cases

(1) The number of the hidden layer neurons:

In the result graph of the previous part, we can find that the shape of the transient region from 0 to 1 are kinds of a sigmoid function.(Fig.13) We named the distorted planes in Fig.13 as the sigmoid planes, and we made an assumption which define the relationship between the number of hidden layer neurons and sigmoid planes in the reconstructed space. (Appendix 1)

Assumption: The number of sigmoid planes in the reconstructed space is equal or less than the number of the hidden layer neurons. The results of XOR, AND, OR functions follow the above assumption.

(2) The initial weights

For systems for general cases (ex. ones have three neurons and more, or the inputs and outputs are no longer 0, nor 1.), we used the following initial weights, because the initial weight used for XOR function doesn't make the outputs the mean value of the expected outputs.

1) choose the weight such that they are uniformly distributed over (0,1).
2) choose the weights randomly.

(3) The training sequence

Besides the training sequence mentioned in part 4, we also used another sequence. Instead of feeding data sequentially, we choose one pattern randomly and train the network with it once, and then change the input pattern for next training. we choose the patterns randomly every time. the initial weights are the same as the converged weights for the previous pattern. The training continues until a certain iteration time (ex. 3000).(Fig.14)

(4) The simulation results

We tried to do simulation for the patterns shown in Fig.15 and 16 For the both case, the number of the hidden layer nodes are four. For Fig.15, we tried the simulations with several conditions. The conditions and results graphs are shown in Fig.17, 18. The results graph for Fig. 16 is shown in Fig. 19.We can't get the convergence for the both cases. It is considered the reason that the initial weights are wrong so that they leads a local minimums. With the correct initial weights for XOR function, we succeeded the simulation for the patterns in Fig.20 and 21, which have small difference from XOR, OR(NOR) functions with two hidden layer nodes. The results are shown in Fig.22 and 23. These results show the difference from these of XOR, OR(NOR) functions.The Matlab program list for Fig. 17 are listed in list. 2.

6. Conclusions

In this project, we have constructed the back-propagation network which perform XOR, AND, OR functions by learning. From the results of these, we suggest the assumption regarding the number of hidden layer neurons, which gives a key to general cases. We also found the successful network can distinguish the difference of the input data.

We can't get the deterministic way to find the initial weights. The assumption is to be proven in mathematics

From the discussion of the sigmoid plane (Appendix 1), we suggest to use the function like the Gaussian function , which has a down slope as well as an up slope, as the transfer function instead of the sigmoid function. (4) Because it is considered to have more possibility that it can fix-dumps and dimple in the surface.

Reference

(1) Suggestion from Mr. . Ph.D student in the Department of Mathmatics,U.S.C

(2) Robert Hecht-Nielsen. "Neurocomputing", Addison Wesley, pp132-133.

(3)L. Wang, J.M.Mendal, Structured Trainable Networks For Matrix Algebra, SIPI, U.S.C

(4) Suggestion from Mr. S. Shimoji. Ph.D student of the Department of Computer Science, U.S.C.

(*) RICOH ltd. Japan, has developed the system to inspect the products of lens using similar neral network.

Appendix 1

The sigmoid plane

In the back-propagation network, the sum of a hidden layer neuron is given;

$Z = aX + bY + c$ (1)

where X, Y are the inputs, a,b,c are the weights for the hidden layer.

Eq. (1) represent a plane in the space (X,Y,Z). The output of the hidden layer, f(*), can be considered to be a distorted plane by the sigmoid function. In this project, we call the one the sigmoid plane. (Fig. A1)

As for the output neuron, the output (surface) is also considered as a sigmoid plane, where the inputs are the sum of the sigmoid plane for the hidden layer.

$W = \alpha Z + \beta V + \gamma$

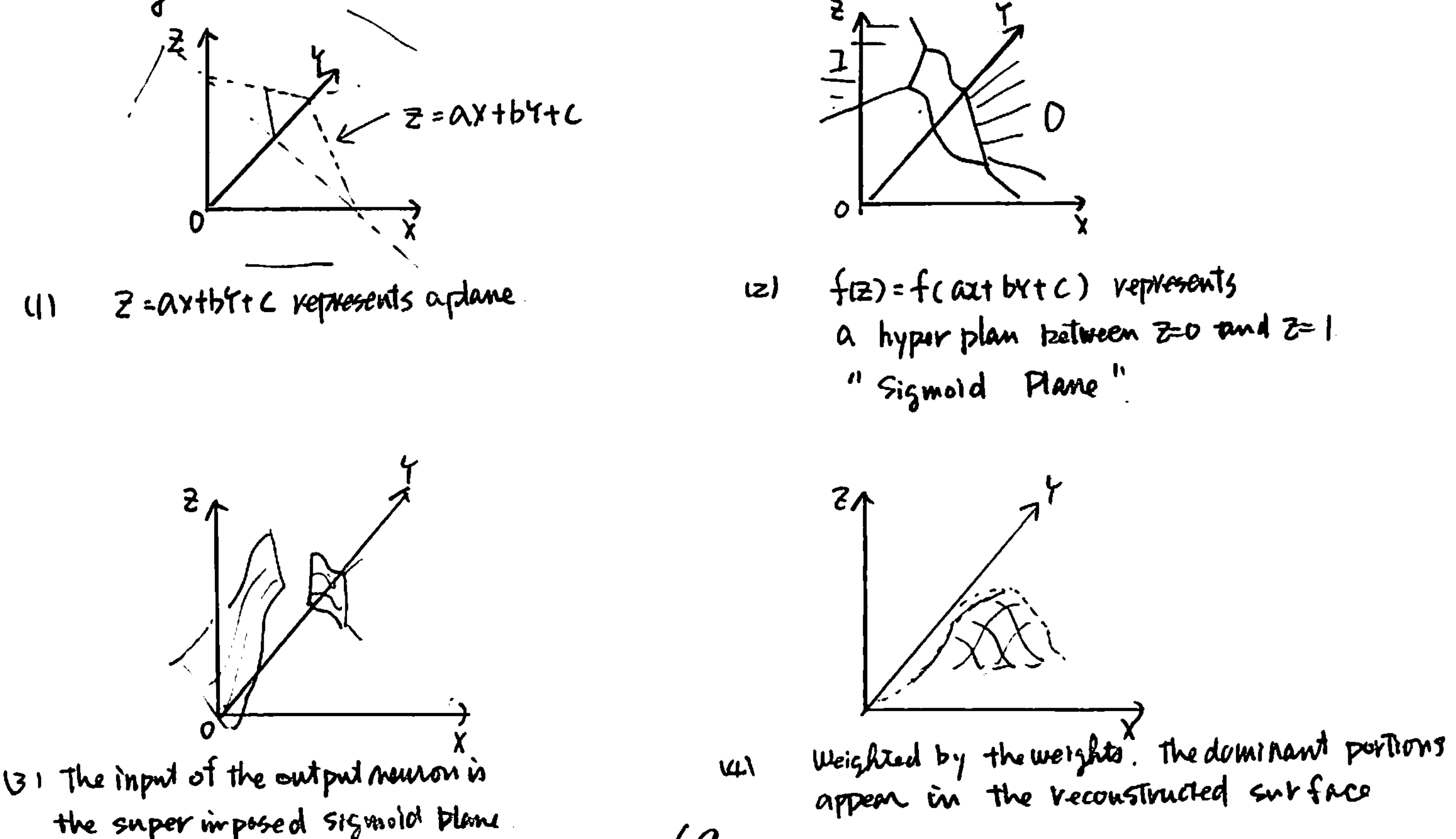
The surface = f(W)

where Z,V are the output of the hidden layer neurons, α, β, γ are the weight for the output layer.

Now we note that the sigmoid function is monotony increasing function. The surface consists of the factor of Z, and V in the sense of superimpose in a linear system.

Therefore, the surface reconstructed by the system with two hidden layer neuron is considered to have two portions made by sigmoid planes of the hidden layer neurons. And this is able to be expanded for general cases.

Fig. A1



(1) Z = aX+bY+c represents a plane

(2) f(Z) = f(aX+bY+c) represents a hyper plan between Z=0 and Z=1 "Sigmoid Plane"

(3) The input of the output neuron is the super imposed sigmoid plane

(4) Weighted by the weights. The dominant portions appear in the reconstructed surface

The Back-propagation Network Algorithm

(1). Learning Rule:

1): Output layer:

weighted error $\delta_i = (t_i - a_i) f'(CS_i)$
where t_i= expected output value for neuron j
a_i = actual output value for neuron j
S_i = weighted sum to neuron j
$f(*)$ = neuron transfer function, = $\frac{1}{1+\exp(-CS_i)}$, where c is a coefficient.
$f'((*)$ = derivation of the transfer function

2): Hidden layer:

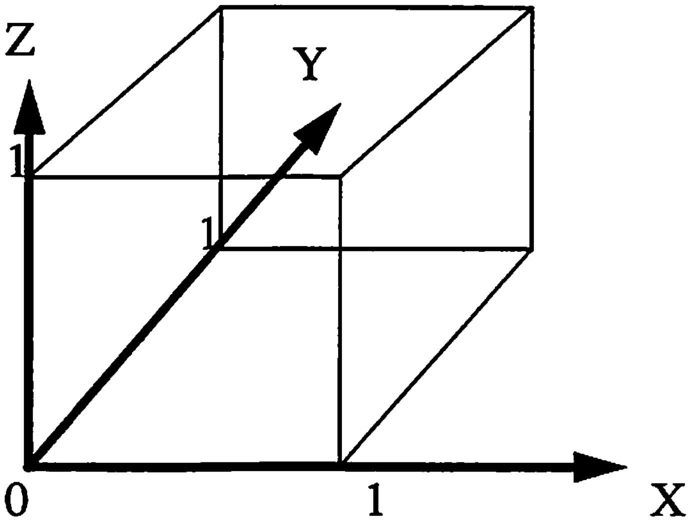
weighted error $_j = \left| \sum_k \delta_k W_{kj} \right| f'(CS_{\cdot}$
where δ_i = weighted error from the next layer
W_k = synapse weight connecting neuron j to neuron k in the next layer

For the both layers,

$$W_{ii}^{new} = W_{ii}^{old} + \alpha a_i \delta_{ij}$$

where $0<0.25<\alpha<0.75<1$

Fig.1 The Back-propagation Network Algorithm



X = xi / max{ xi }
Y = yi / max{ yi }

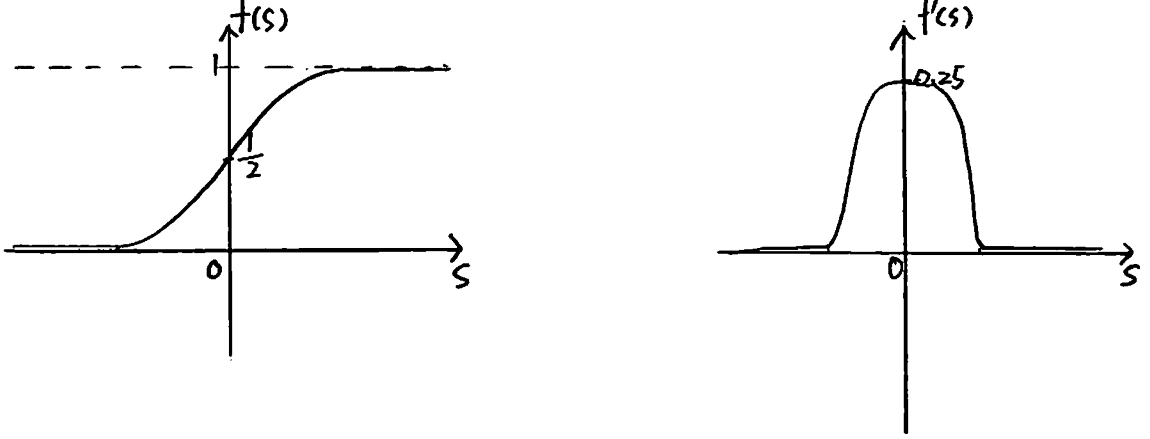


Fig.2 The normalization of the input, output & sigmoid function

X	0	0	1	1
	0	1	0	1
Z	0	1	1	0

Fig. 3 The true table of XOR function

* Initial Condition

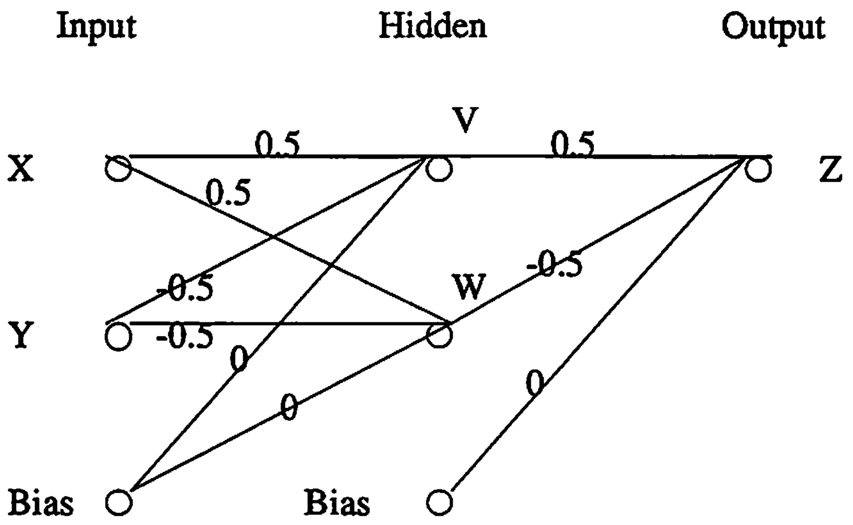
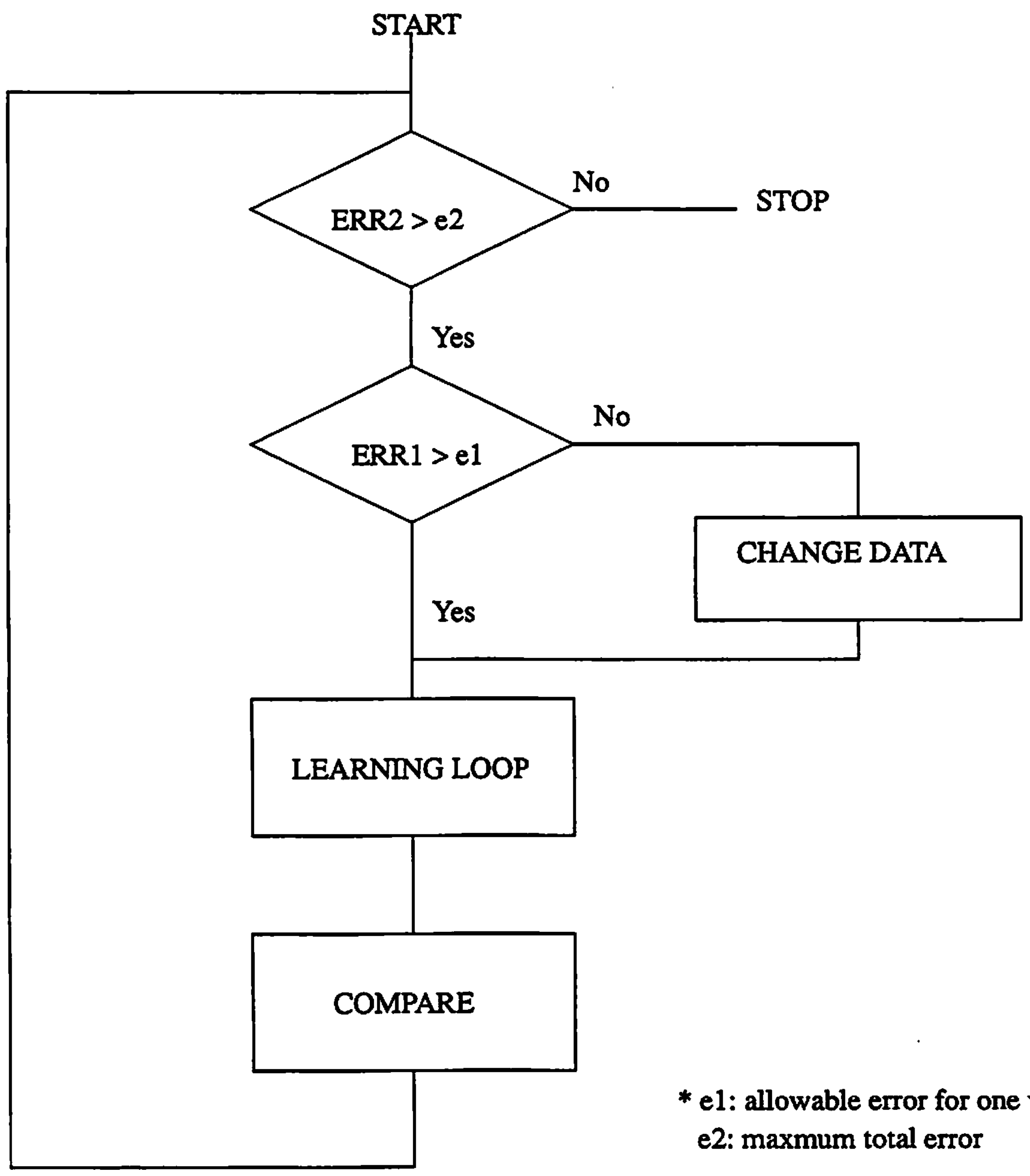


Fig.4 The Initial Conditions

Z	0.5	0.5	0.5	0.5

Fig. 5 The expected outputs for Fig.5



* e1: allowable error for one vector
e2: maxmum total error

Fig. 6 The pattern sequence (1). Wang & mendal's method

	X1	X2	X3	X4
X	0	0	1	1
Y	0	1	0	1
E	0	1	1	0

1) XOR 1

	X1	X2	X3	X4
X	0	1	0	1
Y	1	0	0	1
E	1	1	0	0

2) XOR 2

	X1	X2	X3	X4
X	0	1	0	1
Y	0	1	1	0
E	0	0	1	1

3) XOR 3

	X1	X2	X3	X4
X	0	0	1	1
Y	1	0	1	0
E	1	0	0	1

4) XOR 4

Fig. 7 The Sequence Pattern for XOR

	Iterarion time x1, x2, x3, x4	The final output results x1, x2, x3, x4
XOR 1	7, 4, 6, 3	0.0447, 0.9057, 0.8254, 0.0352
XOR 2	1087, 19, 130, 19	0.9717, 0.9950, 0.0385, 0.0533
XOR 3	8, 6, 11, 4	0.0008, 0.0217, 0.8527, 0.8805
XOR 4	639, 366, 49, 86	0.9871, 0.0260, 0.0243, 0.9526

Fig. 8 The Training Results for Fig. 7

1) 10% accuracy

$$H = \begin{bmatrix} 0.5012 & -0.6370 & -0.2920 \\ 0.6586 & -0.6204 & 0.4009 \end{bmatrix}$$

$$O = \begin{bmatrix} 0.5637 & -0.6363 & 0.2899 \end{bmatrix}$$

$$Z = \begin{bmatrix} 0.0447 & 0.9057 & 0.8254 & 0.0352 \end{bmatrix}$$

# of iterations = 20

2) 1% accuracy

$$H = \begin{bmatrix} 0.7338 & -0.6634 & -0.3945 \\ 0.7489 & -0.8067 & 0.4588 \end{bmatrix}$$

$$O = \begin{bmatrix} 0.9916 & -0.9952 & 0.4954 \end{bmatrix}$$

$$Z = \left[0.0090 \; 0.9907 \; 0.9900 \; 0.0011 \right.$$

# of iterations = 2570

Fig. 9 The training Results for XOR function

Fig. XOR function

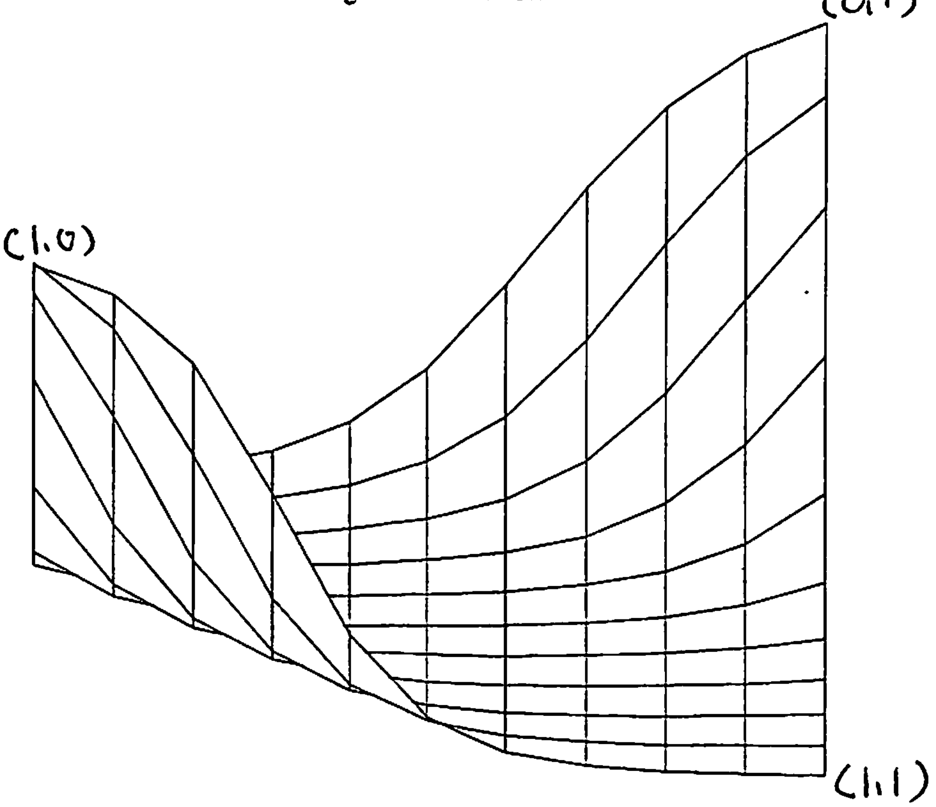


Fig. XOR Function (2)

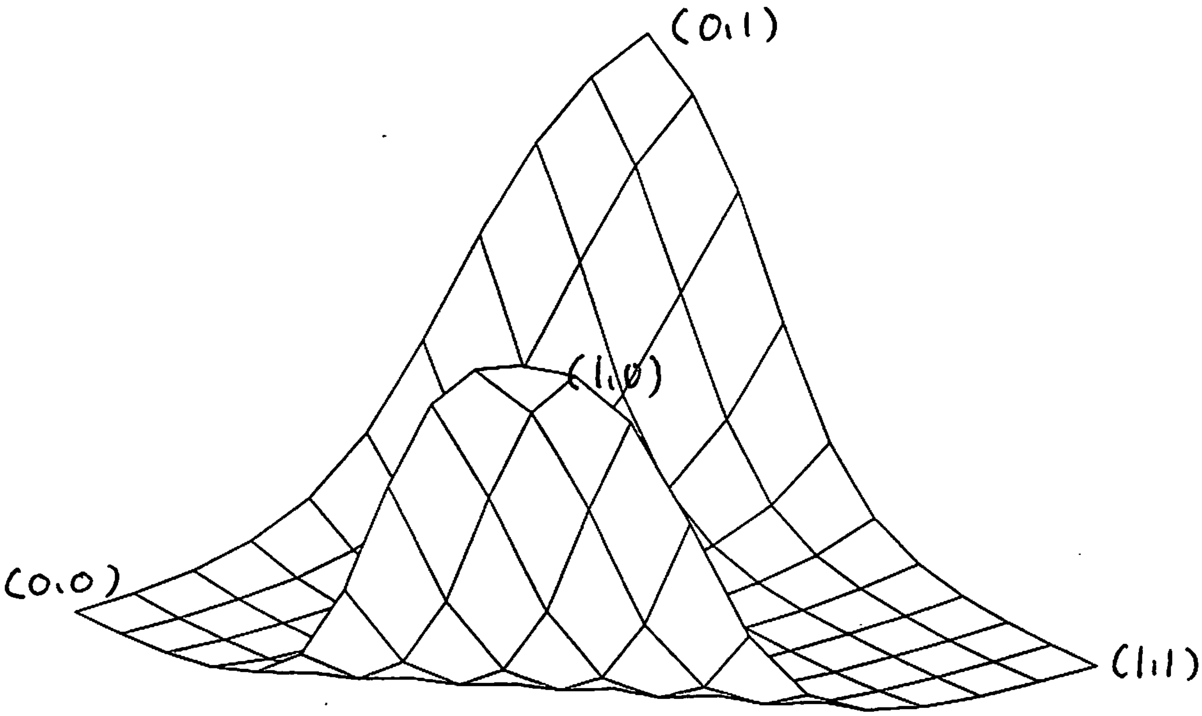


Fig. 10 The Constructed Surface of XOR

(1) 10% Accuracy

Fig. XOR Function. 1% (2)

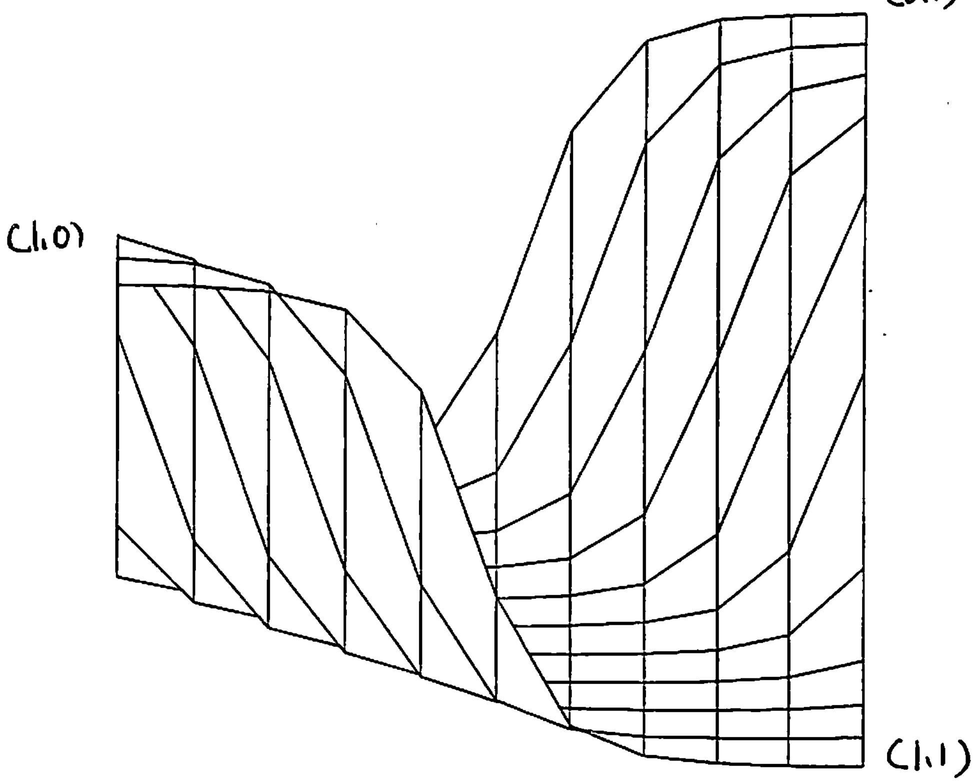


Fig. XOR Function. 1%

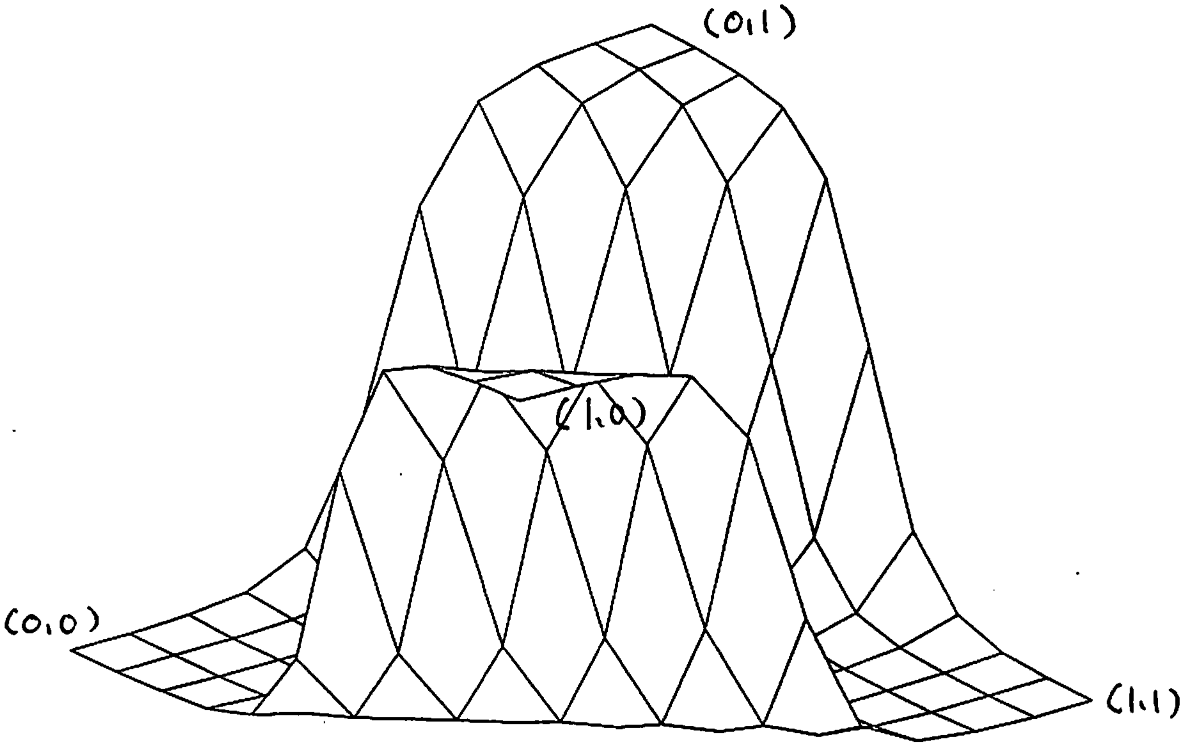


Fig. 10 The Constructed Surface of XOR
(2) 1% Accuracy

(1) OR pattern -(1)

	x1	x2	x3	x4
X	0	0	1	1
Y	0	1	0	1
E	0	1	1	1

The final Results	0.3120, 0,9982, 0.9997, 1.000
Iteration time	439, 12, 3, 3

(2) OR pattern -(2)

	x1	x2	x3	x4
X	1	0	1	1
Y	1	1	0	0
E	1	1	1	0

The final Results	0.9992, 0.9999, 0.9955, 0.0605
Iteration time	15, 7, 55, 1022

(3) AND pattern -(1)

	x1	x2	x3	x4
X	0	0	1	1
Y	0	1	0	1
E	0	0	0	1

The final Results	NO CONVERGENCE
Iteration time	NO CONVERGENCE

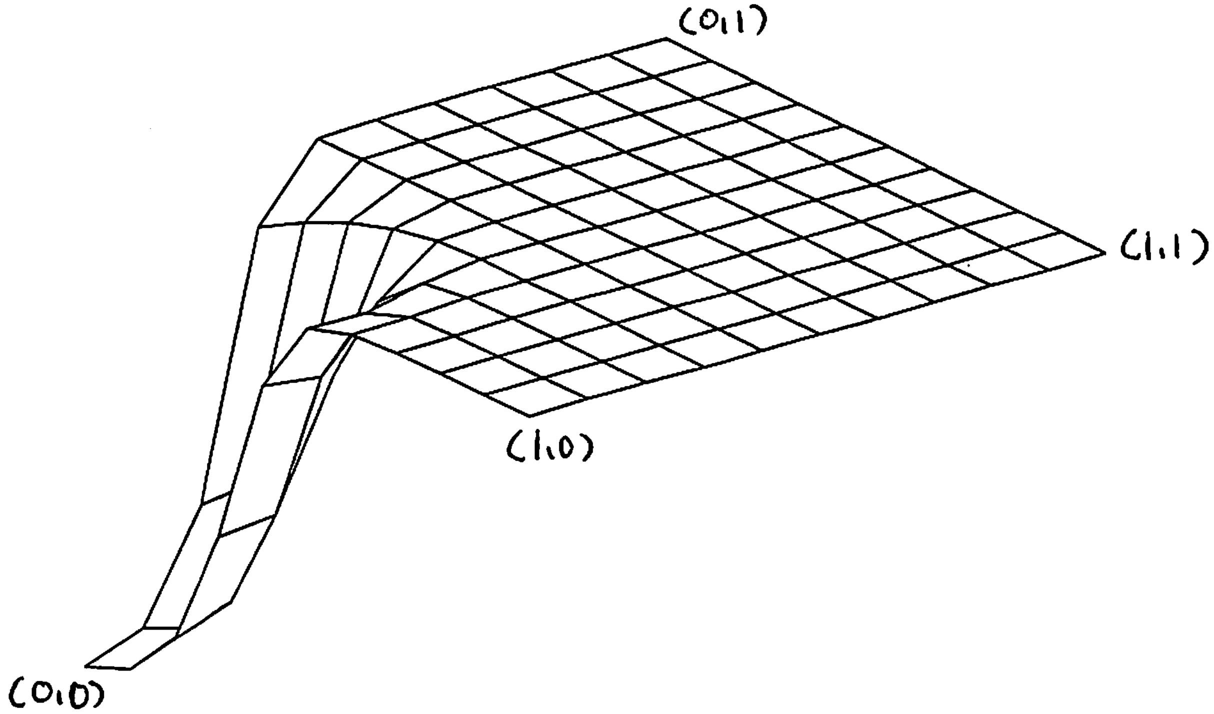
(1) OR pattern -(1)

	x1	x2	x3	x4
X	1	0	1	0
Y	1	1	0	0
E	1	0	0	0

The final Results	0..8004, 0.0394, 0.0237, 0.0287
Iteration time	124, 450, 20, 20

Fig. 11 The pattern sequence and the final outputs of OR, AND functions

The Surface of OR (3)



The Surface of OR Function (4)

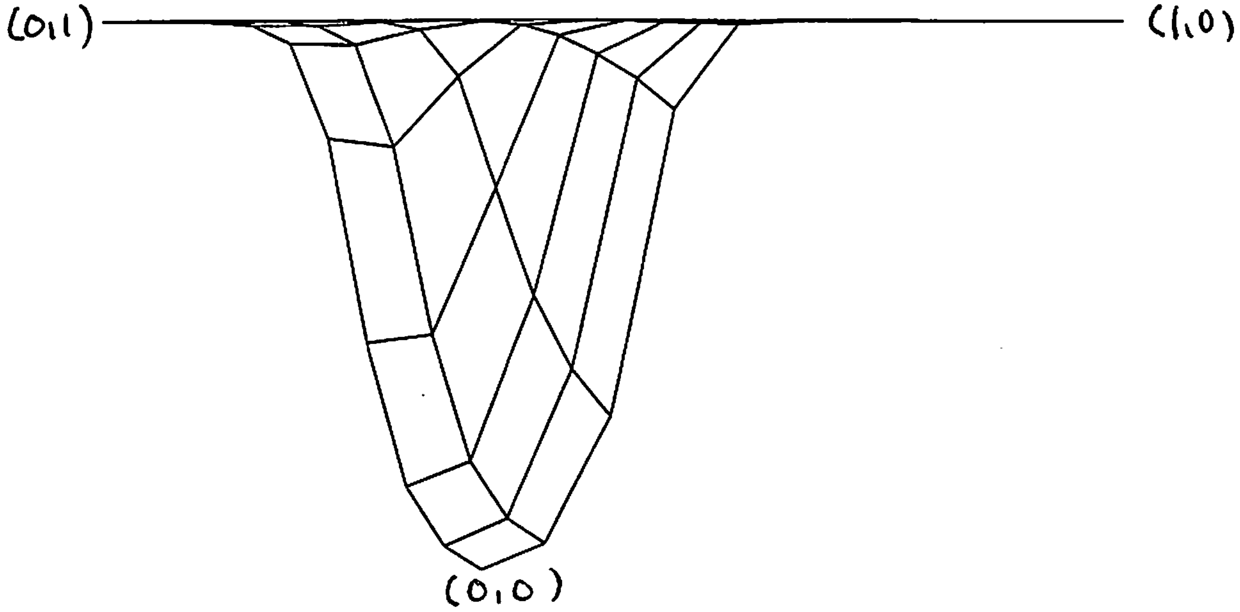
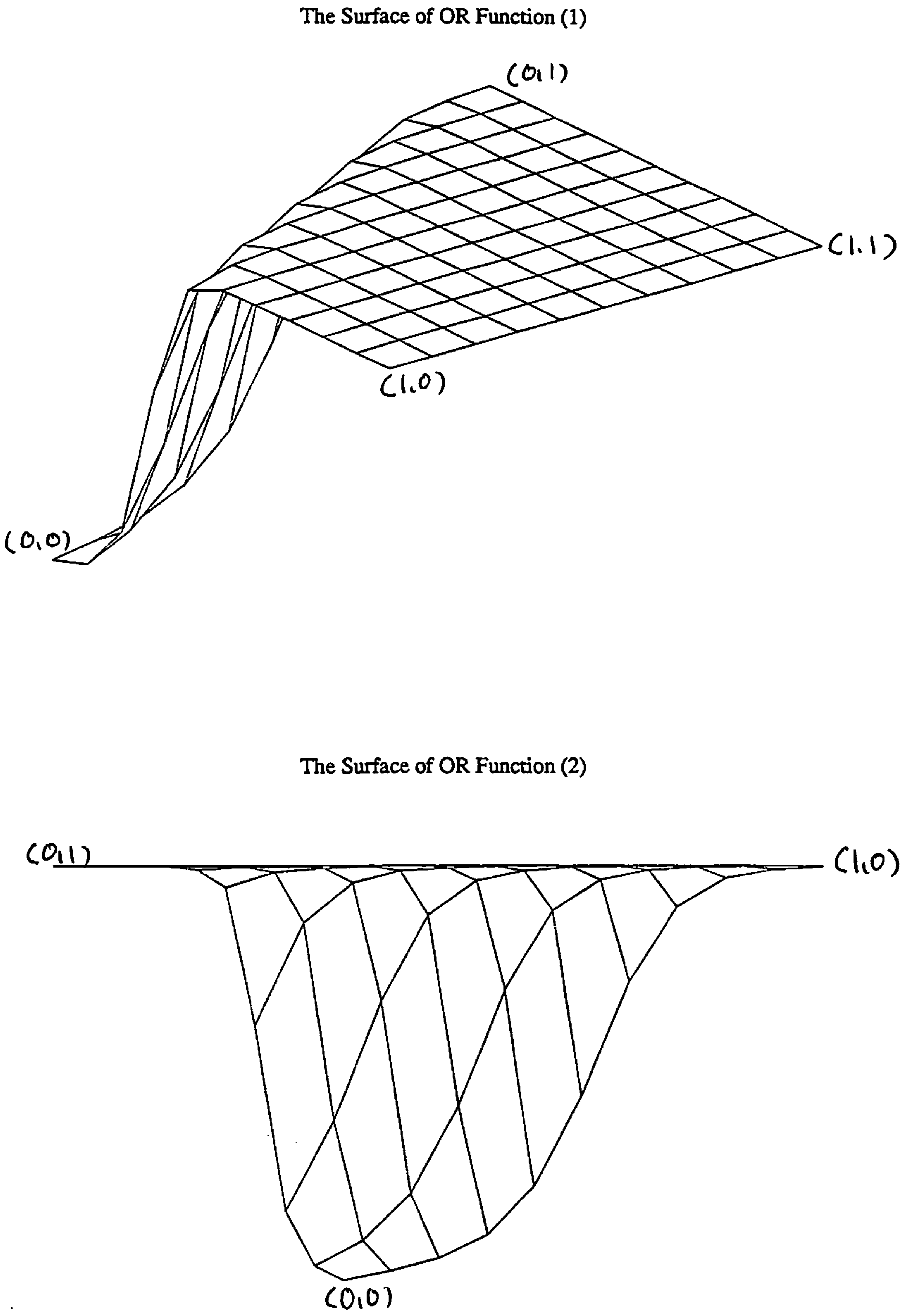


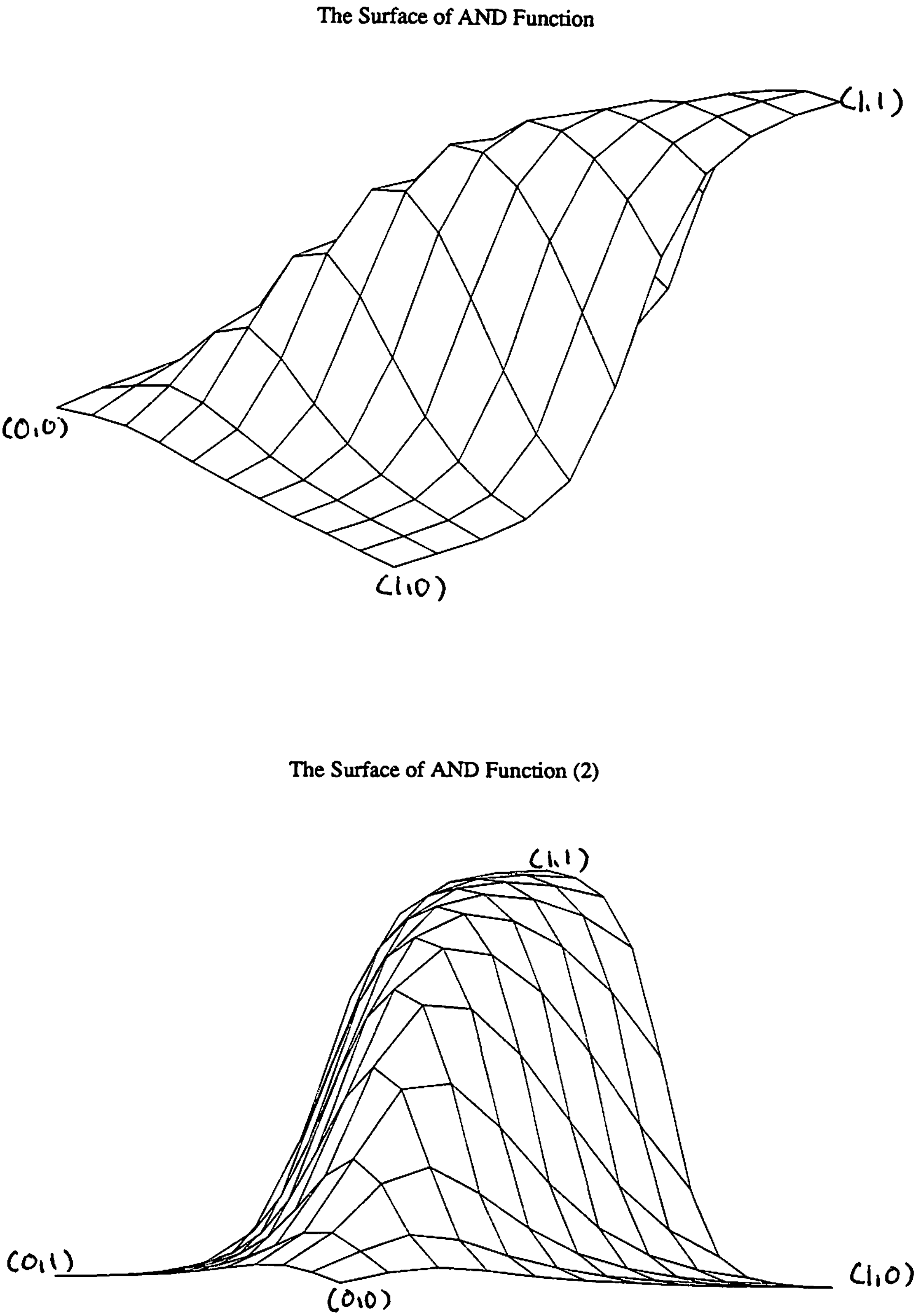
Fig. 12 The Constucted Surface of OR. AND function.

(a) OR pattern (1) in Fig.11


The Surface of OR Function (1)

The Surface of OR Function (2)

Fig. 12 The Constructed Surface of OR. AND function
(b) OR pattern (2) in Fig. 11


The Surface of AND Function

The Surface of AND Function (2)

Fig.12 The Constructed Structur of OR. AND function
(c) AND pattern in Fig.11

Fig. XOR Function. 1% (2)

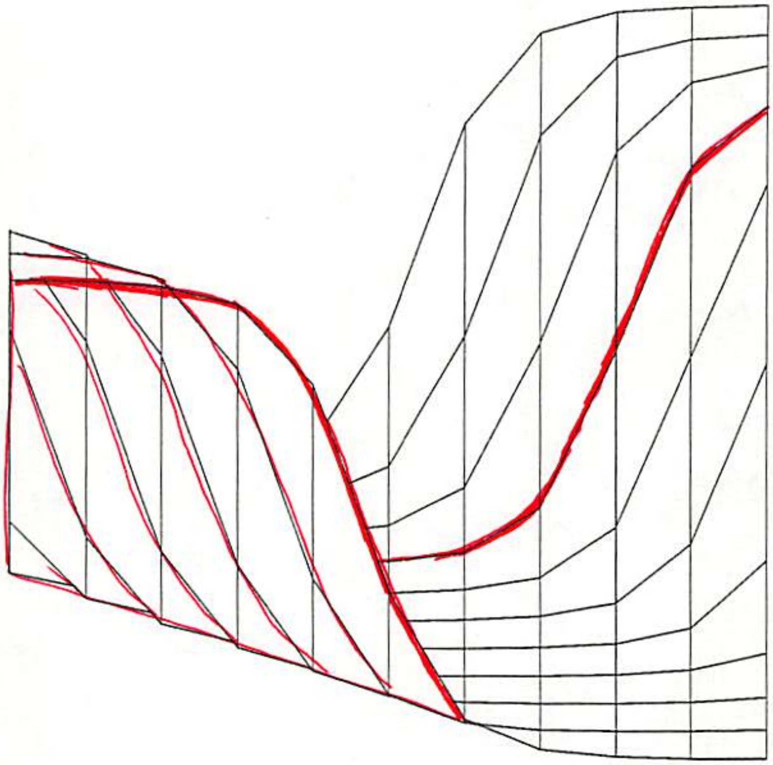
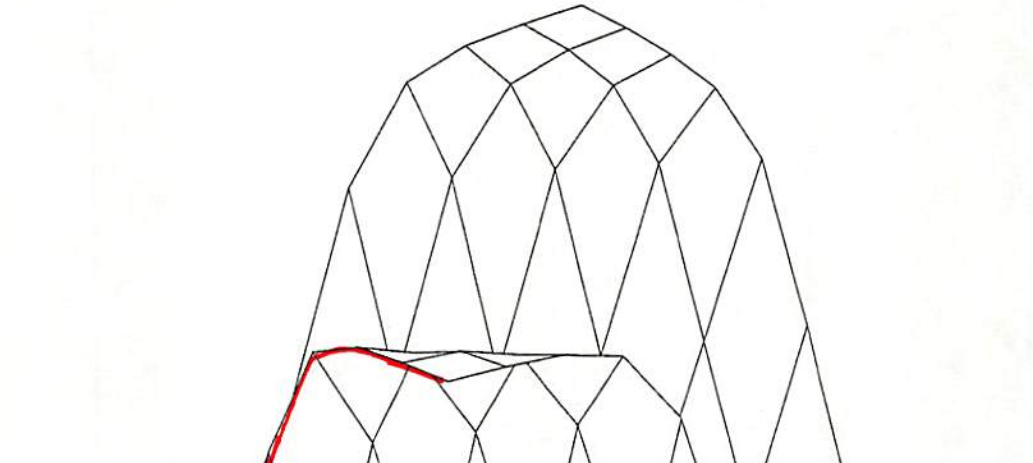


Fig. XOR Function. 1%



The contour can be considered to be sigmoid function

Fig, 13. Sigmoid function in the Reconstructed Surface.

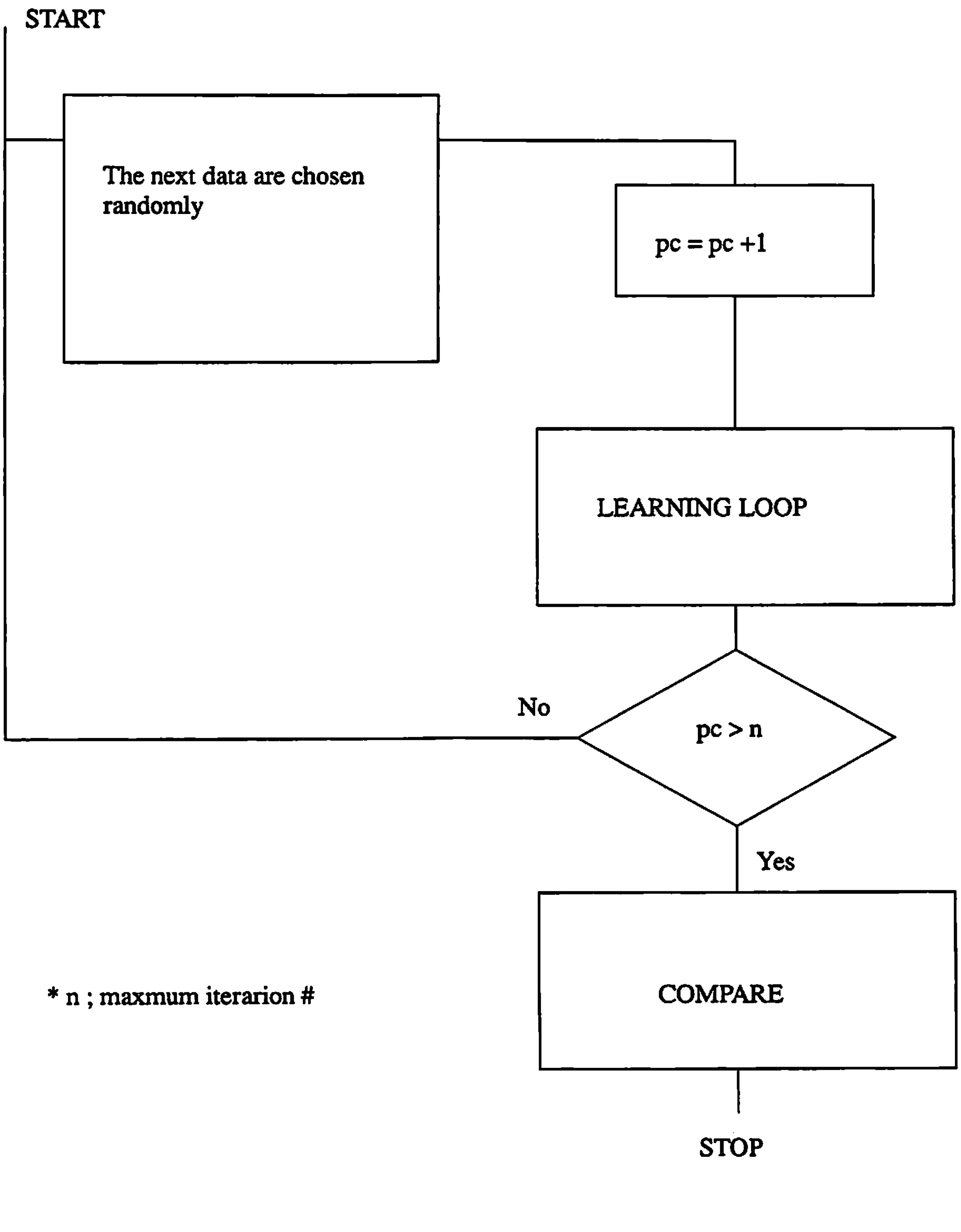


Fig.14 The pattern sequence (2)

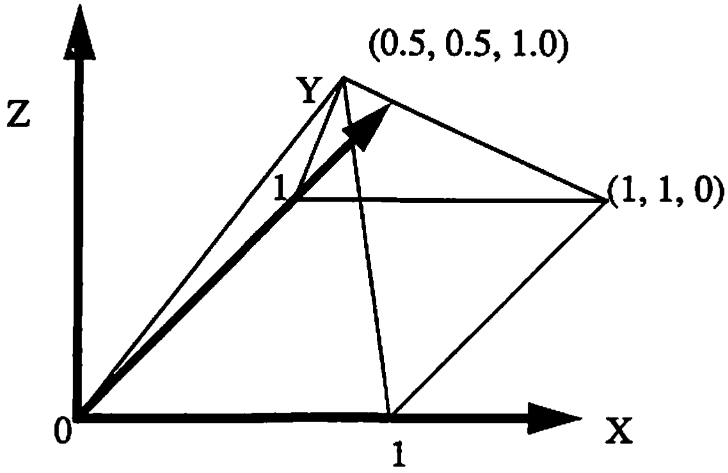


Fig. 15 Input Pattern (A)

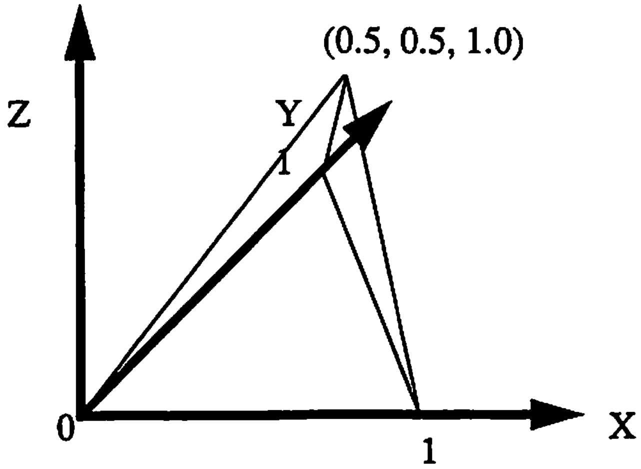


Fig. 20 Input Pattern (B)

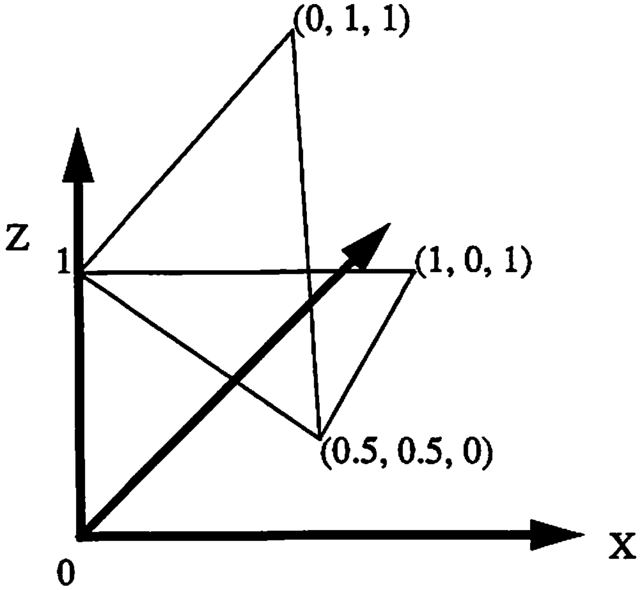


Fig. 21 Input Pattern (C)

The Input data (sinx/x)

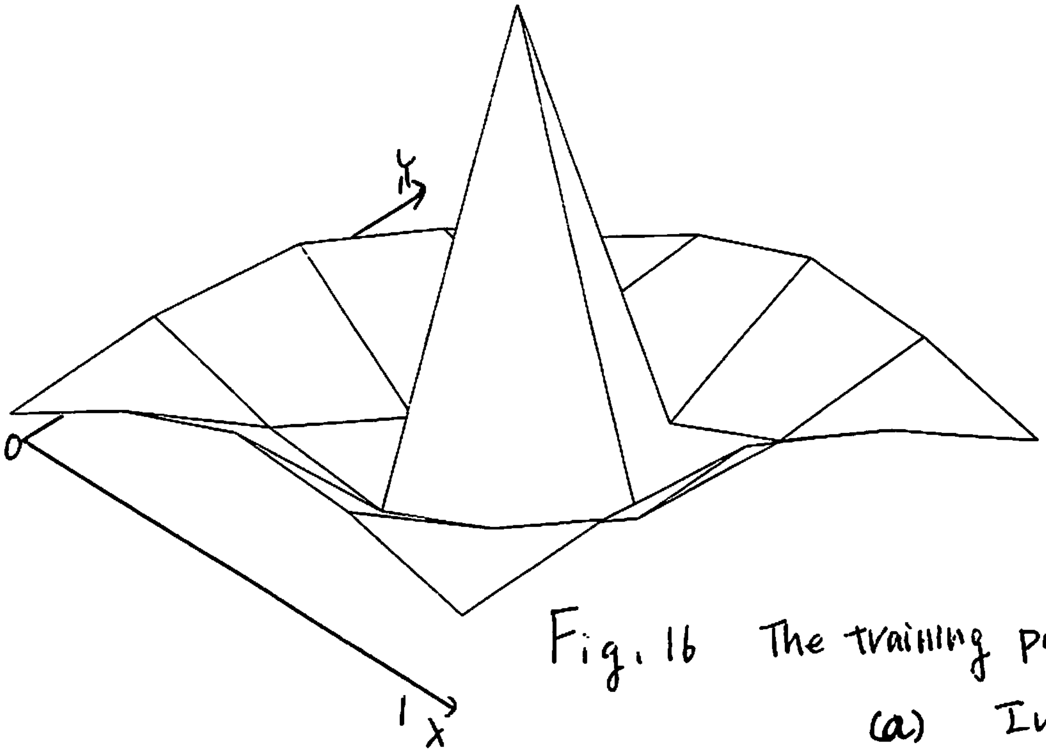
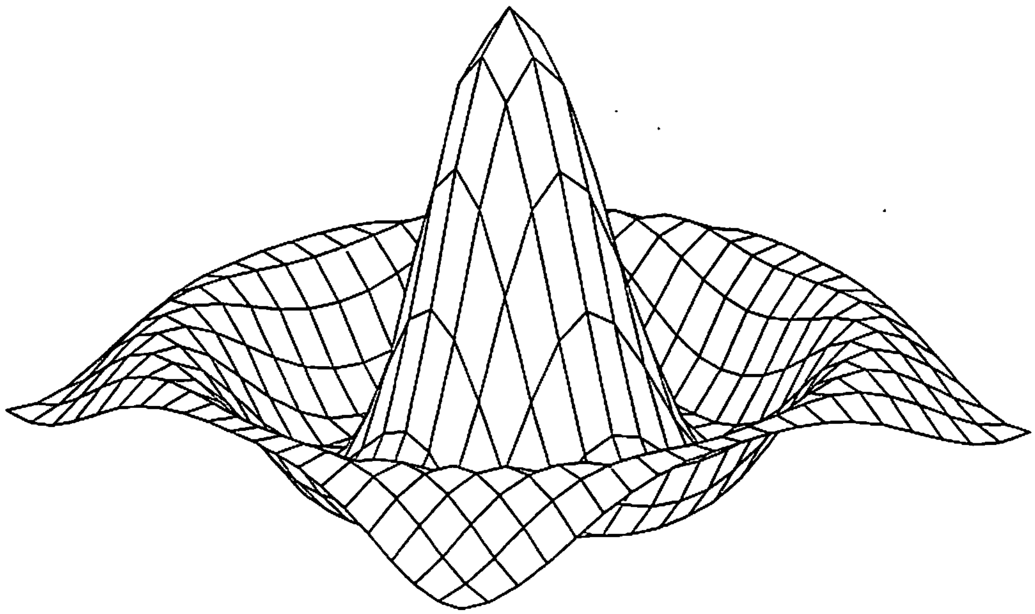


Fig. 16 The training pattern.
(a) Input data

Fig. Z=0.5*SIN(R)/R+0.3, R=sqrt(X^2+Y^2)



(b) Ideal output (Expected Reconstructed surface)

	The Initial Weights Hidden l. Output l.	The Pattern Sequence	* Matlab File Name
1	H : The Same as those of XOR O: Random	Wang & Mendal's method	TT
2	H: The Same as those of XOR O: The Same as those of XOR	Wang & Mendal's method	SS
3	H: The Same as those of XOR O: Random	Random	Ram2
4	H: Random O: Random	Random	Ram3

* refer to list. 2

Fig. 17 The Conditions for the pattern of Fig. 15

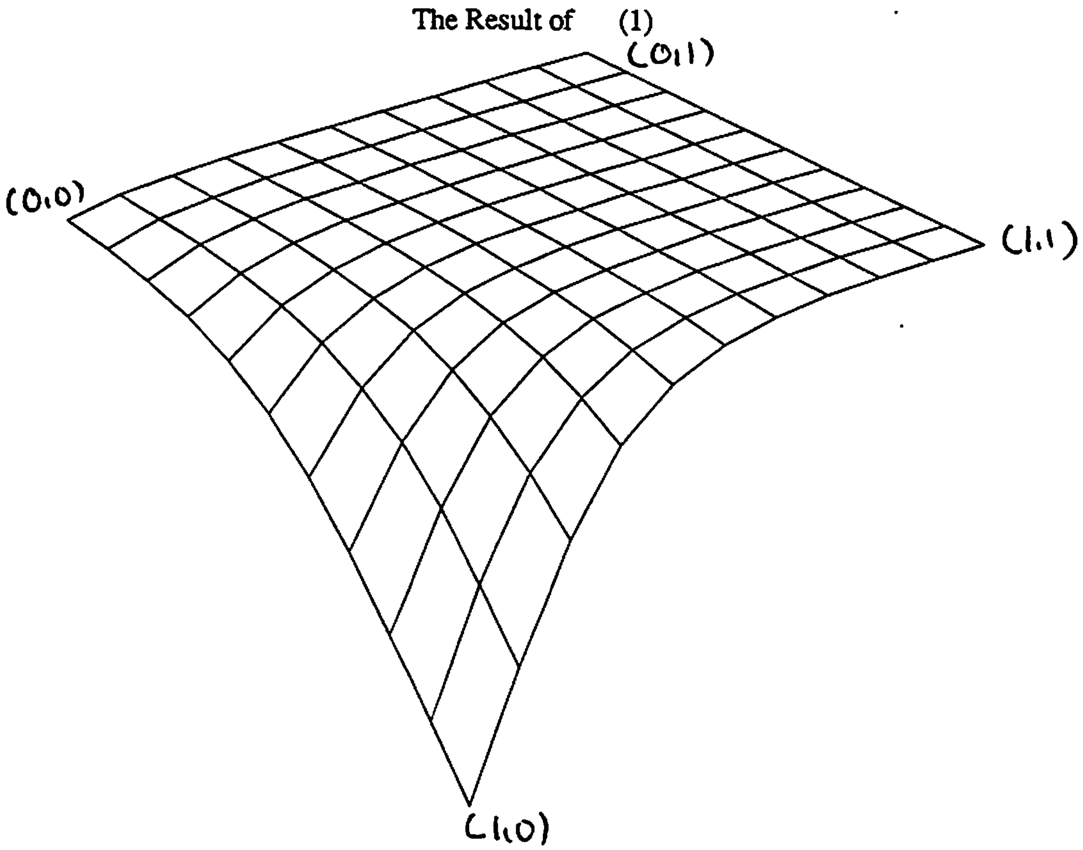
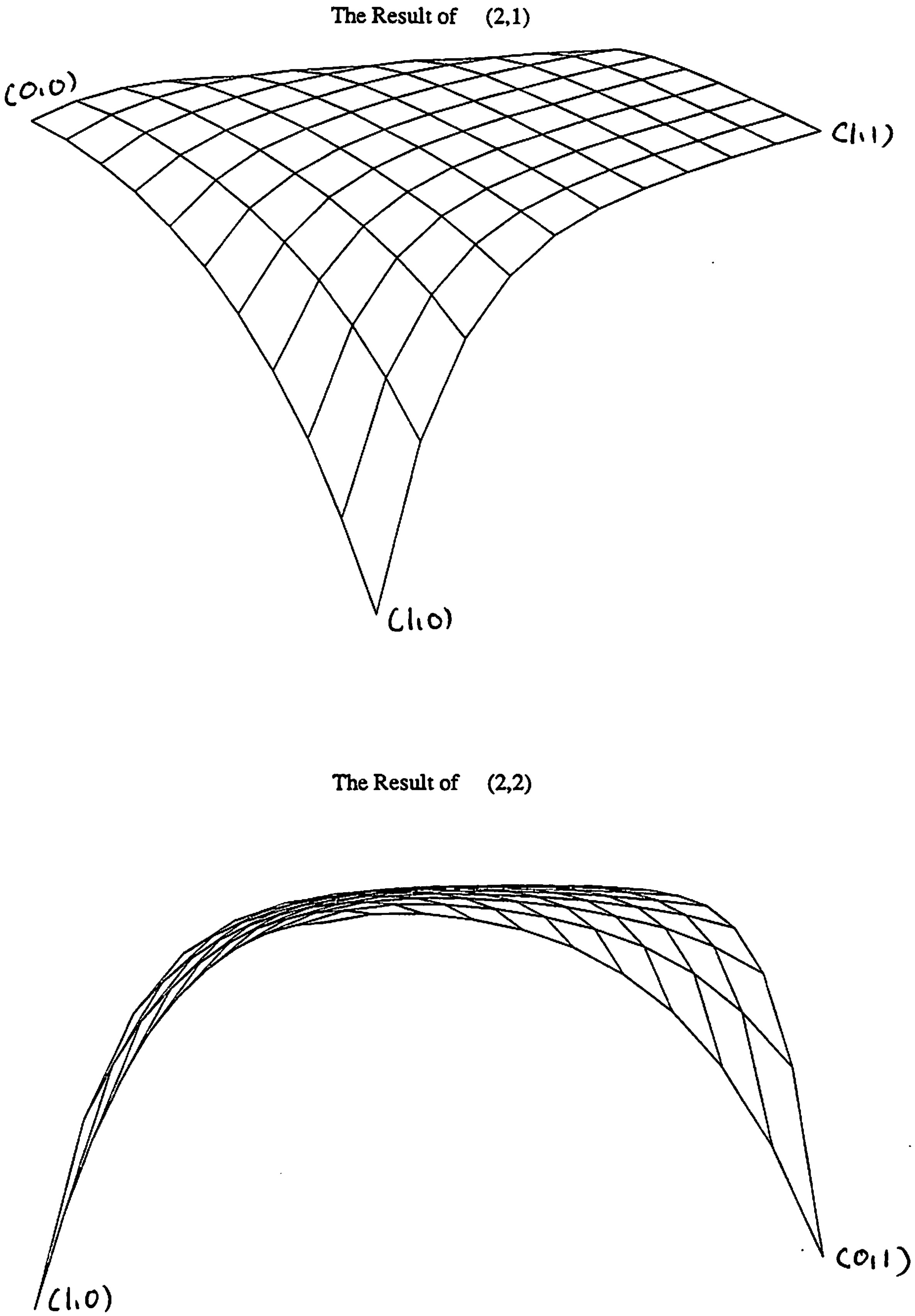


Fig. 18. The Reconstructed Surface

(a), for the case of (a) in Fig. 17

(NOT CONVERGED)


The Result of (2,1)

The Result of (2,2)

Fig. 18. The Reconstructed Surface

(b). for the case of (b) in Fig. 17

(NOT CONVERGED)

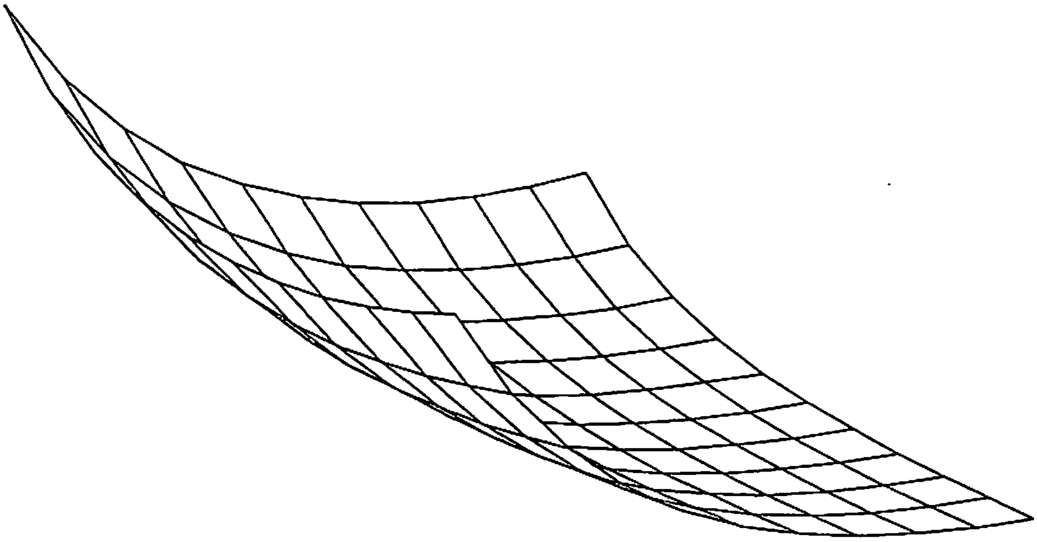


Fig. 18(c) For the case in fig. 17(c)
(Not Converged)

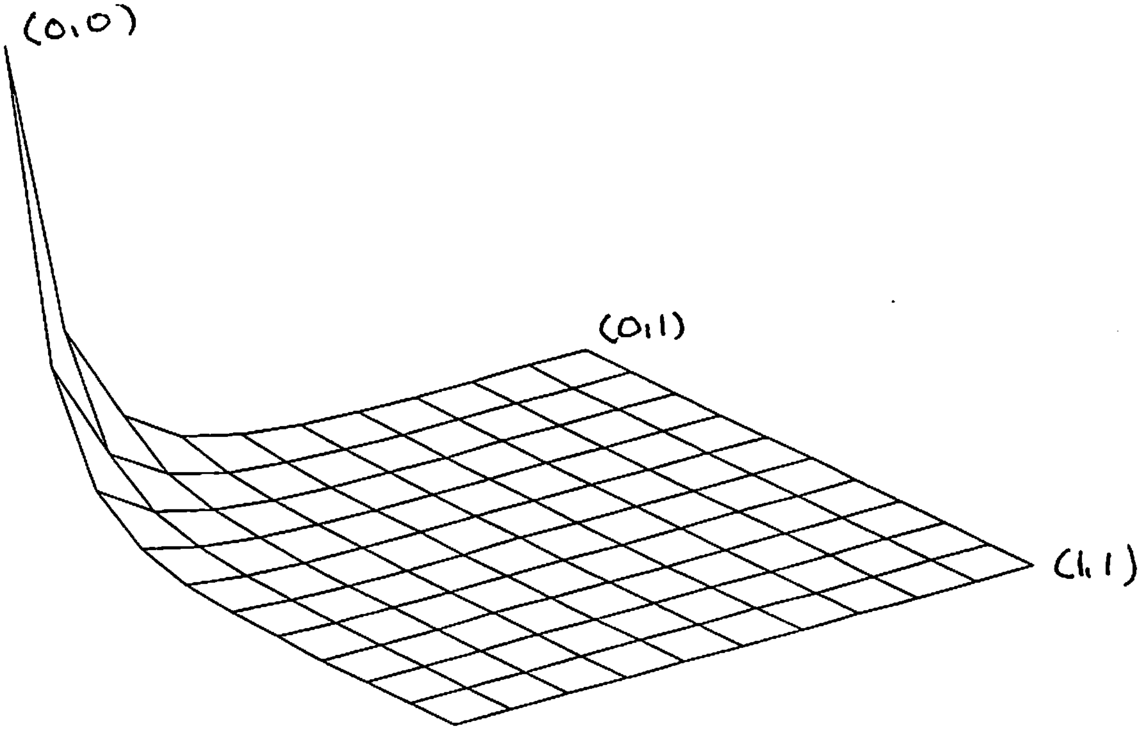


Fig. 18(d) For the case in fig. 17(d)
(Not Converged)

The result of training for sinx/x

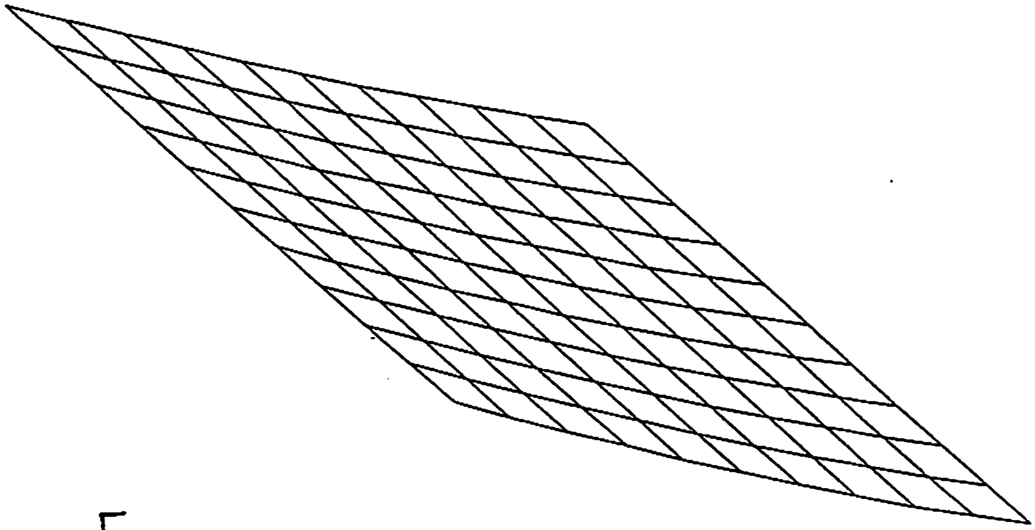


Fig. 19 The Actual Reconstructed Surface for Fig. 16

The Surface of pattern

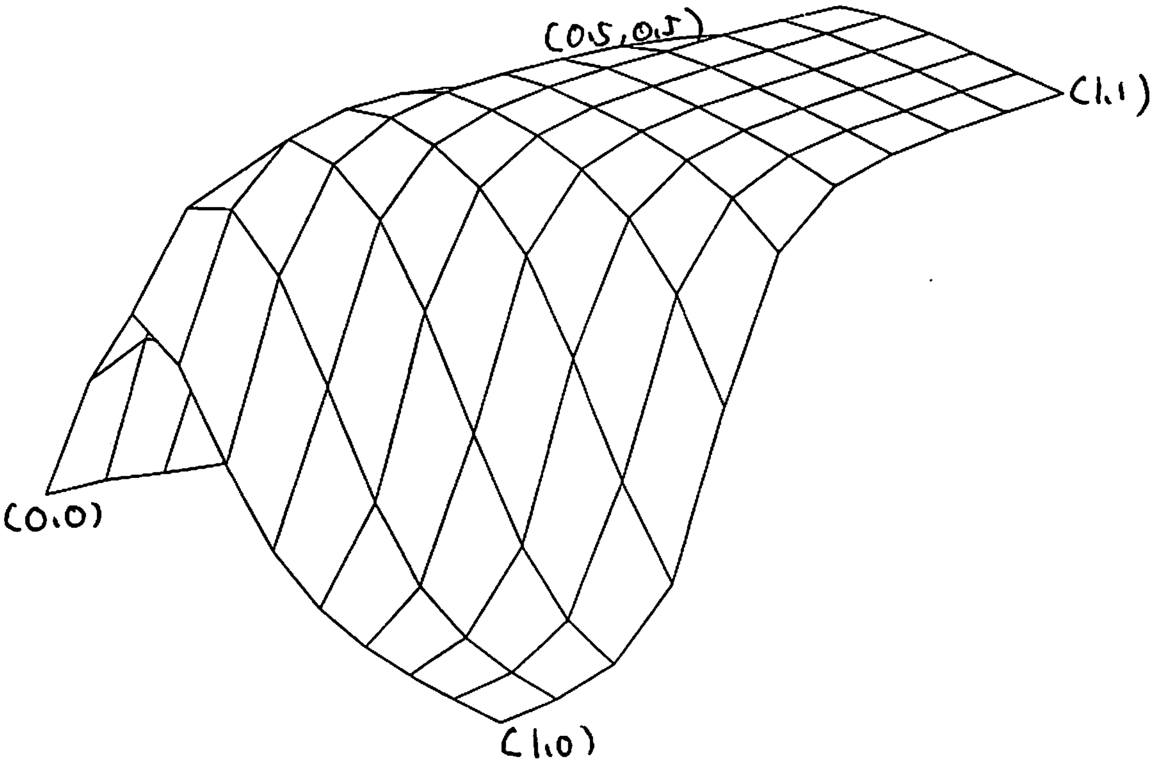
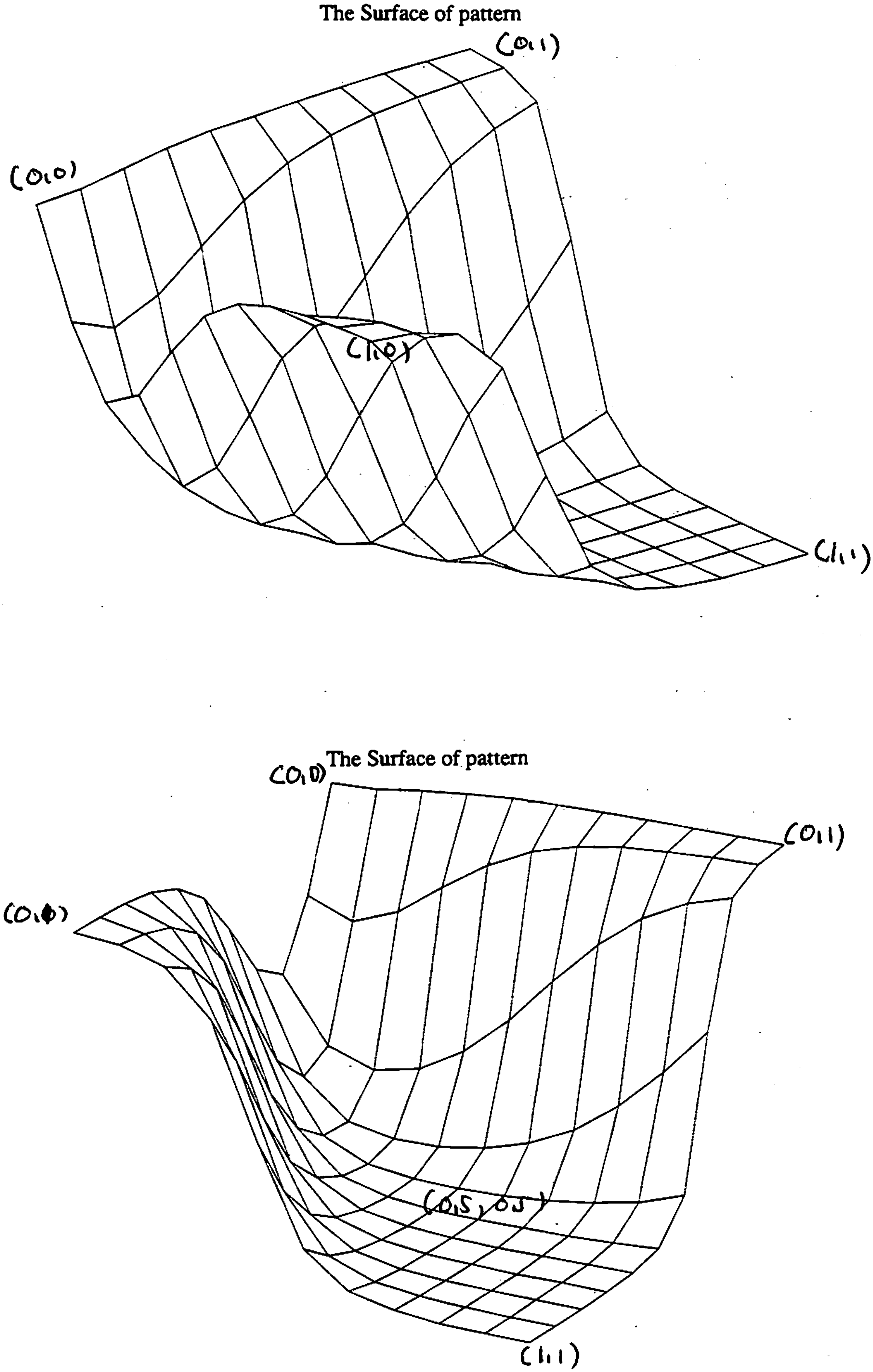


Fig. 22 The Reconstructed Surface for Fig. 20


The Surface of pattern

The Surface of pattern

Fig. 23. The Reconstructed surface for Fig. 21.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                                                              %%
%%           The Network Training for EX-OR function                            %%
%%                                                                              %%
%%           The MATLAB program             ver. 1.2                            %%
%%                                          03/29/91     by Hiro Oakada         %%
%%                                                                              %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  The definition of the intial matrices
%
%%
%clear
C = 10
ni = 4
%
%  The Input Vector X ( x1', x2', x3')
%
%       X = [   0.5     0       1       0
%               0.5     1       0       0
%               1       1       1       1       ]
%
%  The Weight of the Hidden Layer  H
%
        H = [   0.5     -0.5    0.0
                0.5     -0.5    0.0     ]


%
        H1 = [  8.0     -3.9    2.9
                7.7     3.5     0.7     ]
%
%%%%%
%
%  The Weight of the Output layer  O
%
        O = [   0.5     -0.5    0.0     ]
%
        O1 = [  -5.5    5.6     0.1     ]
%
%%%%%%%%
%
%  The Desired Output Vector  D
%
%       D = [   0       1       1       1       ]
%
%%%%%%%%%%%%%%%%%%
%
%  THE INITIAL INTERNAL VALUES
%
        pc =    zeros(D)
        ERR1 =  ones(D)
        Z =     ones(D)
        r = 0.25
%
%
%%%%%%%
%
ERR2 = 0
for i = 1:ni
                        ERR2 = ERR2 + abs( D(i) - Z(i) )
end
%
while   ERR2 > 0.1*ni
```

The initial weights
& Input data

```
%
        for     k = 1:ni
%
                while   ERR1(k) > 0.1
%
                        pc(k) = pc(k)   + 1;
%       Caliculation of the sum of the hidden layer
%
                        S = H * X(:,k);
%
%       Caliculation of the output of the hidden layer
%
                        for i = 1:2
                                y(i) = 1 / ( 1 + exp( -S(i)*C ) );
                        end
                        Y =     [       y(1)    y(2)    1       ];
%
%       Caliculation of the sum of the output layer
%
                        t =  O  * Y';
%
%        Caliculation of the output

                        z = 1 / ( 1 + exp( -t*C ) );
%
%
%        Estimation
%
%        For Output Layer
%
                        Derr1  = ( D(k) - z ) * C * exp ( -t* C )/( 1 + exp( -t*C ) )^2;
%
%        Repacement for the output layer
%
                        O = O + r * Derr1 * Y;
%
%       For Hidden Layer
%
                                for i = 1:2
                                        Derr2(i) = Derr1 * C * O(i) * exp( -C * S(i) )/(
1 + exp( -S(i)*C ))^2 ;
                                end
%
%       Replacement for the Hidden layer
%
                        H = H + r * Derr2' * X(:,k)';
%
%

                        S = H * X(:,k);
%
                                for i = 1:2
                                        Y(i) = 1 / ( 1 + exp( -S(i)*C ) );
                                end
%
                        Y(3) = [        1       ];
%
% Caliculation of the sum of the output layer
%
                        T =   Y * O';
%
% Caliculation of the output
%
                        Z = 1 / ( 1 + exp( -T*C ) );
%
```

The caliculation
of the hidden
Layer neuron.

- Update the weights of the Ridden Layer
- The Caliculation of the output of output N.
- update the weights of the output neuron.

List. 1 The Matlab Program List

```
                        ERR1(k) = abs( D(k) - Z );
%
%
%
                end
        end
%
                if sum(pc) > 2000
                        break
                        end
%       Caliculation of the sum of the hidden layer
%
        S = H * X
%
% Caliculation of the output of the hidden layer
%
        for j = 1:ni
                for i = 1:2
                        Y(i,j) = 1 / ( 1 + exp( -S(i,j)*C ) );
                end
        end
        Y(3,:) =ones( Y(2,:));
%
% Caliculation of the sum of the output layer
%
        T =   Y' * O';
%
% Caliculation of the output
%
        ERR2 = 0
        for i = 1:ni
                        Z(i) = 1 / ( 1 + exp( -T(i)*C ) )
                        ERR1(i) = abs( D(i) - Z(i) )
                        ERR2 = ERR2 + ERR1(i)

        end
H
X
Z
end
%
%
H
X
O
Y
Z
```

The caliculation of the final weights & results.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                                                          %%
%%          The Neural Network For Surface Reconstruction       2           %%
%%             ( Retriveing Phase)                                          %%
%%          The MATLAB program              ver. 1.1                        %%
%%                                          04/22/91     by Hiro Oakada     %%
%%                                                                          %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%
nd = ns * ns
%
s = 0: 1/(ns-1): 1;
XS =s;
YS = s;
for i = 1:ns
        for j = 1:ns
                m = (i-1) * ns + j;
                EX(:,m) = [ XS(i); YS(j) ];
        end
end
if sum(size( H(1,:))) >3
        for i = 1:nd
                EX(3,i) = 1;
        end
end
%
%       Caliculation of the sum of the hidden layer
%
        SE = H * EX;
%
% Caliculation of the output of the hidden layer
%
        for j = 1:nd
                for i = 1:nhln
                        YE(i,j) = 1 / ( 1 + exp( -SE(i,j)*C ) );
                end
        end
        YE(nhln+1,:) = ones( YE(1,:) );
%
% Caliculation of the sum of the output layer
%
        G = size(O)
        if G(2) > G(1)
        O =O'
        end
%
        TE =   YE' * O;
%
% Caliculation of the output
%
        ERR2 = 0
        for i = 1:nd
                        W(i) = 1 / ( 1 + exp( -TE(i)*C ) );
%
        end
%
%
        V = zeros( ns, ns)
%
        for i = 1: ns
                V(:,i) = [ W((i-1)*ns+ 1: (i-1)*ns+ ns)']
        end
```

List 1.2. The Matlab Program to get the surface

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                                                           %%
%%          The Neural Network For Surface Reconstruction                    %%
%%                                                                           %%
%%          The MATLAB program              ver. 2.1                         %%
%%                                          04/21/91     by Hiro Oakada      %%
%%                                                                           %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  The definition of the intial matrices
%
%%
%
%  The Input Vector X ( x1', x2', x3')
%
        X(3,:) = ones( X(2,:))
%
%  The Weight of the Hidden Layer  H
%
for i = 1: 2: nhln;
%       H(i,1)    =  i / (nhln*3);
%       H(i+1,1) = -i/ (nhln*3);
%       H(i,2)    =  i /( nhln*3 + 1 );
%       H(i+1,2) = -i/(nhln*3 + 1 );
%       H(i,3)    =  i /( nhln*3 +2);
%       H(i+1,3) = -i/(nhln*3 +2) ;
%       O(i)     =  i / nhln*4;
%       OO(i+1) = -i/nhln*4;
%end
%       H(nhln+1,1) = [0];
%       H(:,2)    =  H(:,1);
%       H(:,3)    =  H(:,1);
```

uniformly distributed
weights generator

```
%
%
%
%  The Weight of the Output layer O
%
%       O =  HH(:,1) ;
%       O = zeros(H(:,1));
%       OO( nhln+1) = 0;
%       O = OO'
%
for i = 1: nhln
        H(i,1) = 0.5;
        H(i,2) = -0.5;
        H(i,3) = 0;
end
%
        rand('uniform')
        R1 = rand(nhln+1,1);
        R2 = rand(nhln+1,1);
        O   = R1-R2
%
%  The Desired Output Vector  D
%
%
%%%%%%%%%%%%%%%%%%
%
%  THE INITIAL INTERAL VALUES
%
        pc    =  zeros(1,ni);
        ERR1 =  ones(1,ni);
        Z     =  zeros(1,ni);
end
        r = 0.25;
        MAXERR1 = 0.10
        MAXERR2 = MAXERR1 * ni
%
%
%%%%%%%
%
ERR2 = 0
for i = 1:ni
                        ERR2 = ERR2 + abs( D(i) - Z(i) );
end
%
while   ERR2 > MAXERR2
%
        for     k = 1:ni
%
                while   ERR1(k) > MAXERR1
%
                        pc(k) = pc(k)  + 1;
%       Caliculation of the sum of the hidden layer
%
                        S = H  * X(:,k);
%
%       Caliculation of the output of the hidden layer
%
                        for i = 1:nhln
                                Y(i) = 1 / ( 1 + exp( -S(i)*C ) );
                        end
                        Y(nhln+1) =     [       1       ];
%
%       Caliculation of the sum of the output layer
%
                        t =  Y * O;

%        Caliculation of the output

                        z = 1 / ( 1 + exp( -t*C ) );
%
%
%        Estimation
%
%        For Output Layer
%
                        Derrl  = ( D(k) - z ) * C * exp ( -t* C )/( 1 + exp( -t*C ) )^2;
%
%        Repacement for the output layer
%
                        O = O + r * Derrl * Y';
%CC
%       For Hidden Layer
%
                                for i = 1:nhln
                                        Derr2(i) = Derrl * C * O(i) * exp( -C * S(i) )/(
1 + exp( -S(i)*C ))^2 ;
                                end
%
%       Replacement for the Hidden layer
%
                        H = H + r * Derr2' * X(:,k)';
%
%
                        S = H * X(:,k);
%
```

List. 2 The Matlab Program List

```
                                for i = 1:nhln
                                        Y(i) = 1 / ( 1 + exp( -S(i)*C ) );
                                end
%
                        Y(nhln+1) = [    1         ];
%
% Caliculation of the sum of the output layer
%
                        T =    Y * O;
%
% Caliculation of the output
%
                        Z = 1 / ( 1 + exp( -T*C ) );
%
                        ERR1(k) = abs( D(k) - Z );
%
%
%
                end
        end
%
                if sum(pc) > 3000
                break
                end
%
%       Caliculation of the sum of the hidden layer
%
        S = H * X
%
% Caliculation of the output of the hidden layer
%
        for j = 1:ni
                for i = 1:nhln
                        YY(i,j) = 1 / ( 1 + exp( -S(i,j)*C ) );
                end
        end
        YY(nhln+1,:) = ones( YY(1,:) );
%
% Caliculation of the sum of the output layer
%

        TT =    YY' * O;
%
% Caliculation of the output
%
        ERR2 = 0
        for i = 1:ni
                        ZZ(i) = 1 / ( 1 + exp( -TT(i)*C ) );
                        ERR1(i) = abs( D(i) - ZZ(i) );
                        ERR2 = ERR2 + ERR1(i);

        end
H
1
X
ZZ
pc
end
%
%
H
O
ZZ
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                                                            %%
%%          The Neural Network For Surface Reconstruction                     %%
%%                                                                            %%
%%          The MATLAB program             ver. 2.1                           %%
%%                                         04/21/91     by Hiro Oakada        %%
%%                                                                            %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  The definition of the intial matrices
%
%%
%
%  The Input Vector X ( x1', x2', x3')
%
        X(3,:) = ones( X(2,:))
%
%  The Weight of the Hidden Layer  H
%
for i = 1: 2: nhln;
%       H(i,1)    =  i / (nhln*3);
%       H(i+1,1) = -i/ (nhln*3);
%       H(i,2)    =  i /( nhln*3 + 1 );
%       H(i+1,2) = -i/(nhln*3 + 1 );
%       H(i,3)    =  i /( nhln*3 +2);
%       H(i+1,3) = -i/(nhln*3 +2) ;
%       O(i)     =  i / nhln*4;
%       OO(i+1) = -i/nhln*4;
%end
%       H(nhln+1,1) = [0];
%       H(:,2)    =  H(:,1);
%       H(:,3)    =  H(:,1);
%
%
%
%  The Weight of the Output layer O
%
%       O =  HH(:,1) ;
%       O = zeros(H(:,1));
%       OO( nhln+1) = 0;
%       O = OO'
%
for i = 1: nhln
        H(i,1) = 0.5;
        H(i,2) = -0.5;
        H(i,3) = 0;
end
O = H(1,:)'
O(4) = 0
%  The Desired Output Vector  D
%
%
%%%%%%%%%%%%%%%%%%
%
%  THE INITIAL INTERAL VALUES
%
        pc    =  zeros(1,ni);
        ERR1 =  ones(1,ni);
        Z     =  zeros(1,ni);
end
        r = 0.25;
        MAXERR1 = 0.10
        MAXERR2 = MAXERR1 * ni
```

```
%
%
%%%%%%%
%
ERR2 = 0
for i = 1:ni
                        ERR2 = ERR2 + abs( D(i) - Z(i) );
end
%
while   ERR2 > MAXERR2
%
        for     k = 1:ni
%
                while   ERR1(k) > MAXERR1
%
                        pc(k) = pc(k)  + 1;
%       Caliculation of the sum of the hidden layer
%
                        S = H  * X(:,k);
%
%       Caliculation of the output of the hidden layer
%
                        for i = 1:nhln
                                Y(i) = 1 / ( 1 + exp( -S(i)*C ) );
                        end
                        Y(nhln+1) =     [       1       ];
%
%       Caliculation of the sum of the output layer
%
                        t =  Y * O;

%        Caliculation of the output

                        z = 1 / ( 1 + exp( -t*C ) );
%
%
%        Estimation
%
%        For Output Layer
%
                        Derr1  = ( D(k) - z ) * C * exp ( -t* C )/( 1 + exp( -t*C ) )^2;
%
%        Repacement for the output layer
%
                        O = O + r * Derr1 * Y';
%CC
%       For Hidden Layer
%
                                for i = 1:nhln
                                        Derr2(i) = Derr1 * C * O(i) * exp( -C * S(i) )/(
1 + exp( -S(i)*C ))^2 ;
                                end
%
%       Replacement for the Hidden layer
%
                        H = H + r * Derr2' * X(:,k)';
%
%
                        S = H * X(:,k);
%
                                for i = 1:nhln
                                        Y(i) = 1 / ( 1 + exp( -S(i)*C ) );
                                end
%
```

```
                        Y(nhln+1) = [   1        ];
%
% Caliculation of the sum of the output layer
%
                        T =   Y * O;
%
% Caliculation of the output
%
                        Z = 1 / ( 1 + exp( -T*C ) );
%
                        ERR1(k) = abs( D(k) - Z );
%
%
%
                end
        end
%
                if sum(pc) > 3000
                break
                end
%
%       Caliculation of the sum of the hidden layer
%
        S = H * X
%
% Caliculation of the output of the hidden layer
%
        for j = 1:ni
                for i = 1:nhln
                        YY(i,j) = 1 / ( 1 + exp( -S(i,j)*C ) );
                end
        end
        YY(nhln+1,:) = ones( YY(1,:) );
%
% Caliculation of the sum of the output layer
%

        TT =   YY' * O;
%
% Caliculation of the output
%
        ERR2 = 0
        for i = 1:ni
                        ZZ(i) = 1 / ( 1 + exp( -TT(i)*C ) );
                        ERR1(i) = abs( D(i) - ZZ(i) );
                        ERR2 = ERR2 + ERR1(i);

        end
H
1
X
ZZ
pc
end
%
%
H
O
ZZ
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                                               %%
%%          The Neural Network For Surface Reconstruction        %%
%%                                                               %%
%%          The MATLAB program              ver. 3.1             %%
%%                                          04/21/91    by Hiro Oakada   %%
%%                                                               %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  The definition of the intial matrices
%

nhln =3
%%
%
%  The Input Vector X ( x1', x2', x3')
%
        X(3,:) = ones( X(2,:) )
%
%  The  Initial Weight of the Hidden Layer  H
%
%       rand('uniform')
%       R1  = rand(nhln,2);
%       R2  = rand(nhln,2);
%       H   = R1 -R2
for i =1: nhln
        H(i,1) = 0.5;
        H(i,2) = 0.5;
end
        H(:,3) = zeros( H(:,1) )
%
%  The Initial Weight of the Output layer  O
%
        R3  = rand(1, nhln +1);
        R4  = rand(1, nhln +1);
        O   = R3 -R4
```

∘ Random Weights generator

```
%
%%
%
%  THE INITIAL INTERNAL VALUES
%
for     i = 1:ni
        pc(i)   =       [       0       ];
        Z(i)    =       [       0       ];
end
        r = 0.25;
%
%
%%%%%%%
%
%
while   sum(pc) < 100000
%
%
        R5 = rand * 10000;
        R6 = fix( R5 );
        R7 = fix(R6/ni);
        k  = R6  - R7*ni +1;
```

∘ Pickup a pattern randomly.

```
%
                        pc(k) = pc(k)  + 1;
%       Caliculation of the sum of the hidden layer
%
                        S = H  * X(:,k);

%
%       Caliculation of the output of the hidden layer
%
                        for i = 1:nhln
                                Y(i) = 1 / ( 1 + exp( -S(i)*C ) );
                        end
                        Y(nhln+1) =     [       1       ];
%
%       Caliculation of the sum of the output layer
%
                        t =  Y * O';

%       Caliculation of the output

                        z = 1 / ( 1 + exp( -t*C ) );

%
%
%        Estimation
%
%        For Output Layer
%
                        Derr1  = ( D(k) - z ) * C * exp ( -t* C )/( 1 + exp( -t*C ) )^2;
%
%        Repacement for the output layer
%
                        O = O + r * Derr1 * Y;
%CC
%       For Hidden Layer
%
                                for i = 1:nhln
                                        Derr2(i) = Derr1 * C * O(i) * exp( -C * S(i) )/(
1 + exp( -S(i)*C ))^2 ;
                                end
%
%       Replacement for the Hidden layer
%
                        H = H + r * Derr2' * X(:,k)';
%
%
                        S = H * X(:,k);
%
                                for i = 1:nhln
                                        Y(i) = 1 / ( 1 + exp( -S(i)*C ) );
                                end
%
                        Y(nhln+1) = [   1       ];
%
% Caliculation of the sum of the output layer
%
                        T =   Y * O';
%
% Caliculation of the output
%
                        Z = 1 / ( 1 + exp( -T*C ) );
%
                        ERR1(k) = abs( D(k) - Z );
%
%
%
end
%
%       Caliculation of the sum of the hidden layer
%
        S = H * X
%
```

```
% Caliculation of the output of the hidden layer
%
        for j = 1:ni
                for i = 1:nhln
                        YY(i,j) = 1 / ( 1 + exp( -S(i,j)*C ) );
                end
        end
        YY(nhln+1,:) = ones( YY(1,:) );
%
% Caliculation of the sum of the output layer
%

        TT =   YY' * O';
%
% Caliculation of the output
%
        ERR2 = 0
        for i = 1:ni
                        ZZ(i) = 1 / ( 1 + exp( -TT(i)*C ) );
                        ERR1(i) = abs( D(i) - ZZ(i) );
                        ERR2 = ERR2 + ERR1(i);

        end
H
1
X
ZZ
pc
end
%
%
H
O
ZZ
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                                                            %%
%%          The Neural Network For Surface Reconstruction                     %%
%%                                                                            %%
%%          The MATLAB program             ver. 3.1                           %%
%%                                         04/21/91     by Hiro Oakada        %%
%%                                                                            %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  The definition of the intial matrices
%
%%
%
%  The Input Vector X ( x1', x2', x3')
%
        X(3,:) = ones( X(2,:) )
%
%  The  Initial Weight of the Hidden Layer  H
%
        rand('uniform')
        R1  = rand(nhln,2);
        R2  = rand(nhln,2);
        H   = R1 -R2
        H(:,3) = zeros( H(:,1) )
%
%  The Initial Weight of the Output layer  O
%
        R3  = rand(1, nhln +1);
        R4  = rand(1, nhln +1);
        O   = R3 -R4
%
%%
%
%  THE INITIAL INTERNAL VALUES
%
for     i = 1:ni
        pc(i)    =      [       0       ];
        Z(i)     =      [       0       ];
end
        r = 0.25;
%
%
%%%%%%%
%
%
while   sum(pc) < 1000
%
%
        R5 = rand * 10000;
        R6 = fix( R5 );
        R7 = fix(R6/ni);
        k  = R6  - R7*ni +1;
%
                        pc(k) = pc(k)  + 1;
%       Caliculation of the sum of the hidden layer
%
                        S = H  * X(:,k);
%
%       Caliculation of the output of the hidden layer
%
                        for i = 1:nhln
                                Y(i) = 1 / ( 1 + exp( -S(i)*C ) );
                        end
                        Y(nhln+1) =     [       1       ];
%
%       Caliculation of the sum of the output layer
%
                        t =  Y * O';

%        Caliculation of the output

                        z = 1 / ( 1 + exp( -t*C ) );
%
%
%        Estimation
%
%        For Output Layer
%
                        Derr1  = ( D(k) - z ) * C * exp ( -t* C )/( 1 + exp( -t*C ) )^2;
%
%        Repacement for the output layer
%
                        O = O + r * Derr1 * Y;
%CC
%       For Hidden Layer
%
                                for i = 1:nhln
                                        Derr2(i) = Derr1 * C * O(i) * exp( -C * S(i) )/(
1 + exp( -S(i)*C ))^2 ;
                                end
%
%       Replacement for the Hidden layer
%
                        H = H + r * Derr2' * X(:,k)';
%
%
                        S = H * X(:,k);
%
                                for i = 1:nhln
                                        Y(i) = 1 / ( 1 + exp( -S(i)*C ) );
                                end
%
                        Y(nhln+1) = [   1       ];
%
% Caliculation of the sum of the output layer
%
                        T =   Y * O';
%
% Caliculation of the output
%
                        Z = 1 / ( 1 + exp( -T*C ) );
%
                        ERR1(k) = abs( D(k) - Z );
%
%
%
end
%
%       Caliculation of the sum of the hidden layer
%
        S = H * X
%
% Caliculation of the output of the hidden layer
%
        for j = 1:ni
                for i = 1:nhln
                        YY(i,j) = 1 / ( 1 + exp( -S(i,j)*C ) );
                end
        end
        YY(nhln+1,:) = ones( YY(1,:) );
%
% Caliculation of the sum of the output layer
%

        TT =   YY' * O';
%
% Caliculation of the output
%
        ERR2 = 0
        for i = 1:ni
                        ZZ(i) = 1 / ( 1 + exp( -TT(i)*C ) );
                        ERR1(i) = abs( D(i) - ZZ(i) );
                        ERR2 = ERR2 + ERR1(i);

        end
H
1
X
ZZ
pc
end
%
%
H
O
ZZ
```

A Neural-based Digital Communication Receiver for Inter-symbol Interference (ISI) and White Gaussian Noise Channels

Sa Hyun BANG

ABSTRACT EE599/4 (Spr. 1991) Term Project

SS# : 888-00-1634

A neural-based network is applied to the data communication receiver to investigate the feasibility of the network over the inter-symbol interference (ISI) and additive white noise channel environments. With a 3-layered perceptron and backward error propagation training algorithm, it can be shown that it closely approximates the theoretical optimum receiver as the number of network trainings increases. Another interesting result from the simulations is the network operation against the white Gaussian noise, as opposed to the conventional schemes. Once the problem on the network training is solved, the proposed data receiver is a good alternative to the optimum Viterbi channel decoder.

1. INTRODUCTION

Communication systems can be modeled as one shown in Fig.1. The bandlimited and/ or multipath fading channel often introduces an inter-symbol interference(ISI) signal which severely degrades the system performance. This is especially true for narrowband time-division multiple access (TDMA) mobile communications such as the U.S. digital cellular telephone systems. Common technique for removing or reducing the ISI in the receiving end is to use either a channel equalizer or Viterbi maximum-likelihood sequence estimator(MLSE) [1] as shown in Figs.2 and 3. However, the performance of the channel equalizer is not so high as that of the MLSE, i.e., is sub-optimum. This is obvious when the communication channel has non-minimum phase response. The drawback of the Viterbi MLSE is its complexity of the receiver. Furthermore, for its optimum performance it requires an exact knowledge on the channel characteristics which is very difficult to achieve in practice. This is also true for the channel equalizer because the initial tab weights are set by the channel impulse response.

A neural-based receiver is exploited to show the feasibility of the network when applied to such a communication system with the ISI channel. The network used is a multi-layered perceptron with the back-propagation training algorithm. A 3-layer perceptron is selected in this study and it can generate arbitrarily complex decision regions. One obvious advantage of this scheme lies on the fact that the channel estimator is not required because it is embedded in the training operation. In order to facilitate the training algorithm, the transfer function of neurons must be differentiable and preferably non-linear, as opposed to the hard-limiting in the original perceptron network. Since the binary representation of +1/-1 is used instead of 0/1, the sigmoid function is modified to give

$$f(x) = \frac{1 - e^{-x}}{1 + e^{-x}} \qquad (1)$$

Fig.4 shows the plots of $f(x)$ and its derivative $f'(x)$ with respect to x. The network is trained by the stochastic gradient descent algorithm in which the weight w_{ji} and threshold θ_j of the nodes are updated in such a manner as to decrease the sum of squares of the error $e_j = d_j - o_j$

$$E = \frac{1}{2}\sum_j (d_j - o_j)^2 \qquad (2)$$

where d_j and o_j are the desired and actual outputs, respectively.

A simple algebric manipulation gives[2]

$$\delta_j = f'(x_j)\sum_k \delta_k w_{kj}$$

$$w_{ji} = w_{ji} + \eta\, \delta_j s_i \qquad (3)$$

where w_{ji} is the weight between the neuron j and i in the present and preceeding layer, respectively. Also δ_j is the error term in the present layer and δ_k in the next layer, x_j is the weighted sum input to the neuron j, and s_i is the output from the neuron i in the preceeding layer. In this simple algorithm, the iterative method of calculating δ_j [3] and additional momentum term for fast convergence are ruled out.

2. SYSTEM MODEL

Fig.5 shows the system model analyzed in this study. The multipath fading and/or finite bandwidth effects of the channel can be modeled as a finite impulse response (FIR) filter. The white Gaussian noise is added to the signal independently. Given a transmitted signal d(k), the output of the channel can be written as

$$\begin{aligned} y(k) &= h(k)*d(k)+n(k) \\ &= a_0 d(k)+a_1 d(k-1)+ \ldots + a_m d(k-m)+n(k) \\ &=\sum_i a_i d(k-i)+n(k) \qquad (4) \end{aligned}$$

where h(k) is the impulse response of the channel with its z-transform H(z), * represents the convolutional operation, n(k) is the white noise, and a_i's are the tab weights of the FIR filter. The integer m here is the number of symbol intervals for which the ISI takes effect. The white Gaussian noise samples n_i's are assumed to be uncorrelated each other,

$$E\{\mathbf{n}_i\mathbf{n}_j\} = \begin{cases} \sigma^2 & i=j, \\ 0 & \text{otherwise} \end{cases} \qquad (5)$$

where $\mathbf{n}_i$ is a random variable with its value n(i) at time i, E{ } is the expectation operator, and σ^2 is the variance of $\mathbf{n}$. The inputs to the receiver network are just the delayed replica of the received signal y(k),

$$x_{i+1} = y(k-i), \qquad i=0, \ldots , m. \qquad (6)$$

The output of the network is hard-limited to give binary logic levels(+1/-1). General network diagram is depicted in Fig.6 where there is only one output neuron and the number of neurons in the hidden layers is a simulation parameter.

3. SIMULATION

Two receiver models are simulated for various channel impairments, i.e., the ISI and white Gaussian noise. The number of inputs(2 in this case) is simply chosen to see the results in the two-dimensional plane. The extension of the alalysis dimension more than two is straightforward, but it is hard to recognize the graphical results. The network structures for two models are 2-9-5-1 and 2-6-4-1 where each entry is the number of neurons in the layer, except the first being just the number of inputs. Also two channel models

$H_1(z) = 1.0 + 0.5\, z^{-1}$ and

$H_2(z) = 0.5 + z^{-1}$

are used in the simulation. The first represents a typical minimum phase response ISI channel for which any receiver having linearly separable decision regions may perform well. The latter, on the other hand, exhibits the non-minimum phase response, i.e., poles or zeros of H(z) in the outside of the unit circle of the complex z-plane. The flowchart and MATLAB program of the simulation are given in Figs.7.a & 7.b. The initial weights and thresholds are chosen randomly in the range [-0.5, +0.5]. Also the training is done by a sequence of random data, instead of by a collection of known sample sequences. This simple training sequence will reduce the amount of time in the simulation, but may make the network less 'intelligent'.

4. SIMULATION RESULTS

Figs.8.a - 8.c are the equipotential contour plots of the network output for the channel $H_1(z)=1.0+0.5z^{-1}$. At low noise level (Fig.8.a, $\sigma^2=0.01$ or equivalently signal-to-noise ratio SNR=20 dB), the decision boundary is almost a straight line. This is the expected result in the sense that even though the network is trained to the ISI, no enough information is provided to train it for the white noise signal. This decision boundary, however, does not cause any problems because the receiver performance is less vulnerable to the weak noise interferences. The decision regions in Figs.8.b & 8.c in which the noise level is now increased to $\sigma^2=0.25$ (SNR= 6 dB), are close to the theoretical optimum boundary[4]. The numbers of training data symbols in Figs.8.a & 8.b and Fig.8.c are 5,000 and 10,000, respectively. From the figures, the results thus justify the effect of these numbers on the degree of training.

The proper network operation is clear in Figs.9.a - 9.d where the 2nd channel model $H_2(z)=0.5+z^{-1}$ is used. Since the interfering signal is stronger than the original signal, the decision making becomes very complicated and the conventional receiver such as a equalizer may not detect the data correctly in this case. For small number of training data symbols (1,000 in Fig.9.d), it can be seen that the network has not been trained properly. However, as the number of training data increases (5,000 in Figs.9.a - 9.c), the receiver is well-behaved against the ISI as well as white Gaussian noise. The additional performance improvement for the additive white noise over the conventional techniques is based on the fact that the decision boundaries are different for different noise levels ($\sigma^2=0.25$, 0.01, and 0). As a matter of fact, the objective of the matched-filter in the conventional receivers is to reduce the effect of this noise. Since the optimum decision boundary for this non-mimimum phase response channel is very difficult to decide analytically, any qualitative comparisions with the optimum receiver on the performance can not be made at this moment. An extensive amount of efforts for error-rate simulations and statistical analyses may be required to do this.

5. CONCLUSION

In this simple analysis model and simulation, the perceptron-based neural network is shown to be quite feasible as a digital communication receiver. The hardware architecture for an actual implementation is also provided in order to exploit deeper insights into the applications of the neural-based network. Provided that enough time is devoted to network training, the proposed receiver has several advantages over the channel equalizer or Viterbi channel decoder (MLSE). The table given below summrizes the features of each scheme. Further studies may include an exact performance comparision with the optimum receiver through the simulations and the investigation of techniques to reduce the time for network training utilizing the code sequences with special property such as a preample sequence in the practical communication systems.

REFERENCES :

1. John G. Proakis, *"Digital Communications"*, McGraw Hill, 1983.

2. D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Internal Representations by Error Propagation", in D. E. Rumelhart and J. L. McClelland (eds), *"Parallel Distributed Processing: Exprorations in the Microstructure of Cognition"*, MIT Press, 1986.

3. R. Hecht-Nielsen, *"Neurocomputing"*, Addison Wesley, 1990.

4. G. J. Gibson, S. Siu, S. Chen, C. F. N. Cowan, and P. M. Grant, "The Application of Non-linear Architectures to Adaptive Channel Equalization", IEEE Proceedings of ICC, 1990.

Schemes Items	**Channel Equalizer**	**Viterbi MLSE**	**Neural Network**
Error-rate Performance	Sub-optimum	Optimum	Optimum*
Hardware Complexity	Moderate	High	High **
Channel Statistics (Channel Estimator)	Required	Required	Not Required
Pre-filtering (Matched-filter)	Required	Required	Not Required
Estimation Delay	No	Long	No
Adaptation Algorithm	Gradient or Kalman Algorithm	Viterbi Algorithm	Gradient Algorithm
Special-purpose Sequence	Pre-amble	Pre-amble	Training Sequence
Advantage	Simple H/W	Complex H/W	Noise Adaptation
Disadvantage	Sub-optimality	Optimum Error-rate	Long Training

* Based on this basic study
** Depends on the network structure

TABLE. Comparision of 3 Receiver Techniques

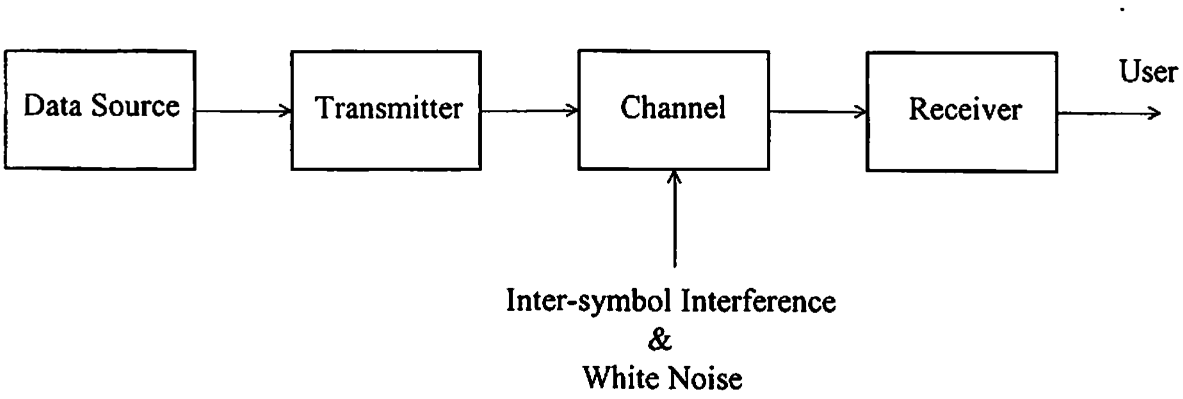


Fig.1 Data Communication System

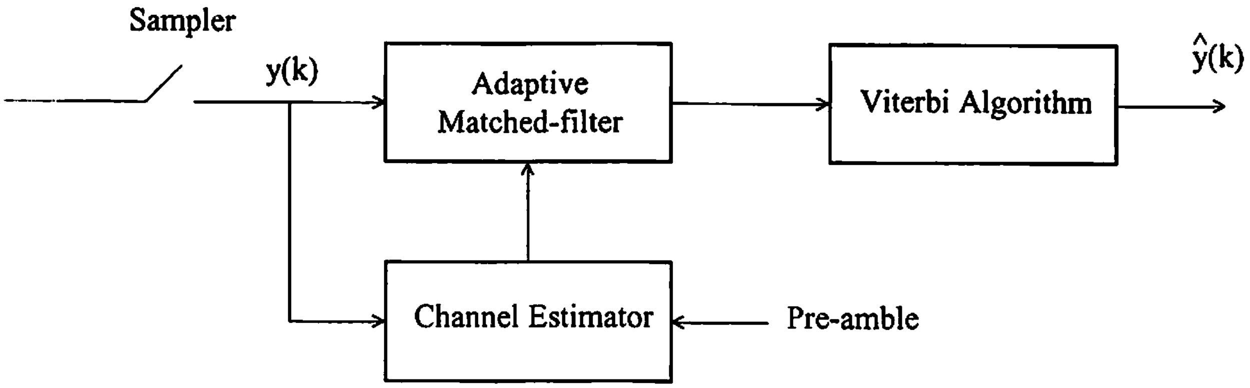


Fig.2 Viterbi Maximum-likelihood Sequence Estimator

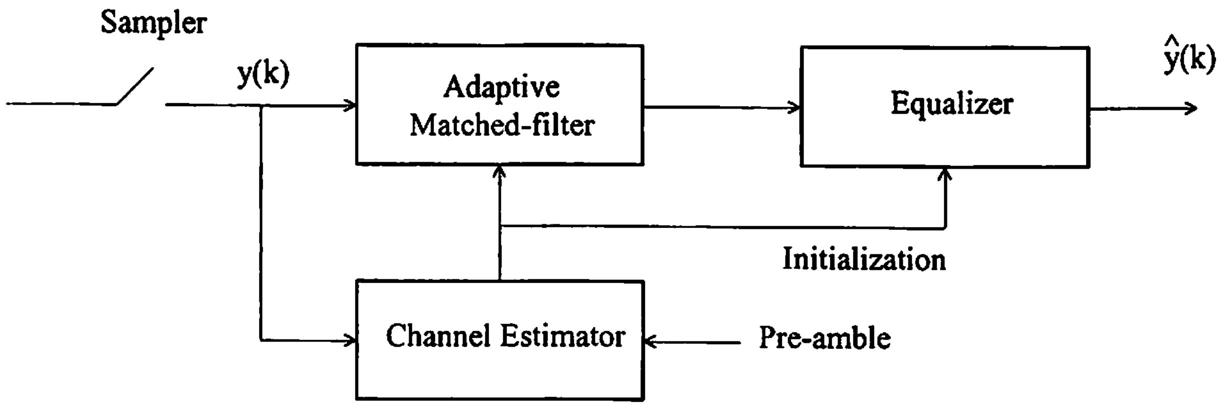
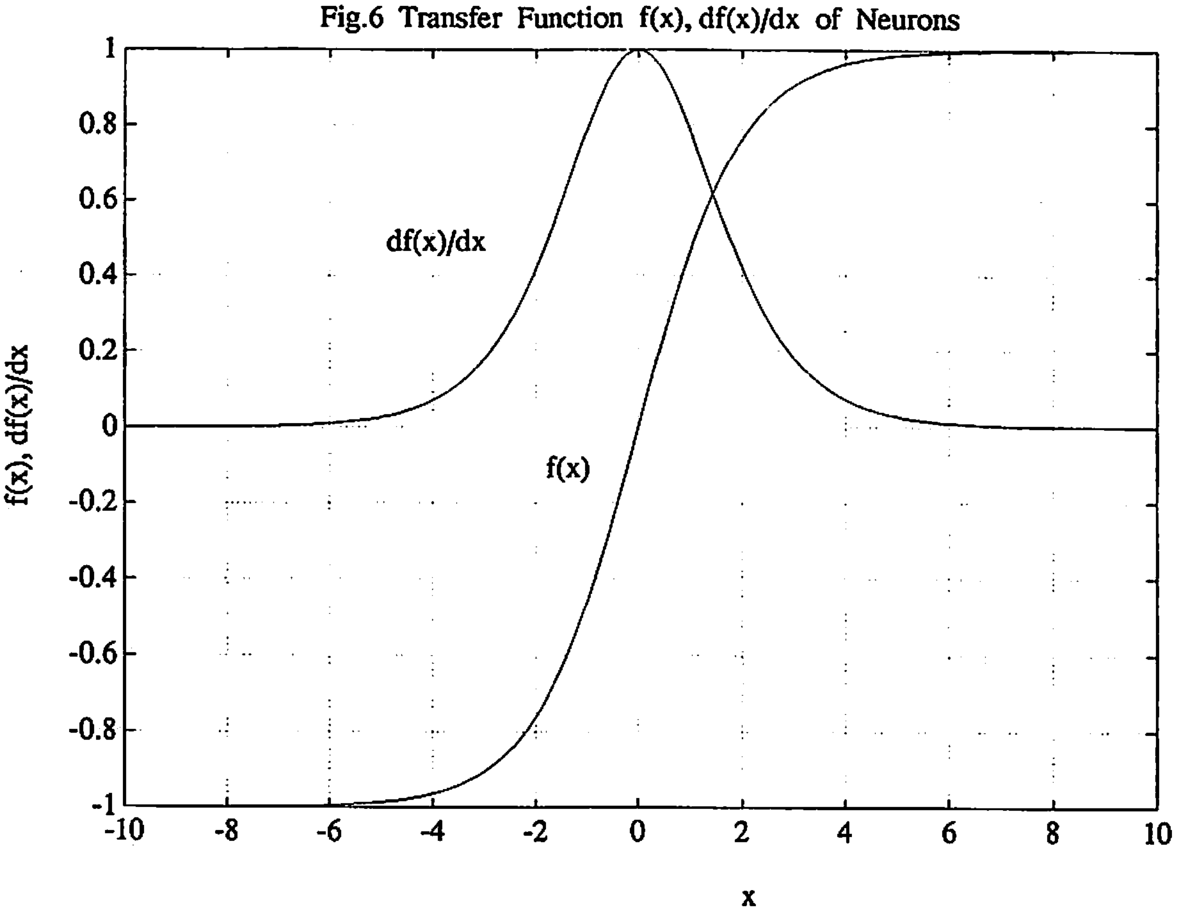
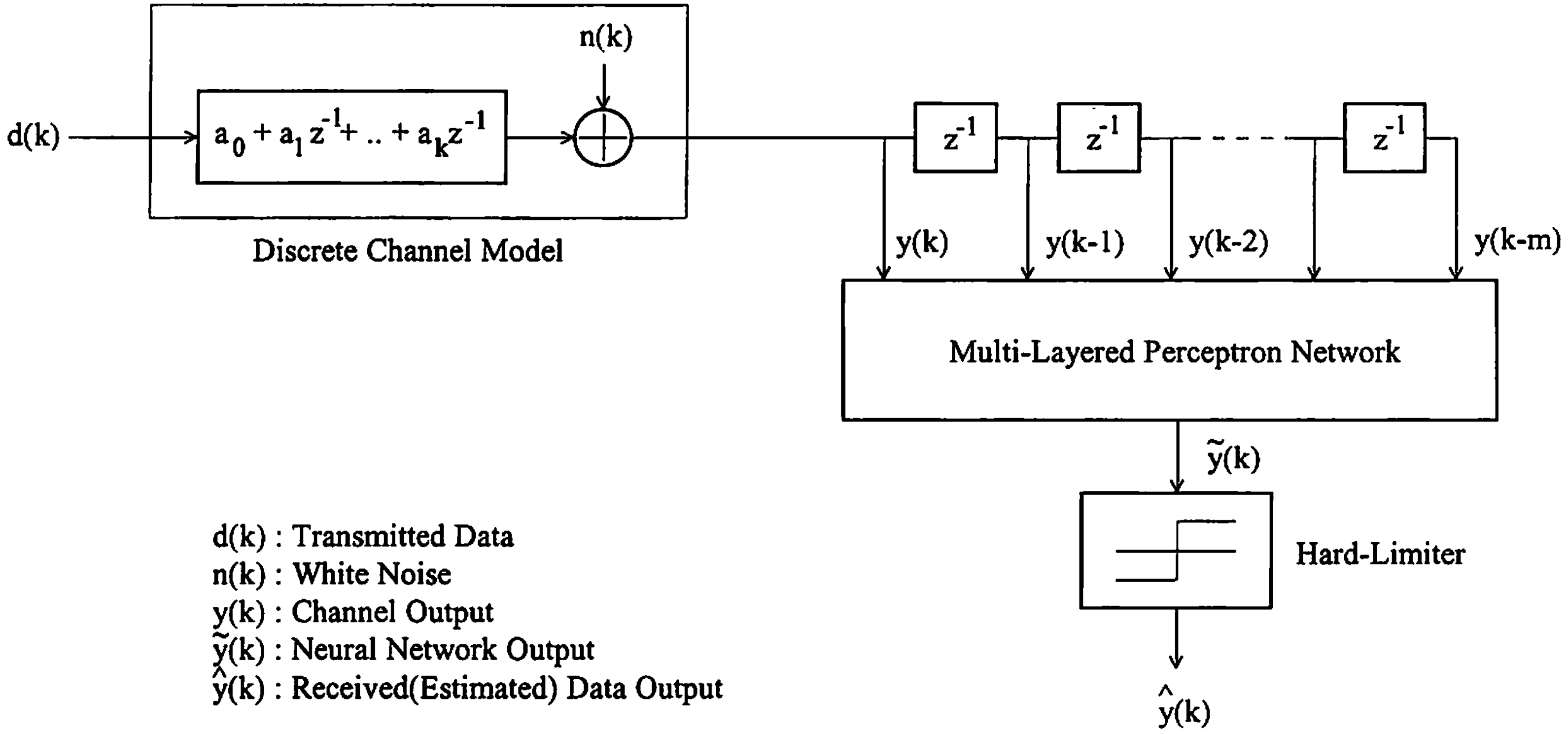


Fig.3 Channel Equalizer Data Receiver



* f'(x) scaled up by 2

Fig.4 Plots of f(x) and its derivative f'(x)



d(k) : Transmitted Data
n(k) : White Noise
y(k) : Channel Output
$\tilde{y}(k)$: Neural Network Output
$\hat{y}(k)$: Received(Estimated) Data Output

Fig.5 Neural-Based Data Receiver for Intersymbol Interference Channel

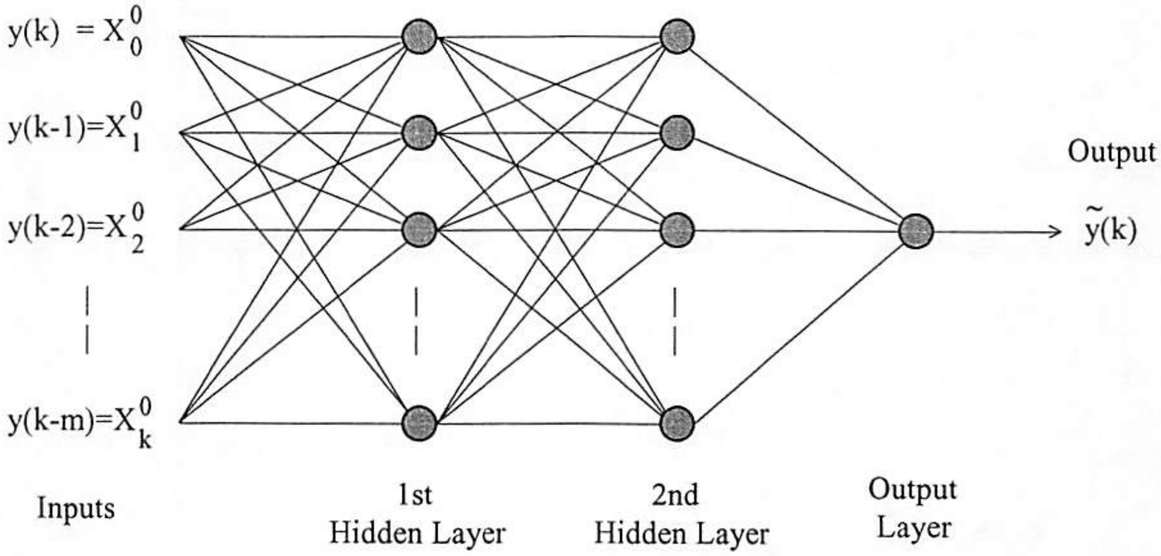


Fig.6.a 3-Layered Perceptron Network for Binary Data

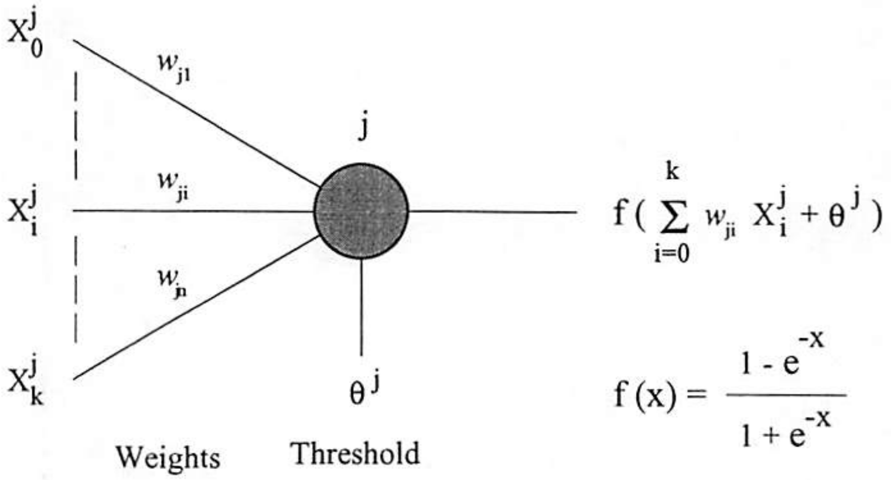

X_0^j
w_{j1}
j
X_i^j
w_{ji}
w_{jn}
X_k^j
θ^j
$f(\sum_{i=0}^{k} w_{ji} X_i^j + \theta^j)$
$f(x) = \frac{1 - e^{-x}}{1 + e^{-x}}$
Weights
Threshold

Fig.6.b Perceptron Node Structure and Transfer Function of Neuron

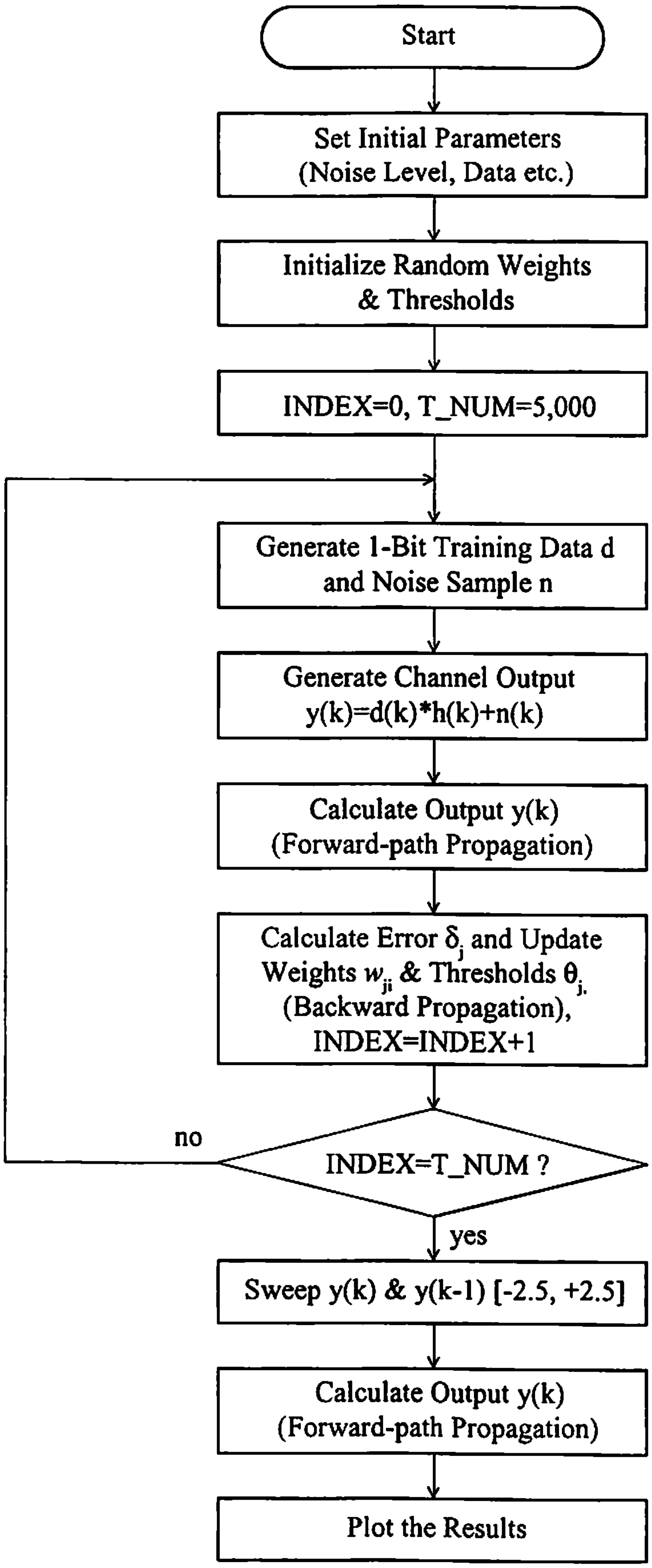

Start
Set Initial Parameters (Noise Level, Data etc.)
Initialize Random Weights & Thresholds
INDEX=0, T_NUM=5,000
Generate 1-Bit Training Data d and Noise Sample n
Generate Channel Output y(k)=d(k)*h(k)+n(k)
Calculate Output y(k) (Forward-path Propagation)
Calculate Error δ_j and Update Weights w_{ji} & Thresholds $\theta_{j.}$ (Backward Propagation), INDEX=INDEX+1
no
INDEX=T_NUM ?
yes
Sweep y(k) & y(k-1) [-2.5, +2.5]
Calculate Output y(k) (Forward-path Propagation)
Plot the Results

Fig.7.a Flowchart of Simulation Program

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%       MATLAB Program for the simulation of a Neural-based Digital
%       Communication Receivers
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
s=0.1;
a=0.5;
t=0;
d=-1;
y=-1.5;
%
%       Initialization
%
rand('uniform')
for i=1:10
        for j=1:3
        if i==10
        W1(i,j)=0;
        else
        W1(i,j)=rand-0.5;
        end
        end
end
for i=1:6
        for j=1:10
        if i==6
        W2(i,j)=0;
        else
        W2(i,j)=rand-0.5;
        end
        end
end
        for i=1:6
        W3(i)=rand-0.5;
        end
%
while t<10000
%
        rand('uniform')
        d1=d;
        d=sign(rand-0.5);
                if d==0
                        d=1;
                else
                end
%
%       Gaussian white noise generation
%
        rand('normal')
        n=s*rand;
        y1=y;
        y=d+d1/2+n;
%
%       Channel Output
%
        R(1)=y;
        R(2)=y1;
        R(3)=1;
%
%       Weighted sum of the 1st layer
%
        N1=W1*R';
        O1=(1-exp(-N1)) ./(1+exp(-N1));
        O1(10,1)=1;
%
%       Weighted sum of the 2nd layer
%
        N2=W2*O1;
        O2=(1-exp(-N2)) ./(1+exp(-N2));
        O2(6,1)=1;
%
%       Weighted sum of 3rd(output) layer
%
        N3=W3*O2;
        O3=(1-exp(-N3))/(1+exp(-N3));
%
%       Back-ward Propagation
%
        del3=(d-O3)*2*exp(-N3)/(1+exp(-N3))^2;
        W3=W3+a*del3*O2';
%
        DEL2=del3*W3' .*(2*exp(-N2) ./((1+exp(-N2)) .^2));
        W2=W2+a*DEL2*O1’;
        for i=1:7
                W2(6,i)=0;
        end
%
        DEL1=(W2’*DEL2) .*(2*exp(-N1) ./((1+exp(-N1)) .^2));
        DEL1(7,1)=0;
        W1=W1+a*DEL1*R;
        for i=1:3
                W1(10,i)=0;
        end
%
        t=t+1;
end
        W1
        W2
        W3
%
for i=1:51
for j=1:51
%
        y=0.1*(j-25.5);
        y1=0.1*(-i+25.5);
%
        R(1)=y;
        R(2)=y1;
        R(3)=1;
%
%       Weighted sum of the 1st layer
%
        N1=W1*R’;
        O1=(1-exp(-N1)) ./(1+exp(-N1));
        O1(10,1)=1;
%
%       Weighted sum of the 2nd layer
%
        N2=W2*O1;
        O2=(1-exp(-N2)) ./(1+exp(-N2));
        O2(6,1)=1;
%
%       Weighted sum of 3rd(output) layer
%
```

```
          N3=W3*O2;
          O3=(1-exp(-N3))/(1+exp(-N3));
%
          Y(i,j)=O3;
%
end
end
          contour(Y)
          grid
          xlabel('             y(k)')
          ylabel('            y(k-1)')
```

Fig.7.b MATLAB Program List

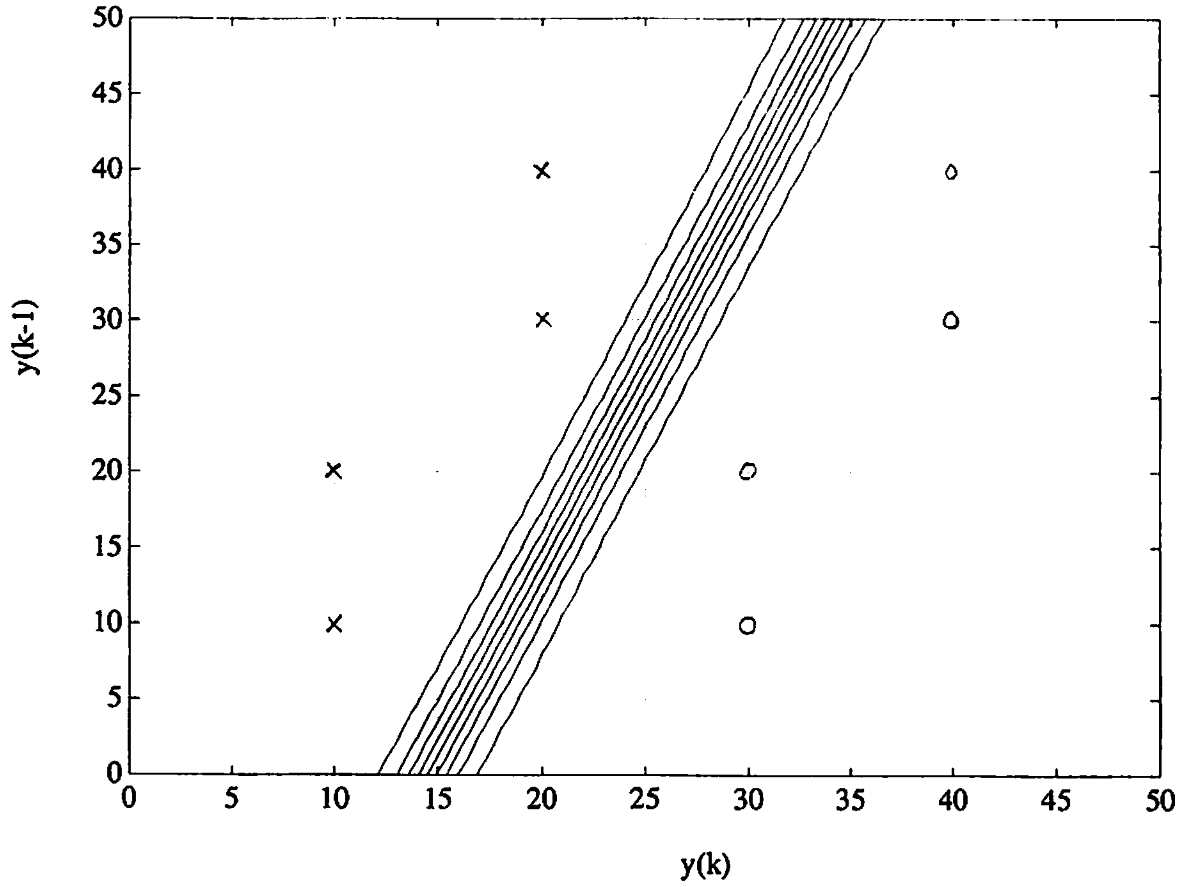


Fig.8.a Contour Plot of Output Equipotentials

($\delta^2=0.01$, $H(z)=1.0+0.5z^{-1}$, 5,000 Training Symbols)

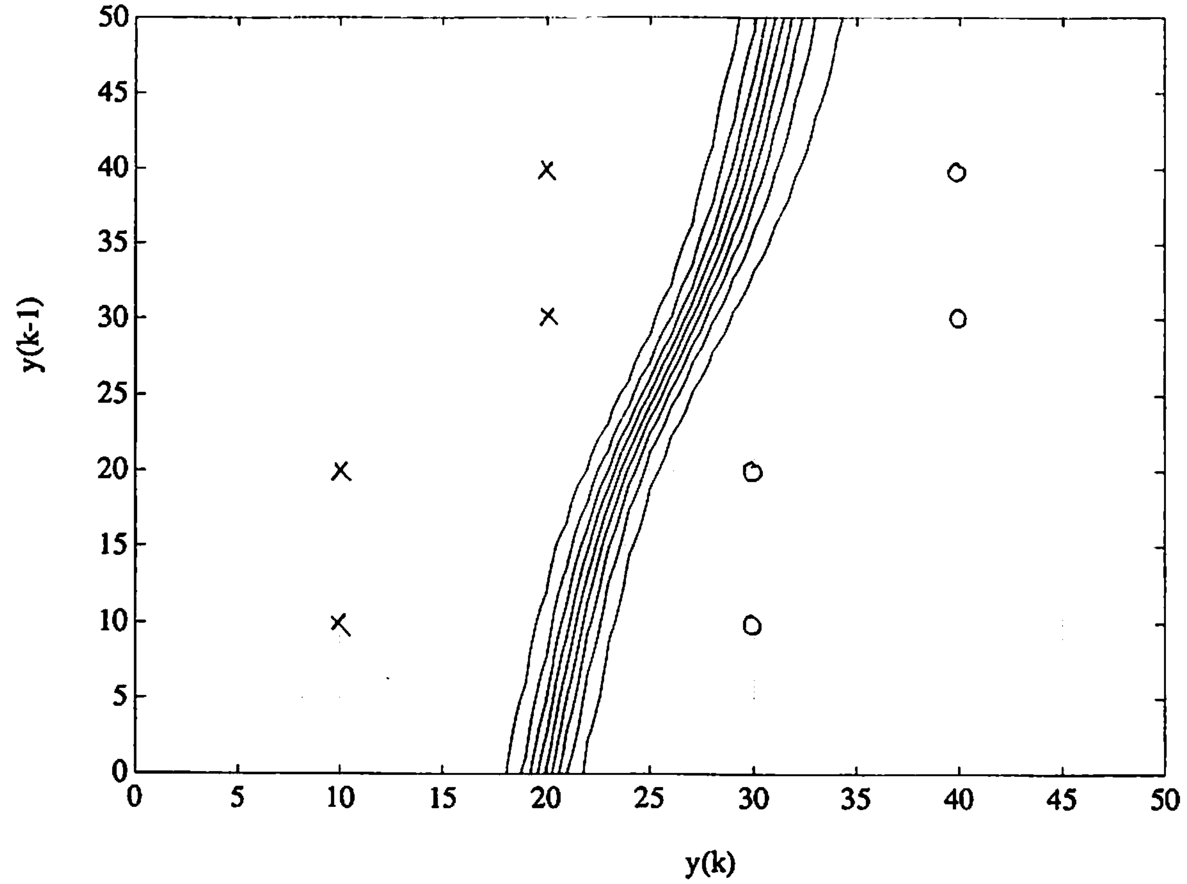


Fig.8.b Contour Plot of Output Equipotentials

($\delta^2=0.25$, $H(z)=1.0+0.5z^{-1}$, 5,000 Training Symbols)

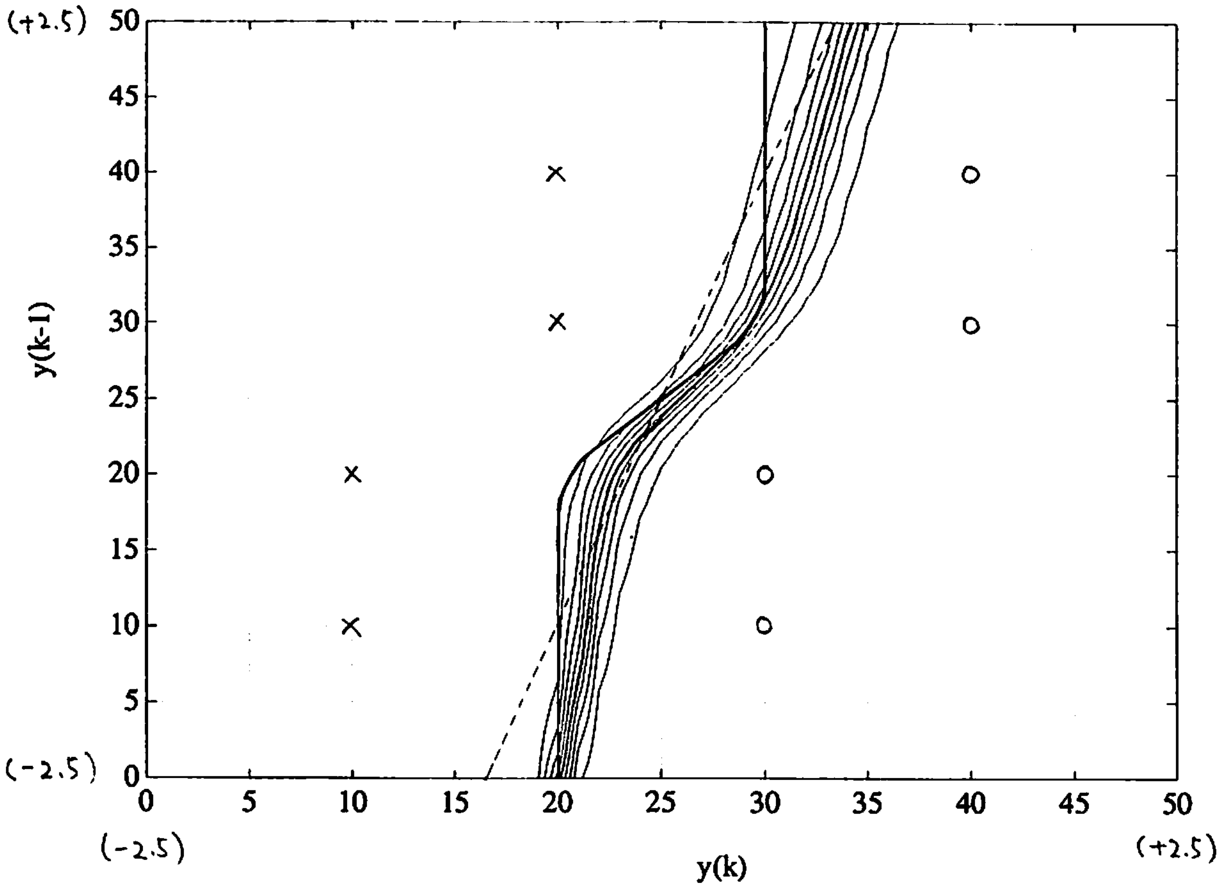


Fig.8.b Contour Plot of Output Equipotentials

($\delta^2 = 0.25$, $H(z) = 1.0 + 0.5z^{-1}$, 10,000 Training Symbols)

------ Equalizer decision boundary

——— Theoretical optimum decision boundary

——— Neural Net decision boundary

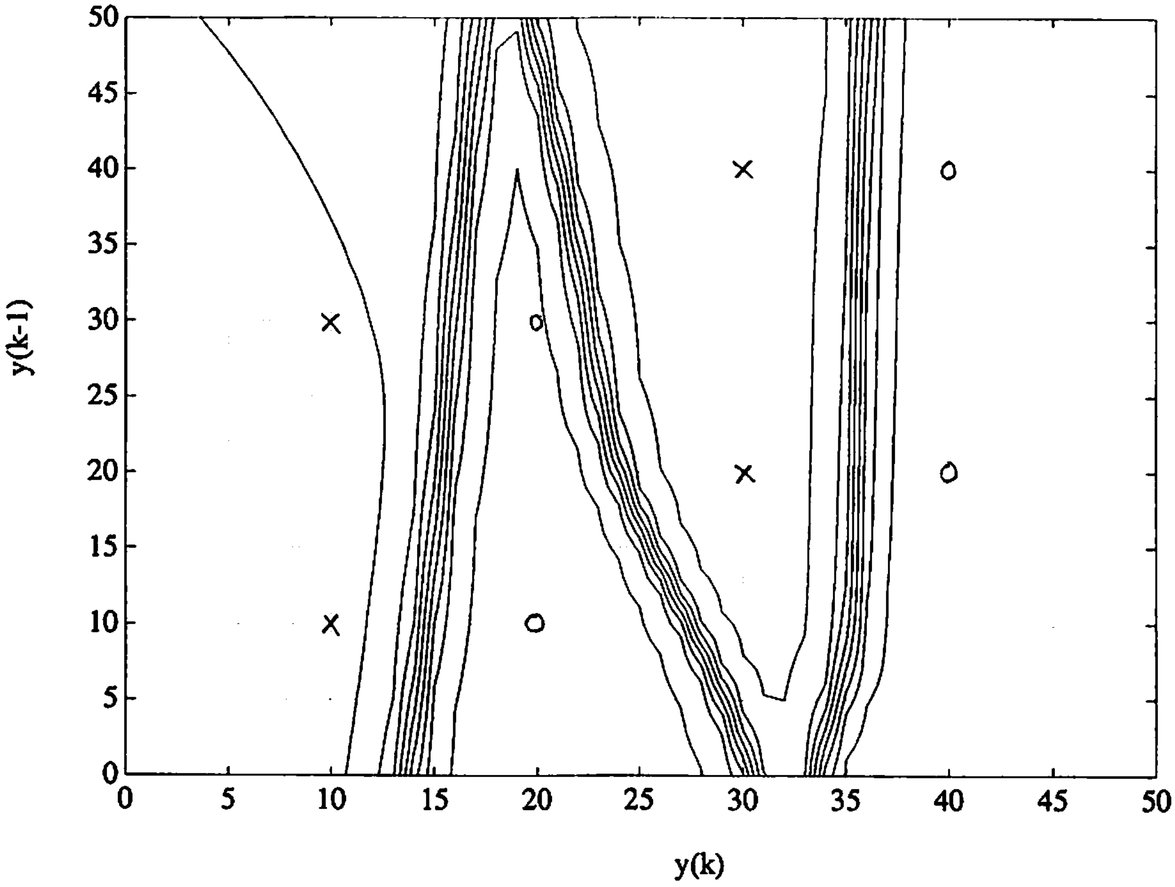


Fig.9.a Contour Plot of Output Equipotentials

($\delta^2=0.25$, $H(z)=0.5+1.0z^{-1}$, 5,000 Training Symbols)

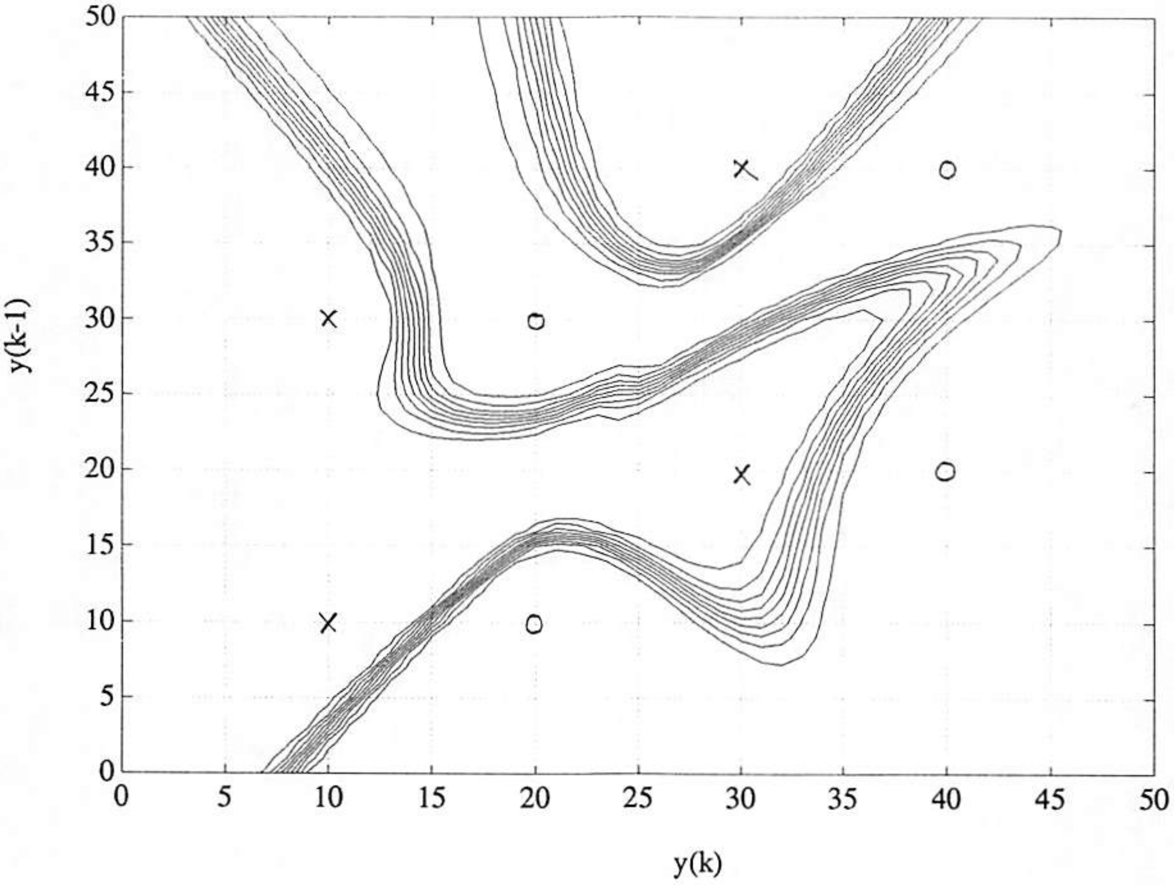


Fig.9.b Contour Plot of Output Equipotentials

($\delta^2 = 0.01$, $H(z) = 0.5 + 1.0z^{-1}$, 5,000 Training Symbols)

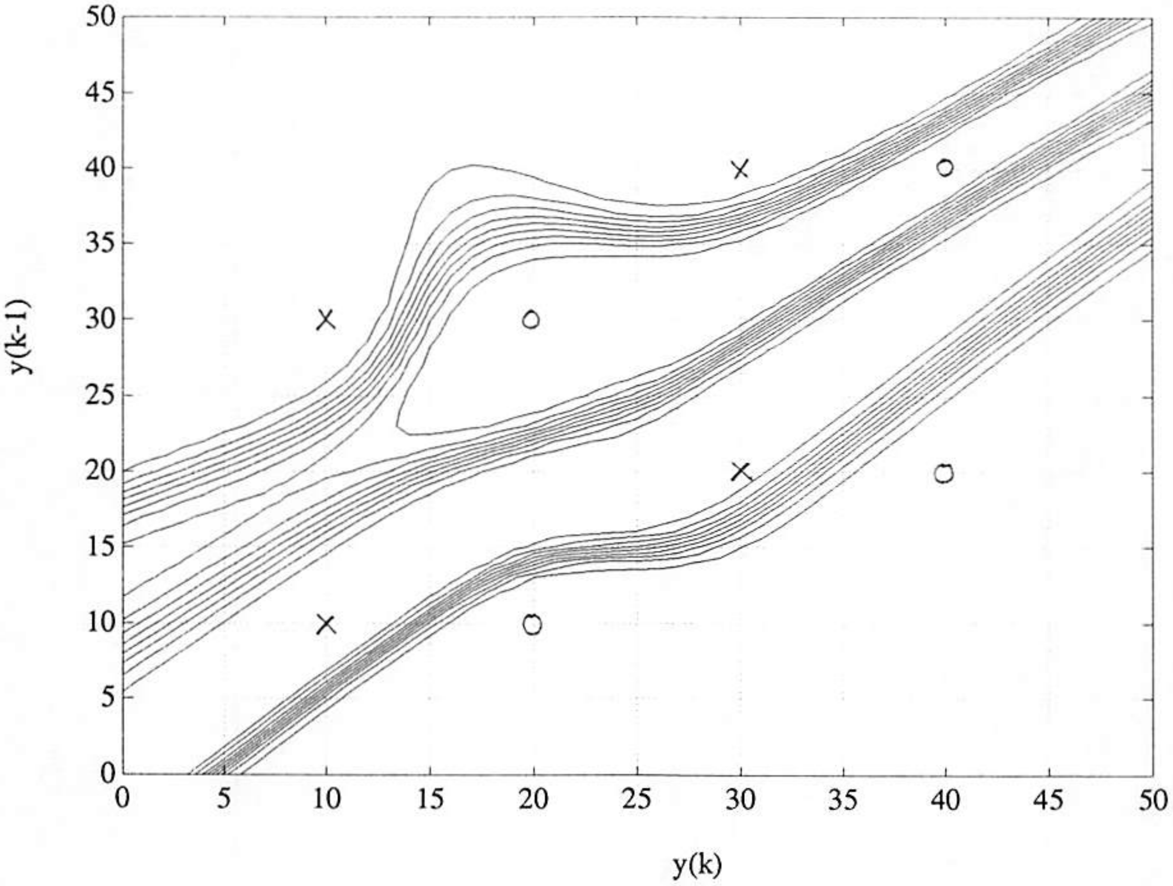


Fig.9.c Contour Plot of Output Equipotentials

($\delta^2 = 0$, $H(z) = 0.5 + 1.0z^{-1}$, 5,000 Training Symbols)

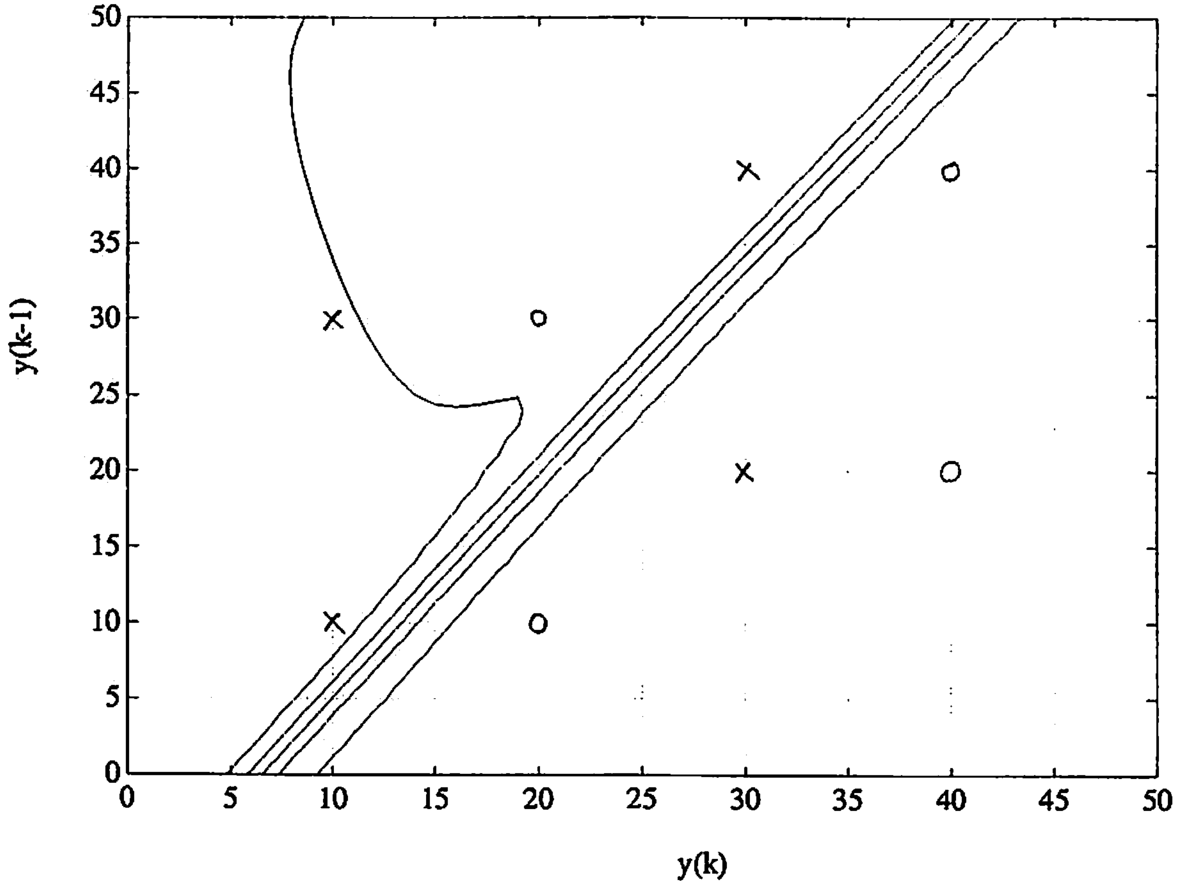


Fig.9.d Contour Plot of Output Equipotentials

(δ^2=0.01, H(z)=0.5+1.0z^{-1}, 1,000 Training Symbols)

EE 599/4

Neurocomputing VLSI

Project

For

Kotleong Shin
547-93-7398

Forecasting Model In The Volatile Stock Market Using Neural Network : A Software Approach

Abstract : *Neural Nework* (NN) has been proven to be very usefil in interpreting a collection of otherwise meaningless data. It is able to do so mainly due to its ability to undergo training with the data. For example, in the field of the volatile stock market, a collection of old prices or indices can be trained by a neural network to give an estimated forecast of the future prices or indices.

For this project, a three layer network is used in conjunction with the error backpropagation algorithm. The results obtained with such trained network is surprisingly accurate. In particular, a collection of old indices give an estimated prediction within 2.5% accuracy.

I) Introduction

Recent growth in the application of neural network technology has been explosive. For example, it has been used extensively in the field of pattern recognition.

In this project, the understudy problem has its root in pattern recognition. In particular, it requires the network to learn and recognize input/output patterns in the training data presented to the network. In addition, it will make an intelligent prediction of the future value based on the trained network. The problem understudy in this project is a prediction problem on stock market prices and indices. And it would be indispensably useful to all those who deal in the stock market.

The required network is a three layer backpropagation architecture. The algorithm used is the standard backpropagation technique suggested by Rumelhart and McClelland in 1986.

To prove its validity, a test model, which the network is used to forecast on an existing value, and an actual real life stock prices prediction are presented. In particular, the test model results have shown to deviate less than 2.5% from actual indices on the test model cases.

For remaining sections of the paper, a view into how this network is constructed, input and trained are presented. At the end, some performance and limitation issues are discussed.

II) Constructing the Network

In constructing the network, an experiment is conducted to optimize the number of input nodes that would give best overall results without incurring huge overhead on host resource. As shown in table II-1 and graph II-1, there is a trend of decreasing number of iterations as the number of input nodes are increased from 5 to about 20. After 20, data seems to show sign of instability. This may be contributed by the fact that local minima is being reached. Besides, at 20, this also trains other data as good as this set of data. To be practical, the number of input nodes is chosen to be 20. This would require about 400 iteration of trainings to reach a global error rate of 10e-6. For this number of iterations, it could very well be performed on an ordinary PC.

In contrast to other networks, this network uses 41 hidden nodes. This is due to direct consequences of the classical result derived by Kolmogorov [1]. This result maintains that any realistic functions can be able to be approximated by a three-layer neural network having a hidden layer with two times the number of input nodes as its hidden nodes.

The network itself is fully connected between the input/hidden and hidden/output layers. There is no connection between the input and the output layer.

The weights associated with these connections are set with some logic. The weights in the upper(hidden/output) layer are set to fall within [-0.5, +0.5] randomly. The reason for this is to be able to use a symmetric logistic activation function such as the sigmoid function. This symmetry has been proven to have a shorter learning times and easier to generalize for the network [2].

In addition, this selection would allow the network to operate within a more linear portion of the sigmoid nodal activation functions. Staying in the linear portion of the function has the benefit of having lower training error levels for a continuous-valued outputs [3]. For this same reason, the inputs as well as the outputs are also scaled to fall within [0, 0.5]. The weights in the lower layer are initially set to be zero.

The output layer just consists of a single node. The network is shown in figure II-1.

III) Training

Initially, a large number of iterations on training the network was thought to be essential. But a check on the error rate on the test model runs proved unnecessary. It was seen that error rates remain pretty much unchanged after about a few hundred runs on a large alpha. So a criteria is set for the network to stop training after it converges within 10e-6 error rate. This is supported by the fact that nothing greater than this accuracy seems necessary. As with any forecast problem, it itself is a problem of unpredictable and inaccurate.

In addition, the adaptation rate(alpha) was studied for its tradeoff. In table III-2, it was shown that for a particular sample from the test model, the network still remained unsaturated beyond alpha = 3. This is due to the fact that the network is properly scaled to fall nicely on the very linear portion of the activation function. But, to play safe, alpha is set to 0.99 for the rest of the runs.

IV) Selecting Data

In order to show the performance of this network on forecasting, 6 samples of old indices obtained from a report released by the Commodities Futures Trading Commission(CFTC) are run. The purpose here is to allow the network to train under 20 inputs of the weekly indices of the S&P500, T-Bond and Gold (2 samples each), and predict on existing indices to see how well this network is capable of. Discussions of the performance will be given at a later section.

In addition, 3 samples of the stock prices of IBM, MCI and Apple are taken into consideration. The prices on these are current valid prices and the predictions are performed on the April 25 prices.

V) Getting Output

The way to forecasting value is derived is as follow. After the network has shown to converge within 10e-6, the desired output, which is the most recently available prices or indices, is moved to become a value in the input. The rest of the input are shifted with the oldest value purged.

Then, a feedforward run is performed with this new set of inputs. The actual output collected at the output node is the predicted value.

As for the test model, this is contrasted with an actual value to show its effectiveness.

VI) Analysis of the Outputs

Runs in the test model data show some promising results. The predicted values are shown to fall less than 1% of error for most cases (5 out of 6). Whereas, one of them has a 2.5% error. This results indeed prove that neural network can be trained and evaluate an effective forecast. This has to be overwhelming.

In addition, the three runs on IBM, MCI and Apple stock prices are hold off to see if it can indeed perfom as well as the test model cases. Should it prove viable, technology such as neural network could really make many investors very happy and rich?

VII) Limitation

There are certain limitation on this neural model.

First, the 0.5 limit imposed on the input and output could limit how far the prices or indices can go up. In theory, the network can never be trained to give actual output that exceeds its input and output limits. Consequently, this network is perilous to sharp change in prices or indices. However, a simple fix to this would be to make a bias input. This bias input would also serve to limit how far the output can go by arbitrary setting it to a higher value. Sharp increases may be possible.

Secondly, this network may get stuck at a local minima [4]. This arises because local minima does exist and there are few proven and practical methods to pull it out of the local minima. This problem is further aggravated by the fact that little is known about them [4]. A fix suggested is maybe to add noises to the weight updates. But there is some evidence [5] that this use of momentum has less significance on the learning algorithm.

VIII) Conclusions

This project has shown an effective forecasting model based on the neural network. Its usefulness derives from the inherent properties of the error backpropagation algorithm. The point proved here is that such a highly specialized technology like neural network has such meaningful application.

However, a note of cautions should be addressed here. This model exists purely as an technical application of technology. It is not intended to be used as an end to a mean. In other words, it only serves as a tool to be used in technical analysis but not means as a speculative tool to be abused. To believe that it really works precisely is a fool. It can be easily argued that the data presented here may not be substantial enough to prove its true validity. Even if the data is substantial, stock market conditions are often as rough as the ocean. Consequently, the usefulness of this forecast model in technical analysis is a thing that only time can tell.

References :

[1] : R. Hecht-Nielsen, 'Theory of the backpropagation neural network.' In "Proceedings of the 1989 International Joint Conference on Neural Networks." IEEE Service Center, Piscataway, NJ, 1989.

[2] : S. Ahmad and G. Tesuro, 'Scaling and generalization in neural networks : a case study.' In "Proceedings of the 1988 Connectionist Models Summer School," T. J. Sejnowski (Eds.).

[3] : P. J. B. Hancock, 'Data representation in neural networks : an empirical study.' In "Proceedings of the 1988 Connectionist Models Summer School," T. J. Sejnowski (Eds.).

[4] : R. Hecht-Nielsen, 'Mapping Networks : Multi-Layer Data Transformation Stuctures.' In Chapter 5 of the book "Neurocomputing," published by Addison-Wesley.

[5] : S. Becker and Y.le Cun, Improving the Convergence of Back-Propagation Learning with Second Order Methods. In "Proceedings of the 1988 Connectionist Models Summer School," T. J. Sejnowski (Eds.).

Table II - 1 : The following table contains the experimental results performed on finding the number of suitable input nodes. The set 20 seems to indicate where the advantages (in terms of # iterations and % difference) levels off. After 20, it shows instable behavior which could be due to the network hitting some local minima. The project assumes an input nodes of 20. Note that the following data is obtained through the T-Bond 2nd set data. It assumes a learning rate (alpha) of .25. The number of iterations indicates the number of trainings needed to achieve 10e-6 accuracy. The % difference means the actual difference of the actual forecasted output and the real exeisting index. For data set information, refer to **Table IV - 3.**

Number of Inputs Nodes	Number of Iterations	% difference
5	> 9000	1.43
10	> 9000	.41
15	472	.76
20	429	.19
25	343	3.20
30	28	3.66
35	186	.072
40	55	2.72
42	94	.38

Table II - 2 : The following table illustrates the efforts to select a learning rate based on a 20 input nodes, 41 output nodes and 1 output node. The number of iterations is a number where the trained network converges to 10e-6 accuracy. The % difference is an indication of how far off the forecasted index differs from the real existing index. Again, the following results are obtained from the T-Bond's 2nd set of data (refer to **Table IV - 3**).

Alpha (α)	Number of iterations	% difference
.05	> 5000	.12
.1	572	.12
.15	381	.12
.2	286	.12
.25	226	.12
.35	161	.12
.45	124	.12
.55	101	.12
.75	72	.12
.95	56	.12
1.0	53	.12
1.5	33	.12
2.5	17	.12
3.0	13	.12
3.5	11	.12

Table IV-3: This table contains the indices of the three major indicators. This sets of data are obtained from the CTCC report. This data are presented to the network as testing model. Its function serves as a performance analysis of the network in forecasting future indices. The validity of these six sets of data can be shown on **Fig V - 2.**

Data Set # 1 For The Test Model

Date	S&P500	T-Bond	Gold
10/10/86	$ 235.10	$ 96.10	$ 434.50
10/17/86	237.95	94.56	422.20
10/24/86	237.75	95.90	409.20
10/31/86	247.70	98.09	405.50
11/07/86	245.90	96.31	410.60
11/14/86	245.30	98.15	398.80
11/21/86	244.85	99.65	379.30
11/28/86	248.65	99.71	391.40
12/05/86	251.95	99.71	389.50
12/12/86	247.00	99.77	392.40
12/19/86	247.85	100.71	392.40
12/26/86	246.80	101.21	389.60
01/02/87	245.85	100.87	400.60
01/09/87	258.60	102.56	404.10
01/16/87	265.80	102.09	413.90
01/23/87	268.40	101.34	401.90
01/30/87	273.25	100.65	401.70
02/06/87	280.30	101.49	400.20
02/13/87	281.15	100.46	393.50
02/20/87	284.75	101.27	401.50
02/27/87	282.65	102.40	400.20
03/06/87	290.30	102.27	400.60

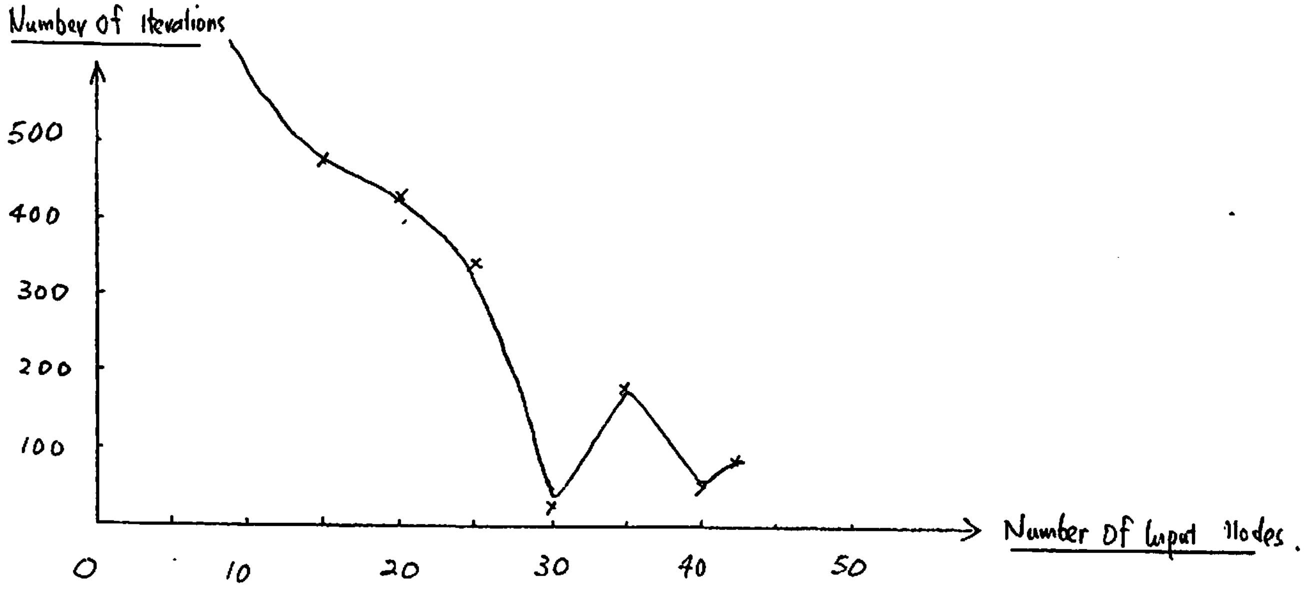
Data Set # 2 For The Test Model

02/27/87	$ 282.65	$ 102.40	$ 400.20
03/06/87	290.30	102.27	400.60
03/13/87	288.45	102.71	397.80
03/20/87	297.05	102.62	399.10
03/27/87	294.60	101.65	416.00
04/03/87	300.70	99.71	412.40
04/10/87	290.65	96.65	452.20
04/17/87	286.15	96.18	435.80
04/24/87	278.20	91.52	454.00
05/01/87	287.15	93.77	448.00
05/08/87	291.90	94.18	445.20
05/15/87	284.60	90.84	464.70
05/22/87	279.10	91.40	453.10
05/29/87	286.75	94.09	438.60
06/05/87	290.55	93.68	441.40
06/12/87	299.40	95.43	441.50
06/19/87	304.30	95.49	431.30
06/26/87	306.15	94.49	432.70
07/03/87	302.35	95.58	431.40
07/10/87	304.85	95.30	428.60

TableIV - 4 : Data for the current prices forecasting model. This is extracted from the New York Times, Business Section. This twenty one sets (the 21st set of data serves as the desired output for the purpose of training the data) of data is used to give a forecast prices of these three corporations on the data April 25, 1991.

Date	IBM	MCI	Apple
03/27/91	$112.75	$26.75	$69.25
03/28/91	114.250	25.625	68.000
03/29/91	113.750	26.500	70.000
04/01/91	113.500	27.000	71.750
04/02/91	113.000	27.625	72.750
04/03/91	113.120	27.250	70.000
04/04/91	113.500	26.625	71.500
04/05/91	112.625	26.250	69.375
04/08/91	113.000	26.625	70.000
04/09/91	111.125	26.250	68.750
04/10/91	111.125	26.000	66.875
04/11/91	109.250	27.000	69.250
04/12/91	108.375	27.625	71.750
04/15/91	107.000	27.625	62.250
04/16/91	109.250	28.250	64.250
04/17/91	109.750	29.500	63.250
04/18/91	109.625	29.375	61.000
04/19/91	109.375	28.625	59.625
04/22/91	108.750	28.500	60.000
04/23/91	108.375	28.375	61.500
04/24/91	108.250	28.250	60.750
Data Obtained From The Software : (see Fig V - 3)			
04/25/91	**108.200**	**60.640**	**28.270**

Graph II - 1 : Graph for Table II -1 (see explanation in Table II -1)



Graph II - 2 : Graph for II - 2 (see exploration in Table II -2)

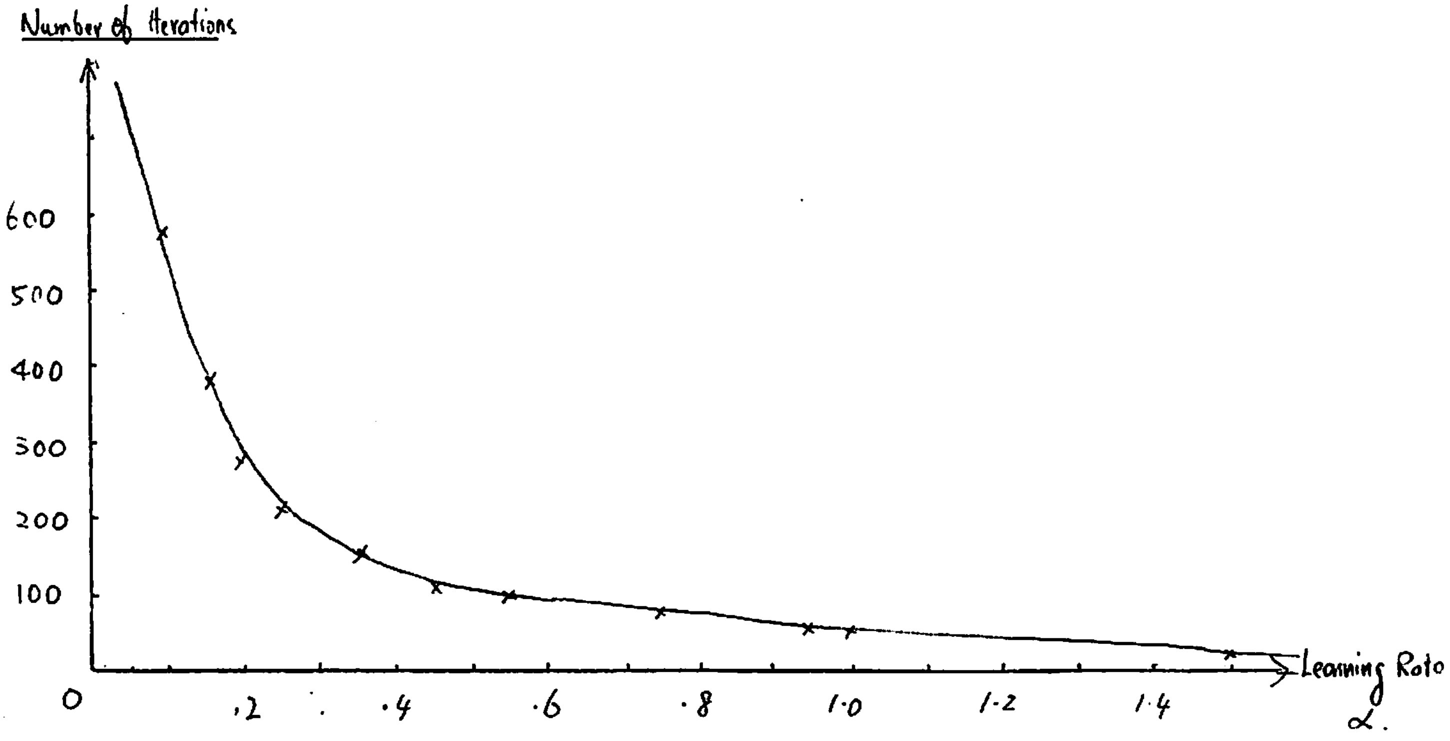
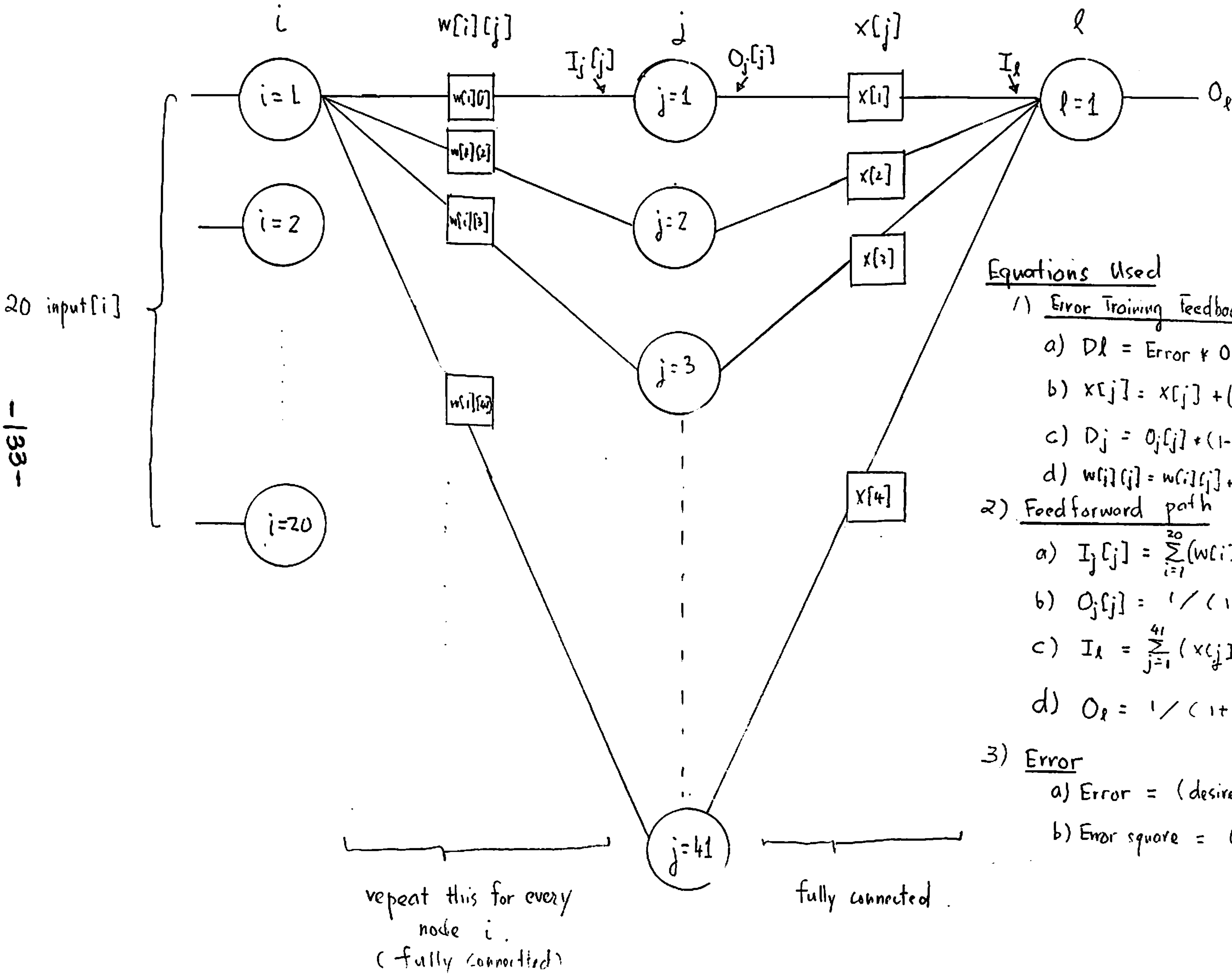


Fig II - 1 : The 3-layer backpropagation network used.



Equations Used

1) Error Training Feedbackword Path

a) $D\ell = \text{Error} * O\ell * (1 - O\ell)$

b) $x[j] = x[j] + (\alpha * D\ell * O_j[j])$

c) $D_j = O_j[j] * (1 - O_j[j]) * x[j] * D_\ell$

d) $w[i][j] = w[i][j] + (\alpha * D_j * \text{input}[i])$

2) Feedforward path

a) $I_j[j] = \sum_{i=1}^{20} (w[i][j] * \text{input}[i])$

b) $O_j[j] = 1 / (1 + \exp(-I_j[j]))$

c) $I_\ell = \sum_{j=1}^{41} (x[j] * O_j[j])$

d) $O_\ell = 1 / (1 + \exp(-I_\ell))$

3) Error

a) Error = (desired o/p - O_ℓ)

b) Error square = (desired o/p - O_ℓ)^2

Fig. V - 2 : The following contains the results of the testing model runs. The predicted outcome is compared to an existing indices and its difference is shown next to it. This difference shows how good this forecasting model is to predict indices. The information on the statistics of parameters are as follow. The number of iterations performed to reach 10e-6 accuracy with its respective learning rate (alpha) are presented on the same line. Next, the number of input and hidden nodes used in the network are also presented. The number of output node is always one. The error terms are the diference between the desired output and the actual trained output after iterations times. For the input of these results, see **Table IV - 3.**

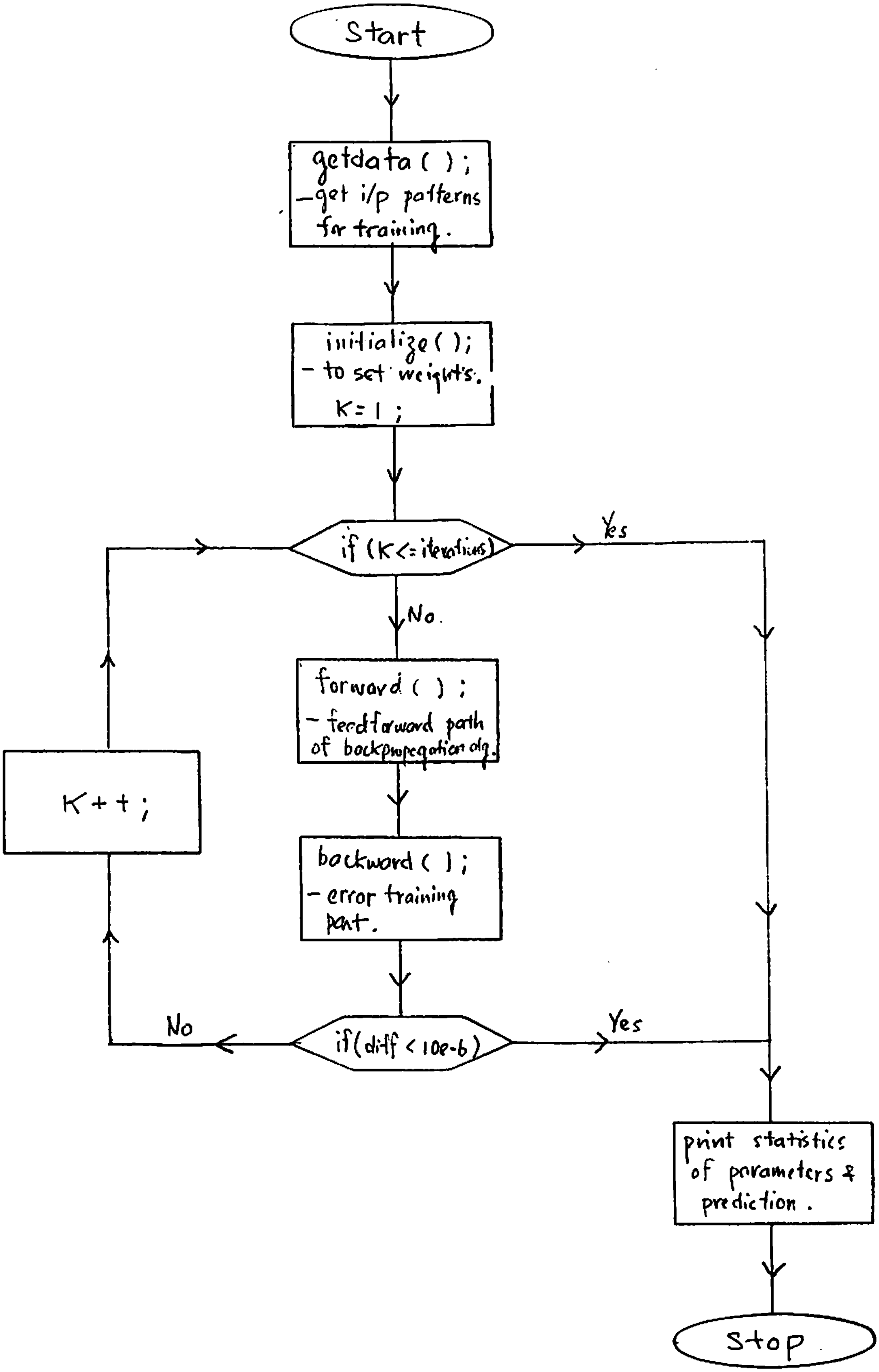
```
**********************************************************************
*                   S&P 500
**********************************************************************

Statistics Of Parameters.....
Num(IterationsPerformed) = 355 ; Alpha = 0.990000
Num(InputNodes) = 20 ; Num(HiddenNodes) = 41
Error = 0.000001 ; Error Square= 0.000000


The Predicted Outcome :  [ 282.984098 ] ; Difference : [ 2.520114 ]
Thank You For Using Kshin's Software.....

************************************************** 2nd set of data

Statistics Of Parameters.....
Num(IterationsPerformed) = 41 ; Alpha = 0.990000
Num(InputNodes) = 20 ; Num(HiddenNodes) = 41
Error = 0.000001 ; Error Square= 0.000000


The Predicted Outcome :  [ 302.132582 ] ; Difference : [ 0.891397 ]
Thank You For Using Kshin's Software.....

**********************************************************************

**********************************************************************
*                   T-Bond
**********************************************************************

Statistics Of Parameters.....
Num(IterationsPerformed) = 105 ; Alpha = 0.990000
Num(InputNodes) = 20 ; Num(HiddenNodes) = 41
Error = 0.000001 ; Error Square= 0.000000


The Predicted Outcome :  [ 102.464676 ] ; Difference: [ -0.190358 ]
Thank You For Using Kshin's Software.....

************************************************** 2nd set of data

Statistics Of Parameters.....
Num(IterationsPerformed) = 54 ; Alpha = 0.990000
Num(InputNodes) = 20 ; Num(HiddenNodes) = 41
Error = 0.000001 ; Error Square= 0.000000
```

```
The Predicted Outcome :  [ 95.217078 ] ; Difference : [ 0.118456 ]
Thank You For Using Kshin's Software.....

**********************************************************************

**********************************************************************
*                         GOLD
**********************************************************************

Statistics Of Parameters.....
Num(IterationsPerformed) = 90 ; Alpha = 0.990000
Num(InputNodes) = 20 ; Num(HiddenNodes) = 41
Error = 0.000001 ; Error Square= 0.000000


The Predicted Outcome :  [ 399.787988 ] ; Difference: [ 0.202701 ]
Thank You For Using Kshin's Software.....

*************************************************** 2nd set of data

Statistics Of Parameters.....
Num(IterationsPerformed) = 103 ; Alpha = 0.990000
Num(InputNodes) = 20 ; Num(HiddenNodes) = 41
Error = 0.000001 ; Error Square= 0.000000


The Predicted Outcome :  [ 431.740992 ] ; Difference : [ -0.732848 ]
Thank You For Using Kshin's Software.....

**********************************************************************
```

Figure V - 3 : The following is obtained from the software for the current prices rundown. It gives detailed information about the forecast prices of the respective company. The information on the statistics of the parameters is about the condition of the network before the forecast feedforward run. In particular, it tells the number of iterations needed to achieve 10e-6 accuracy, the alpha used, the number of input nodes and output nodes used and the actual error difference between the desired output and the actual output of the trained network. The actual input to the network can be viewed at Table IV - 4.

```
*****************************************************************
*       Prediction for IBM stock on 4/25/91 Friday
*****************************************************************

Statistics Of Parameters.....
Num(IterationsPerformed) = 65 ; Alpha = 0.990000
Num(InputNodes) = 20 ; Num(HiddenNodes) = 41
Error = 0.000001 ; Error Square= 0.000000


The Predicted Outcome : [ 108.195088 ]
Thank You For Using Kshin's Software.....

*****************************************************************
*       Prediction for MCI stock on 4/25/91 Friday
*****************************************************************

Statistics Of Parameters.....
Num(IterationsPerformed) = 41 ; Alpha = 0.990000
Num(InputNodes) = 20 ; Num(HiddenNodes) = 41
Error = 0.000001 ; Error Square= 0.000000

The Predicted Outcome : [ 28.268447 ]
Thank You For Using Kshin's Software.....

*****************************************************************
*       Prediction for Apple stock on 4/25/91 Friday
*****************************************************************

Statistics Of Parameters.....
Num(IterationsPerformed) = 36 ; Alpha = 0.990000
Num(InputNodes) = 20 ; Num(HiddenNodes) = 41
Error = 0.000001 ; Error Square= 0.000000


The Predicted Outcome : [ 60.636073 ]
Thank You For Using Kshin's Software.....

*****************************************************************
```

Fig 4: Flow chart for the software.



```
/****************************************************************
*

*                           Neurocomputing Project

*                                    (c) 1991  kshin@aludra

*

*           This project uses the error back propagation technique
*  on a PDP type of network to predict indices from the stock
*  market.

*           It is a three layer PDP network with twenty input nodes
*  and fourty one hidden nodes.  Though, it only has one output
*  node.

*           It takes twenty samples of the stock indices and train
*  itself to a desired output (the most recently available
*  index).  It will train itself for 5000 times unless the error
*  between the desired and actual output is less than 10**(-6).
*           Note that this program also normalize all the inputs
*  values so that they (inputs) will fall on the linear portion
*  of the sigmoid function.  But, the predicted result is
*  renormalized before being output.

*

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

# include "stdio.h"
# include "math.h"

# define iterations 5000                /* default iterations */
# define alpha .99                              /* optimized alpha */
# define INPUT 20                               /* Number of input nodes */
# define HIDDEN (2*INPUT+1)             /* 2n + 1 */


/*  Notations : I - input ; O - output ; i - input layer ;
                o - output layer ; w - input/hidden weights ;
                x - hidden/output weights ; D - delta          */

        float   input[INPUT+1], w[INPUT+1][HIDDEN+1];
        float   Ij[HIDDEN+1], Oj[HIDDEN+1], x[HIDDEN+1];
        float   I1, O1, D1;
        float   diff, d, NormalizeFactor, correct;

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

*                                       BACKWARD ()

*

*           This function evaluates the backward cycle of the error
*  back propagation algorithm.

*

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

void    backward()
{
        int     i, j;

        diff = d - O1;
        D1 = diff * O1 * (1 - O1);
        for (j=1; j<=HIDDEN; j++)                            /* update upper layer */
                x[j]  = x[j] + (alpha * D1 * Oj[j]);

        for (i=1; i<=INPUT; i++)                             /* update lower layer */
                for (j=1; j<=HIDDEN; j++)
                   w[i][j] = w[i][j]+(alpha*(Oj[j]*(1-Oj[j])*x[j]*D1)*input[i]);
}

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

*                                                    FORWARD ()

*

*               This function evaluates the forward cycle of the error
*  back propagation algorithm.

*

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

void    forward()
{
        int     i, j;

        for (j=1; j<=HIDDEN; j++)                            /* cal Ij[i] */
        {
                Ij[j] = 0;
                for (i=1; i<=INPUT; i++)
                        Ij[j] = w[i][j] * input[i] + Ij[j];

                Oj[j] = 1 / (1 + exp((-1) * Ij[j]));
        }

        I1 = 0;
        for (j=1; j<=HIDDEN; j++)                            /*  cal I1 */
                I1 = I1 + x[i] * Ij[i];

        O1 = 1 / (1 + exp((-1) * I1));
}

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

*                                                    PREDICT  ()

*

*               This function will predict the next value by first
*  moving the desired output to become a member of the input.
*  Then, it will make a forward pass of the network and obtain
*  the predicted output by taking the actual output at the end
*  output.  Note that it would have to be renormalized to be
*  meaningful.

*               For the purpose of this simulation, the predicted value
*  is compared to the actual value and a percentage difference
*  is output.

*
```

```
*****************************************************************/

void    predict()
{
        int     i;
        float   difference;

        for (i=1; i<INPUT; i++)
                input[i] = input[i+1];
        input[INPUT] = d;

        forward();
        difference = ( (correct - (NormalizeFactor*O1)) / correct ) * 100;
        printf("\nThe Predicted Outcome :  [ %2f ] ; Difference : [ %f ]\n",(Nor
malizeFactor*O1),difference);
}

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

*                                                       GETDATA  ()

*

*               This function will get input in the standard i/o format.
*  Input includes : 1) raw data to be input ; 2) Desired output
*  3) Actual correct value .

*

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

void    getdata()
{
        int     i;

        NormalizeFactor = 1;
        for(i=1; i<=INPUT; i++)
        {
                printf ("Input # %5d : ",i);
                scanf ("%f", &(input[i]));
                if (input[i] > NormalizeFactor)
                        NormalizeFactor = input[i];
        }
        NormalizeFactor = NormalizeFactor + 10;
        for (i=1; i<=INPUT; i++)
                input[i] = (input[i] / NormalizeFactor);
        printf ("Input Desired Output  :  ");
        scanf ("%f", &d);
        d = d / NormalizeFactor;
        printf ("Input Correct Value  :  ");
        scanf ("%f", &correct);
}

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

*                                                       INITIALIZE  ()

*

*               This function will set the two sets of weights.
*

*****************************************************************/
```

```
void    initialize()
{
        int     i, j;

/*  Setting upper layer's weights  */

        for (j=1; j<=HIDDEN; j++)
        {
                x[j] = rand() % 10000;
                x[j] = x[j] / 20000;
                j++;
        }
        for (j=2; j<=HIDDEN; j++)
        {
                x[j] = rand() % 10000;
                x[j] = x[j] / 20000 * (-1);
                j++;
        }

/*  Setting lower layer's weights  */

        for(i=1; i<=INPUT; i++)
                for(j=1; j<=HIDDEN; j++)
                        w[i][j] = 0.;
}

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

*                                                MAIN  DRIVER

*

*               This driver function will call various related functions
*  to coordinate in order to run the error back propagation
*  algorithm.

*

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

main()
{
        int     k;

        getdata();                                      /* get raw data from use
r */
        initialize();                           /* setting the two sets of weigh
ts */

        for (k=1; k<=iterations; k++)
        {
                forward();                              /* feedforward parts */
                backward();                             /* feedbackward parts */

        /* if error of the trained network is less than 10**(-6)
           jump out of this loop */

                if (diff < .000001)
                        break;
        }

        printf("\nStatistics Of Parameters.....\n");
        printf("Num(IterationsPerformed) = %d ; Alpha = %f\n",k-1,alpha);
        printf("Num(InputNodes) = %d ; Num(HiddenNodes) = %d\n",INPUT,HIDDEN);
        printf ("Error = %8f ; Error Square= %8f\n\n",diff,(diff*diff));
        predict ();
        printf ("Thank you \n");
```

Project Report

King, Li-Ping
(545-67-2395)

A Voice Control System

April 29, 1991

Introduction

In recent years, the advances in integrated circuits and digital computers has established a growing effort to resemble some aspects of brain function both in hardware and software. Although we do not know how brains work, but some clues on how information propagates from one nerve cell, or neuron, to the next have been found and modeled mathematically. Based on the enough information, constructing mathematical models for networks of highly interconnected neurons that are at least reasonably consistent with biological observations has been done.

In a brain, one nerve cell is typically connected to approximately 10,000 other nerve cells. A human brain has about 100 billion neurons; a simple animal brain, like that of a mollusk, has about a million neurons. In contrast, the models studied so far have at most a few thousand neurons, and the circuit chips based on these models now have at most a few hundred electronic neurons. Due to the reason that the real neurons respond only at millisecond time scale, a million times slower than transistors, it is apparent that the computation abilities of brains arise from the large number of neurons and the huge number of interconnections.

A biological neuron has an input structure called the dendrite and an output structure called the axon. The connection between the dendrite and axon is called synapse. When a neuron is exited by its inputs, it produces a train of pulses. When pulses from other neurons arrive at the synapses of a given neuron, they either increase the pulsing rate or decrease it depending on whether the connections are inhibitory of excitatory. [reference 3]

In a simple model of neural function the neuron pulsing rate is determined by a sigmoid function of a weighted sum of the inputs. The pulsing rate vi of neuron i is given by

$$vi = f(\sum_{j} vj\ Tij)$$

Here f is a sigmoid function, and Tij is the strength of the connection from neuron j to neuron i. The above equation can be implemented electronically. The Tij are the conductances of the array of input resistors, the summing is done on a wire dendrite, f is provided by the transfer function of the amplifier, and the output is broadcast to other neurons by a wire axon. The neuron pulsing rate is represented by the amplifier output voltage.

As mentioned before, the computational ability of brains are based upon the large number of neurons and the huge number of interconnections. The major features of the artificial neural networks are also based on this property-- the numerous processing/memory elements. Of course, a neural network can be successfully trained rather than programmed. The training of the network is based on learning rules/laws.

Design of intelligent systems is based on massive parallelism. The ultimate level is parallelism in which the number of computational units is so large that it can be treated as a continuous quantity. It is clear that processing elements on this level of parallelism cannot be individually programmed; they must be controlled all together. This leads to a new model of computation, based on the transformation of continuous scalar and vector fields. And this is role where neural networks play.

Project Overview

Since neural network can be successfully trained to recognize the features in a signal, speech, or image, by observing samples, neural networks are fully capable of discovering hidden information in process control. As a result, the network can learn the data and develop a control law with minimal assistance from specialists.

The goal of this project is to transfer voice to digital signals. And with the digital signals, we are able to control the desired output devices connected to the system. In another word, this project will utilize the computer as an intelligent analog-to-digital conversion device with the ability of recognizing the input voice signals. A design of an interface controller card for the IBM PC/XT is the concern of this project. A block diagram of the design is shown in *figure 1.*

Microphone is used as a transducer to convert nonelectrical source into an electrical wave form. The output of the microphone will be analog signals which contains desired message as well as noise signals. The output voltage level of the microphone is too low for subsequent use, thus an amplifier is used to boost the

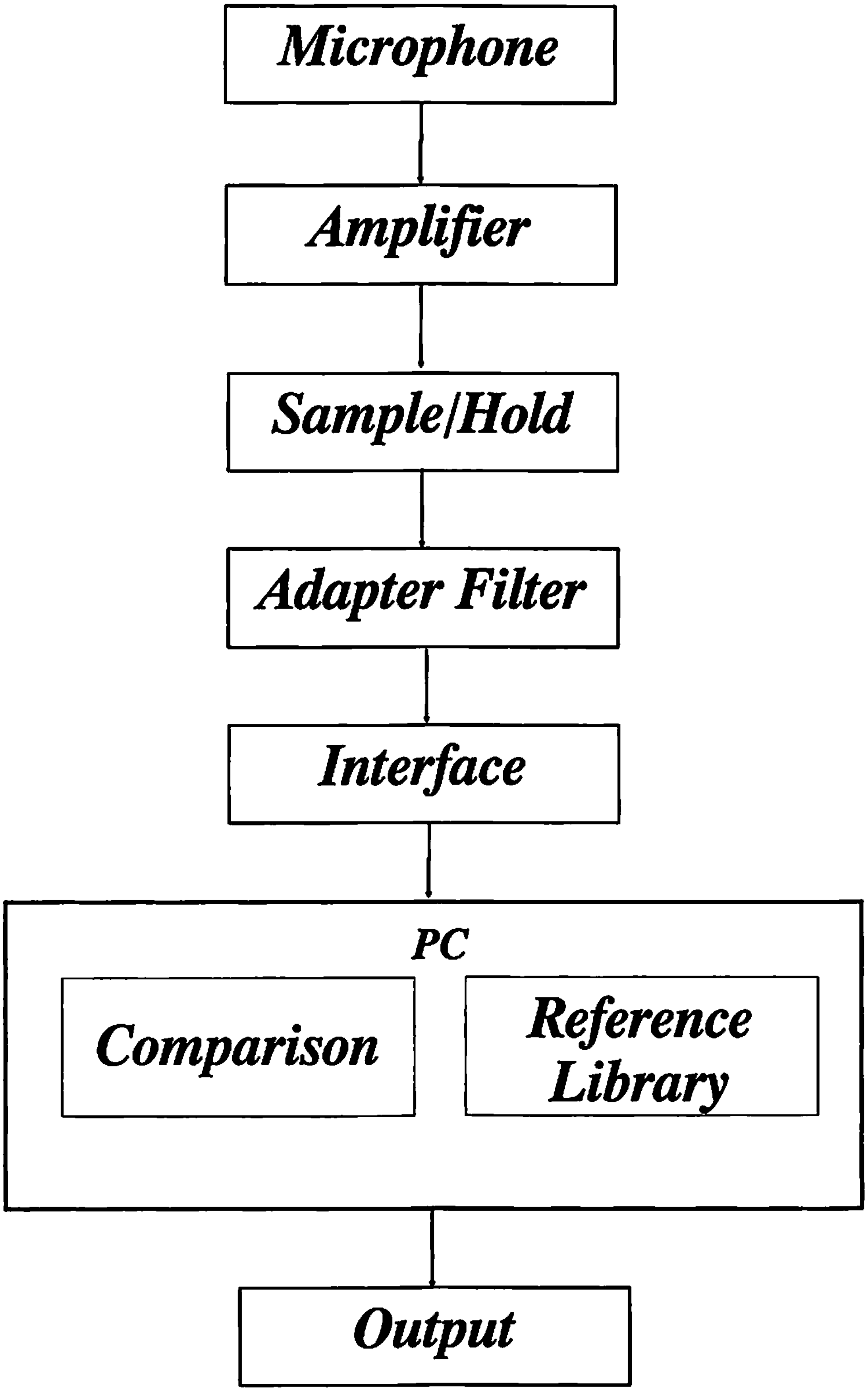


Figure 1: System Block Diagram

speech voltage level from the microphone to a higher usable values for subsequent digitization. The output of the amplifier is processed by the sample-and-hold circuit.

Voice recognition system must sample or learn a word in the way that humans speak it. Thus, filtering out undesired noise would be a primary concern. An adaptive linear neuron called Adaline was developed by Widrow. The output of Adaline is compared with a desired response, the difference or error is used to adjust the weights, and the Widrow-Hoff least mean square algorithm is used to minimize the error. This adaptive neurons can be used as part of an adaptive filter.

The output of the adaptive filter then goes to the analog-to-digital converter. The digitized data will be stored in buffer (reference library) as a sample after the training phase or as a data to be compared with the samples
in the reference library. Once the data is recognized, then, a control signal will be sent to control the devices connected to the system.

Adaptive Filter

An adaptive digital filter is shown in *figure 2*. The sampled input signal is applied to a string of delay boxes, each delaying the signal by one sampling period. An adaptive linear combiner produces an output that is a linear combination of the current and past input signal samples. The desired response is supplied during training. The weights are adjusted so that the output signal provides the best least square match over time to the desired response. [reference 8]

Figure 3 shows the adaptive filter used for noise canceling. This approach is viable only when an additional reference input is available containing noise n1, which is correlated with the original corrupting noise n0. The adaptive filter filters the noise n1 and subtracts the result from the primary input s+n0. For this adaptive filter, the noisy input s+n0 acts as the desired response. The "system output" acts as the error for the adaptive filter. [reference 8]

Since an additional reference input is needed, an extra microphone is to be hooked up to catch the surrounding environmental noise.

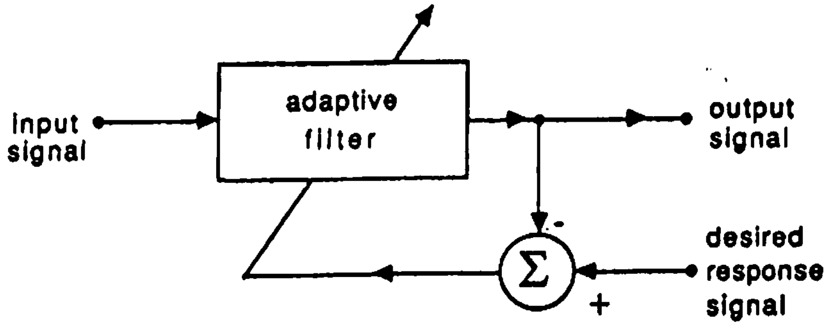


Figure 2: Adaptive Filter

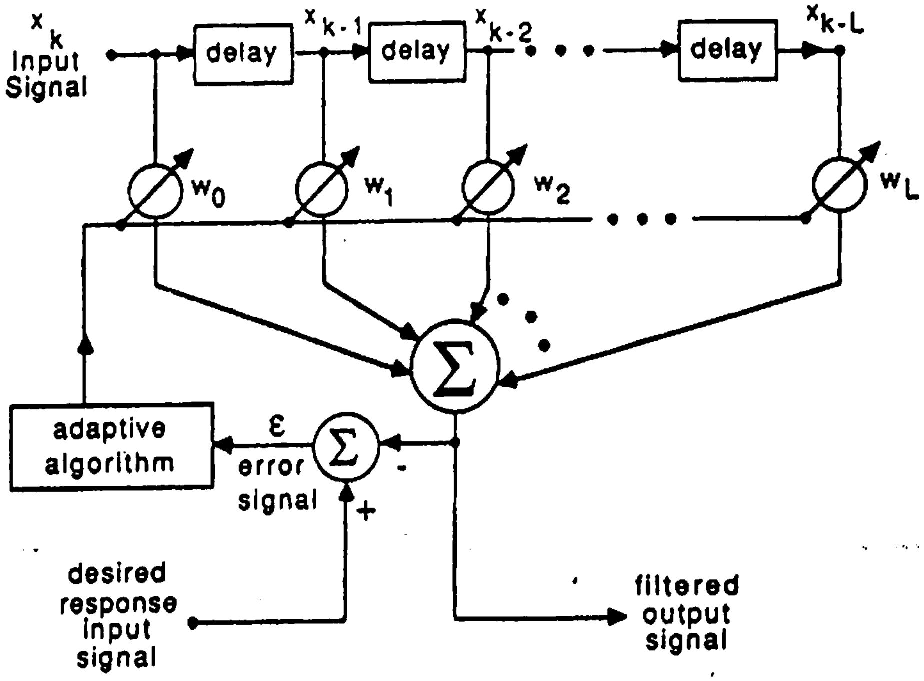


Figure 3: Details of a Adaptive Digital Filter

B. Widrow and R. Winter have explained that little or no prior knowledge of s, $n0$, or $n1$ of their interrelationships is required before the filter could adapt to produce the noise-canceling signal y. Their argument is shown as follow:

s, $n0$, $n1$ and y are assumed statistically stationary and have zero means. Further assume s is uncorrelated with $n0$ and $n1$, and $n1$ is correlated with $n0$. The output then is given as

$$e = s + n0 - y$$

and squaring both sides,

$$e^2 = s^2 + (n0-y)^2 + 2s(n0-y)$$

Taking the expectation of both sides, and realizing that s is uncorrelated with $n0$ and with y, which yields

$$\begin{aligned} E[e^2] &= E[s^2] + E[(n0-y)^2] + 2E[s(n0-y)] \\ &= E[s^2] + E[(n0-y)^2] \end{aligned}$$

Adapting the filter to minimize $E[s^2]$ will not affect the signal power $E[s^2]$. Accordingly, the minimum output power is

$$Emin[e^2] = E[s^2] + Emin[(n0-y)^2]$$

When the filter is adjusted so that $E[e^2]$ is minimized, $E[(n0-y)^2]$ is therefore also minimized. The filter output y is then a best least-squares estimate of the primary noise $n0$. Moreover, when $E[(n0-y)^2]$ is minimized, $E[(e-s)^2]$ is also minimized, since, from the first equation,

$$e - s = n0 - y$$

Adjusting or adapting the filter to minimize the total output power is equivalent to causing the output e to be a best least-squares estimate of the signal s for the given structure and adjustability of the adaptive filter and for the given reference input. [reference 8]

Sample-and-Hold & A/D Converter

The speech digitization process in which the speech analog waveform is converted to a serial string of 8-bit bytes for each data sample. What is occurring is that at periodic intervals a sample-and-hold circuit captures a snapshot of the analog incoming voltage, send the voltage to the adaptive filter, and what comes out from the adaptive filter is immediately converted to a voltage-equivalent parallel data byte by an analog-to-digital converter.

Upon the command from the computer, the circuit that has been continuously sampling the input waveform holds the current analog voltage during the adaptive filtering and digitization process. Following the release of the hold command, the sample-and-hold circuit returns to the sample mode (the sample time is 20 usec for the LF198 sample-and-hold chip used). During the time that the sample-and-hold circuit is holding the analog voltage, the 8-bit analog-to-digital converter performs the actual digitization of the held speech waveform. Conversion time for the A/D converter used (ADC0809) is 100 usec. Thus, after 100 usec, the controlling computer may now input the digital byte corresponding to the temporary voltage level and repeat the entire cycle, reset the sample-and-hold circuit to its sample mode and delay for the next periodic conversion time out, until the full desired waveform has been captured.

Interface

In many interfacing applications, it becomes necessary to accept or transmit data to or from an interface. The rates at which data is transmitted is higher than possible with a simple programmed I/O loop that uses IN and OUT instructions. However, the speed of the I/O port was taken into consideration first to see if it was fast enough for the A/D converter.

For the sample-and-hold circuit, the sample time is 20 usec and the conversion time of the A/D converter is 100 usec. Fortunately, one I/O read or write takes only 5 clock cycles which is 1.05 usec. Moreover, for the memory write or read bus cycle, it takes 0.84 usec each. As a result, for complete cycle of recording a data from the A/D converter to the memory location of the computer plus some delay to allow the sample-and-hold circuit to sample a new data, it still has plenty of time left to allow the I/O loops to come back for the next data latch from the A/D converter. Thus, using I/O ports for the design of the encoder is decided.

In order to communicate with the computer, an encoder is needed here. A simplified circuit diagram of the encoder is provided in *figure 4*. For this project, the use of I/O port addressing is chosen. Because most of the support devices and the I/O adapter in the PC are controlled and sensed by using the digital input and output ports. These ports are addressed using the I/O port address space of the 8088 microprocessor. Data can be sent to these ports by using the OUT instruction of the 8088 microprocessor. Data can be or read from these ports by using the 8088 IN instruction. The 8088 microprocessor architecture supports an I/O port address space of 65,536 unique port addresses. The PC design does not use the full address space. Only the lower 10 bits of the address bus are used in the PC. Thus, bit 0 through bit 9 of the address bus are used to decode interface card or port addresses. Bit 9 of the I/O port address has special meaning in the PC design. When this bit is inactive, data cannot be received on the system bus from the expansion card slots on the system board. When inactive, this bit enables data only from the devices and the I/O port addresses on the system board. When this bit is active, it enables data from the expansion card slot.

The simplest way to decode an I/O port address or group of addresses for an interface design is to find a block of unused addresses and then construct the proper encoder decoder; for this project, the ports decoded are 250H to 257H. The signal AEN (address enable) provided in the expansion slot should be connected to the interface design to degate the decode. This is required to prevent an invalid port address decode during DMA cycles.

In *figure 4 (a and b)*, the address ports are decoded by using an SN74LS688 octal comparator. Bit 0 to bit 9 and AEN are attached on one side. The ports decoded are set by the dip switches on the other side of the comparator. The reason why the dip switches are used is to prevent further feature cards designed by IBM so that the ports can be easily moved to other available port addresses. When the address set by the dip switches is the same as the address bus, the P=Q signal is turned low and used as a group select signal.

S/H A/D & DECODER

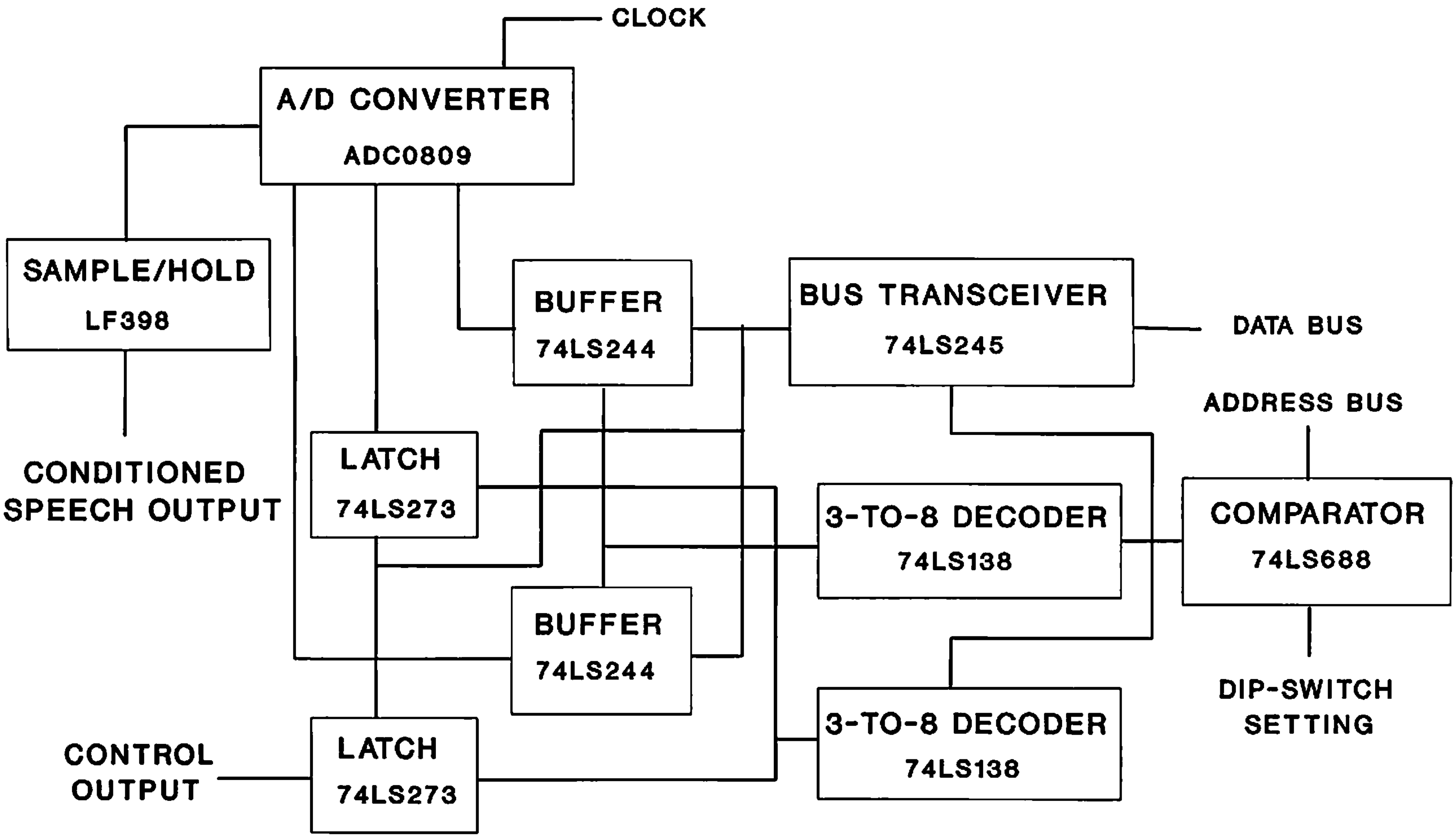


Figure 4a: Circuit Block Diagram

S/H & A/D & DECODER

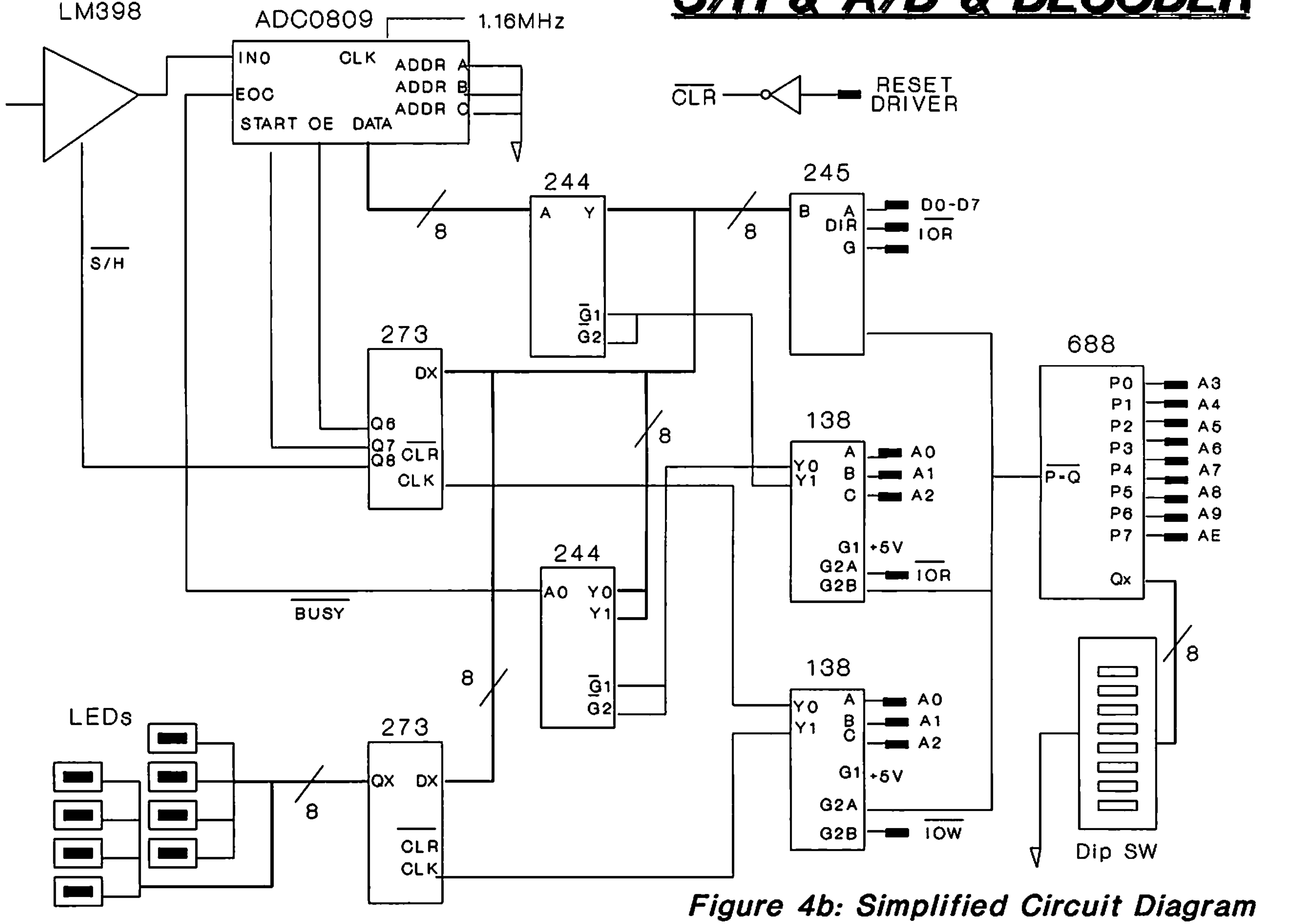


Figure 4b: Simplified Circuit Diagram

The data from and to the data bus is controlled by the octal bus transceivers, SN74LS245, with tri-state output. Hence it is compatible with the computer bus. The direction of the data bus are controlled by the signal IOR provided by the expansion slot. When IN instruction is performed, IOR is taken to be low which sets the direction of the transceiver form B to A. It is then reading data. When OUT instruction is executed, the IOR signal is taken high and which sets the direction from A to B; this is data writing.

Two 3 x 8 decoders, SN74LS138, are used to accomplish the write or the read cycles form the data bus. The upper 138 is used to read data because it is connected to the IOR signal. The lower one is to write data cycles since the IOW signal is attached.

For the output signals, two SN74LS273, octal D-type flip-flop with clear are used. The upper SN74ls273 is used to latch the control signals from the computer to control the sample-and-hold and A/D converter. An octal buffer and line driver with tri-state outputs, SN74LS244, are used to take data from the A/D converter. The lower SN74LS273 is used here to latch the device control signal from the CPU to the LEDs. The eight LEDs indicates the devices connected to the system. Hence, the on/off LEDs represent the control signals to turn on/off the devices accordingly.

Software Development

The purpose of the software is to train the real-time computer to recognize the input voice and output the correct signal to control external devices. The software of the real-time voice control system consists four primary phases: there are training phase, voice input phase, comparison phase and signal output phase. The first one phases and the last three phases can be written in two independent programs, which are called training program and control program respectively.

A copy of the flow charts (*figure 5.1a,b,c,d,e*) of these four phases are included at the end of this paper. The source code is written in 8086 assembly language and is also attached to the paper.

In the first phase, the training phase, to input the number of different voice and training time for each voice are required at the beginning of the phase. Then, voices have to be generated from external microphone. The real-time computer gets the voices and records the digitized data. Finally, the digitized data are stored in the main memory or hard disk and formed the database of the voice. The step of recording the voice will be described in the following paragraph.

In the second phase, the voice input phase, the real-time computer gets a voice from external environment. For recording the voice, the program consists the controlling of the A/D conversion and S/H mode determination. The program will start A/D conversion by commanding S/H circuit to be on Hold mode and A/D converter on conversion mode. A 100uS delay is used to allow the conversion time needed for the A/D converter(8089 needs a maximum of 100uS of conversion time). The program then will set the A/D converter on output mode by enable the OE pin of the converter. While doing so, S/H circuit is set to be on sample mode for the next data conversion (a 20uS delay is used to allow the sampling time required by LF398). The program then will read in the digitized data and store the data in the temporary location.

In the third phase, the comparison phase, the digitized data which is generated in the second phase will be compared with the each data of the database of voice. If either these two 8-bit digitized voice data match each other completely or only one bit in the 8-bit digitized voice data is not the same as the bit in the other 8-bit digitized for the same bit position, the real-time voice computer will recognize the input voice.

In the last phase, the signal output phase, if the input voice is recognized by the real-time computer, the proper action will be taken, that is, the real-time computer will send the control signal to handle the external devices. If the input voice is not recognized after comparing all data in the voice database, the real-time computer will send a message to announce that there are not any digitized voice data can be matched and found.

The software of project is divided into four phases. Establishing the voice database, getting an external command, searching the closest digitized voice from voice database, and sending the control signal to external devices are done by these four phases.

Conclusion

The goal is to develop a R/T voice (recognition) control system that has a very high computation rate yet has not sacrificed accuracy. Since the choice of dissimilarity or distortion measure is extremely important that the final recognition decision is generally based entirely upon the cleanness of the input signal. Many speech recognition system relied upon powerful computer system to achieve the high computation rate they required. For the sake of hardware limitation and time constrain, the adaptive filter portion was not finished. For this part, more research was needed to actual design the filter.

Reference

1. Cater, John P., Electronically Hearing: Computer Speech Recognition. Howard W. Sam & Co., Inc., 1984.

2. Effebrecht, Lewis C., Interfacing to IBM Personal Computer. Howard W. Sama & Company, 1988.

3. Jackel, L.D., Graf, H. P., and Howard, R. E., "Electronic neural network chips," Applied Optics, vol. 26, no. 23, Dec, 1987.

4. Lathi, B. P., Modern Digital and Analog Communication Systems. Holt, Rinehart and Winton, 1983.

5. Norton, Peter, Inside the IBM PC. Rober J. Brady Co., 1983.

6. Soucek, Branko, Neural and Concurrent Real-Time Systems. John Wiley & Sons, 1989.

7. Rabiner, L. R., Levinson, S.E., Rosenberg, A.E., and Wilpon, J. G., "Speaker independent recognization of isolated words using clustering techniques," IEEE Trans. Acoust., Speech, Signal Processing, vol. assp -27, Aug. 1979.

8. Widrow, Bernard and Winter, Rodney, "Neural Nets for Adaptive Filtering and Adaptive Pattern Recognition," Computer 25-39, March 1988.

Figure 5.1.a : Flowchart of Software

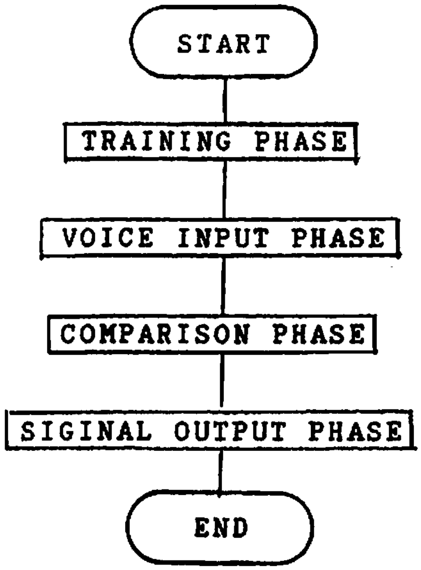
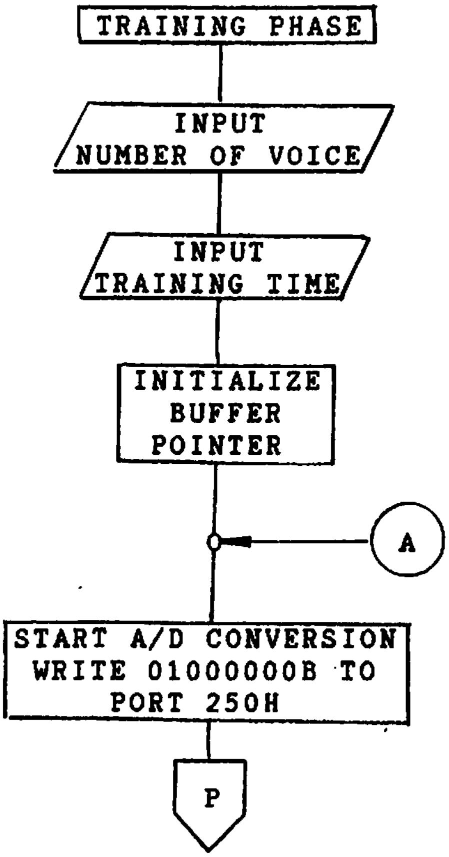
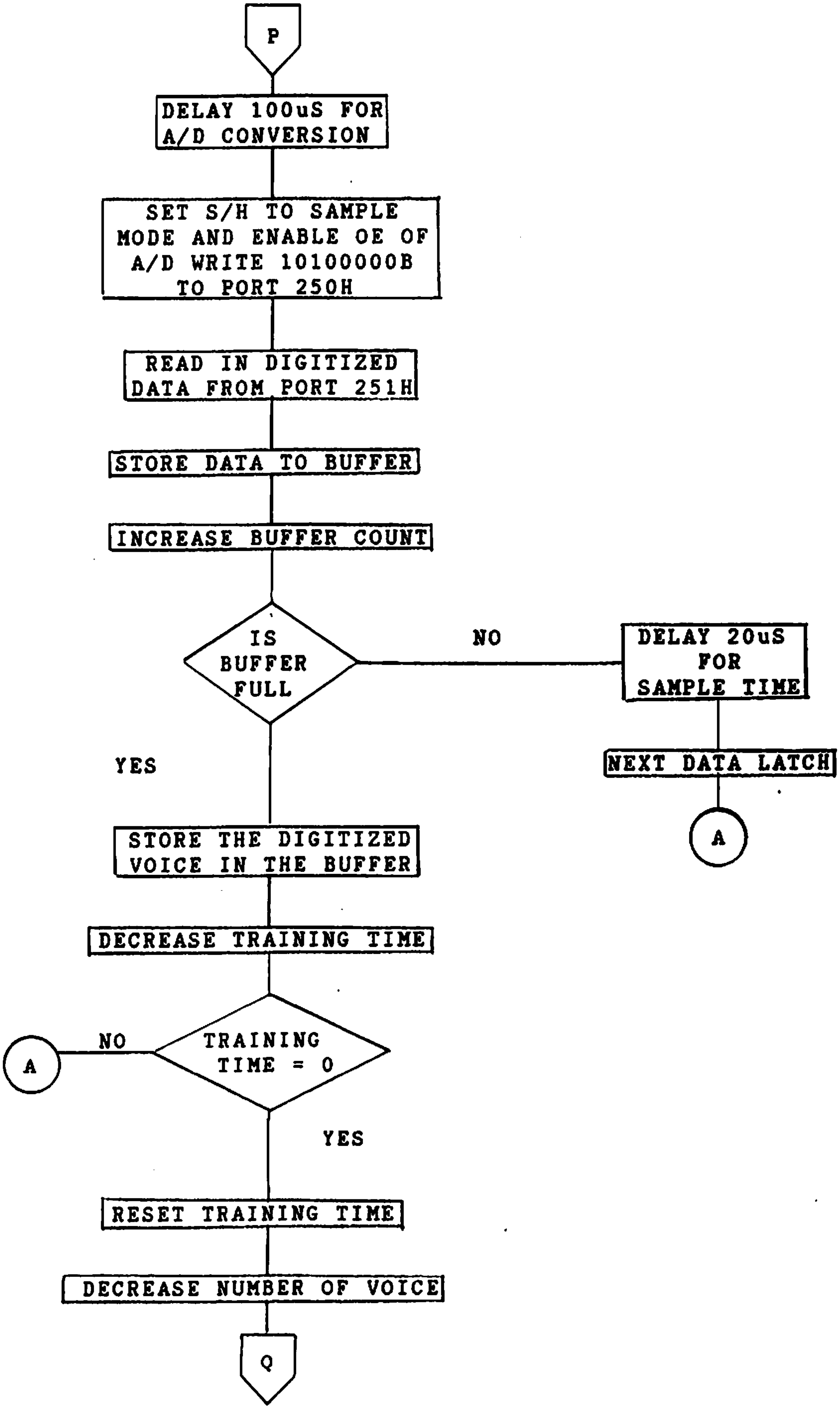
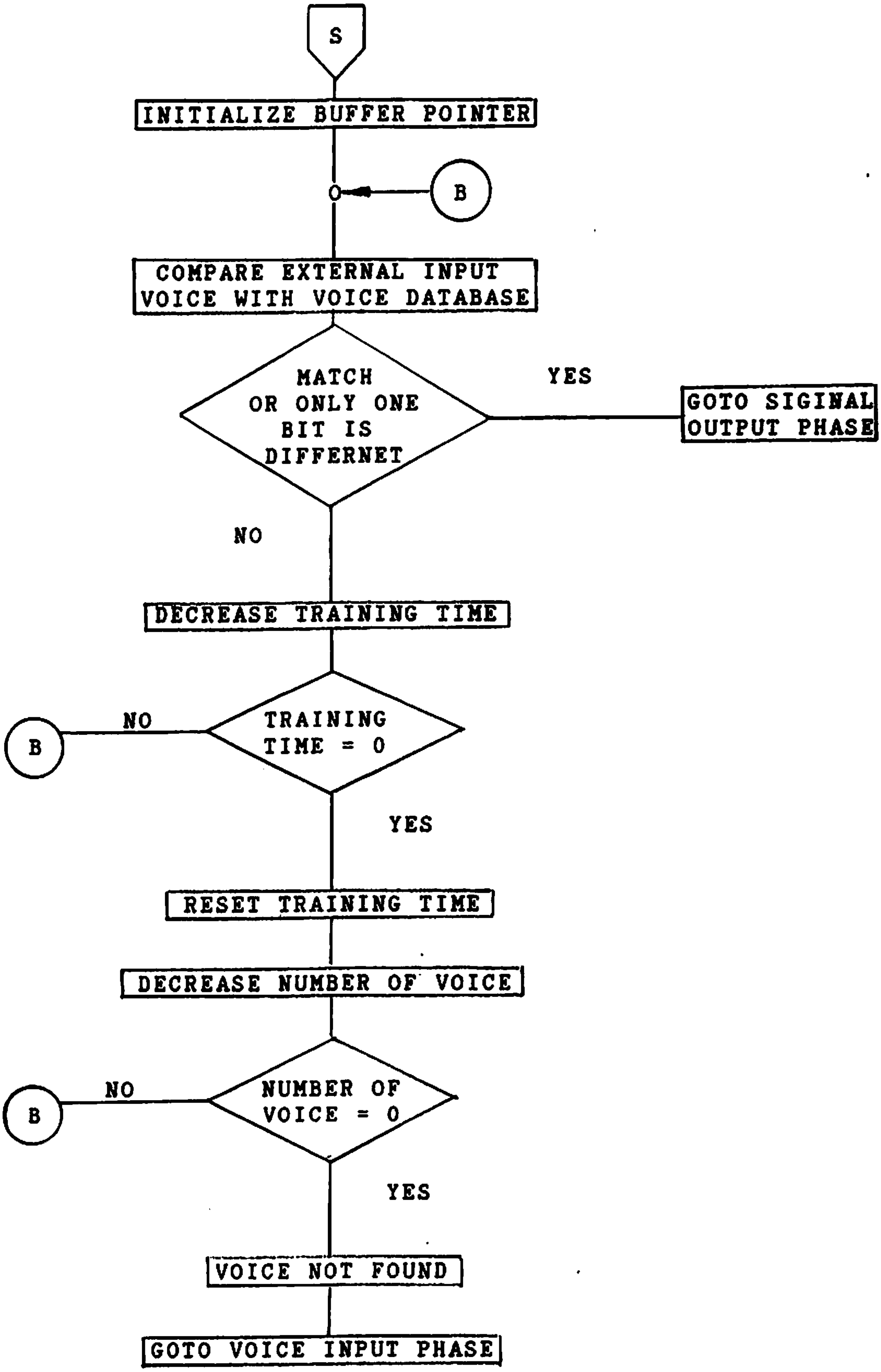


Figure 5.1.b : Flowchart of TRAINING PHASE


TRAINING PHASE
INPUT NUMBER OF VOICE
INPUT TRAINING TIME
INITIALIZE BUFFER POINTER
A
START A/D CONVERSION WRITE 01000000B TO PORT 250H
P


P
DELAY 100uS FOR A/D CONVERSION
SET S/H TO SAMPLE MODE AND ENABLE OE OF A/D WRITE 10100000B TO PORT 250H
READ IN DIGITIZED DATA FROM PORT 251H
STORE DATA TO BUFFER
INCREASE BUFFER COUNT
IS BUFFER FULL
NO
DELAY 20uS FOR SAMPLE TIME
NEXT DATA LATCH
A
YES
STORE THE DIGITIZED VOICE IN THE BUFFER
DECREASE TRAINING TIME
A
NO
TRAINING TIME = 0
YES
RESET TRAINING TIME
DECREASE NUMBER OF VOICE
Q


S
INITIALIZE BUFFER POINTER
B
COMPARE EXTERNAL INPUT VOICE WITH VOICE DATABASE
MATCH OR ONLY ONE BIT IS DIFFERNET
YES
GOTO SIGINAL OUTPUT PHASE
NO
DECREASE TRAINING TIME
NO
B
TRAINING TIME = 0
YES
RESET TRAINING TIME
DECREASE NUMBER OF VOICE
NO
B
NUMBER OF VOICE = 0
YES
VOICE NOT FOUND
GOTO VOICE INPUT PHASE

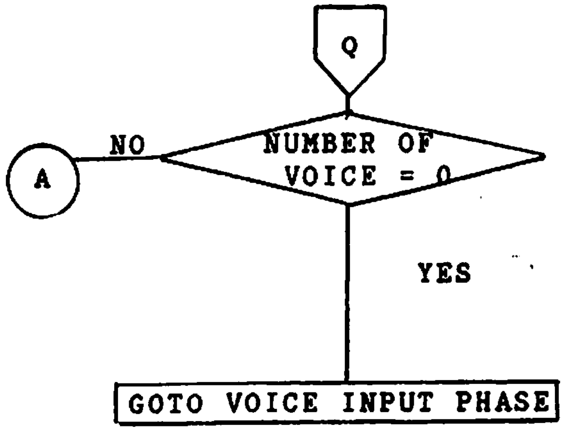
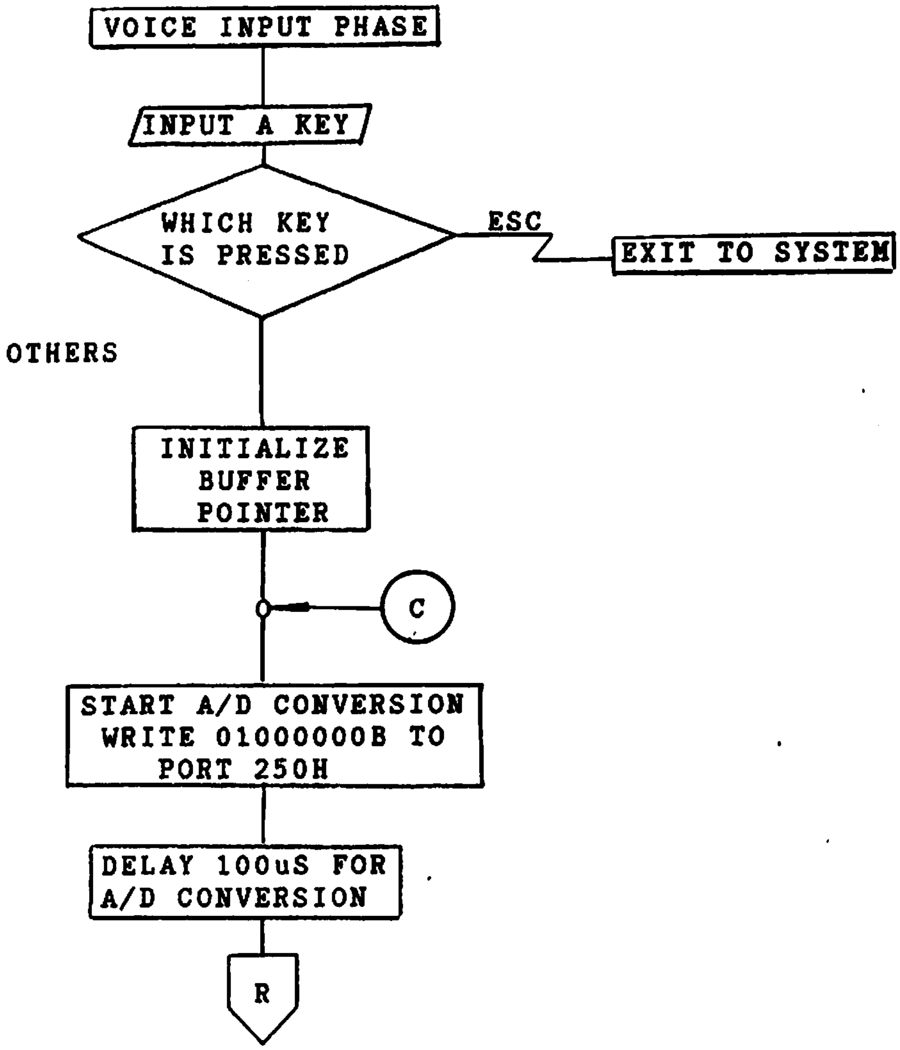
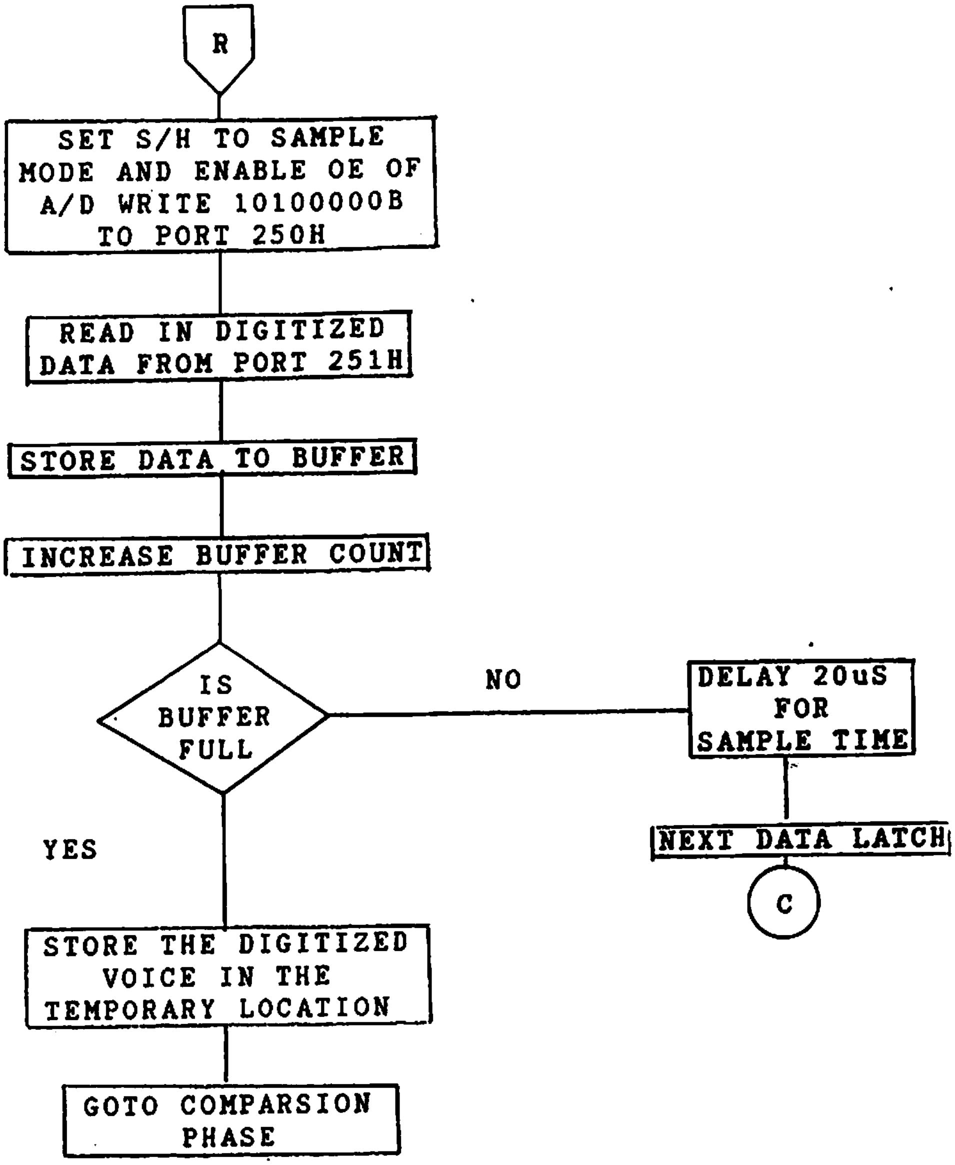
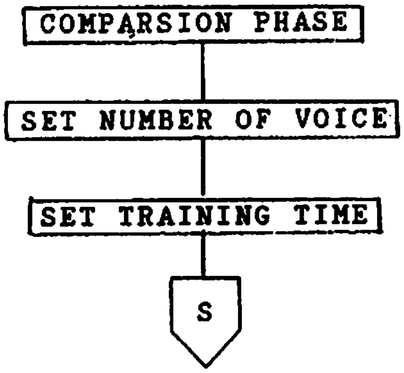


Figure 5.1.c : Flowchart of VOICE INPUT PHASE


VOICE INPUT PHASE
INPUT A KEY
WHICH KEY IS PRESSED
ESC
EXIT TO SYSTEM
OTHERS
INITIALIZE BUFFER POINTER
C
START A/D CONVERSION WRITE 01000000B TO PORT 250H
DELAY 100uS FOR A/D CONVERSION
R


R
SET S/H TO SAMPLE MODE AND ENABLE OE OF A/D WRITE 10100000B TO PORT 250H
READ IN DIGITIZED DATA FROM PORT 251H
STORE DATA TO BUFFER
INCREASE BUFFER COUNT
IS BUFFER FULL
NO
DELAY 20uS FOR SAMPLE TIME
NEXT DATA LATCH
C
YES
STORE THE DIGITIZED VOICE IN THE TEMPORARY LOCATION
GOTO COMPARSION PHASE



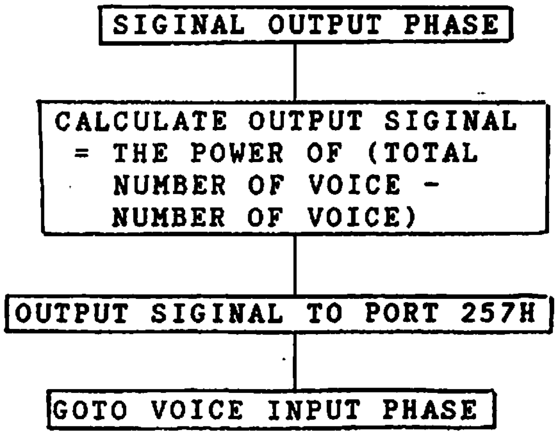

SIGINAL OUTPUT PHASE
CALCULATE OUTPUT SIGINAL = THE POWER OF (TOTAL NUMBER OF VOICE - NUMBER OF VOICE)
OUTPUT SIGINAL TO PORT 257H
GOTO VOICE INPUT PHASE

Figure 5.1.d : Flowachart of COMPARSION PHASE

Figure 5.1.e : Flowchart of SIGINAL OUTPUT PHASE

```
;***************
;
;         EE599
;
;***************
;
;*********************************************************************
;                                                                   *
; NOTES :                                                           *
;         [ 1 ] : PORTS DECODED -- 250H ~ 257H                      *
;         [ 2 ] : A/D CONVERTER INPUT PORT -- IN PORT 251H          *
;         [ 3 ] : S/H CKT & A/D CONTROL PORT -- OUT PORT 250H       *
;         [ 4 ] : D/A CONVERTER -- OUTPUT PORT 251H                 *
;         [ 5 ] : CONTROL DEVICE PORT -- OUTPUT PORT 257H           *
;                                                                   *
;*********************************************************************
;
STACK   SEGMENT STACK
        DB      128 DUP(0)              ;STACK SPACE
STACK   ENDS
;
DATA    SEGMENT PARA PUBLIC 'DATA'
;
BUFFER  DB      16000D DUP(0)           ;MAX MEMORY SPACE 64KB
VOICE_B DB      16000D DUP(0)           ;BUFFER FOR VARIED SOUNDS
;
MAXLEN  DB      4                       ;MAX LENGTH OF STRING
ACTLEN  DB      ?                       ;ACTUAL LENGTH OF STRING
CHAR    DB      4 DUP(?)                ;INPUT STRING
;
TEN     DB      10                      ;CONSTANT 10
VOICE   DB      ?                       ;NUMBER OF VOICE
TIME    DB      ?                       ;NUMBER OF TRAINING
O_VOICE DB      ?                       ;HARDCOPY OF VOICE
O_TIME  DB      ?                       ;HARDCOPY OF TIME
SOUND   DB      ?                       ;INPUT SOUND
O_SOUND DB      ?                       ;HARDCOPY OF SOUND
TOTAL   DB      0                       ;NUMBER OF BIT 1
POINTER DW      0                       ;POINTER OF BUFFER
E_KEY   DB      65H                     ;EXIT PROGRAM
;
MSG0    DB      0AH,0AH
        DB      '**********************************************',0DH,0AH
        DB      '*                                            *',0DH,0AH
        DB      '*   WELCOME TO THE REAL-TIME CONTROL SYSTEM   *',0DH,0AH
        DB      '*                                            *',0DH,0AH
        DB      '**********************************************',0DH,0AH
        DB      0AH,0AH,'PRESS ANY KEY TO BEGIN $',10,13
MSG1    DB      'PLEASE INPUT THE NUMBER OF DIFFERENT VOICE YOU WANT TO '
        DB      'TRAIN THE REAL-TIME VOICE CONTROL COMPUTER -- [1~8]
$',0DH,0AH
MSG2    DB      'PLEASE INPUT THE TRAINING TIME OF EACH VOICE -- '
        DB      '[1~8] $',0DH,0AH,0AH,0AH
MSG3    DB      'BEGIN TO RECORD AND INPUT THE FIRST VOICE ',0DH,0AH,0AH,0AH
```

```
        DB      'AFTER PRESSING ANY KEY $',0DH,0AH,0AH,0AH
MSG4    DB      'PLEASE INPUT ANOTHER VOICE AFTER PRESSING ANY KEY
                $',0DH,0AH,0AH,0AH
MSG5    DB      'PLEASE INPUT THE SAME VOICE AFTER PRESSING ANY KEY
                $',0DH,0AH,0AH,0AH
MSG6    DB      'BEGIN CONTROL PHASE......',0DH,0AH,0AH,0AH
        DB      'PRESS "e" --- EXIT PROGRAM ',0DH,0AH,0AH,0AH
        DB      'OTHER KEYS -- GET A VOICE $',0DH,0AH,0AH,0AH
MSG7    DB      'THE INPUT VOICE DOES NOT MATCH TRAINED VOICE $',0DH,0AH
MSG8    DB      'BYE-BYE!!!!!$',0DH,0AH,0AH,0AH
DATA    ENDS
;
CODE    SEGMENT PARA PUBLIC 'CODE'
START   PROC    FAR
;
;*********************************************************************
;                                                                    *
;   TRAINING PHASE -- INPUT THE NUMBER OF VOICE AND TRAINING TIME    *
;                     AND STORE THE VOICE TO BUFFER                  *
;                                                                    *
;*********************************************************************
;
;DEFINE SEGMENT ADDRESSABILITY
;
        ASSUME  CS:CODE,SS:STACK        ;
        PUSH    DS                      ;SAVE PSP SEG ADDR
        MOV     AX,0                    ;
        PUSH    AX                      ;SAVE RET ADDR OFFSET (PSP+0)
        MOV     AX,DATA                 ;
        MOV     DS,AX                   ;ESTABLISH EXTRA SEG ADDRESSABILITY
        ASSUME  DS:DATA                 ;
;
        MOV     AH,0                    ;SET DISPLAY MODE (40*25)
        MOV     AL,0                    ;
        INT     10H                     ;
        MOV     DX,OFFSET MSG0          ;DISPLAY MSG0 ON SCREEN
        MOV     AH,9H                   ;
        INT     21H                     ;
;
        MOV     AH,0                    ;SETUP READ KEYBOARD
        INT     16H                     ;READ KEYBOARD INPUT
;
V_AGAIN:
        MOV     AH,0                    ;SET DISPLAY MODE (40*25)
        MOV     AL,0                    ;
        INT     10H                     ;
        MOV     DX,OFFSET MSG1          ;DISPLAY MSG1 ON SCREEN
        MOV     AH,9H                   ;
        INT     21H                     ;
;
        LEA     DX,MAXLEN               ;LOAD ADDRESS OF INPUT BUFFER
        MOV     AH,0AH                  ;SET DOS FUNCTION CALL
        INT     21H                     ;CALL DOS FUNCTION TO GET INPUT
;
```

```
        MOV     AL,0                    ;CLEAR AL
        MOV     BX,0                    ;CLEAR BX
;
N_VOICE:
        SUB     CHAR[BX],30H            ;CONVERT ASCII TO REAL VALUE
        MUL     TEN                     ;AL=AL*10
        ADD     AL,CHAR[BX]             ;AL=AL+CHAR[BX]
        INC     BX                      ;INCREASE CONTENT OF BX REG
        DEC     ACTLEN                  ;DECREASE THE CHARACTER COUNTER
        JNE     N_VOICE                 ;IF COUNT <> 0,JUMP TO N_VOICE
;
        CMP     AL,0                    ;CHECK IT IS LESS THAN 0
        JBE     V_AGAIN                 ;IF ILLEGAL,INPUT AGAIN
        CMP     AL,8                    ;CHECK IT IS GREATER THAN 8
        JA      V_AGAIN                 ;IF ILLEGAL,INPUT AGAIN
;
        MOV     VOICE,AL                ;RESTORE TO VOICE
        MOV     O_VOICE,AL              ;BACKUP ONE HARDCOPY
;
T_AGAIN:
        MOV     AH,0                    ;SET DISPLAY MODE (40*25)
        MOV     AL,0                    ;
        INT     10H                     ;
        MOV     DX,OFFSET MSG2          ;DISPLAY MSG2 ON SCREEN
        MOV     AH,9H                   ;
        INT     21H                     ;
;
        LEA     DX,MAXLEN               ;LOAD ADDRESS OF INPUT BUFFER
        MOV     AH,0AH                  ;SET DOS FUNCTION CALL
        INT     21H                     ;CALL DOS FUNCTION TO GET INPUT
;
        MOV     AL,0                    ;CLEAR AL
        MOV     BX,0                    ;CLEAR BX
;
N_TIME:
        SUB     CHAR[BX],30H            ;CONVERT ASCII TO REAL VALUE
        MUL     TEN                     ;AL=AL*10
        ADD     AL,CHAR[BX]             ;AL=AL+CHAR[BX]
        INC     BX                      ;INCREASE CONTENT OF BX REG
        DEC     ACTLEN                  ;DECREASE THE CHARACTER COUNTER
        JNE     N_TIME                  ;IF COUNT <> 0,JUMP TO N_VOICE
;
        CMP     AL,0                    ;CHECK IT IS LESS THAN 0
        JBE     T_AGAIN                 ;IF ILLEGAL,INPUT AGAIN
        CMP     AL,8                    ;CHECK IT IS GREATER THAN 8
        JA      T_AGAIN                 ;IF ILLEGAL,INPUT AGAIN
;
        MOV     TIME,AL                 ;RESTORE TO TIME
        MOV     O_TIME,AL               ;BACKUP ONE.COPY
;
;RECORDING
;
        MOV     AH,0                    ;SET DISPLAY MODE (40*25)
        MOV     AL,0                    ;
        INT     10H                     ;
        MOV     DX,OFFSET MSG3          ;DISPLAY MSG3 ON SCREEN
        MOV     AH,9H                   ;
        INT     21H                     ;
;
        JMP     INIT                   ;GOTO RECORDING LOOP
;
ANOTHER_VOICE:
        MOV     AH,0                    ;SET DISPLAY MODE (40*25)
        MOV     AL,0                    ;
        INT     10H                     ;
        MOV     DX,OFFSET MSG4          ;DISPLAY MSG4 ON SCREEN
        MOV     AH,9H                   ;
        INT     21H                     ;
;
        JMP     INIT                    ;GOTO RECORDING LOOP
;
SAME_VOICE:
        MOV     AH,0                    ;SET DISPLAY MODE (40*25)
        MOV     AL,0                    ;
        INT     10H                     ;
        MOV     DX,OFFSET MSG5          ;DISPLAY MSG5 ON SCREEN
        MOV     AH,9H                   ;
        INT     21H                     ;
;
INIT:
        MOV     SI,0                    ;INITIALIZE BUFFER POINTER
;
        MOV     AH,0                    ;SET AL = 0
        INT     16H                     ;GET A CHARACTER
;
AGAIN:
;
;START A/D CONVERSION
;
        MOV     DX,0250H                ;NO KEY PRESSED => START RECORDING
        MOV     AL,01000000B            ;A/D=CONVERTING MODE,S/H=HOLD MODE
        OUT     DX,AL                   ;CONTROL FROM PORT 250H
;
;DELAY 100uS FOR CONVERTING TIME
;
        MOV     CX,21D                  ;DELAY FOR 100 uS
DELAY:  NOP                             ;
        LOOP    DELAY                   ;
;
;GET DIGITIZED DATA
;
        MOV     DX,0250H                ;
        MOV     AL,10100000B            ;A/D=OUTPUT MODE,S/H=SAMPLE MODE
        OUT     DX,AL                   ;CONTROL FROM PORT 250H
;
        MOV     DX,0251H                ;READ IN DATA FROM PORT 251H
        IN      AL,DX                   ;
        MOV     BUFFER[SI],AL           ;STORE DATA INTO MEMORY
        INC     SI                      ;INCREASE DATA POINTER
        CMP     SI,16000D               ;BUFFER FULL?
        JA      VOICE_ANALYSIS          ;IF BUFFER IS FULL, ANALYSIS VOICE
;
;20uS DELAY FOR SAMPLING ON S/H CKT
;
        MOV     CX,03D                  ;DELAY FOR 20uS
SAMPLE: NOP                             ;
        LOOP    SAMPLE                  ;
;
        JMP     AGAIN                   ;
;
VOICE_ANALYSIS:
;
        MOV     SI,POINTER              ;LOAD POINTER OF VOICE BUFFERR
        MOV     AL,BUFFER[64]           ;LOAD THE 64 BYTE TO AL
        MOV     VOICE_B[SI],AL          ;LOAD THE 64 BYTE TO VOICE BUFFER
        INC     POINTER                 ;INCREASE POINTER
;
        MOV     SI,16000D               ;SET MAX MEMORY SIZE
        MOV     CX,16000D               ;
CLEAR:  MOV     BUFFER[SI],0H           ;CLEAR BUFFER
        DEC     SI                      ;DECREASE SI
        LOOP    CLEAR
;
        DEC     TIME                    ;DECREASE TIME
        JNZ     SAME_VOICE              ;RETRAIN THE SAME VOICE
;
        MOV     AX,0                    ;CLEAR AX
        MOV     AL,O_TIME               ;STORE O_TIME TO AX
        MOV     TIME,AL                 ;REFRESH TIME
;
        DEC     VOICE                   ;DECREASE VOICE
        JNZ     TEMP_ANOTHER_VOICE      ;TRAIN AONTHER NEW VOICE
        JMP     CONTINUE1               ;
TEMP_ANOTHER_VOICE:
        JMP     ANOTHER_VOICE           ;

CONTINUE1:
;
;*********************************************************************
;                                                                    *
;       VOICE INPUT PHASE -- GET A VOICE (COMMAND) FROM              *
;                                     EXTERNAL ENVIRONMENT           *
;                                                                    *
;*********************************************************************
;
CONTROL_PHASE:
;
        MOV     AH,0                    ;SET DISPLAY MODE (40*25)
        MOV     AL,0                    ;
        INT     10H                     ;
        MOV     DX,OFFSET MSG6          ;DISPLAY MSG6 ON SCREEN
        MOV     AH,9H                   ;
        INT     21H                     ;
;
        MOV     AH,0                    ;SETUP READ KEYBOARD
        INT     16H                     ;READ KEYBOARD INPUT
        CMP     AL,E_KEY                ;E_KEY => EXIT PROGRAM
        JE      TEMP_EXIT               ;IF E_KEY IS PRESSED ,EXIT PROGRAM
        JMP     CONTINUE2               ;
TEMP_EXIT:
        JMP     EXIT
CONTINUE2:
;
        MOV     SI,0                    ;SET SI TO 0
;
;GET A VOICE
;
GET_A_VOICE:
;
;START A/D CONVERSION
;
        MOV     DX,0250H                ;NO KEY PRESSED => START RECORDING
        MOV     AL,01000000B            ;A/D=CONVERTING MODE,S/H=HOLD MODE
        OUT     DX,AL                   ;CONTROL FROM PORT 250H
;
;DELAY 100uS FOR CONVERTING TIME
;
        MOV     CX,21D                  ;DELAY FOR 100 uS
DELAY_C:NOP                             ;
        LOOP    DELAY_C                 ;
;
;GET DIGITIZED DATA
;
        MOV     DX,0250H                ;
        MOV     AL,10100000B            ;A/D=OUTPUT MODE,S/H=SAMPLE MODE
        OUT     DX,AL                   ;CONTROL FROM PORT 250H
;
        MOV     DX,0251H                ;READ IN DATA FROM PORT 251H
        IN      AL,DX                   ;
        MOV     BUFFER[SI],AL           ;STORE DATA INTO MEMORY
        INC     SI                      ;INCREASE DATA POINTER
        CMP     SI,16000D               ;BUFFER FULL?
        JG      ANALYSIS_INPUT_VOICE    ;ANALYSIS INPUT VOICE
;
;20uS DELAY FOR SAMPLING ON S/H CKT
;
        MOV     CX,03D                  ;DELAY FOR 20uS
SAMPLE1:NOP                             ;
        LOOP    SAMPLE1                 ;
;
        JMP     GET_A_VOICE             ;
;
;SEARCH DATA FROM BUFFER
;
ANALYSIS_INPUT_VOICE:
;
```

```
        MOV     AL,BUFFER[64]           ;MOVE THE 64TH BYTE TO AL
        MOV     SOUND,AL                ;MOVE AL TO SOUND
;
;*********************************************************************
;                                                                    *
;       COMPARE PHASE --                                             *
;                                                                    *
;       COMPARE THE TRAINED VOICE WITH INPUT VOICE                   *
;                                                                    *
;*********************************************************************
;
        MOV     AX,0                    ;CLEAR AX
        MOV     AL,O_VOICE              ;COPY O_VOICE TO AL
        MOV     VOICE,AL                ;REFRESH VOICE
;
        MOV     AX,0                    ;CLEAR AX
        MOV     AL,O_TIME               ;COPY O_TIME TO AL
        MOV     TIME,AL                 ;REFRESH TIME
;
        MOV     SI,0                    ;CLEAR SI
;
NEXT_DATA:
        MOV     AL,VOICE_B[SI]          ;GET BUFFER DATA TO AL
        INC     SI                      ;INCREASE SI
        XOR     AL,SOUND                ;GENERATE THE NUMBER OF BIT 1
        MOV     SOUND,AL                ;RESTORE TO SOUND
        AND     AL,00000001B            ;GET THE LESS BIT
        MOV     TOTAL,AL                ;COPY TO TOTAL
;
        MOV     CX,0                    ;CLEAR CX
        MOV     CL,7                    ;SET SHIFT POINTER
;
NEXT_SHIFT:
        MOV     AX,0                    ;CLEAR AX
        MOV     AL,SOUND                ;COPY SOUND TO AL
        SHR     AL,CL                   ;RIGHT SHIFT CL BITS
        AND     AL,00000001B            ;GET LESS BIT
        ADD     TOTAL,AL                ;TOTAL=TOTAL+AL
        DEC     CL                      ;DECREASE CL
        JNZ     NEXT_SHIFT              ;IF POINTER <> 0 ,SHIFT AGAIN
;
        CMP     TOTAL,1H                ;COMPARE WITH 1
        JBE     OUTPUT_SIGINAL          ;IF ERROR LESS THAN ONE BIT
;
        DEC     TIME                    ;DECREASE TIME
        JNZ     NEXT_DATA               ;IF TIME <> 0,COMPARE WITH NEXT DATA
;
        MOV     AX,0                    ;CLEAR AX
        MOV     AL,O_TIME               ;COPY O_TIME TO AL
        MOV     TIME,AL                 ;REFRESH TIME
;
        DEC     VOICE                   ;DECREASE VOICE
        JNZ     NEXT_DATA               ;IF VOICE <> 0,COMPARE WITH NEXT
DATA
```

```
;
        MOV     AH,0                    ;SET DISPLAY MODE (40*25)
        MOV     AL,0                    ;
        INT     10H                     ;
        MOV     DX,OFFSET MSG7          ;DISPLAY MSG7 ON SCREEN
        MOV     AH,9H                   ;
        INT     21H                     ;
        JMP     CONTROL_PHASE           ;GOTO MENU
;
;*********************************************************************
;                                                                    *
;       SIGINAL OUTPUT PHASE --                                      *
;       OUTPUT THE CONTROL SIGINAL TO HANDLE EXTERNAL DEVICES        *
;                                                                    *
;*********************************************************************
;
OUTPUT_SIGINAL:
        MOV     CX,0                    ;CLEAR CX
        MOV     CL,O_SOUND              ;COPY O_SOUND TO CL
        SUB     CL,SOUND                ;GET WHICH VOICE IS MATCHED
;
        MOV     AX,0                    ;CLEAR AX
        MOV     AL,00000001B            ;SET AL = 1
        SHL     AL,CL                   ;GET THE POWER CL OF AL
;
        MOV     DX,0257H                ;SET OUTPUT PORT = 257H
        OUT     DX,AL                   ;OUTPUT AL TO PORT 257H
;
        JMP     CONTROL_PHASE           ;GOTO MENU
;
EXIT:
        MOV     AH,0                    ;SET DISPLAY MODE (40*25)
        MOV     AL,0                    ;
        INT     10H                     ;
        MOV     DX,OFFSET MSG8          ;DISPLAY MSG8 ON SCREEN
        MOV     AH,9H                   ;
        INT     21H                     ;
;
        MOV     CX,0FFFFH               ;NONSENCE LOOP
NONE:   NOP                             ;
        NOP                             ;
        NOP                             ;
        LOOP    NONE                    ;
;
        MOV     AH,0                    ;
        MOV     AL,2                    ;
        INT     10H                     ;SET DISPLAY MODE (80*25)
        RET                             ;RETURN TO DOS
;
START   ENDP
CODE    ENDS
        END     START
```

VLSI Neurocomputing Hardware for Matrix Algebra Using N64000 Chips

John Vranich
047-66-6692
EE599 Final Project

Srinivas S. Reddy

Vinai Kolli

St-Id 888-03-8813

Abstract

In this project I investigated the implementation of the matrix algebra algorithms presented by the Wang and Mendel [1] on a commercially available neural network chip. The neurocomputing hardware is shown to fit on a single board that can be connected to a host computer bus; interface software routines running on the host allow application programs to access the hardware through a subroutine call. A system performance evaluation, carried out in conjunction with my project partners Srinivas Reddy and Vinai Kolli, showed that the neurocomputing hardware compares favorably to a serial computer running the same algorithm. Finally, the primary shortcoming of the selected approach, namely a restriction to integer math, is overcome with proposed enhancements to the neurocomputer architecture.

Discussion

The target configuration of our neurocomputing hardware was a single-board coprocessor that could be connected to the bus of a host computer. The X1 chip from Adaptive Solutions [2] (to be marketed by Inova Microelectronics [3] as the N64000) was chosen as a processor since it is a fully digital network implementation and it has the capability of on-chip learning. The digital implementation reduces the amount of external hardware required to interface the chip to a host computer and also provides more accuracy than analog approaches; the on-chip learning capability is required by the matrix algebra algorithms.

Hardware

The hardware required, in addition to the X1 chip, to build a functioning system consists of: 1) bus interface logic, 2) local memory to hold the input and output matrices, 3) a microsequencer to run the matrix algebra algorithms and provide instructions to the X1 chip, 4) program memory storage for both the X1 chip and the microsequencer, and 5) specialized logic to support the unique features of the X1 chip and the vector operations common to matrix/neurocomputing algorithms. Figure 1 shows a system block diagram. The key features of the hardware design are detailed below.

The program memory on the board is partitioned into read-only and read/write sections for maximum hardware flexibility. The read-only memory contains general-purpose routines which control the interface to the host computer and the transfer of algorithms into the read/write memory from the host computer. In this way, the host computer can configure the hardware to perform almost any algorithm by downloading a new set of instructions to the read/write program memory.

A second important element of the design is the logic which supports the serial and parallel buses on the X1 chip. The parallel unit is needed to convert back and forth between 16-bit and 8-bit data so that the X1 chip can communicate with the local memory; it is also used to feed-back the X1 output bus onto its input bus during certain phases of the learning algorithm. The serial unit is required to generate commands and status messages that can be passed from one processor to the next within the X1 chip and between X1 chips if more than one is used at a time.

Finally, the zero-overhead loop hardware is an extension to the basic sequencer which allows it to efficiently perform vector operations. The hardware can be programmed with a loop parameter (which is essentially a vector length) and then a sequence of one or more micro-operations can be automatically performed for the specified number of times without the need for explicit compare and branch instructions. In the short loops found in the matrix algebra microcode this can result in a 2-to-1 speed improvement. Figure 2 summarizes the operation of the zero-overhead loop.

The total amount of hardware required to implement the system is approximately 25 chips. The total board area is estimated to be 35 square inches. This count assumes that Programmable Gate Array chips can be used to implement the majority of the logic needed for the functions described above. Table 1 summarizes the memory requirements of the system. Table 2 details the chip count.

Software

Only a few short software routines are needed to allow a user program to make use of the neurocomputing hardware. The routines would typically be stored in a library that the user links into his application code at compile time. One routine is used to configure the hardware with parameters of the algorithm to be run such as learning rate, error bounds, and matrix size. A second routine is used to download the user's matrix to the hardware and return the processed results from the hardware. A third routine, which would be transparent to the user, would download new algorithms to the hardware based on the operation that the user wishes to perform. Figure 3 shows the flowchart for the host computer download routine.

While these routines are short in static code size, they prove to be bottlenecks when examined from a run-time perspective. Because the X1 chip can only operate on integer data, the host software interface routines must convert the user's data, which will typically be in a floating-point format, to a fixed-point format for transmission to the hardware. Similarly, the results returned from the hardware must typically be converted back to floating point format for use by the user's application. Both of these operations are O(N2) and therefore time consuming. Major performance increases could be realized if the X1 chip were enhanced to allow floating-point processing.

Summary of Team Effort

The matrix LU decomposition algorithm was studied in detail by our project team. The algorithm was mapped onto the X1 chip and clock cycles for executing a single pass of the learning algorithm were counted. A formula relating clock cycles to matrix dimensions was derived, and using this formula and an implementation of the algorithm running on a Sun workstation we were able to calculate estimated execution times for our neurocomputing hardware. Figure 4 shows the LU decomposition algorithm developed for the hardware, and Figure 5 shows the microcode written for the X1 chip that allows it to execute the algorithm. Figures 6a and 6b show a graph of the error versus training iteration for various matrix sizes. Table 3 shows a breakdown of the average number of training cycles spent per vector for various matrix sizes. Table 4 shows a time comparison of the algorithm executed on our hardware versus the algorithm executing on a Sun workstation.

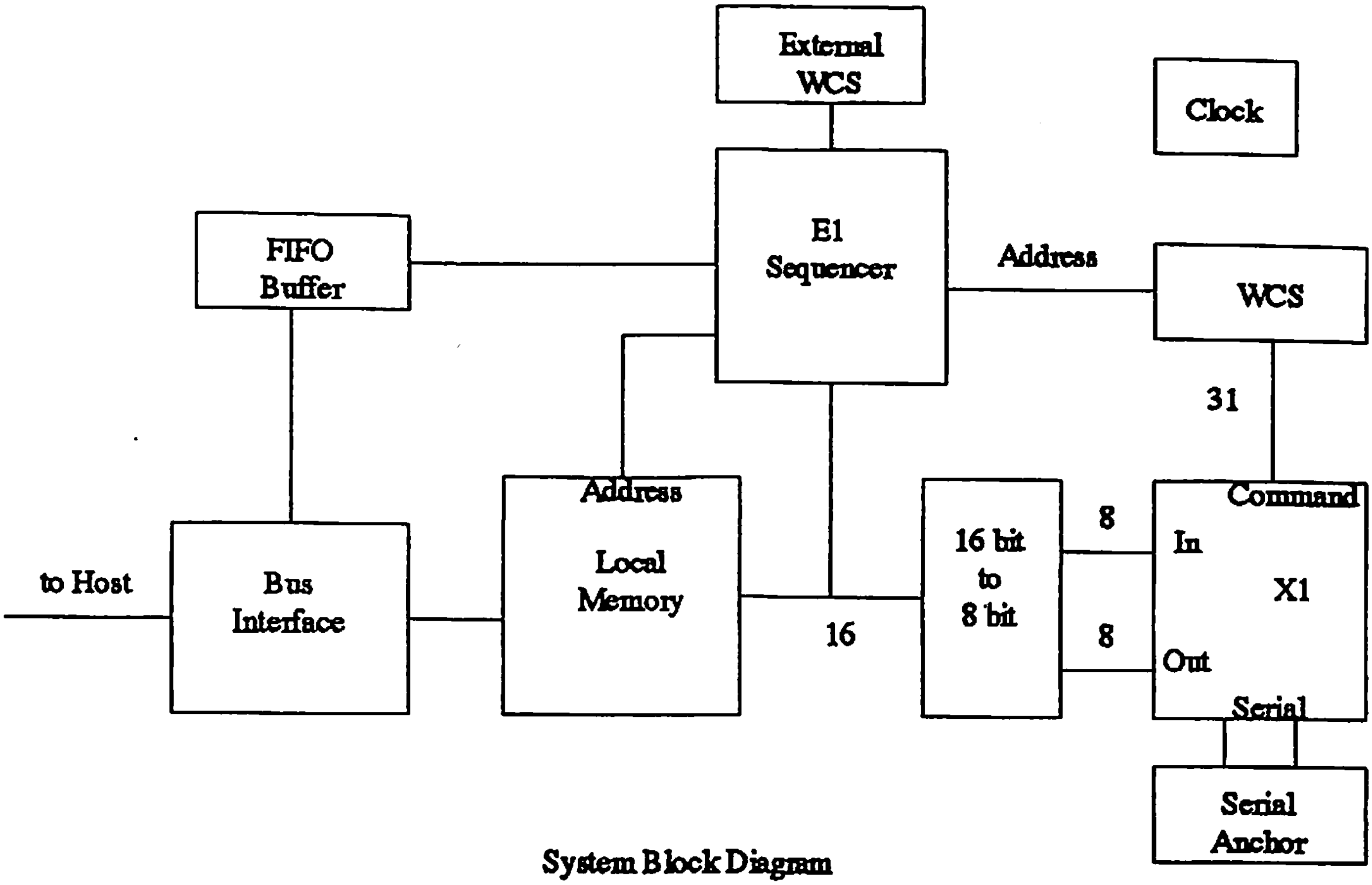
A few important conclusions can be drawn from our findings: First, the current hardware solution is less than ideal because of the lack of accuracy in the integer math and the extra time spent to convert from floating-point representations to fixed-point representations. We estimate that floating-point units would be twice the size of the current fixed-point units, thereby reducing the number of processing elements per chip by half. Second, the training time rises sharply for matrices larger than 7 by 7, and the algorithm did not even converge to a solution for 8 by 8 and larger matrices. This may be due to our starting with poorly conditioned matrices as test cases -- all of our matrices were generated using a pseudo-random number generator. The cause of the non-convergence must be explored further before the neurocomputing algorithm for decomposing matrices can be considered useful. Third, the bus interconnection between processing elements inside the X1 chip proves to be less than ideal when performing the input-layer weight updating in the back-propagation algorithm. This portion of the algorithm runs in O(N2) time on the chip while all other portions of the algorithm are completed in O(N) time.

References

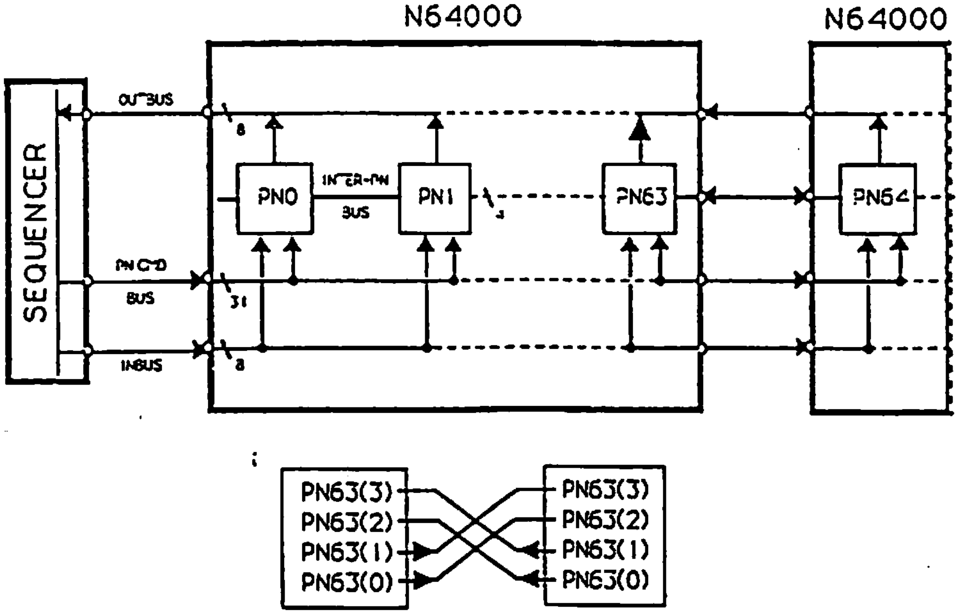
1. Wang, L. and Mendel, J.M. "Structured Trainable Networks for Matrix Algebra", SIPI, USC.

2. Hammerstrom, D. "A VLSI Architecture for High-Performance, Low-Cost, On-chip Learning", 1990 IJCNN.

3. Inova Microelectronics, N64000 data sheet, "Digital Neural Network Processor."



System Block Diagram



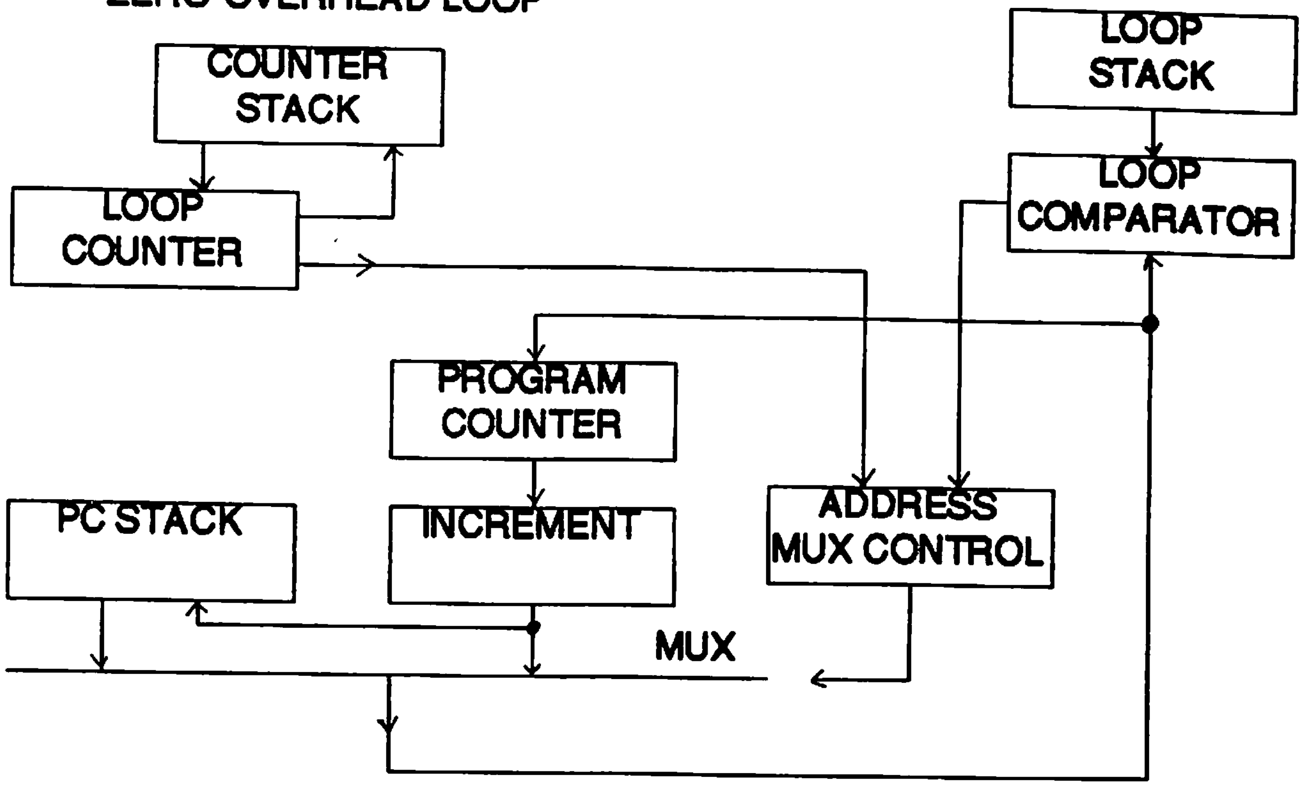
INTER-PN BUS COMMUNICATION

Multiple N64000s Arranged in Parallel

FIGURE 1

HARDWARE DESCRIPTION (CONTINUED)

- ZERO-OVERHEAD LOOP



HARDWARE DESCRIPTION (CONTINUED)

- ZERO-OVERHEAD LOOP

	MICROINSTRUCTION	DESCRIPTION
	LOAD LOOP COUNTER	
	DO label UNTIL CNT==0	PC STACK ‹ - next LOOP STACK ‹ - label
next:	*ANY MICROINSTRUCTION*	
	...	
label:	*ANY MICROINSTRUCTION*	CNT ‹ - CNT - 1 CNT == 0 PC ‹ - PC + 1 CNT != 0 PC ‹ - PC STACK

FIGURE 2

HOST COMPUTER S/W FLOWCHART

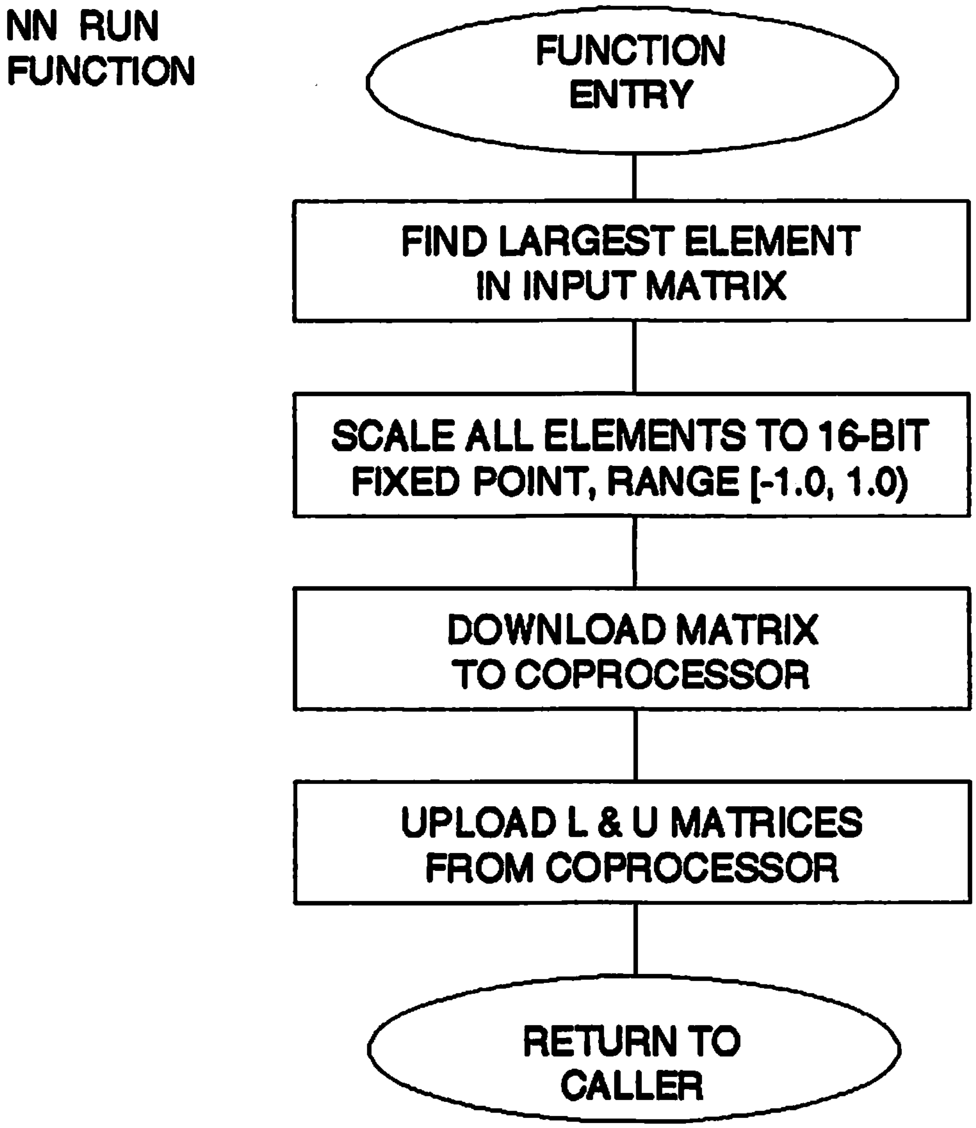


FIGURE 3

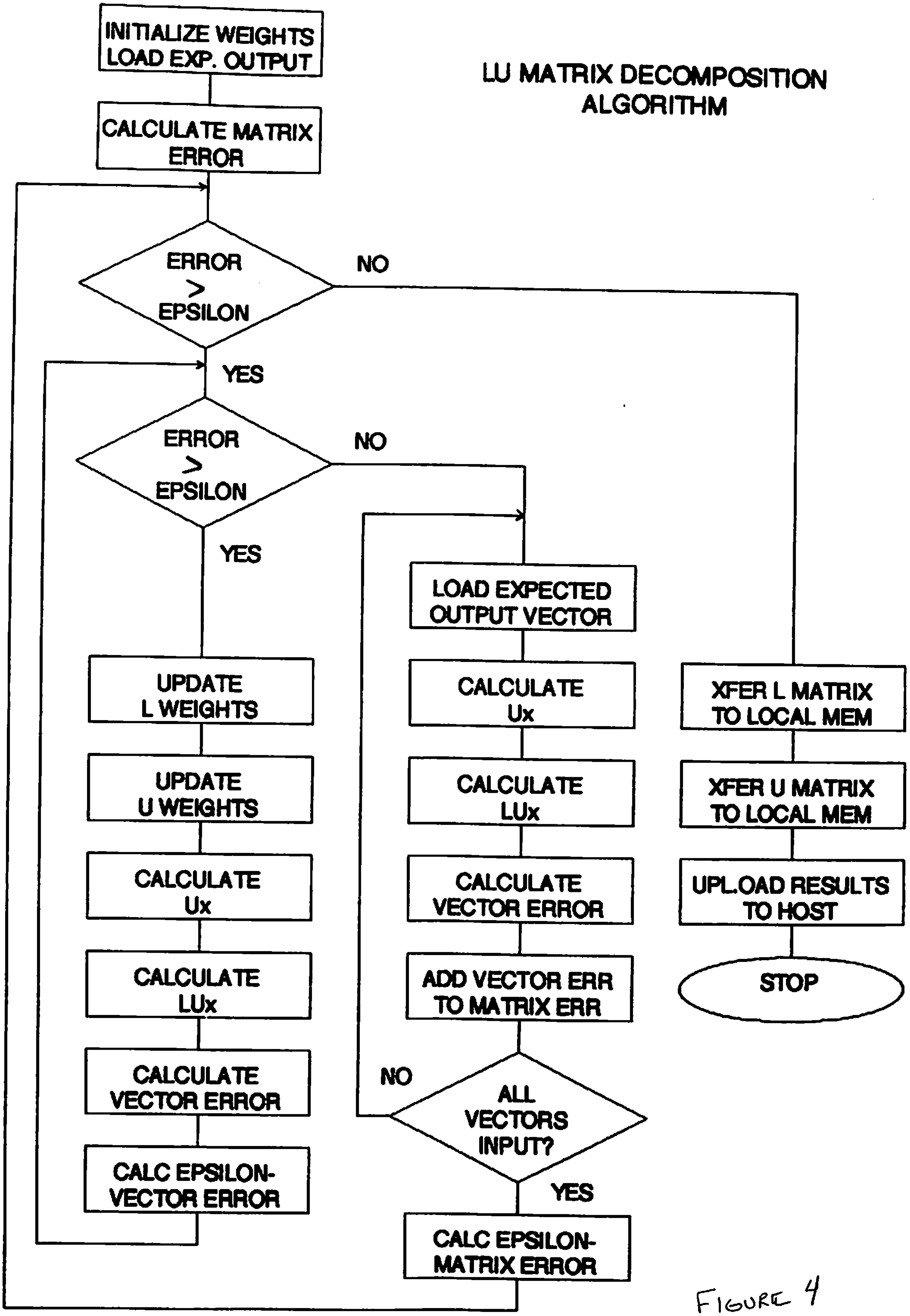

LU MATRIX DECOMPOSITION
ALGORITHM
INITIALIZE WEIGHTS
LOAD EXP. OUTPUT
CALCULATE MATRIX
ERROR
ERROR
>
EPSILON
NO
YES
ERROR
>
EPSILON
NO
YES
UPDATE
L WEIGHTS
UPDATE
U WEIGHTS
CALCULATE
Ux
CALCULATE
LUx
CALCULATE
VECTOR ERROR
CALC EPSILON-
VECTOR ERROR
LOAD EXPECTED
OUTPUT VECTOR
CALCULATE
Ux
CALCULATE
LUx
CALCULATE
VECTOR ERROR
ADD VECTOR ERR
TO MATRIX ERR
NO
ALL
VECTORS
INPUT?
YES
CALC EPSILON-
MATRIX ERROR
XFER L MATRIX
TO LOCAL MEM
XFER U MATRIX
TO LOCAL MEM
UPLOAD RESULTS
TO HOST
STOP

FIGURE 4

LU Decomposition Backpropagation Propagation MicroCode:

X1 chip assembly language description

Register description

regA ACC accumulator
regB hidden layer output activation
regC expected output
regD error terms (expected - actual)
regE alpha (learning rate)
regF Sum [alpha*(d-y)*l(t+1)]
regG matrix error (sum of regD over all input vectors)
regH epsilon 1 (vector error target)
regI epsilon 2 (matrix error target)
WA weight memory address register
WM weight memory modulus register
PROD multiplier product register

processor control instructions

DISABLE ALL disables all PNs
DISABLE # disables only PN#
ENABLE ALL enables all PNs
ENABLE # enables only PN#
PASS EN TOKEN passes enable token to next processor
PASS OE TOKEN passes output enable token to next processor

data movement instructions

FETCH reg <- bus loads specified register from specified bus
LOAD reg <- M[addr_reg] loads specified register from memory location pointed to by addr_reg
MOVE dreg <- sreg moves source register (sreg) to destination
OUTPUT OUTBUS <- sreg place contents of source register on OUTBUS
STORE M[addr_reg] <- reg stores contents of register in memory location pointed to by addr_reg

computation instructions

ADD ACC <- ACC + reg add contents of register to accumulator
SUB ACC <- ACC - reg subtract contents of register from accumulator
MAC ACC <- ACC + src1*src2 add product of src1 and src2 to accumulator
MULT PROD <- src1 * src2 compute product of src1 and src2 and store result in PROD register
INC reg increment the register, valid only for ACC and WA
CLR ACC clear the accumulator

FIGURE 5

Load Expected Output function

```
PN0 <- ENABLE TOKEN
      for X= 1 to #(rows-1)
          FETCH regC <- INBUS, PASS ENABLE TOKEN
      end
```

Matrix Output Routine

```
/* Matrices are output by column, L matrix then U matrix */
 ENABLE ALL, PN0<- OE TOKEN;                              ;L Matrix Start Address
 FETCH WA <- INBUS
    for 1 to # (rows-1)
          for 1 to # (columns -1)
             OUTPUT: OUTBUS <- M[WA], INC WA
           end
         PASS OE TOKEN
     end

     FETCH WA <- INBUS, PASS OE TOKEN                     ;U Matrix Start Address
    for 1 to #(rows-1)
        for 1 to #(columns-1)
          OUTPUT: OUTBUS <- M[WA], INC WA                 ;Each Processor stores a ;
                                                          ;row of the weight matrix
        end
      PASS OE TOKEN
    end
```

Matrix Error Calculation

```
FETCH regG <- INBUS;                                      ;clear Reg G
    for 1 to # (columns-1)                                ;(for all input patterns)
          call FORWARD_PROP                               ;Calculates Ux,LUx,  .
                                                          ; &Vec
                                                          ;tor Error
          ADD ACC <- ACC + regG
           STORE regG <- ACC                              ;regG=Matrix Error
     end
SUB ACC <- regI - ACC                                     ;regI=Epsilon2
OUTPUT OUTBUS <- ACC
```

L matrix updating

```
ENABLE ALL, PN0 <- OE TOKEN
FETCH WA <- INBUS                                         ; set address to beginning
                                                          ; of L matrix
```

```
    for1 to #( rows-1)
         LOAD ACC <- M[WA]                              ;load old weight l(t)
          OUTPUT OUTBUS <- regB                         ;output the activation
          FETCH BBUS <- INBUS, MULT PROD <- BBUS * regD
          MAC ACC <- ACC + PROD * regE, PASS OE TOKEN
         STORE M[WA] <- ACC, PASS ENABLE TOKEN
   end
```

U matrix weight update

```
ENABLE ALL, PN0 <- OE TOKEN;
FETCH WA<-INBUS, PASS OE TOKEN, CLR ACC
    for 1 to #( rows -1)
         MULT PROD <- regD * M[WA], INC WA
           for 1 to # valid output layer synapse connections
               OUTPUT OUTBUS <- PROD, PASS OE TOKEN
               FETCH BBUS <- INBUS, MAC ACC <- ACC + regE*BBUS
           end
STORE regF <- ACC, PASS ENABLE TOKEN;
    end

/* all nodes now have Sum[ alpha*(d-y)*l(t+1) ] */

ENABLE ALL, FETCH WA <- INBUS, PN0 <- ENABLE TOKEN   ;WA=U Matrix
                                                     ; start address;
    for 1 to # (columns-1)
         LOAD ACC <- M[WA], INC WA;
          DISABLE ALL,
         FETCH BBUS <- INBUS, MAC ACC <- ACC +BBUS*regF      ;INBUS=IN
                                                             ;PUT Vector
          STORE M[WA] <- ACC, ENABLE ALL, PASS EN TOKEN      ;Enable next
                                                             ;PE in addi
                                                             ;tion to those
                                                             ;already En
                                                             ;abled
    end
```

Routine to load L

```
DISABLE ALL
PN0 <-ENABLE TOKEN
   for X=1 to# (Columns-1)
         FETCH WA <-INBUS                        ;WA=Weight address;
         PASS ENABLE TOKEN
   end
ENABLE ALL                                       ;( PN_Command bus)
FETCH weight_modulus <-INBUS                     ;weight_modulus=# rows of
                                                 ; matrix
```

```
STORE M[WA]<-INBUS
     for X=0 to # (Columns-2)
          STORE M[WA]<-INBUS, WA<-WA+INCR;
     end
```

Routine to load U matrix

```
FETCH WA<- INBUS                                  ;Load inital Address in
                                                  ;Weight Memory
     for X=1 to # (Columns-1)
          STORE M[WA]<-INBUS, WA=WA+INCR          ;;Assume modulo addressing
     end
```

Routine to calculate Ux in the forward propagation

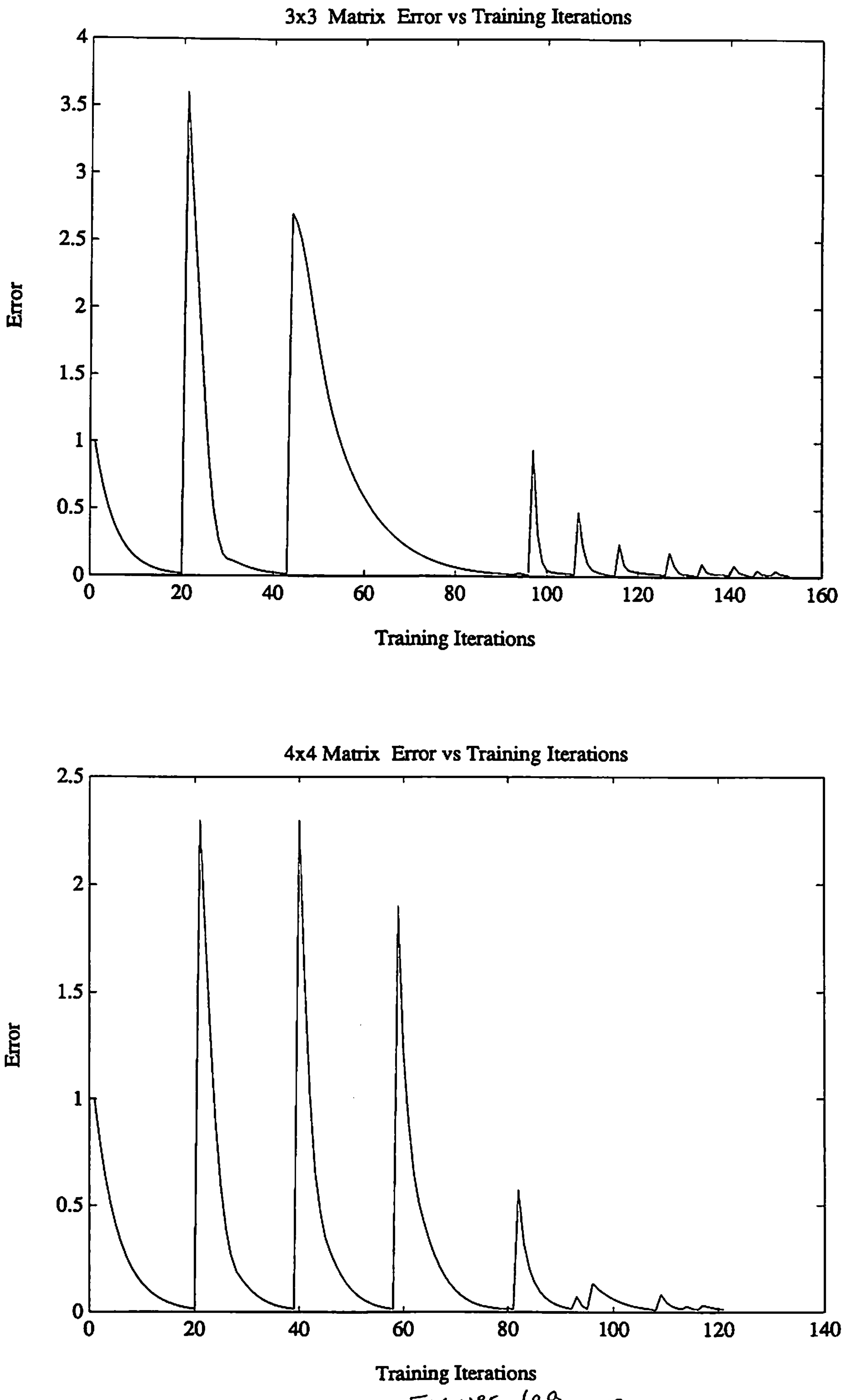
```
FETCH WA <-INBUS(U matrix start address), ACC<-0;
     for X= 1 to # (Columns-1)
          FETCH BBUS<-INBUS, MAC: ACC<-(BBUSxM[WA])+ACC;
     end
regB<-ACC
```

Routine to calculate LUx matrix

```
ACC<-0
FETCH WA <-INBUS                                  ;L matrix start address
PN0<-OE TOKEN
     for 1 to # (columns-1)
        OUTPUT: OUTBUS<-regB, PASS OE TOKEN;
        FETCH BBUS<-INBUS, MAC: ACC<-ACC + (BBUS *M[WA]);
     end
ADD ACC<- expected value(regC)+ACTUAL             ;ACC<-(dk+yk)(where actual
                                                  is A[-x])
ACC<-absolute error_val(ACC)
regD<-ACC                                         ;sstore each nodes error D<-
                                                   dk-yk(
```

Sum the Vector Error

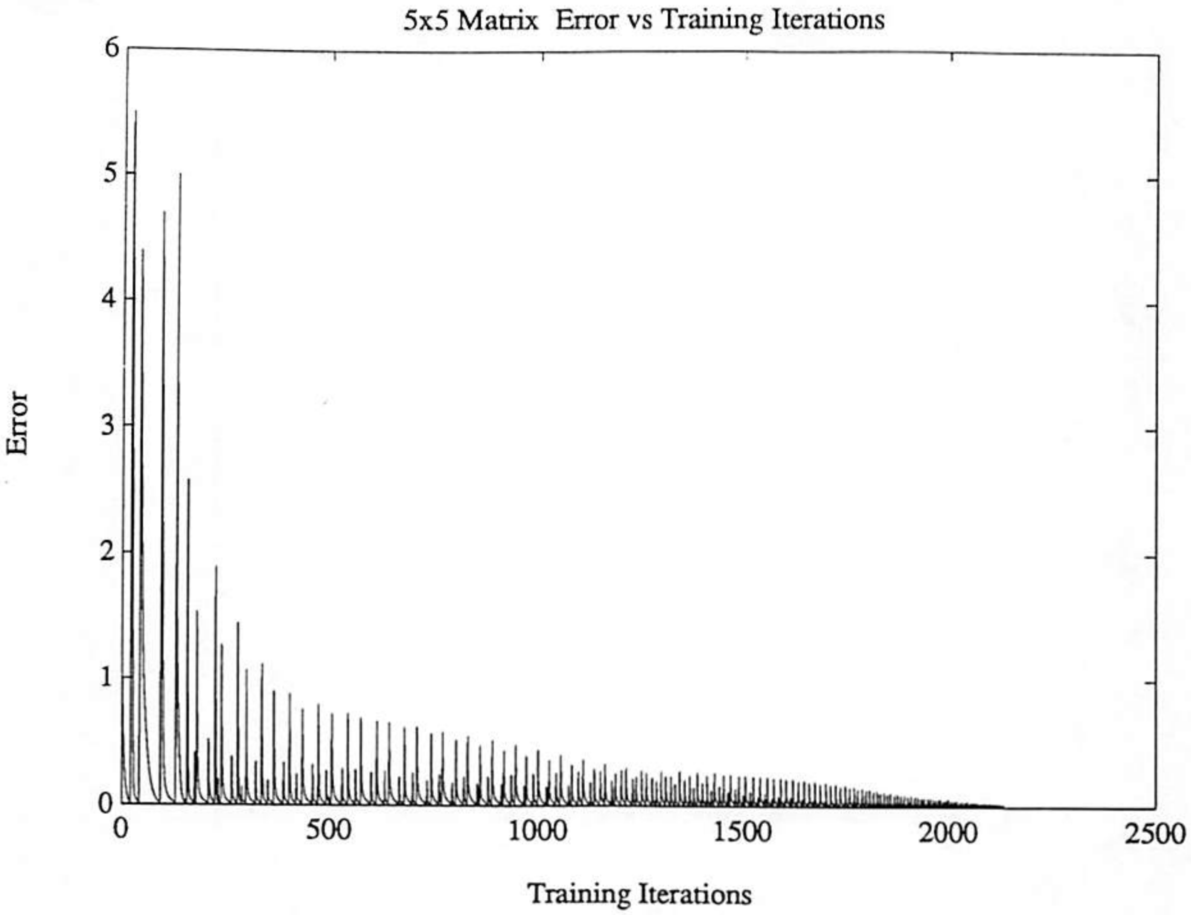
```
DISABLE TOKEN TO ALL, PN0<-OE TOKEN
ENABLE PN0,PN1<- OE TOKEN
     for 1 to # Columns
          OUTPUT OUTBUS<- regD, PASS OE TOKEN
          FETCH BBUS<-INBUS, ADD ACC<-ACC+ BBUS
     end
SUB epsilon1 - ACC, PN0<-OE TOKEN
OUTPUT: OUTBUS<-ACC
```



3x3 Matrix Error vs Training Iterations

4x4 Matrix Error vs Training Iterations

FIGURE 6a


5x5 Matrix Error vs Training Iterations
Error
Training Iterations
0
1
2
3
4
5
6
0
500
1000
1500
2000
2500

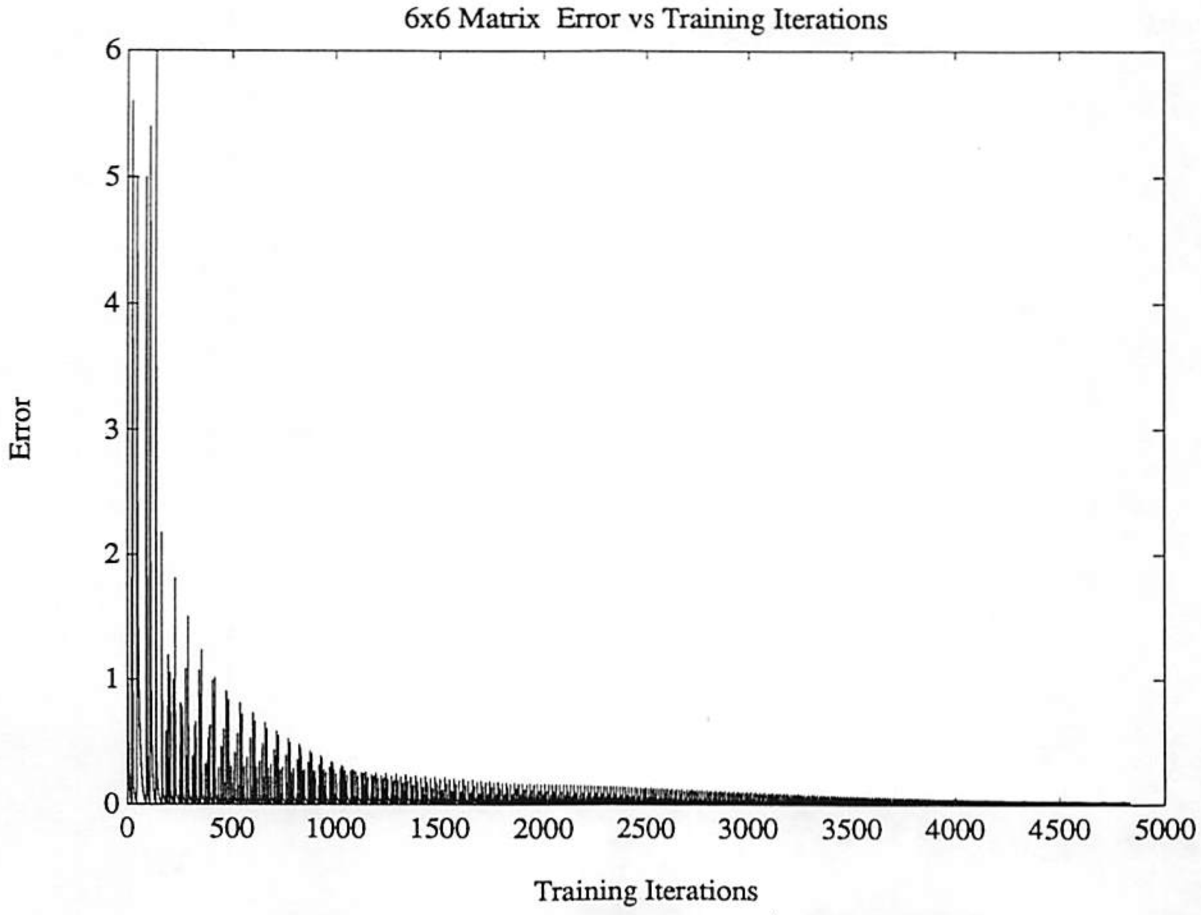

6x6 Matrix Error vs Training Iterations
Error
Training Iterations
0
1
2
3
4
5
6
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000

FIGURE 6b

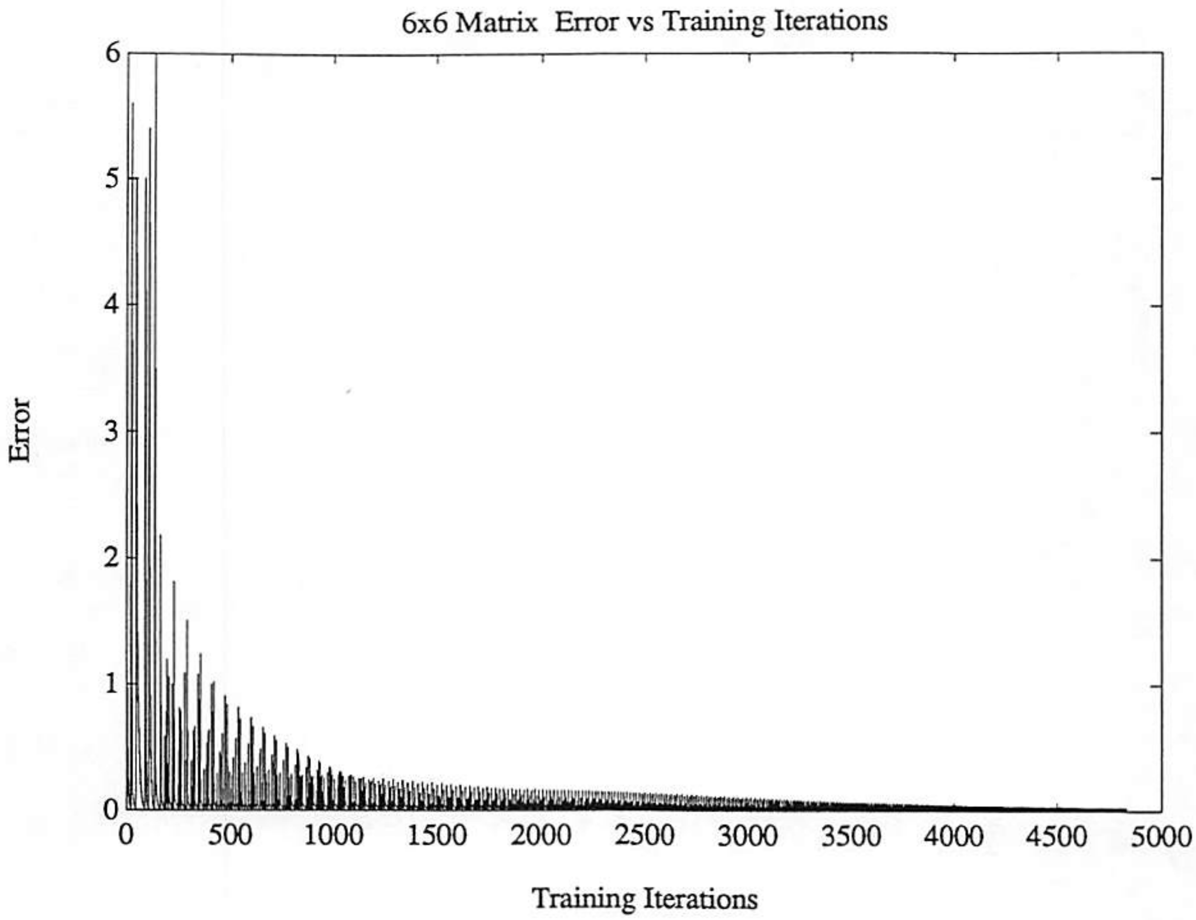

6x6 Matrix Error vs Training Iterations
Error
0
1
2
3
4
5
6
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
Training Iterations

TABLE 1

HARDWARE DESCRIPTION (CONTINUED)

- MEMORY REQUIREMENTS VS MATRIX SIZE

MATRIX SIZE	LOCAL MEMORY
64 x 64	16 KBYTES
128 x 128	64 KBYTES
256 x 256*	256 KBYTES
512 x 512	1024 KBYTES
1024 x 1024	4096 KBYTES

* X1 ON-CHIP 16-BIT WEIGHT MEMORY LIMIT REACHED

TABLE 2

- CHIP COUNT (APPROX. 35 IN^2)
  - 10 1-MBYTE (128K x 8) STATIC RAM CHIPS
  - 2 PROGRAMMABLE GATE ARRAY CHIPS (3K GATES EACH)
  - 2 X1/E1 PROCESSOR/SEQUENCER CHIPSET
  - 2 BUS INTERFACE
  - 3 OCTAL TRANSCEIVERS (HOST INTERFACE)
  - 4 8Kx8 PROM CHIPS

Table 3

Training Cycles per Vector

Matrix Size	Epochs	Patterns	Avg. Cycles to Train a Pattern
4x4	92	860	9.3
5x5	75	714	9.5
6x6	966	3857	4.0
7x7	275	1645	5.9
8x8	189	5057	26.7 *
8x8	823	35297	42.9 *
8x8	727	21514	29.6 *

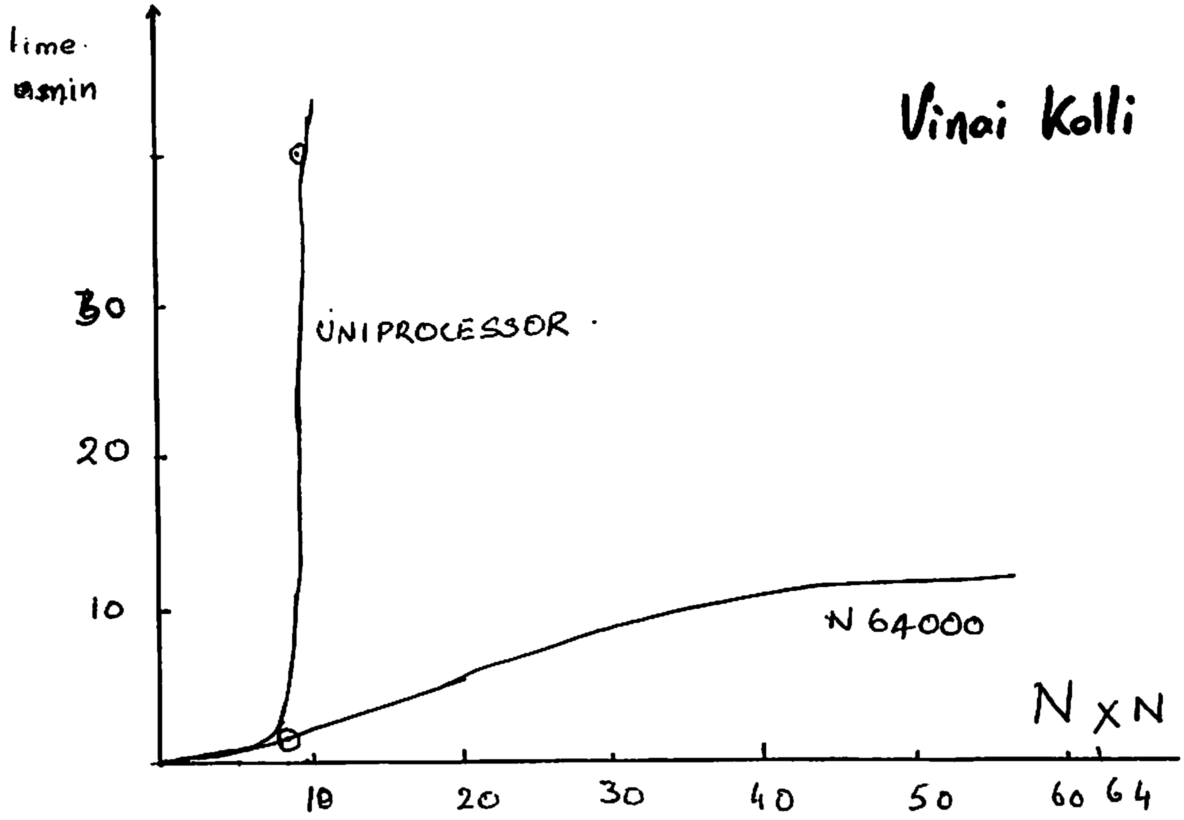
* Did not converge

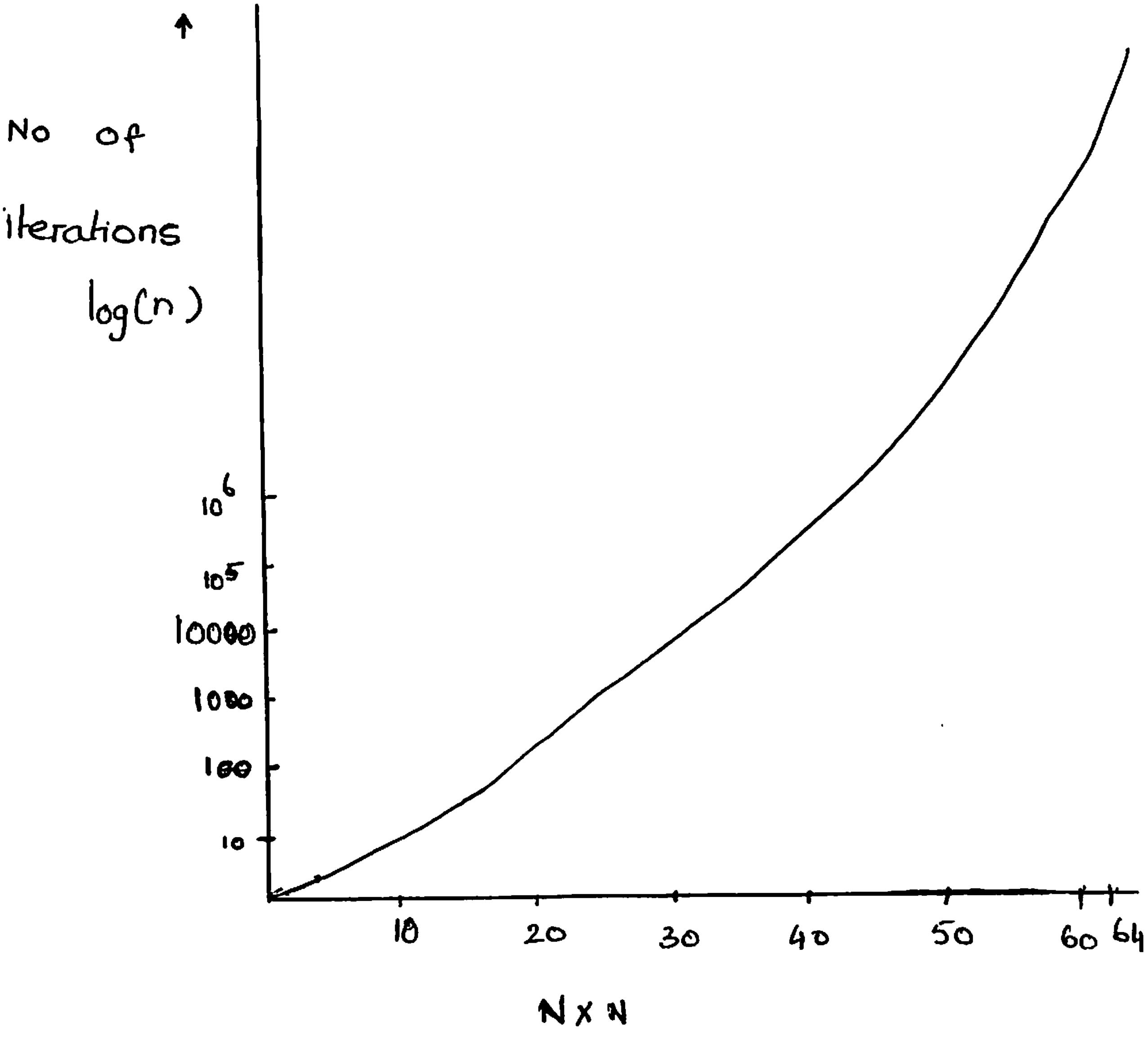
COMPARISON TABLE

N X N	no of iterations	Sparc	clock cycles in N64000 41 +17*N+10N(N+1)/2	time on N64000	speed up
4 x 4	121	1sec	32549	1.3ms	769
5 x 5	2136	2sec	62500	2.5ms	800
6 x 6	4837	20sec	475000	19ms	1029
7 x 7	68762	1.48min	2875000	75ms	1169
8 x 8	2219875	41 min	5.1756 E 7	2.075sec	1185

TABLE4

Vinai Kolli


time
min
30
20
10
UNIPROCESSOR
N 64000
N x N
10
20
30
40
50
60
64


No of
iterations
log(n)
10^6
10^5
10000
1000
100
10
10
20
30
40
50
60
64
N x N

```
/* ma.c   John Vranich  047-66-6692
 * EE 599 Hwk 4
 * April 1, 1991
*/
/* matrix algebra LU decomposition using neural network, reference "Structured
 * Trainable Networks for Matrix Algebra" by Wang and Mendel, SIPI, USC
*/

#include <stdio.h>
#include <math.h>
#include "ma.h"

#define  SIZE   7
#define  INPUTS SIZE                    /* number of input nodes in network    */
#define  OUTPUTS SIZE                   /* number of output nodes in network   */
#define  PATTERNS SIZE                  /* number of training patterns         */
#define  INFILE  "ma.data"              /* data file containing input matrix   */
#define  ERR_FILE "pattern.error"       /* magnitude of pattern error          */

void     calc_LU(double x[INPUTS][INPUTS]);

int      inputs=INPUTS;
int      outputs=OUTPUTS;
int      patterns=PATTERNS;

double   activation[INPUTS];            /* activation of input layer nodes     */
double   zactivation[INPUTS];           /* activation of hidden layer nodes    */
double   yactivation[INPUTS];           /* activation of output layer nodes    */
double   error[OUTPUTS];                /* output layer errors                 */
double   zerror[OUTPUTS];               /* hidden layer errors                 */
double   lweight[INPUTS][INPUTS];       /* weight[to][from], Lower diagonal Matrix*/
double   uweight[INPUTS][INPUTS];       /* weight[to][from], Upper diagonal Matrix*/
double   target[OUTPUTS];               /* target outputs for current pattern  */
double   out[PATTERNS][OUTPUTS];        /* training outputs                    */
int      first_lweight_to[INPUTS];      /* describes connections               */
int      last_lweight_to[INPUTS];       /* describes connections               */
int      first_uweight_to[INPUTS];      /* describes connections               */
int      last_uweight_to[INPUTS];       /* describes connections               */

double   alpha=0.20;                    /* learning rate parameter             */
double   pattern_epsilon=0.015;         /* pattern error limit, i.e. epsilon1  */
double   epoch_epsilon=0.06;            /* epoch error limit, i.e. epsilon2    */
unsigned long tr_count;                 /* count of training iterations        */
unsigned long epoch_count;              /* count of training epochs            */

void main()
{
   int   done=0;
   int   c;

   init_network();
   while(!done)
   {
      display_menu();
      scanf("%d",&c);
      switch (c)
      {
         case 0: train();            break;
         case 1: display_results();  break;
         case 2: display_weights();  break;
         case 3: init_network();     break;
         case 4: set_alpha();        break;
         case 5: set_pat_epsilon();    break;
         case 6: set_epoch_epsilon();  break;
         case 9: done=1;               break;
         default:                      break;
      }
   }
}


void  train()
{
   int      i,j;
   double   pattern_error, epoch_error;
   FILE     *err_file;        /* file of error magnitudes during training     */

   if (NULL == (err_file = fopen(ERR_FILE,"w")))
    {
       puts("\nunable to open error output file\n");
       exit(0);
    }

   i=0;
   tr_count=0;
   epoch_count=0;
   do                         /* train until epoch_error bound is reached     */
   {
      set_activation(i);      /* set input activation and output target for   */
      set_target(i);          /* pattern i                                    */
      do             /* train on pattern until pattern_error bound is reached */
      {
         compute_output();
         pattern_error = compute_error();
         update_weights();
         tr_count++;
         fprintf(err_file,"%e\n",pattern_error);
      } while (pattern_error > pattern_epsilon);

      epoch_error = 0.0;
      for (j=0; j<patterns; j++) /* check epoch error after training on each  */
      {                          /* pattern                                   */
         set_activation(j);
         set_target(j);
         compute_output();
         epoch_error += compute_error();
      }

      i = (i+1) % patterns;      /* go to next pattern   */
      epoch_count++;
   } while (epoch_error > epoch_epsilon && epoch_count < 50000);
   fclose(err_file);
}

void  set_activation(int i)
{
   int j;
   for (j=0; j<inputs; j++)
      activation[j]=(j==i)? 1.0: 0.0;
}

void  set_target(int i)
{
   int j;
   for (j=0; j<outputs; j++)
      target[j]=out[i][j];
}

void  compute_output()
{
   int i,j;

   for (i=0; i<inputs; i++)                                 /* hidden layer   */
   {
      zactivation[i] = 0.0;
      for (j=first_uweight_to[i]; j<last_uweight_to[i]; j++)
         zactivation[i] += activation[j] * uweight[i][j];
   }

   for (i=0; i<inputs; i++)                                 /* output layer   */
   {
      yactivation[i] = 0.0;
      for (j=first_lweight_to[i]; j<last_lweight_to[i]; j++)
         yactivation[i] += zactivation[j] * lweight[i][j];
   }
}

double   compute_error()
{
   int    j;
   double output_error=0.0;

   for (j=0; j<outputs; j++)  /* compute all output nodes */
   {
      error[j] = target[j] - yactivation[j];
      zerror[j] = error[j];            /* set zerror because of unity weight  */
      output_error += fabs(error[j]);  /* connection along l-matrix diagonal  */
   }
   return (output_error);
}

void  update_weights()
{
   int   i,j;
/* note that hidden layers' initial error is set equal to output layers'   */
/* error in compute_error() because of constant unity weight on diagonal   */

   for (i=0; i<outputs; i++)                                   /* output layer */
      for (j=first_lweight_to[i]; j<last_lweight_to[i]; j++)
         if (i!=j)   /* don't update weights on diagonal of l-matrix */
         {
            lweight[i][j] += zactivation[j] * error[i] * alpha;
            zerror[j] += error[i] * lweight[i][j]; /* add this node's portion to */
         }                                         /* hidden layer error total   */

   for (i=0; i<inputs; i++)                                    /* hidden layer */
      for (j=first_uweight_to[i]; j<last_uweight_to[i]; j++)
         uweight[i][j] += activation[j] * zerror[i] * alpha;
}

void  init_network()                   /* initialize network weights and      */
{                                      /* target training patterns            */
   FILE *fp;
   if (NULL==(fp=fopen(INFILE,"r")))
   {
      printf("\nUnable to open input file %s\n",INFILE);
      exit(0);
   }
   init_weights();
   init_connects();
   init_train_patterns(fp);
   fclose(fp);
}

void  init_train_patterns(FILE *fp)    /* only need to set target patterns    */
{                                 /* input patterns are set algorithmically    */
   int   i,j;

   for (i=0; i<inputs; i++)            /* target patterns are the columns of  */
      for (j=0; j<inputs; j++)         /* the matrix to be decomposed when    */
         fscanf(fp,"%lf",&out[j][i]);  /* the input vectors are chosen to be  */
                                       /* [1 0 0...0] [0 1 0...0] [0 0 1...0] */
}

void  init_connects()
{
   int   i;

/* initialize network interconnection pointers assuming no connection where   */
/* the weights are identically zero.  Note that the Lower triangular matrix   */
/* weights that are constant unity are included in the connection description.*/
/* nodes are numbered on each layer from left to right, starting at zero      */
/* The arrays are indexed by the current node number.  The arrays contain the */
/* node number of the first or last node on the layer below the current one   */
/* that is connected to the current node.                                     */

   for (i=0; i<outputs; i++)                   /* output layer, L-matrix */
   {
      first_lweight_to[i]=0;
      last_lweight_to[i] =i+1;                 /* actual node number + 1 */
   }
   for (i=0; i<outputs; i++)                   /* hidden layer, U-matrix */
   {
      first_uweight_to[i]=i;
      last_uweight_to[i] =inputs;              /* actual node number + 1 */
   }
}

void  init_weights()
{
   int   i,j;

   for (i=0; i<inputs; i++)         /* initialize both to identity matrix  */
      for (j=0; j<inputs; j++)
      {
         lweight[i][j] = (i==j) ? 1.0 : 0.0;
         uweight[i][j] = 0.0;
      }
}

void  display_results()
{
   int   i,j;

   printf("\n");
   for (i=0; i<patterns; i++)
   {
      for (j=0; j<inputs; j++)
      {
         printf("%d ", (i==j) ? 1   : 0);
         activation[j]=(i==j) ? 1.0 : 0.0;
      }
      printf("\t");
      compute_output();
      for (j=0; j<outputs; j++)
         printf("%6.3lf ",yactivation[j]);
      printf("\n");
   }
   printf("\nepochs: %ld\tpatterns: %ld\n",epoch_count,tr_count);
}

void  display_weights()
{
   int   i,j;
   double LU[INPUTS][INPUTS];

   printf("\nL matrix:\n");
   for (i=0; i<outputs; i++)
   {
      for (j=0; j<outputs; j++)
         printf("%6.3lf ",lweight[i][j]);
      printf("\n");
   }

   printf("\n\nU matrix:\n");
   for (i=0; i<outputs; i++)
   {
      for (j=0; j<outputs; j++)
         printf("%6.3lf ",uweight[i][j]);
      printf("\n");
   }

   calc_LU(LU);
   printf("\n\nLU matrix:\n");
   for (i=0; i<outputs; i++)
   {
      for (j=0; j<outputs; j++)
         printf("%6.3lf ",LU[i][j]);
      printf("\n");
   }
   printf("\n");
}

void  calc_LU(double LU[INPUTS][INPUTS])  /* calculates matrix product of     */
{                                         /* lower and upper diagonal matrices*/
   int   i,j,k;

   for (i=0; i<INPUTS; i++)
      for (j=0; j<INPUTS; j++)
      {
         LU[i][j] = 0.0;
         for (k=0; k<INPUTS; k++)
            LU[i][j] += lweight[i][k] * uweight[k][j];
      }
}

void  set_alpha()                               /* learning rate parameter    */
{
   printf("\nCurrent alpha: %3.2lf  Enter new alpha > ",alpha);
   scanf("%lf",&alpha);
```

```
}

void  set_pat_epsilon()                          /* each pattern's error bound */
{
   printf("\nCurrent pattern epsilon: %4.3lf  Enter new epsilon > ",pattern_epsilon);
   scanf("%lf",&pattern_epsilon);
}

void  set_epoch_epsilon()      /* error bound for full set of training patterns */
{
   printf("\nCurrent epoch epsilon: %4,3lf  Enter new epsilon > ",epoch_epsilon);
   scanf("%lf",&epoch_epsilon);
}

void  display_menu()
{
   printf("\n(0) train, (1) results, (2) weights, (3) init, (9) quit > ");
}
```

EE599/4 Term Project April 29,1991

Title : System Integration of Structured Trainable Networks for Matrix Algebra (LU Decomposition)

Name : Joseph C. Wong

Student Number : 557-19-6066

Remote : TRW

Abstract

This project describes a hardware implementation of the structured networks for LU decomposition. The neural VLSI chip used in this design is the Adaptive Solutions / Inova Neural Network Execution Engine. A single chip, which contains 64 processor nodes, is used to solve problems of 32x32 matrix size. The hardware is implemented as a microsequencer based design. This provides the flexibility to adapt the processor broad for other matrix problems. Computation speed of O(n) is observed when compared to the single CPU O(n^2).

1. Introduction

In the paper "Structured Trainable Networks for Matrix Algebra", L. Wang and J. Mendel proposed a novel approach in solving matrix algebra problems. The approach calls for the use of structured networks. By training the network with a particular algorithm and rule, solutions to the matrix problem can be obtained. The training algorithm used in this approach is the error back-propagation algorithm used in neural networks. This project will propose a hardware implementation of the network for solving LU decomposition.

2. Selection of available neural chips for the implementation

Tradeoffs study was made to select the best compromise in available VLSI chips. Since the primary use of this hardware is for solving matrix problems, a digital approach would have an advantage over an analog neural chip. The reason is two fold. First, most, if not all, matrix problems are presented in numbers, digitally. Using an analog neural chip would involve both D-to-A and A-to-D conversions. Second, higher accuracy can be achieved and maintained with the digital format. Now that the chip is digital, the second issue has to do with the speed of the hardware. Since most matrix calculations involve vector arithmetic, a neural chip with vector like capability, such as multiply and accumulate, would be advantageous. Size and power is also of consideration. A high density chip with multiple processing elements would fare better. After trading of pros and cons with the available neural chips, the Adaptive Solutions / Inova neural network execution engine is chosen.

3. Adaptive Solutions / Inova Neural Network Execution Engine

This chip is a 11-million device, 25MHz processor capable of performing 1.6 billion connections/sec. The processor array is composed of 64 processor nodes (PNs) plus 16 spare PNs for redundancy. The chip uses a SIMD architecture with the 64 PNs linear array. The PNs are connected with an 8-bit input bus, an 8-bit output bus and a 31-bit command bus (PNCMD). Each PN has two 16-bit internal bus. The reason for the two internal buses is to facilitate single clock multiplication. Each PN has a

9x16 flash multiplier with a 16-bit CSA to create a 24-bit product. The 24-bit product can be fed into the 32-bit adder for an effective single clock cycle multiply and accumulate. Each PN contains 4K bytes of weight memory which can be configured into 2K words (16-bit wide). It also contains a weight address generator and a 32-word 16-bit register file. Since this is an execution engine, an external controller / sequencer is needed to coordinate the operation of the chip.

4. System Architecture of Matrix Computation hardware

The hardware has to be supported by a host computer. A 16-bit data bus is assumed. All registers on the board are memory mapped to the host computer. Data is downloaded to the board before execution. Driver program which is not yet developed should be interrupt-driven. This will allow the processor board to process data without tying up the host computer. Upon completion, the processor board will interrupt the host computer and awaits further instruction.

The processor board, Fig.2, is a microsequencer based machine utilizing the neural network execution engine as it main processing element. A microprogram will control the sequence of the operation of the engine and provide control of all datapath on the board. Apart from the neural network execution engine, there is a set of register buffers. These buffers are used to store the X vector (Xi), the desired outputs (di), the delta values (i.e. the difference between desired output and actual output) and some intermediate results (Zi). Since the host computer can support 16-bit data bus and the execution engine I/O bus is only 8-bit wide, 16-to-8 bit translators and 8-to-16 bit translators, Fig.4, are used to interface between board resources and the execution engine. To provide flexibility to the design, the error thresholds ($\varepsilon 1$ and $\varepsilon 2$) are to be specified by the user. An error threshold circuit, Fig.4, will keep track of the delta values (di-yi). It will halt further training if the error threshold criteria is met. This is done by an interrupt signal sent to the host computer.

Each PN in the execution engine contains 4K bytes (2K words) of weight memory. Since LU decomposition is a 3 layer network, 2 sets of weights are required, each should be 1K word big. This translates to a maximum matrix size of 32x32 for a single chip. The output layer and the hidden layer are implemented with the same PN. In other the words, the two layers of summation are performed with the same PN at different times. This arrangement, although seems to be slower in throughput, is actually more efficient since less communication in weight values is required.

5. Operation Sequence

The error back-propagation algorithm is implemented on this hardware.

Calculation of Zi's and yi's

$$Zi = \Sigma\, Uij * Xj$$
$$yi = \Sigma\, lij * Zj$$

Weight update

$$lij(t+1) = lij(t) + \alpha\ (di - yi)\ Zj$$
$$uij(t+1) = uij(t) + \alpha\ [\ \Sigma\ (dk\text{-}yk)\ lki(t+1)]\ Xj$$

These computations are implemented with loop sequences for the microsequencer. There are 5 basic loop sequences. Two sequences are used to calculate Zi and yi. The third sequence calculates the delta values (di-yi) for both error threshold and weight update. The last two sequences are for updating uij and lij with error back-propagation. The algorithm for generating μcode for these five sequences is listed after Fig.1. The algorithms take advantage of the vector nature of the problem to achieve fast multiply and accumulate. A flowchart of the overall flow of the operation is shown in Fig. 1.

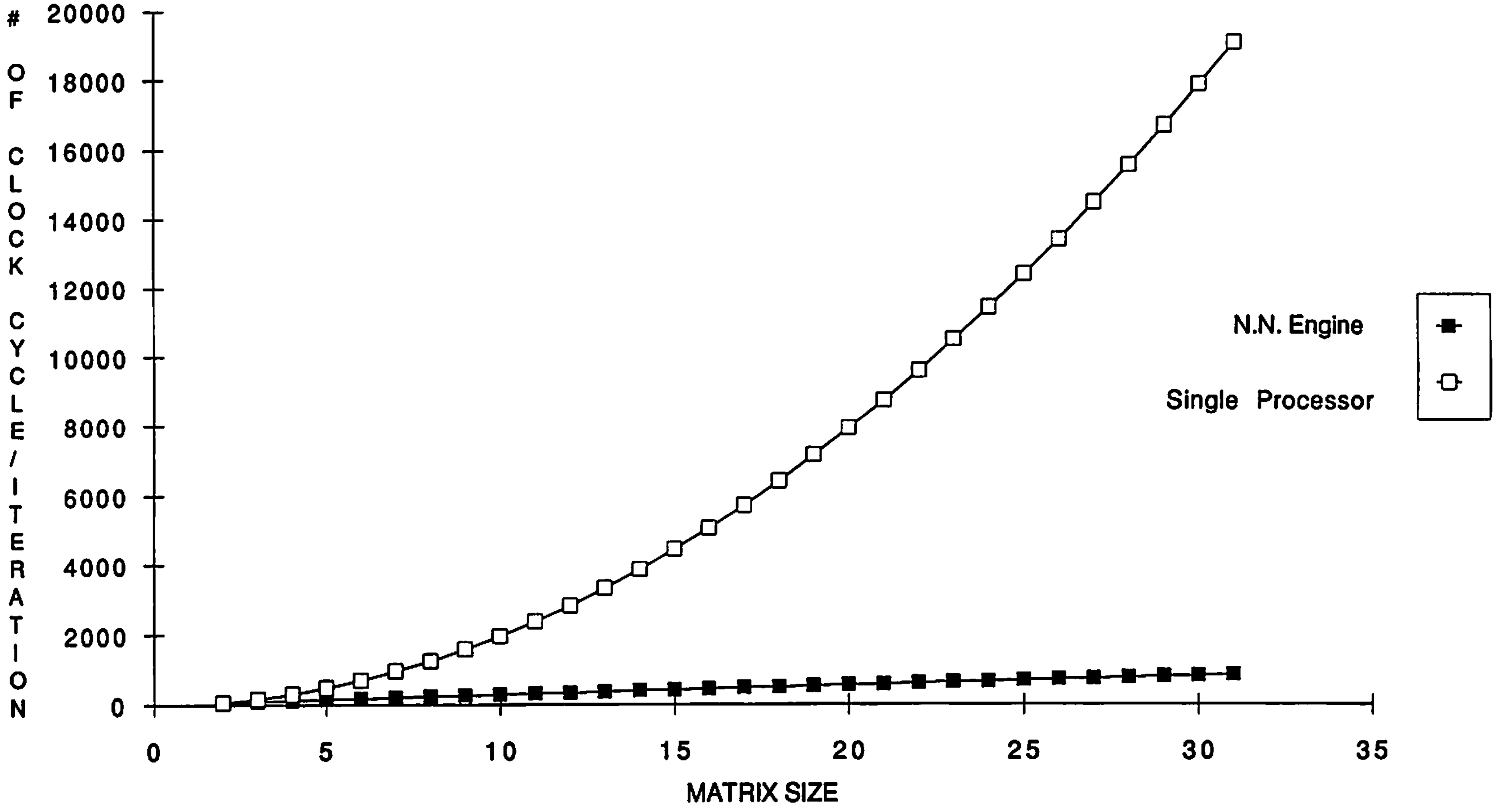
6. Results

The advantage of using neural network vs using single CPU computer to solve matrix problem was studied. Due to the vector nature of the matrix operation and the parallel processing capability of the neural network execution engine, the speed advantage would clearly favor the N.N. engine. Calculation time of a single training iteration is used as the basis of the comparison. It is found that the time needed by the N.N. engine is on the order of O(n) where n is the matrix size. However, it will take a single CPU computer $O(n^2)$ clock cycles per training iteration. The speed advantage of the neural network and parallel processing has been illustrated.

7. Conclusion

The hardware implementation of the structured networks with N.N. execution engine is shown in this project. Because of the use of a microsequencer in controlling the neural engine, this hardware can be used in different applications just by replacing the microcode sequence and the driver program. Most of the board hardware would remain intact. This will prove to be useful in an environment where a "generic" neural execution engine is needed to solve various problems. The only drawback to the Adaptive Solutions/Inova N.N. engine is the 8bit I/O buses. Eventhough the designer claims that 8 bit is sufficient in neural processing, it would definitely provide extra speed improvement if 16bit I/O bus is adopted. A change to 16 bit I/O bus would not be drastic to the chip since internal architecture is 16bit based already. Lastly, $O(n)$ computation speed of the N.N. engine vs the $O(n^2)$ speed of the CPU computer is evident.

CALCULATION SPEED OF NEURAL CHIP vs SINGLE PROCESSOR COMPUTER



COMPARISON OF CALCULATION SPEED FOR DIFFERENT MATRIX SIZE

(# OF CLOCK CYCLES PER ITERATION)

MATRIX SIZE	N.N. ENGINE	SINGLE CPU	MATRIX SIZE	N.N. ENGINE	SINGLE CPU
			17	489	5729
2	84	74	18	516	6426
3	111	171	19	543	7163
4	138	308	20	570	7940
5	165	485	21	597	8757
6	192	702	22	624	9614
7	219	959	23	651	10511
8	246	1256	24	678	11448
9	273	1593	25	705	12425
10	300	1970	26	732	13442
11	327	2387	27	759	14499
12	354	2844	28	786	15596
13	381	3341	29	813	16733
14	408	3878	30	840	17910
15	435	4455	31	867	19127
16	462	5072	32	894	20384

NOTE: In both cases

16x16 MULTIPLY = 2 clock cycles
ADD = 1 clock cycle
16 bit LOAD = 2 clock cycles

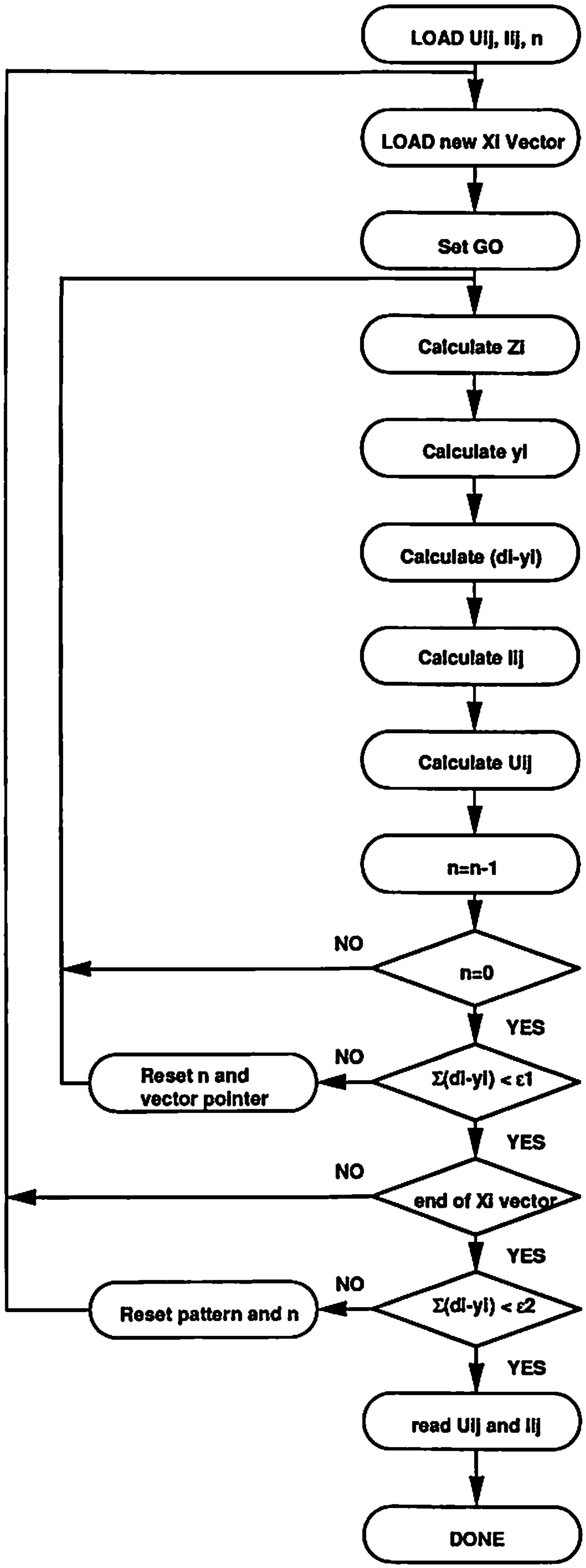


Fig. 1 OPERATION SEQUENCE

ALGORITHM FOR GENERATING MICROCODE SEQUENCE

```
    INBUS = $\alpha$                          /* broadcast $\alpha$
    PNReg1 = $\alpha$                         /* store $\alpha$

* Calculate Zi
    LOAD weight base addr
    LOAD weight offset addr
    LOAD loop count
    begin loop:
        INBUS = Xj                     /* load X
        PROD = INBUS *Uij              /* Xj*Uij
        SUM = SUM + PROD               /* $\Sigma$ (Xi + Uij)
    end loop;
    PNReg2 = SUM                       /* store Zi in intern. reg

* Calculate yi
    LOAD weight base addr
    LOAD weight offset addr
    LOAD loop count
    begin loop:
        OUTBUS = PNReg2j               /* output Zj
        Store Zj in ext. buffer        /* store Zj for later use
        INBUS = OUTBUS                 /* broadcast Zj to all PNs
        PROD = INBUS * lij             /* Zj * lij
        SUM = SUM + PROD               /* $\Sigma$ (Zj *lij)
    end loop;
    PNReg3= SUM                        /* store yi in intern. reg

* Calculate (di - yi)
    Set PN internal shift register for PNID operation
    LOAD weight base addr
    LOAD weight offset addr
    LOAD loop count
    begin loop:
        INBUS = di                     /* broadcast di
```

```
          If PNID matches
               SUM=INBUS - PNReg3          /* di-yi
               OUTBUS = SUM                /* store (di-yi) ext. buffer
               PROD = PNReg1 *SUM          /* α * (di-yi)
               PNReg4 = PROD               /* store result intern. reg
     end loop;

*Update lij's
     LOAD weight base addr
     LOAD weight offset addr
     LOAD loop count
     begin loop:
          INBUS = Zj                       /* broadcast Zj
          PROD = PNReg4 * INBUS            /* α(di-yi)Zj
          SUM = lij + PROD                 /* lij + α(di-yi)Zj
          lij = SUM                        /* update lij
     end loop;

* Update Uij's
     LOAD weight base addr
     LOAD weight offset addr
     LOAD loop count
     begin loop:
          LOAD base address
          INBUS = (di - yi)                /* broadcast di-yi
          PROD = INBUS * lij               /* (di-yi)*lij
          SUM = SUM + PROD                 /* Σ (di-yi)*lij
          Increment base address
     end loop;
          PROD = PNReg1 * SUM              /* α * Σ (di-yi)lij
          PNReg5 = PROD                    /* store for intern. use

     LOAD weight base addr
     LOAD weight offset addr
     LOAD loop count
     begin loop:
          INBUS = Xi                       /* broadcast Xi
          PROD = INBUS * PNReg5            /* α * Xi * Σ (di-yi)lij
          SUM = uij + PROD                 / * uij+α*Xi*Σ(di-yi)lij
          Uij = SUM                        /* update Uij
     end loop;
```

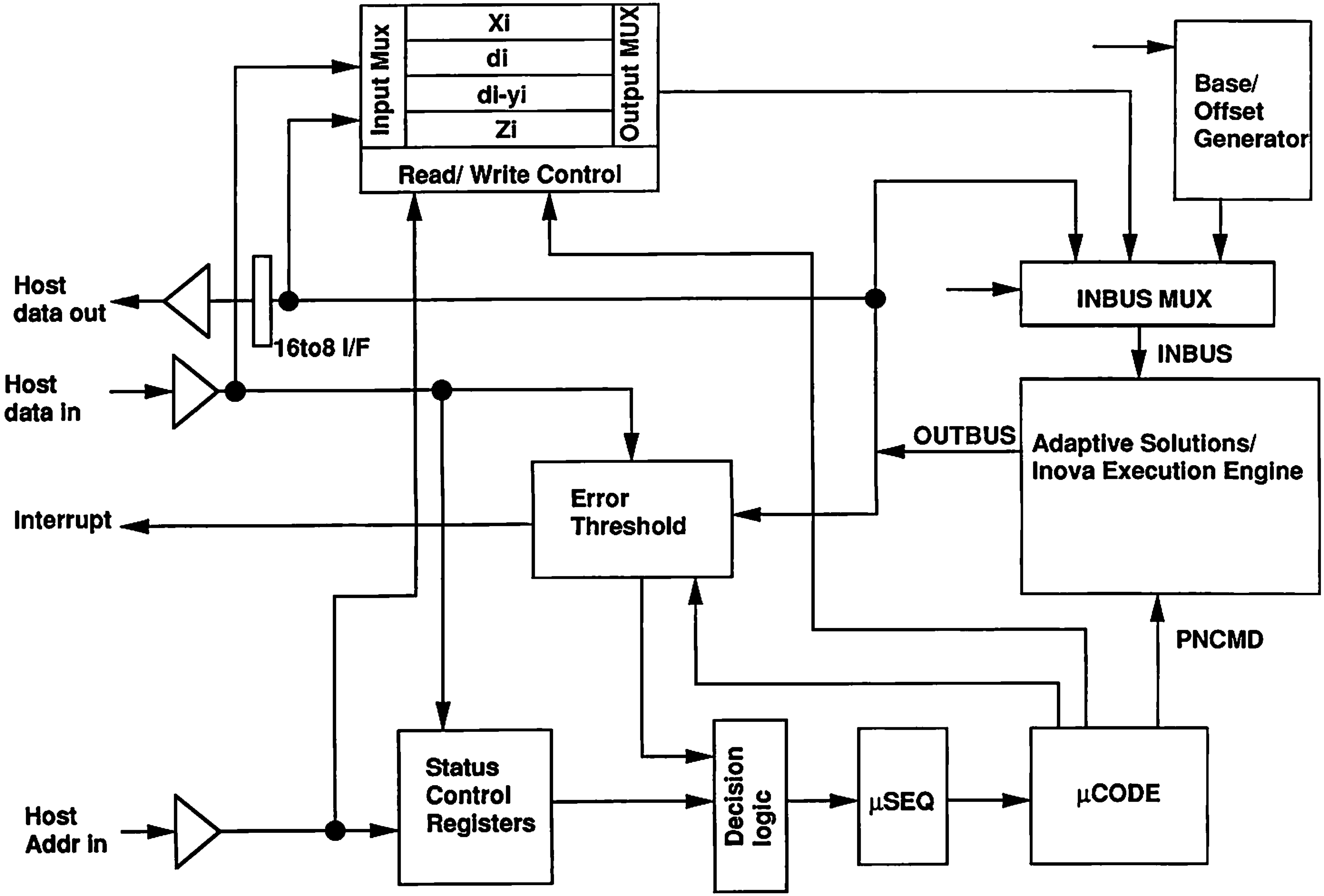
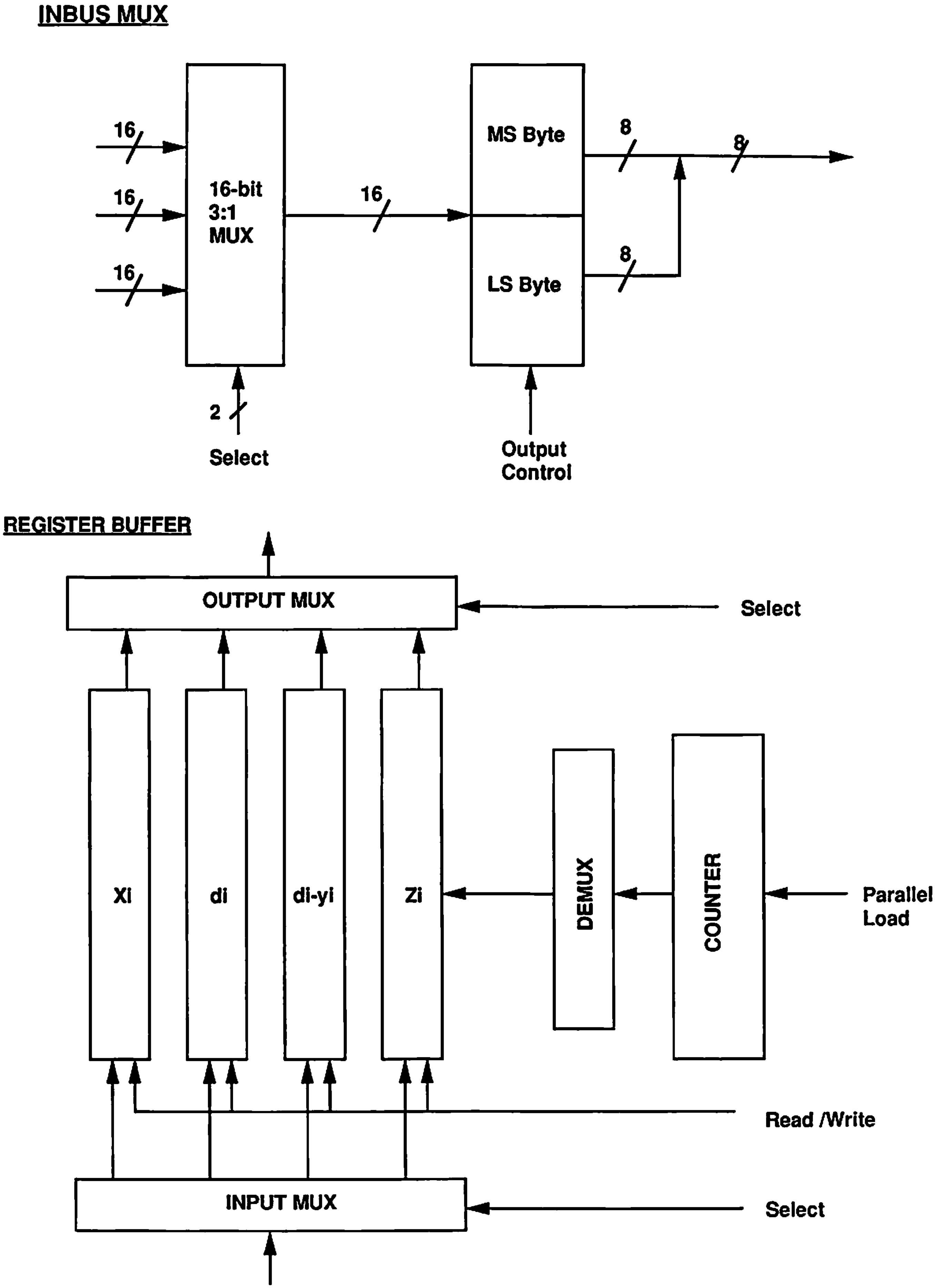


FIG.2 PROCESSOER BOARD BLOCK DIAGRAM

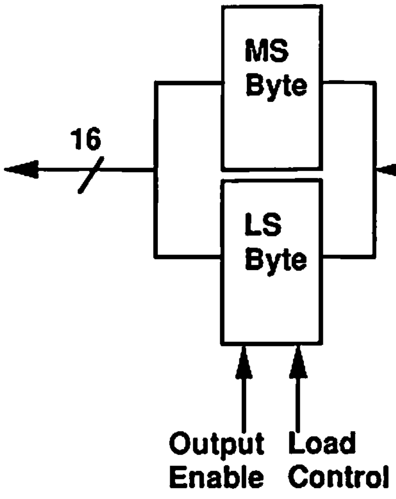


INBUS MUX

REGISTER BUFFER

Fig. 3 INBUS MUX and BOARD REGISTER BUFFER

8bit to 16bit TRANSLATOR



ERROR THRESHOLD CIRCUIT

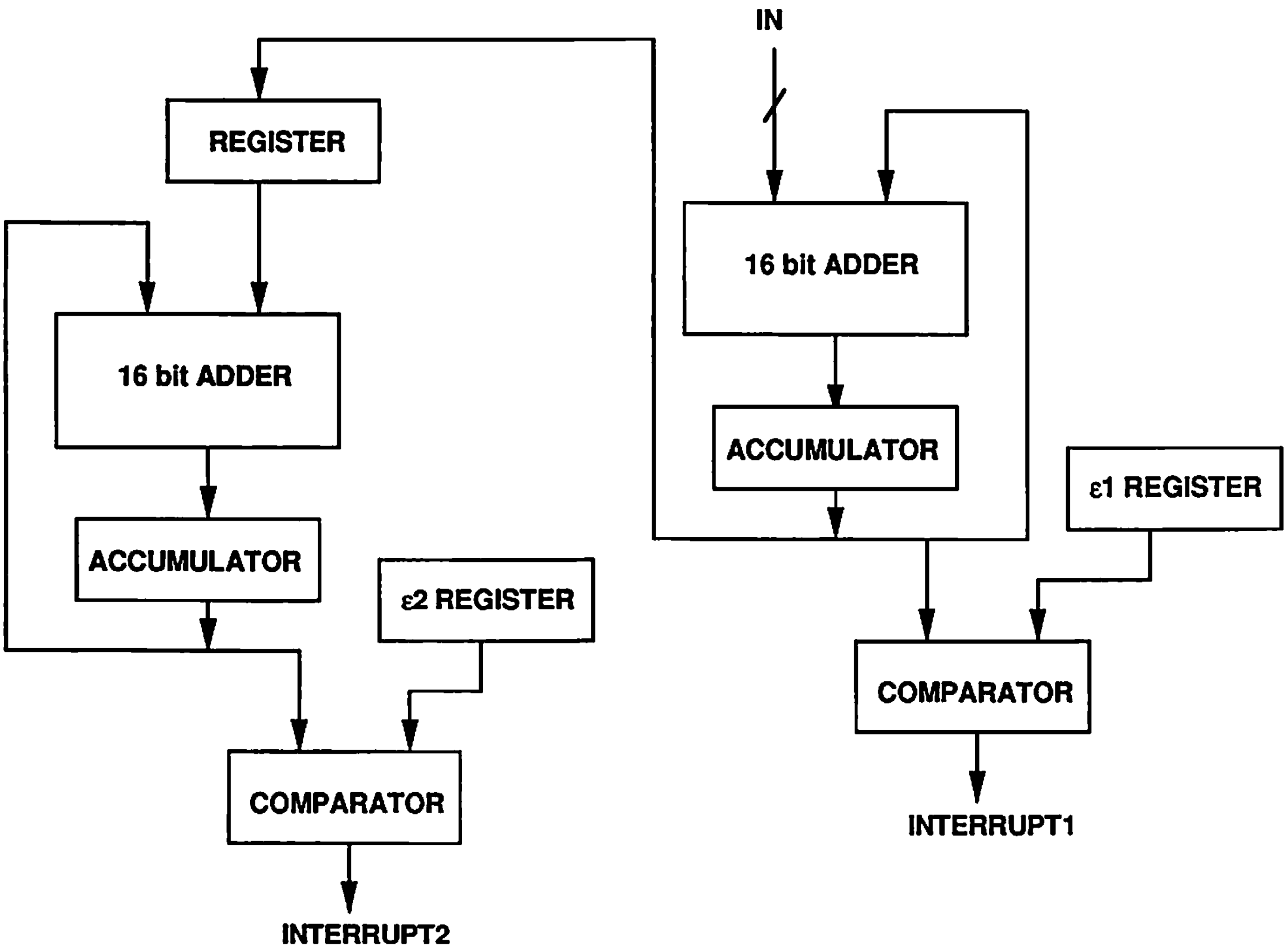


Fig. 4 8-TO-16 TRANSLATOR AND ERROR THRESHOLD CIRCUIT

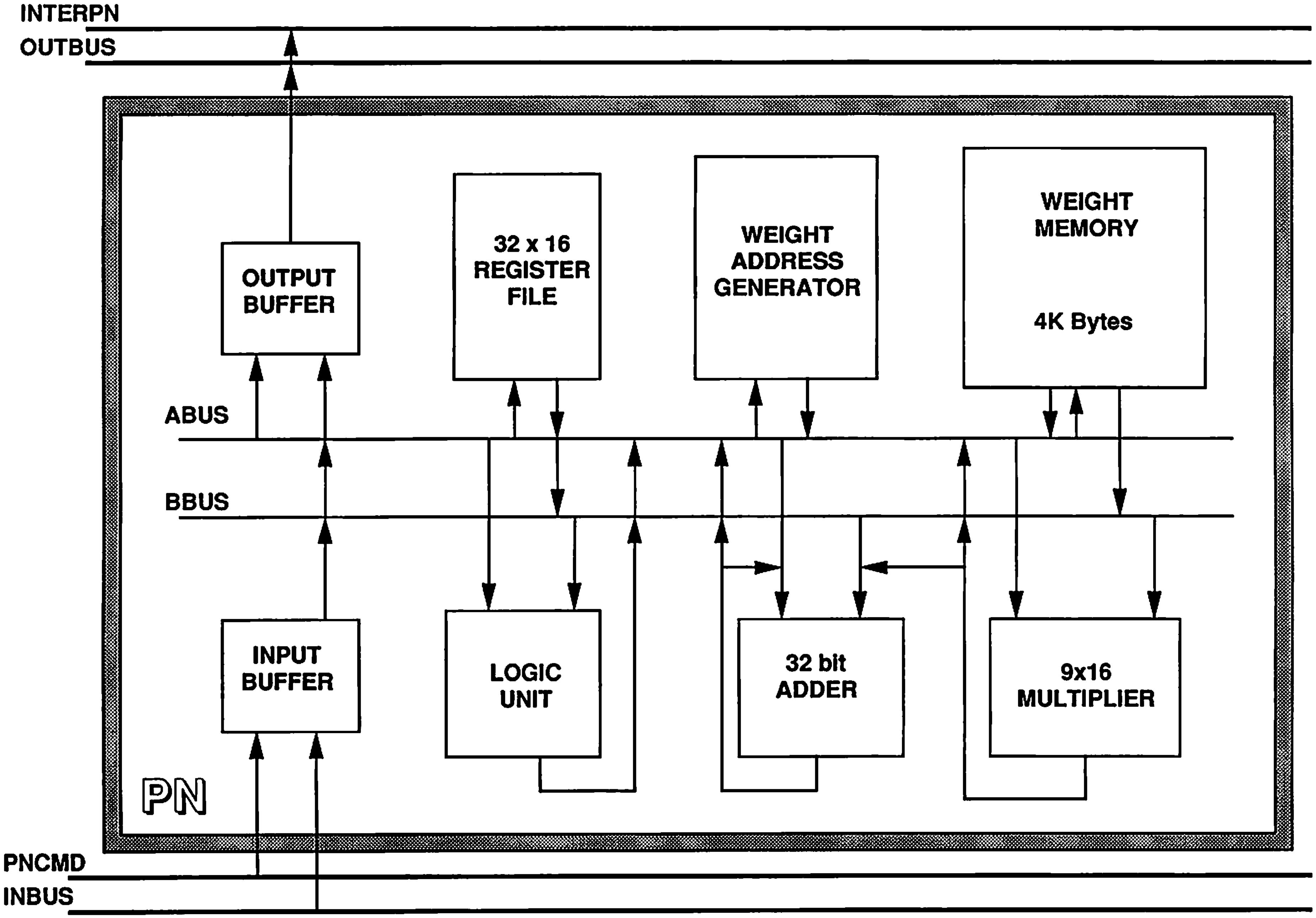


FIG.5 NEURAL NETWORK EXECUTION ENGINE -- PROCESSOR NODE BLOCK DIAGRAM