Dataflow Processor for Back Propagation Neural Networks: Architecture and Performance Evaluation

by

Maher Hamdan Khalil Abu-Mutlaq

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER ENGINEERING

February, 1995

Dataflow Processor for Back Propagation Neural Networks: Architecture and performance Evaluation

BY

Maher Hamdan Khalil Abu-Mutlaq

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

Computer Engineering

February 1995

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHRAN 31261, SAUDI ARABIA

COLLEGE OF GRADUATE STUDIES

This thesis written by

Maher Hamdan Abu-Mutlaq

under the direction of his Thesis Advisor and approved by his Thesis Committee,

has been presented to and accepted by the Dean of the College of Graduate Studies,

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER ENGINEERING

Thesis Committee:

[illegible]

Dr. Rafiq Braham (Chairman)

Sadiq Sait. M

Dr. Sadiq M. Sait (Co – Chairman)

[illegible]

Dr. Habib Youssef (Member)

[illegible]

Department Chairman

[illegible]

Dean, College of Graduate Studies

3/6/9[illegible]

Date

كلية الدراسات العليا
COLLEGE OF GRADUATE STUDIES

COMPUTER ENGINEERING DEPARTMENT

To my Family

Acknowledgments

All praise be to Allah for his limitless help and guidance. Peace and blessings of Allah be upon His prophet Muhammad.

Acknowledgment is due to King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia, for the generous support for this research. I would like to express my profound gratitude and appreciation to my advisor, Dr. Rafiq Braham for his support and patience throughout this thesis. His continuous support and encouragement can never be forgotten. I would like to thank Dr. Sadiq M. Sait and Dr. Habib Yousef for their consistent support and valuable discussions.

I also wish to thank faculty, graduate students, and the staff members of the Computer Engineering Department for their support, especially the department chairman, Dr. Samir Abdul Jauwad. The encouragement and good wishes of my friends, A. Rayhan, E. Zaharna, I. Alshaikh, and M. Al Sharif are also worthy acknowledged. Special thanks must be given to my family for their encouragement and moral support.

Contents

List of Tables

List of Figures

Abstract

Name: Maher Hamdan Abu-Mutlaq

Title: Dataflow Processor for Back Propagation Neural Networks: Architecture and Performance Evaluation

Major Field: Computer Engineering

Date of Degree: February, 1995

Real time applications of neural networks demand high performance systems. Neural networks may be naturally represented by macro dataflow graphs. Dataflow machines therefore offer suitable platforms for simulation of these networks. In this thesis, the hardware requirements for neural computing are first discussed. The rationale of dataflow approach to neural computing is presented. A new neural dataflow processor architecture based on *argument-fetching principles* is proposed. The proposed architectural model is static and flexible to exploit different levels of parallelism offered by neural networks through the use of software pipelining. The architecture is studied by extensive simulations using some neural network examples with various parameters. Back propagation and Hopfield networks are transformed into dataflow graphs in order to execute on the machine. The simulation shows good performance results.

Key Words: Neural networks, Dataflow architecture, Argument-fetching, Performance Evaluation.

Master of Science Degree

King Fahd University of Petroleum and Minerals

Dhahran, Saudi Arabia

February, 1995

خلاصة الرسالة

الاسم : ماهر حمدان أبومطلق.
عنوان الرسالة : معالج بنظام انسياب المعلومات للشبكات العصبية التي تعلمت بطريقة التمرير الخلفي للأخطاء: النموذج وتقييم الأداء.
التخصص : هندسة الحاسب الآلي.
تاريخ الشهادة : فبراير 1995م.

إن تطبيقات الشبكات العصبية في الوقت الحقيقي تتطلب أنظمة حاسبات عالية الآداء. الشبكات العصبية الإصطناعية يمكن أن تمثل بواسطة رسم حسب نظام ماكرو انسياب المعلومات. لذلك يمثل الحاسوب العامل بنظام انسياب المعلومات آلة مناسبة لمحاكاة الشبكات العصبية.

إن هذه الاطروحة، تناقش أولا حاجيات الشبكات العصبية من الناحية الحسابية، ومن ثم تقدم الأسس لاستخدام نظام انسياب المعلومات في محاكاة الشبكات العصبية. تقدم هذه الاطروحة نموذجا جديدا من هذا النوع من الحواسيب. النموذج المقترح من النوع الإستاتيكي لانسياب المعلومات المعتمد على المتغيرات المجلوبة وهو قابل لاستغلال عدة مستويات من العمليات المتوازية المتوفرة في الشبكات العصبية. درس النموذج المقترح على شكل موسع باستخدام عدة أمثلة لشبكات عصبية مثل التي تعلمت بطريقة التمرير الخلفي للأخطاء. واعتمادا على هذه الأمثلة يتضح أن النموذج له فعالية عالية عندما يستخدم في محاكاة الشبكات العصبية.

درجة الماجستير في العلوم
حامعة الملك فهد للبترول و المعادن
الظهران - المملكة العربية السعودية
فبراير 1995م

Chapter 1

Introduction

1.1 Principles of neural networks

Artificial Neural Networks (ANNs) are inspired by biological nervous systems. A neural network is composed of simple processing elements called neurons operating in parallel. The processing elements are interconnected via unidirectional signal channels [19]. Each connection has a value, defined as a connection weight. The weights between the neurons represent the network architecture and determine to a large extent the network function. Although neural networks have been known for decades, successful applications have been shown only recently. NNs have been employed, for instance, in pattern recognition, classification, vision and control systems [18].

There are two phases in neural information processing: a recall (or retrieving) phase and a learning (or training) phase. In the recall phase, the final neuron outputs representing the output of the network are computed based on the dynamic equations of the model [36]. During the learning phase, the weights are updated using learning rules and information extracted from training patterns, and desired outputs if given. Learning can be supervised or unsupervised. In supervised networks, such as back propagation (BP) networks, actual network outputs are compared to desired output values and an error signal is produced. Then, weights are updated according to the learning rules in order to minimize the error signal. An unsupervised network adapts itself according to statistical associations between input patterns [13]. In this case, outputs are not known in advance.

The neuron model is described by its inputs, weights and a nonlinear transfer function. Each individual input is multiplied by its corresponding weight, then all weighted inputs are added and fed to the transfer function (Figure 1.1).

Several neurons may constitute a layer and a network may contain more than one layer (Figure 1.2). Feed forward networks, for example, consist of ordered layers with no feedback paths. The lowest layer is the input layer and the highest layer is the output layer. Each layer sends outputs to higher layers only, and receives inputs from lower layers only [13].

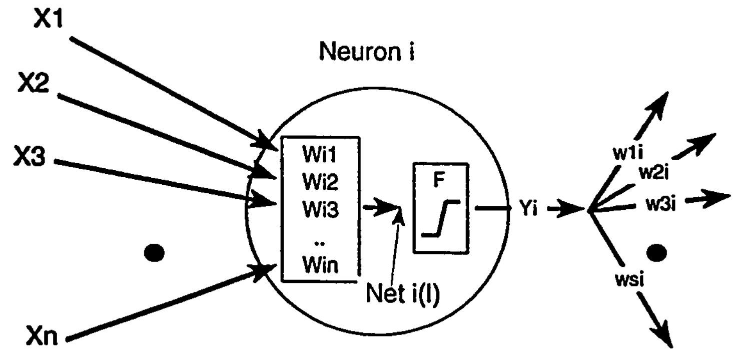


Figure 1.1: Neuron model.

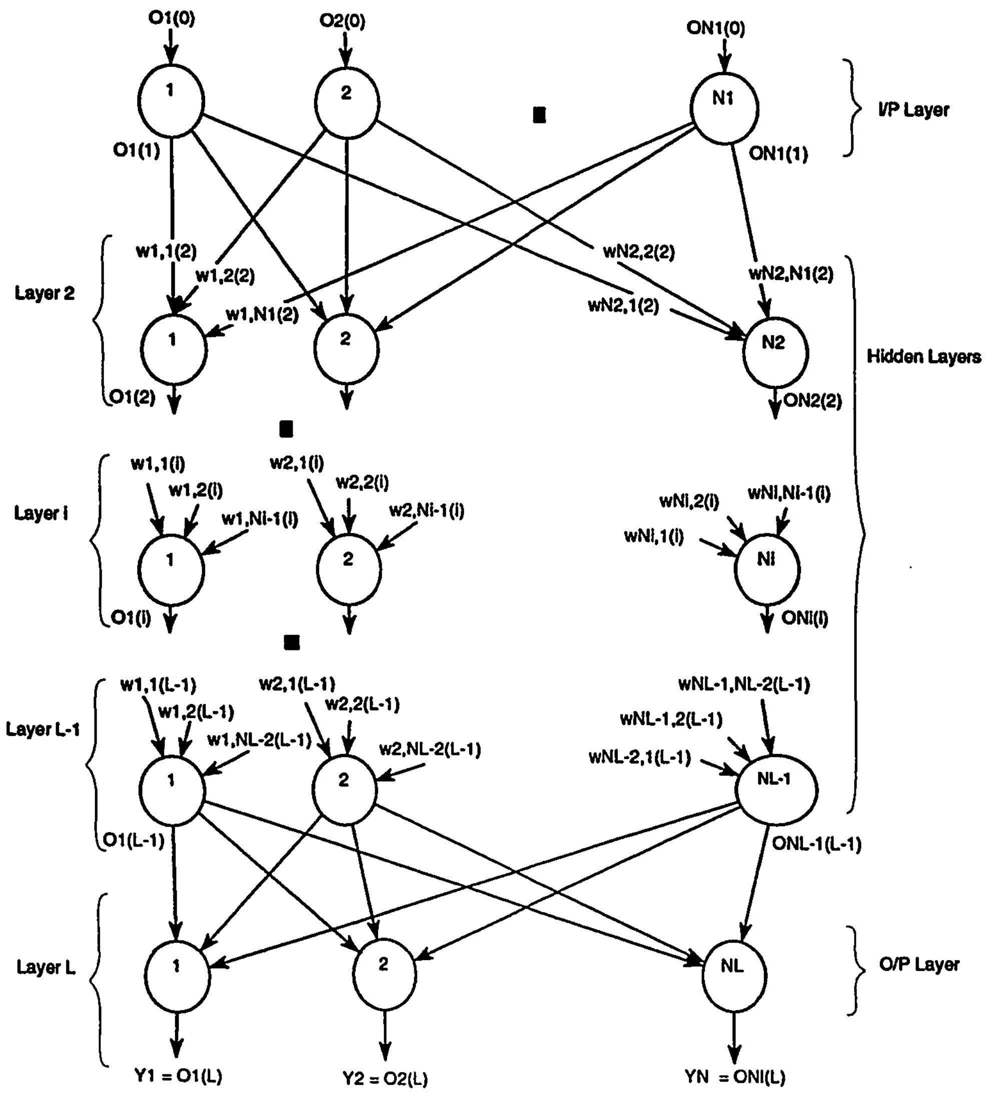


Figure 1.2: Back propagation neural network model.

1.2 Multi-layer back propagation networks

Back propagation (BP) algorithm is a supervised learning method in which the output error signal is fed back through the network [36]. The weights are altered so that the error is minimized. This algorithm is the most common neural network learning algorithm. It will be used in this thesis to illustrate the principles and operations needed for the majority of neural networks.

A typical BP network consists of three layers: input layer, hidden layer, and output layer. The input layer receives the data pattern and passes it to the hidden one. The procedure is then continued through higher layers until the result is presented in the output layer. In other BP network architectures, there can be many hidden layers. The output of each layer goes to the next layer only, and consequently each layer receives its inputs from the previous layer only. The recall phase is characterized by the following equations [36]:

$$Net_i(l) = \sum_{j=1}^{N_{l-1}} w_{ij}(l) \cdot O_j(l-1) + \Theta_i(l) \tag{1.1}$$

$$O_i(l) = F(Net_i(l)) \quad 1 \leq i \leq N_l \tag{1.2}$$

The following notation is used to describe the algorithm in Figure 1.2:

L	: number of layers.
N_l	: number of neurons in layer l.
$w_{ij}(l)$	: weight from neuron j to neuron i in layer l.
$Net_i(l)$	: linear summation of neuron i in layer l.
F	: nonlinear transfer function.
$O_i(l-1)$	: nonlinear output of neuron i in layer $l-1$.
Y_i	: actual output of neuron i in the output layer L, that is equivalent to $O_i(L)$.
T_i	: desired output at neuron i in the output layer.
N	: number of neurons at the output layer.
M	: number of data patterns.
$\delta_i^m(l)$	: error signal of neuron i in layer l with pattern m.
$\Theta_i(l)$	: bias or threshold value of neuron i in layer l.

The weights are updated in order to minimize the least-square-error between actual and desired values [36]:

$$E = \frac{1}{2}\sum_{m=1}^{M}\sum_{i=1}^{N}(T_i^m - Y_i^m)^2 \tag{1.3}$$

The error signal for output units is given by Equation 1.5 and for hidden units by Equation 1.6. The weights are updated according to the basic gradient type learning formulas as in Equation 1.7 [36]. The procedure iterates until desired error is achieved.

$$\delta_i^m(L) = (T_i^m(L) - Y_i^m(L)) \cdot \bar{f}^m(Net_i(L)) \quad \text{output unit} \tag{1.4}$$

$$\delta_j^m(l) = \overline{f}^m(Net_j(l)). \sum_{i=1}^{N_{l+1}} [\delta_i^m(l+1) \cdot w_{ij}^m(l+1)] \text{ hidden unit} \tag{1.5}$$

$$\Delta w_{ij}^m(l) = \eta . \delta_i^m(l) \cdot O_j^m(l-1) \tag{1.6}$$

$$w_{ij}^m(l) = w_{ij}^m(l) + \Delta w_{ij}^m(l) \tag{1.7}$$

The procedure described above shows that neural computations involve highly iterative computations. For a three-layer network ($n \times n \times n$), the computation complexity is shown in Table 1.1. The learning and the recall phases have the same computation complexity.

Table 1.1: Computation complexity of three-layer back propagation neural network.

Operation	Recall Phase	Learning Phase
Multiply	$O(2n^2)$	$O(3n^2 + 4n)$
Add.	$O(2n^2)$	$O(3n^2 + n)$
Nonlinear function	$O(2n)$	$O(2n)$

1.3 Hopfield networks

The Hopfield network is a feedback fixed-weight associative memory network [21, 22]. It is the most popular auto-associative network. In such a network, an input space is mapped mathematically to an output space. When the dimensions of the input and the output spaces are equal, the association is called auto-association. In a feedback network, a pattern is not recalled in one shot but after many iterations via the same network.

All processing elements in a Hopfield network are fully interconnected such that each receives inputs from all others (Figure 1.3). The processing elements outputs ($O_{i's}$) represent the state of the network. The network is usually characterized by an energy function of the form:

$$E = -\frac{1}{2}\sum_{j=1}^{n}\sum_{i=1}^{n} w_{ij} \cdot O_j \cdot O_i + \sum_{i=1}^{n} O_i \cdot \Theta_i \tag{1.8}$$

where,

n	: number of neurons.
w_{ij}	: weight from neuron j to neuron i.
Net_i	: linear sum of neuron i.
O_i	: nonlinear output of neuron i.
Θ_i	: threshold value of neuron i.
k	: iteration number.

The synaptic weights ($w_{ij's}$) are determined in advance by a set of input vectors. Initially, the system's state ($O_1, O_2, O_3, \cdots, O_n$) is set to an n binary-valued (0/1) input pattern. Whenever the processing elements change state, the energy function decreases. The system's state iterates until a local minimum of the energy function is reached.

There are two Hopfield models: a sequential (asynchronous) Hopfield model and a parallel (synchronous) Hopfield model [27]. The computation complexity of the Hopfield model is listed in Table 1.2.

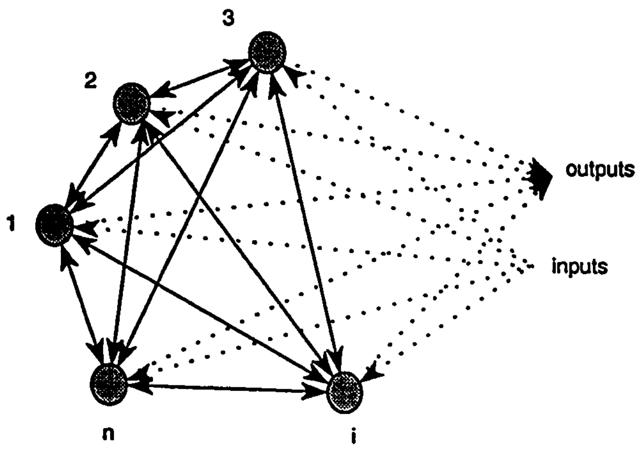


Figure 1.3: n-neuron Hopfield neural network model.

Table 1.2: Computation complexity of n-neuron Hopfield neural network.

Operation	Complexity
Multiply	$O(n^2)$
Add	$O(n^2)$
Nonlinear function	$O(n)$

1.3.1 Sequential (asynchronous) Hopfield model

Here one neuron only updates its output at a time. During the kth iteration, the network performs state updates in a sequential manner for $i = 1, 2, \cdots, n$. The n states are thus sequentially updated using Equations 1.9 and 1.10. The process is repeated until convergence.

$$Net_i(k+1) = \sum_{j=1}^{n} w_{ij}.O_j(k) + \Theta_i \tag{1.9}$$

$$O_i(k+1) = \begin{cases} 1 & Net_i(k+1) > 0 \\ 0 & Net_i(k+1) < 0 \\ O_i(k) & Net_i(k+1) = 0 \end{cases} \tag{1.10}$$

1.3.2 Parallel (synchronous) Hopfield model

In this case during the kth iteration, the net values for $i = 1, 2, \cdots, n$ are computed in parallel as in Equation 1.9. The n states are also updated in parallel (simultaneously) as in Equation 1.10.

Chapter 2

Neural Network Hardware Requirements

Most future neuro-applications will demand powerful machines. This chapter discusses neuro-computer design considerations. Some of the existing neuro-computers are also presented.

2.1 Analog versus digital implementation

The implementation of a neuro-computer can be categorized in three groups: analog, digital, and hybrid. In the analog case, the neuron's output can be expressed as a voltage from 0 to 5 Volts. Intel, for example, has designed an Electrically Trainable Artificial Neural Network (ETANN) using the "floating gate" non-volatite memory technology for analog storage of weights [20]. Advantages and drawbacks of both analog and digital implementations are summarized in Tables 2.1 and 2.2 respectively [35, 6]. The decision of the circuitry type is application dependent as shown in Figure 2.1 [35].

Table 2.1: Advantages and disadvantages of the analog implementation.

Advantages	Disadvantages
Neuron is an amplifier	Susceptible to noise, crosstalk and temp.
Synapses are resistors, etc..	Programming the weights R is not easy
High speed	Interfacing to the system environment
Area efficient. Many can fit in one chip	Low precision limited to 8 bits
Real time continuous application	

Table 2.2: Advantages and disadvantages of the digital implementation.

Advantages	Disadvantages
Flexibility in precision	Relatively low speed
Supported by alot of CAD tools	Bulk VLSI layout area
Convenient to block design for system design	
Programming and learning are flexible	

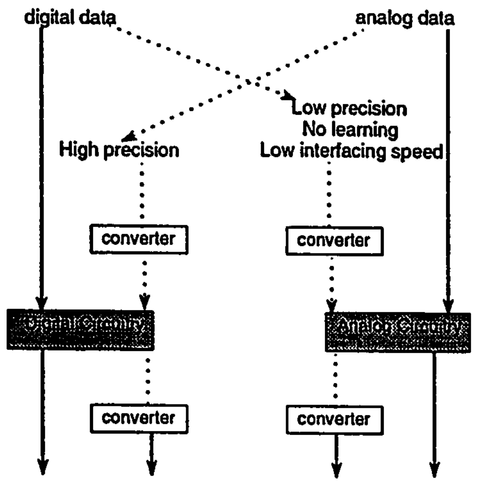


Figure 2.1: Analog versus digital implementation decision [35].

2.2 Neuro-computer design considerations

Recently, neural networks (NNs) have been applied successfully into several applications, especially in the areas of pattern recognition, vision, and control systems. When these applications run in real time, they demand high performance systems. Special purpose neural network hardware is warranted to meet this demand. But prior to hardware implementation of neural networks, many factors need to be considered. Such factors include speed, precision, scalability, generality, and programmability [6, 7, 19, 32].

Speed

Each connection in the recall phase requires a Multiply operation and an Add operation. Connections Per Second (CPS) or how many connections are evaluated per second is often used as a speed measure in neural network simulation. This refers often to the amount of maximum parallelism that can be exploited. It depends on other factors, e.g., precision of operands. Connection Update Per Second (CUPS) is another figure which is used to indicate how fast the connections can be updated during learning. Typically the CUPS number is 20-50% of the CPS number [32]. Real time applications, such as vision, demand very high processing rates. Table 2.3 shows five typical applications and their memory and speed requirements [5]. However, biological neurons fire pulses at a rate from few to 200 Hertz meaning that the weights are updated and the outputs of neurons are calculated once every 5 ms [8]. That is about 200 CUPS.

Precision

Several studies have been done to compare fixed-point and floating point models of computations [38]. Fixed-point error rates are equivalent to what is achieved through floating-point representations. In particular, precision of weight, input, neuron output, error term, intermediate values and weighted sum are given.

Applications	Memory Size	Word bits	In/Out Band-width	Cps Measure (Recall)	MFlops Meas. (Recall)
Radar Pulse Identification	<32 kW	1/neu. 32/sy.	2.5 MBaud	625 MCps	1.2 GFlops
Robot Arm Movement	4.5 kW	32	< 600 W/s	< 350 kCps	< 0.7 MFlops
Isolated Words Recognition (1024 words)	300 kW	32	2.5 kW/s	24 MCps	48 MFlops
Low Level Vision 100 connec./neu.	6.4 MW	1/neu. 8/sy.	1.6 MBaud	156 MCps	312 Mops
Risk Evaluation	4.1 MW	?	< 7 kW/s	4 MCps	8 MFlops

Table 2.3: Hardware requirements related to implementation of the five typical applications [5].

It has been found that for neural activations 8-bit representations are adequate [8]. Based on biological arguments, 8 bits are needed to differentiate between pulses at a rate of 200 Hz. For artificial neural networks, the same precision suffices a wide range of applications [38]. The quantization error in the [-1,+1] range is 2^{-7}. The dynamic range using 2's complement is given in Inequality 2.1.

$$-1 \leq x_i \leq 1 \quad (I_w = 1, I_f = 7) \tag{2.1}$$

where, I_w : integer bits, and I_f : fraction bits.

Regarding weights, a 16-bit resolution is sufficient for most applications. For example, 16-bit weight values are used in SPERT and MA16 [34, 39]. Its dynamic range is shown in Inequality 2.2.

$$-2^{I_{w-1}} \leq w_{ij} \leq 2^{I_w} - 2^{I_f} \quad (I_w = 1, I_f = 15) \tag{2.2}$$

Some techniques such as dithering and software manipulations can be used to improve the network performance using 8-bit weight precision. The same precision can be used for the weight increment and the error rate during learning. The quantization error is 2^{-15}. Low precision selection does not represent truly the sigmoid function and its derivative. Therefore, it may make the network untrainable.

For intermediate summations, weighted inputs are summed through 32-bit operand values. The weighted sum is truncated to 16-bit and fed as an input to the nonlinear function. The precision requirements are summarized in Table 2.4. Digital circuitry currently is most suitable for implementing high precision networks.

Table 2.4: Precision requirements for wide range of applications.

Operand	Precision
Weight	16b
Input and neuron output	9b
Intermediate values	32b
Weight Increment	16b

Scalability

This is another important issue in neuro-computer design. It refers to how many processing elements can be cascaded in a system at a reasonable cost. Emulation of biological networks and simulation of large artificial neural networks for real time applications make scalability essential for future neuro-computers.

Generality

Special-purpose neuro-computers are dedicated to implementation of only certain neural network models. For example, TInMANN VLSI chip is designed for implementing Kohonen self-organizing feature map neural network [30]. On the other hand, general-purpose neuro-computers should implement any neural network model. Examples of general-purpose neuro-computers are described in Section 2.4.

Interfacing Speed

System interfacing plays an important role in choosing the type of circuitry for neural network implementation. Analog circuitry, for instance, is not suitable for vision applications because it requires high speed converters [35]. As shown in Table 2.2, if the application calls for the on-chip support of learning, then a digital design should be preferred [35].

2.3 Nonlinear sigmoid function

Neural networks require a nonlinear activation function at the output of each neuron. The nonlinear sigmoid function is one of the most popular activation functions for many algorithms such as back propagation and Hopfield networks (Figure 2.2). Several expressions are available for evaluating the sigmoid function. One expensive technique in terms of computation time is through the summation of truncated series expansion. The logistic function is $f(x) = \frac{1}{1+e^{-x}}$. Another nonlinear function which has been used has the following form [9]:

$$F(x) = \begin{cases} 0 & x \leq \mu_0 \\ \frac{1}{2}[1 - \cos(ax - \alpha_0)] & a = \frac{\Pi}{\mu_1 - \mu_0},\ \alpha_0 = a\mu_0,\ \mu_0 \leq x \leq \mu_1 \\ 1 & x \geq \mu_1 \end{cases} \tag{2.3}$$

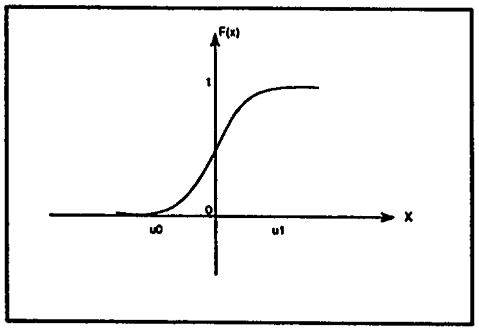


Figure 2.2: Sigmoid function.

The nonlinear sigmoid function may be implemented using a lookup table which can be stored in RAM or ROM. A lookup table of 16-bit inputs and 8-bit outputs demands 64 KB of memory which occupies a relatively large silicon area. SPERT, a general-purpose neuro-chip, implements the nonlinear function as a lookup table of 64KB [39].

An alternative approach is to approximate the nonlinear function by piecewise linearities. The break points are stored in memory. The function evaluation involves a combination of lookup table accesses and a linear interpolation. It has been shown, however, that the back propagation network did not perform well when used during learning. In other words, the training was unstable and failed to converge. This is due to the fact that the first derivative of the region around $x = 0$ is higher than the true sigmoid derivative function. For [-8,+8] input range, the function is represented by 13 segments [31].

A modified curve based on power of 2 calculations has been proposed with only 7 segments. The break points are: -8,-4,-2,-1,1,2,4, and 8 [31]. The modified piecewise linear approximation gives comparable results to what is achieved through true sigmoid function. The 16-bit 2's complement input value is mapped to 9-bit 2's complement output value. An efficient implementation for digital VLSI neural networks has been proposed without using a lookup table. The basic idea is that the circuit directly implements the positive input portion. For the negative portion, the input is bypassed through 2's complement at the input and the output of the positive piecewise linear approximation circuit using a mirroring technique.

A possible precision representation is $I_w = 4, I_f = 12$ for the input of the function and $I_w = 2, I_f = 7$ for the function output. The derivative of the sigmoid function is simply $\overline{f}(x) = f(x)(1 - f(x))$.

Proper segmentation of a nonlinear function minimizes the number of break points and the error of the estimated function. Each segment defines a linear interpolation with $y = a_i(x - x_i) + b_i$ (see Figure 2.3). In addition, mirroring technique exploits the symmetry of the nonlinear function around the y-axis or the xy-axis by halving the number of break points.

A small-size associative memory can be used to implement a nonlinear (curved) function (Figure 2.4), for example Gaussian function. The size of the memory is determined by the number of break points on the positive portion on the x-axis. Eight break points should be sufficient. The derivative of the nonlinear function is treated as a stand alone function. The general form is $y = a_i(X - x_i) + b_i$.

where,

X	: input to the nonlinear function.
x_i	: break point i.
a_i	: slope of segment i.
b_i	: y-offset of the interpolated segment i.
S	: symmetry bit. 0 and 1 values represent y-axis and xy-axis symmetry respectively.

A neural network model is usually characterized by a single nonlinear function. Therefore, two blocks of 8-word associative memory should be adequate to represent a nonlinear function and its derivative. The contents of the memory are loaded only once prior to the hardware simulation of the neural network. In addition, the implementation is not only restricted to the nonlinear function of the neuron's output. It can implement other nonlinear computations required by some neural network models.

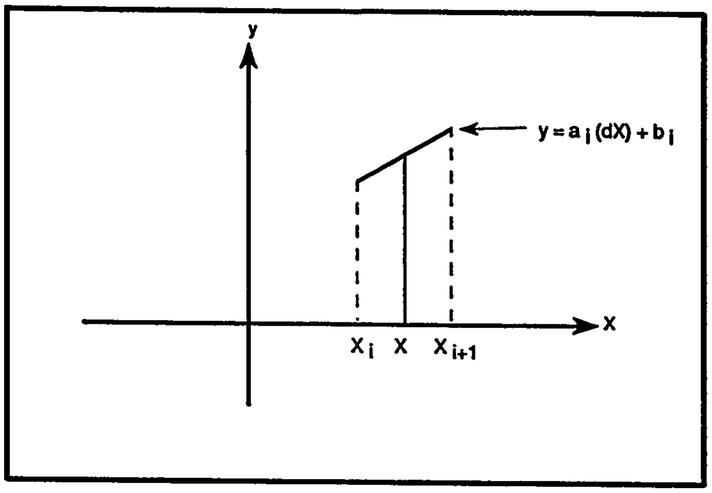


Figure 2.3: Segment interpolation of a nonlinear function.

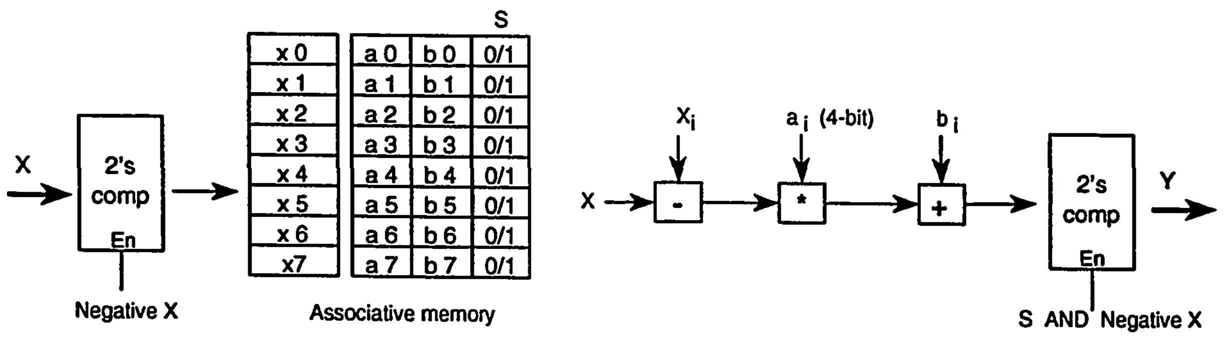


Figure 2.4: Block diagram of a general nonlinear function implementation.

2.4 Review of hardware implementations

Neural networks can be *fully* or *virtually* implemented in hardware [19]. *Full* implementation of neural networks provides dedicated hardware for each information processing element (neuron) and each weight of the network. For modest-sized, low precision, and special-purpose neural network applications, *full* implementation in analog hardware is an attractive solution.

In *virtual* implementation, the hardware used to implement a number of neurons is time multiplexed. The neural network is typically partitioned and mapped into a fewer number of physical processors. Each processor is allocated a sub-network to be simulated locally. The *virtual* implementation is essentially done in software. It is an established practice to simulate large neural networks. Moreover, the maximum generality in both transfer function and connection configuration is often achieved through *virtual* neuro-computers [19]. *Virtual* implementation is more flexible for implementing a wider range of neural algorithms.

Neural networks have been simulated on general purpose commercial computers (Figure 2.5) [33]. Simulations on dataflow machines are described in Section 3.5. In addition, many special hardware implementations have been developed for neural network simulation. A selection of several neuro-computers is briefly discussed next.

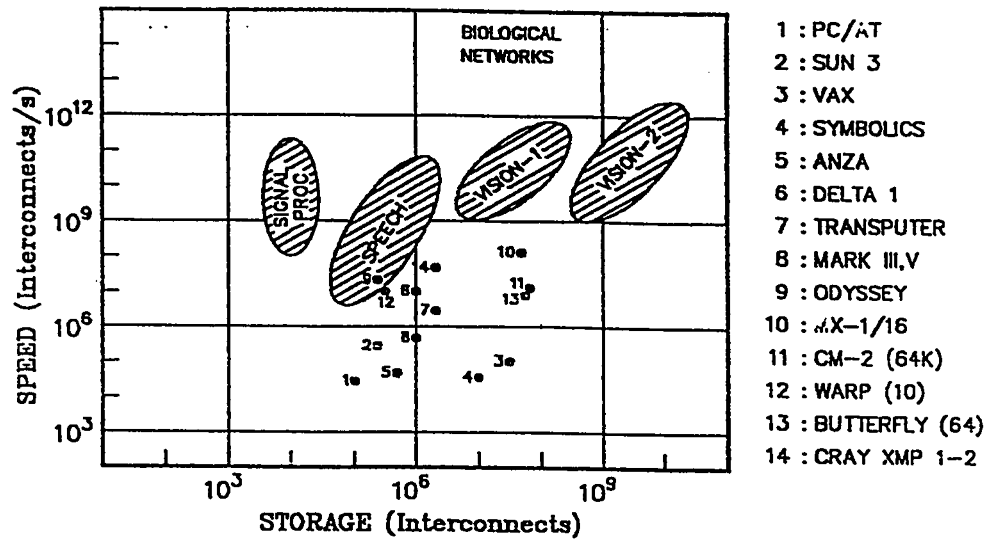


Table 2.5: Computational capabilities of neural network simulators versus computational requirements of some applications [33].

Hitachi has developed a wafer scale integration neural network [29]. The wafer can have 576 neurons, each mapped to a physical processing element. The processing element contains a multiplier, an adder, and a register-file holding up to 64 8-bit weights. The processing elements are connected through a time-sharing digital bus. The 9-bit neuron's output is broadcast through the bus to all processing elements in a time step of 464ns. The implementation is more appropriate for Hopfield model. The wafer speed performance is 1.2×10^9 CPS. No on-chip learning is supported.

Siemens has developed a synthesis of neural algorithms on a parallel systolic engine called *synapse* [34]. It is a general purpose neuro-computer that follows a multiprocessor and memory architecture. Systolic array approach is used at both chip level, called MA16, and at board level. The systolic design is based on three formulae that summarize a range of neural network models. MA16 chip implements 4×4 scalar product chains using 16 multipliers. The chip operates at 40MHz offering a performance of the order of 500×10^6 CPS. Two DRAM banks are interleaved, supplying MA16 with enough dataflows. These memories are dedicated for storing the 16-bit synaptic weights. The number of chips and memory boards in this architecture may vary according to the application requirements.

Nonintensive operations are computed on a Data Unit, DU. MC68040 CPU is used to perform such operations and take over the communication controls between the MA16 chips and the DU.

SPERT is a neuro-processor that uses the single instruction multiple data (SIMD) architecture of an array of eight integer data paths. SPERT is connected to an external SRAM memory that holds weight vectors and the program code. In order to increase the memory bandwidth, very large instruction word (VLIW) format is used with 128-bit memory width. The weight resolution is 16-bit. Basically, each data path implements multiply and add instruction (MAD). The SIMD array execute simultaneously eight MAD operations over the memory latched data. SPERT includes a cache and a general scalar data path. It uses 64 K entry lookup table representing the sigmoid nonlinear function of the system.

For three layer back propagation network, a peak performance of 350×10^6 CPS and 100×10^6 CUPS has been achieved for the recall and the learning phase respectively.

A more extensive survey on neural simulation hardware can be found in [32]. In general, neural network simulation systems follow two approaches. The first approach is the simulation using highly and massively parallel systems with a large number of processors such as CM-2 and message-passing computers. The other approach is the simulation on relatively small number of extremely powerful processors for example SPERT and MA-16.

Chapter 3

Dataflow Computer Architectures

In conventional von Neumann processor architecture, a program counter is needed to control the sequence of instructions. The instructions are executed sequentially. Special operators like BRANCH are used explicitly to transfer the control of the sequence. Two main issues called *Memory Latency* and *Synchronization* must be addressed in a multiprocessor environment [4]. Memory Latency is defined as the elapsed time between issuing a memory request and getting its response. Synchronization is known as the need to order the instructions according to their data dependency. In conventional parallel processors, explicit operators such as Fork and Join are used to synchronize instruction execution among different processors [23]. As a result, large overhead is wasted even with the use of different techniques like context switching, which are aimed at reducing the overhead effect.

Data-driven or dataflow approach is an alternative architecture that addresses the issues of memory latency and synchronization. The fundamental feature of dataflow is in its execution paradigm. An instruction is enabled for execution based on the availability of its operands. The program is transformed into a dataflow graph (DFG). DFG represents the data dependency between the instructions. Each node of the dataflow graph often represents an asynchronous instruction. Synchronization between instructions is achieved implicitly by the flow of data rather than by using explicit operators. It also allows concurrency between the operations. Once the instruction is executed, the result is contained in a token. The token is forwarded to subsequent instructions as an operand through communication paths (or arcs) defined by the DFG. Consequently, both issues are treated by the dataflow architecture. Furthermore, dataflow supports the building of highly parallel and asynchronous architectures. There are two types of dataflow architectures: static and dynamic.

3.1 Static architecture

DFGs generated by the compiler are loaded into memory during initialization and allow one instance of a node to be executed. A node in a DFG is executed according to the following rule:

A node is fired as soon as tokens are present at all input arcs and there is no token at any of its output arcs [16].

In order for a node to fire, all its tokens in the output arcs should be consumed. A maximum of one token is allowed per arc and there is no overlapping of concurrent unfolded iterations. Acknowledgment signals are fed back to the node from subsequent instructions to indicate that the node result has been consumed.

MIT static dataflow architecture was a major step toward the development of new dataflow architectures [16]. This static dataflow model comprises a set of processing elements (PEs) interconnected through a communication network. A single PE architecture of the static dataflow model is described next (Figure 3.1). At the machine level, DFG is represented by a collection of Activity Templates stored in the Activity Store. Each template contains instruction Op-code, Operands, and the Destination addresses. The destination addresses contain both the addresses of subsequent instructions for storing the result and the acknowledgment addresses. Whenever a result packet is received by PE, the Update Unit finds the corresponding template and places the result value in the appropriate operand slot. If the instruction is ready to execute, its template address is placed on the FIFO Instruction Queue. The Fetch Unit fetches the first address, reads its corresponding template, sends it for execution, and resets the template. Once the instruction is executed, the result is put into a result packet. The packet is passed to the Send Unit which decides whether to send it to the local PE or to an external PE through the network. The architecture implements a circular pipeline.

This model has some drawbacks, however. The parallelism is limited to folding iteration of the loop body. Code sharing of consecutive and concurrent unfolded iterations is not allowed. Only pipelining effect is achieved. The parallelism is limited to the critical path of the loop body. Moreover, acknowledgment signals double the number of tokens and the destination lists.

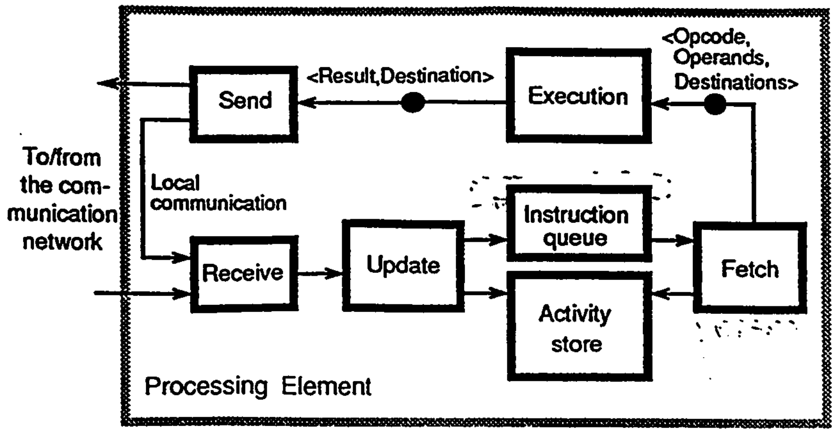


Figure 3.1: Block diagram of a single processing element [12].

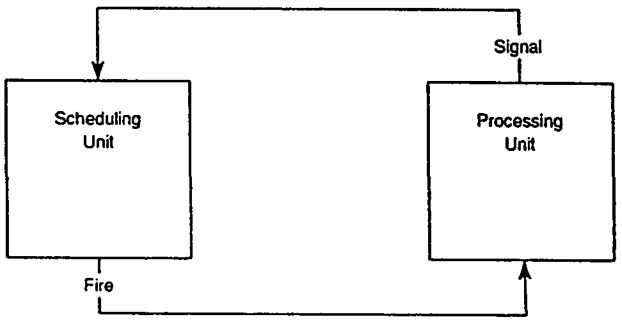


Figure 3.2: Block diagram of the argument-fetching dataflow processor.

3.2 Argument-fetching architecture

Recent static dataflow architectures implement the principles and issues of the basic pipelined static dataflow model. The principles of *argument-fetching* architecture have been incorporated in a modern static architecture [16]. Argument-fetching architecture has two parts: Dataflow Instruction Scheduling Unit (DISU) and a Pipelined Instruction Processing Unit (PIPU) [12].

3.2.1 Dataflow instruction scheduling unit

The program is represented by a pair of graphs (P,S). P graph represents the instruction list of the program and S graph represents the instruction sequence. Each node in P corresponds to an actor in the DFG of the program. It does not contain information about the sequence of the instructions. On the other hand, the sequence is represented by a signal graph S held by DISU. S has the following features:

1. Each node in S contains a reference to an instruction.
2. S has no information about instruction execution.
3. S arcs are signal arcs corresponding to the instruction sequence.
4. Each node in S has two fields: *counter* field and *reset* field.

The counter indicates the number of signal arrivals to an actor. The counter counts both the predecessor ready instructions and the acknowledgement of the successor instructions. Each signal arrival decrements the counter. When the counter reaches zero, a *fire* signal is generated by DISU indicating the availability of the instruction to be executed. Counter value is replaced by the reset value when the node is enabled.

The signal graph has to identify the subset of enabled instructions among all. DISU sends *fire* signals with the referenced address to PIPU. DISU supplies PIPU with enabled instructions. When the instruction is executed in PIPU, *done* signal is generated by PIPU holding the referenced address of the result (Figure 3.2) [12].

3.2.2 Pipelined instruction processing unit

This is a conventional pipelined processor. It consists of four sections: instruction fetch, operand fetch, operation execution, and result store. There can be no pipeline gaps in the PIPU as long as DISU continuously supplies enabled instructions. The sequence of the dataflow guarantees the absence of data conflict in PIPU. Each of the four PIPU sections may consist of several pipeline stages in order to match the timing requirements of memories and operation units [12].

3.3 Dynamic architecture

The dynamic model maximizes parallelism by allowing concurrency between unfolded consecutive iterations. More than one token is allowed per arc. Each token has a distinct tag assigned to it. It requires a matching unit for distinguishing the labels of an instance. The hardware cost of the matching unit makes it impractical even with the use of hashing techniques. MIT tagged-token dataflow model was the first to use the dynamic dataflow principles [16].

The concept of an Explicitly Addressed Token Store has been proposed as a part of the Monsoon architecture in order to eliminate the need for associative memory search [16]. The basic idea is to allocate a separate memory frame for every active loop iteration. The number of concurrent iterations is limited by the number (k) of frames. Only k consecutive iterations may be active at a time.

3.4 Notation and terminology

A DFG is a directed graph in which the vertices (nodes) denote entities called actors; links (arcs) represent paths that carry either data or control values between nodes. There are five primitive operations: copy, operator, decider, switch, and merge (Figure 3.3) [11].

Copy or Link: It is a node with one input actor and two or more output arcs. Its function is simply to copy a token from its input arc to all its output arcs.

Operator: It performs arithmetic or logic function. It fires by consuming all its input tokens and produces a token on its output arc.

Decider: It fires by consuming all its input tokens and produces a Boolean token based on a decision operation.

Switch: It has two inputs one of which is a conditional Boolean token. The second input value is passed to either one of the output arcs based on the Boolean input condition.

Merge: It has a conditional input arc and two inputs. The output value takes one of the two inputs based on the conditional Boolean value.

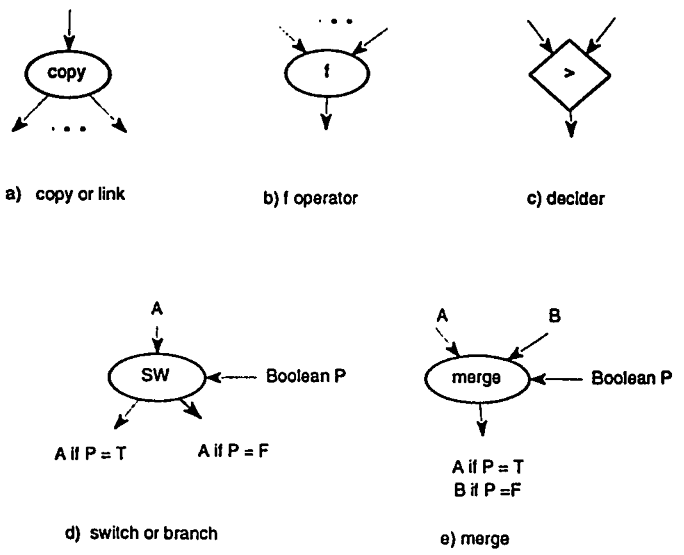


Figure 3.3: Primitive dataflow operations [11].

3.5 Simulation of neural networks on dataflow machines

A back propagation network has been transformed and simulated on the MIT-tagged dataflow machine [26]. As mentioned earlier, the major feature of the dynamic machine is that it allows concurrency between different iterations using tagging technique. One serious drawback of this approach is the lack of locality of computations, since all the data arrays are stored in a remote memory. Another constraint is the implementation of the non-linear function by a relatively large number of instructions. Due to the single assignment rule, a dataflow graph generates large intermediate values.

Alhaj and Terada propose a data-driven implementation of back propagation learning algorithm [2]. They transform the algorithm into a parallel implementation employing Q-v1 which is a general data driven processor developed by Osaka university, Sharp and Mitsubishi corporations. The implementation is based on partioning the network into sub-networks. Each sub-network is allocated to a processor. Different processors, connected in a two dimensional torus configuration, cooperate in computing the weighted sums.

The authors simulated an image data compression back propagation network on the system. The network consists of three layers with 256 neurons in the input and the output layers and 128 neurons in the hidden layer. With 16 processors, the network simulation was 14.68 times faster than with a single processor, thus achieving 92% system utilization rate. Recently, they achieved a speed performance of 50 MCUPS with 64 processors [3].

In addition, an estimated performance of neural network simulation on the Monsoon architecture is 2-3 MCPS per processor [37].

Chapter 4

A New Neural Dataflow Processor Architecture

4.1 Rationale of dataflow approach to neural computing

This section presents the motivation for building neural dataflow machines. A neural network is a parallel and distributed information processing structure that consists of highly interconnected processing elements. Each processing element performs local computations. Neural networks can be mapped into a large number of physical processors in different ways [2, 17, 19, 27]. The major mapping criterion is to minimize the messages to be transferred between physical processors, as well as to maintain the locality of distributed computations. The distributed computation structure of neural networks makes shared memory systems not suitable for neural network simulations [19, 24]. Highly and massively parallel computers with large number of processors, on the other hand, may offer attractive solutions for simulating the distributed computations of neural networks.

Ghosh and Hwang [17] have investigated the architectural requirements for neural network simulation on highly and massively parallel multiprocessors. They have developed an asynchronous, multi-message model in order to estimate the volume of interprocessor communications based on neural network demands. The study has been performed on different processor interconnection networks such as hypernets, hypercubes, and toruses. It supports the building of distributed-memory, multi-neuro-processor.

As mentioned earlier, simulation of large neural networks for real time applications make *scalability* an essential demand in neural system design.

Neural networks offer parallelism at different levels: bit, node (or neuron), layer, pattern, training example, and training session [32]. Many of the dedicated hardware architectures that have been developed and implemented for several neural network models exploit only the node parallelism [29, 32, 34, 39].

Many neural network algorithms (or models) have been suggested. The essential characteristics of these models are: network topology, processing element characteristics, and training rules [24]. The *connectivity* between the processing elements can be *fully* or *sparsely connected* [24, 32]. The Hopfield model, for example, has full neuron connectivity. The adjacent layers are also *fully* connected in the back propagation model. The processing elements read the weight and the output value, multiply them, and accumulate the result. SIMD architectures such as systolic arrays allow simultaneous calculation of many weights through broadcasting [24, 29, 34]. Therefore, they provide best hardware simulation for such type of networks.

In *sparsely connected networks*, many neural connections do not exist. In SIMD architectures, the *sparsely connected networks* can be treated as if they were *fully connected networks* by representing the weights of sparse connections with zero values for consistency [24].

Sparse activations mean that a small number of neurons change their output values at any time. Many computations are wasted due to the ignorance of this feature. Not surprisingly, the percentage of active neurons at any time is very low, 1-10% in competitive learning networks [24].

For a successful neural system design, the following rule is considered:

An application can be at best speed up by a factor of 1/(fraction of nonconnectionist computations) [39].

To permit *generality* and simulation efficiency, the neuro-computer should be flexible for a variety of neural network models including new models that are continuously emerging.

Tuning the hardware architecture for neural computing involves complex tasks in order to build successful neuro-computers. Intensive computations, for example multiply operation and nonlinear function evaluation, should be supported. The localized and distributed computations should be maintained.

Neural networks can be naturally represented by dataflow graph (DFG) [37]. They can be considered as macro dataflow graphs. They seem to be a natural subclass of MIMD systems such as dataflow architectures, which support the flexibility of simulating different neural models with large varieties of connectivity. The dataflow approach offers a suitable platform for simulating DFGs. More parallelism is exploited in this approach. Furthermore, dataflow machines are scalable and highly concurrent.

Therefore, dedicated neural dataflow processors seem promising. The purpose of this thesis is to determine the neural computation requirements and propose a new dataflow processor architecture dedicated to this computation. This architecture is then evaluated by extensive simulations.

4.2 Dataflow graph model

A neural network is decomposed and transformed into fine grain low level dataflow graph (DFG) in order to execute on the proposed dataflow machine. This follows *virtual* implementation of neural networks. For a specific application, the dataflow graph can be *fully* implemented with the use of hardware pipelining.

Dedicated DFG primitives have been selected based on neural network computation requirements (Figure 4.1). The following primitives have been defined: (1) arithmetic operations: Mul, Add, Sub, Nonlinear (NL) function, (2) IF operation, (3) logical operations: And and Not, (4) control: Copy and Join. The arcs of the DFG are of two types: data and control arcs. A data arc corresponds to a data token and a signal. These are used by the processing unit and the scheduling unit, respectively (see Section 4.3). The data token is to be stored in the data memory, while the signal is to activate a subsequent instruction, indicating the availability of the data token. The control arc holds only a signal address to update a subsequent instruction. The data arc and the control arc can be represented graphically by a normal arrow and a dotted arrow respectively as shown in Figure 4.2.

All arithmetic operations are carried with 2's complement fixed-point representation. The number of input arrows for an actor determines the number of waiting signals. Each actor may wait for a maximum of two signals.

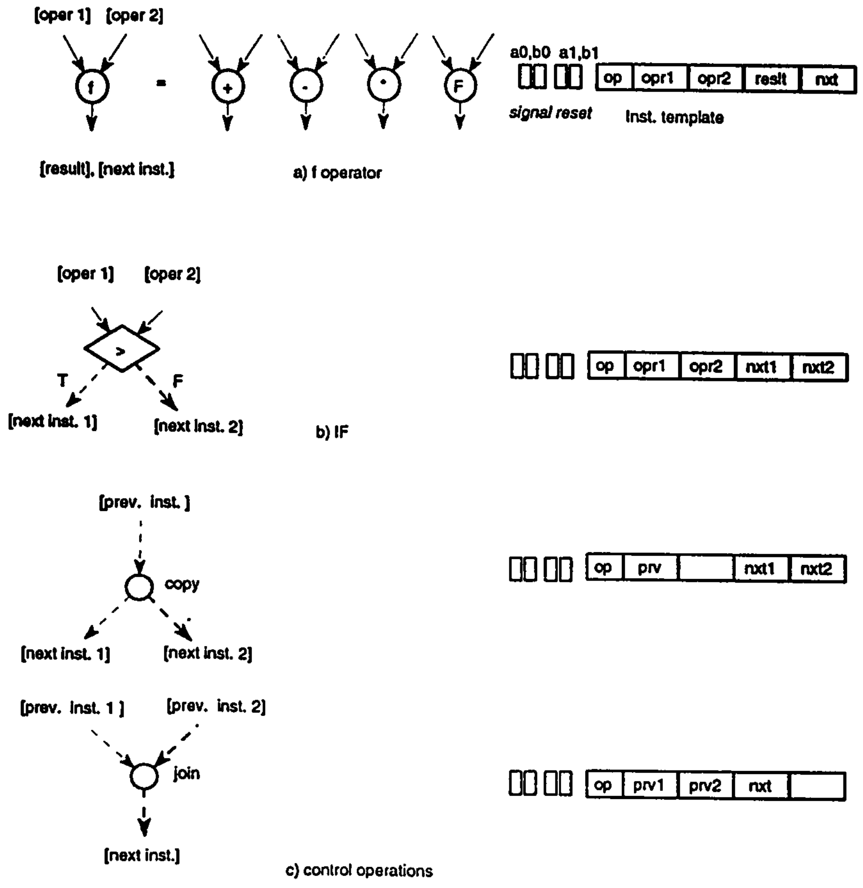
4.2.1 Dataflow graph primitives

The DFG primitives (Figure 4.1) are described as follows:

Operator: An operator can be one of the following: Multiply, Add, Subtract and nonlinear function. It should be flexible for other simple operations such as absolute value operation.

IF: During learning, the loop body iterates till desired error is obtained. Therefore, IF instruction compares two operands and signals one of the two output arcs based on the condition value. The instruction enables one of next two instruction templates. It should be noted that IF statement replaces both the decider and the switch primitives.

Copy: Its function is to signal the arrival of a token to more than one next instruction. In our case, the copy instruction signals two next instructions. For example, if the instruction calls for signaling next four instructions, three copy instructions are needed. The copy instruction operates locally to the scheduler. The instruction deals only with the program memory.



[oper1],[oper2] and [result] refer to operands in Data Memory.

[prev. inst 1],[prev. inst. 2],[next inst. 1] and [next inst. 2] refer to signal addresses in SU.

Figure 4.1: Dataflow graph primitives for neural networks.

Join: The architecture is static and flexible to exploit different levels of parallelism through software pipelining. In other words, the acknowledgment signal is not necessarily sent back directly after the consumption of a token. However, a number of neurons might be acknowledged at a later stage, e.g., at the output layer. One major drawback of the basic static architecture is that it duplicates the number of tokens. Moreover, the scalar product operation, common in neural networks, requires large number of acknowledgment addresses. Fan-in points at neuron outputs in DFG are adequate for join instruction. As a result, join instruction is used to collect the arrival of different signals, e.g., at the output of a layer, then it acknowledges the previous instructions, e.g., the input of that layer through copy instructions. This is to reduce the acknowledgment token overhead due to the static dataflow model.

4.2.2 Software pipelining

The proposed model supports software pipelining at fine-grain instruction level and at different network levels, e.g. layer and pattern levels. More parallelism is exposed explicitly by the use of copy and join operations. As an example, DFG of the recall phase of back propagation XOR network is shown in Figure 4.2. Different patterns can be pipelined using explicit control instructions.

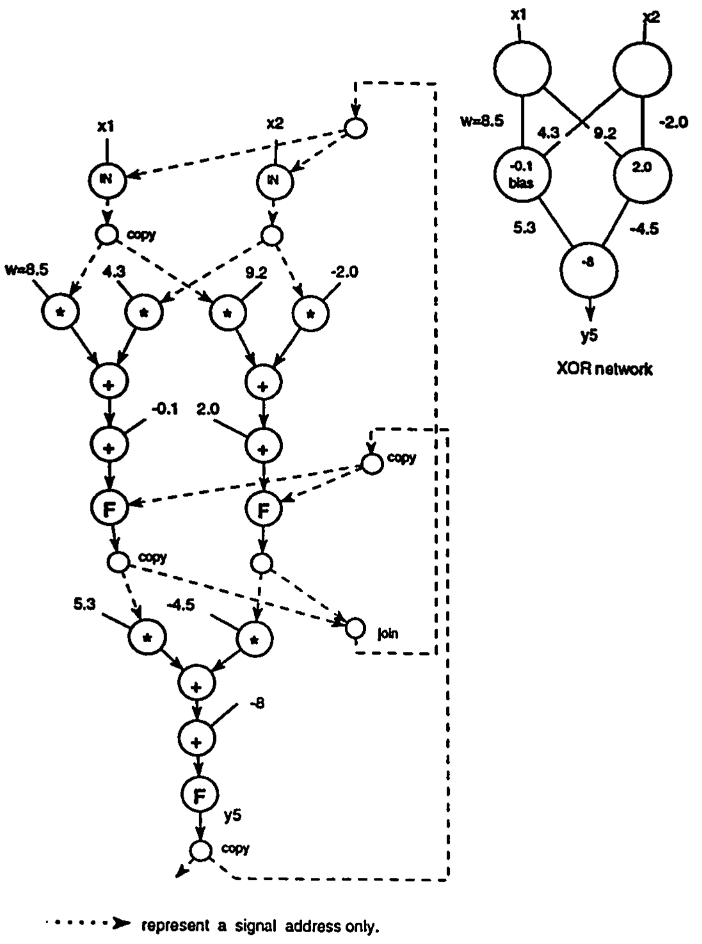


Figure 4.2: DFG of the recall phase of XOR2 network.

4.3 Proposed architecture

4.3.1 Basic principles

The proposed architectural model is based on the *argument-fetching* principles. The scheduling unit of the modern static architecture has incorporated the *argument-fetching* principles [12, 16]. The main feature of this architecture is that data never flows while the scheduling unit remains data-driven. Once a data result is stored in the data memory, a signal is sent to the scheduling unit. The scheduling unit is to signal all instructions waiting for this data result and supply ready instructions to instruction execution unit, without moving the data. This leads to a clear separation between the instruction execution unit and the scheduling unit. Therefore, the model is divided semantically into two parts: instruction scheduling unit (SU) and instruction processing unit (PU), as described earlier. In this architecture, the instruction (or program) memory belongs to the scheduling unit, and the data memory is exclusively available for the processing unit.

Scheduling unit: The node in the DFG represents a low level instruction that is composed of two parts: program part (P) and signal instruction part (S). The S-instruction holds two fields that are part of the dataflow instruction scheduling mechanism. It has two *signal* bits (a_0,b_0) indicating the arrival of a maximum of two operands, and *reset* bits (a_1,b_1), to be placed on the *signal* bits when the node is enabled (fired) for execution. Each signal arrival indicates the availability of a data operand. The sequence of the instructions can also be represented by a separate directed signal graph. The scheduling unit identifies instructions that are available for execution and supplies them to the processing unit.

Processing unit: This is a conventional pipelined processor. It executes the pool of enabled instructions. There can be no pipeline gaps in the processing unit, as long as scheduling unit continuously supplies enabled instructions. The sequence of the dataflow guarantees the absence of data conflict in the processing unit. After the instruction is executed and the result is stored, the processing unit delivers a signal address (done) to the scheduling unit, indicating the availability of a data token for a subsequent instruction. Both the processing unit and the scheduling unit implement a single circular pipeline dataflow path with separate data and instructions.

4.3.2 Machine architecture

The dataflow processor (DFP) described in this thesis is intended to be a basic component in large message-passing neural systems. The DFP has its local program memory that contains a sub-network graph instructions allocated to it. Moreover, the DFP has its own local data memory. The processor implements a single datapath on a ring pipelined architecture that consists of five sections: instruction update, operand fetch, ready instruction queue, exec, and result store as shown in Figures 4.3, and Figure 4.5.

Program memory

Each DFG node corresponds to a fixed instruction cell in the program memory (Figure 4.1). The instruction may wait for one or two signals represented by two *signal* bits. An instruction is fired upon the arrival of the required signals and sent to the execution unit. Once the instruction is fired, the *signal* bits are reset. A typical arithmetic cell contains the following fields:

- a_0,b_0: two *signal* bits indicating the arrival of a maximum of two input signals required to fire the instruction.
- a_1,b_1: two bits that represent the *reset* bits of the instruction.
- opcode: 4-bit opcode are assumed.
- A: address of the first operand in data memory.
- B: address of the second operand in data memory.
- C: address of the result in data memory.
- D: address of the subsequent signal instruction in the program memory, signaling a data token arrival.

Other instruction formats are shown in Figure 4.1. The program memory has an update unit which updates a_0 and b_0 bits. If the instruction is fired, it is supplied

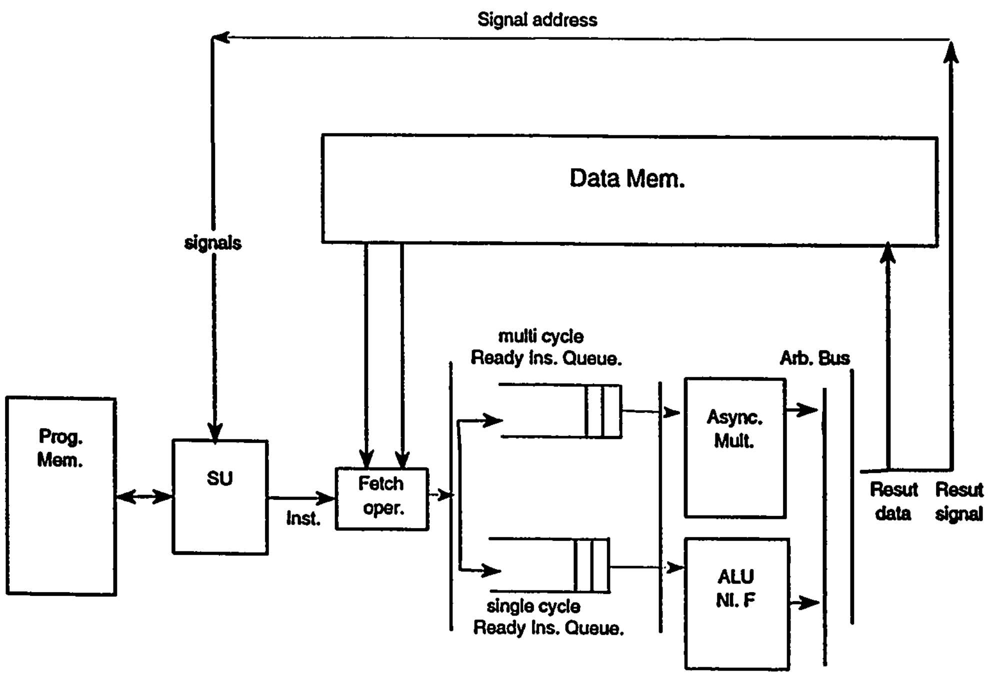


Figure 4.3: Block diagram of the proposed DFP architecture.

to the next stage where operands are fetched and supplied to the proper Ready Instruction Queue.

Signal queue

Once the processing unit stores the result in the data memory, it should signal a subsequent instruction. A signal address is written into the FIFO signal queue. The address refers to one of the *signal* bits a_0 or b_0 of an instruction cell. The update unit on the scheduling unit is busy with signal updating so long as the signal queue is not empty.

Copy and join instructions are not executable. They signal other instructions while the result stored in the data memory never flows. They also feed the queue with additional signal addresses.

Update unit

It belongs to the scheduling unit. It resets the *signal* bit of an instruction cell referenced by a signal address fetched from the signal queue (Figure 4.4). If the *signal* bits are zero, the instruction is fired. The executable fired instruction is supplied to the next fetch operand stage. The two *signal* bits are replaced by the *reset* bits of the instruction. If the *signal* bits of the instruction are not zero, it means that the instruction waits for another signal address. Thus, the *signal* bits are stored and no executable instruction is fired. The update unit identifies instructions that are available for execution.

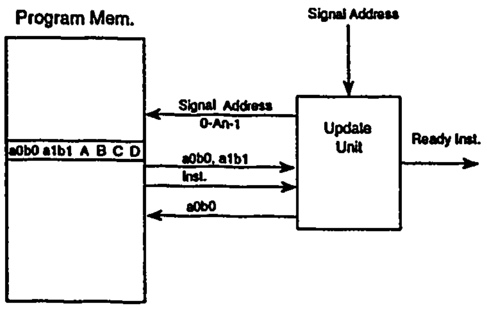


Figure 4.4: Block diagram of the connectivity of the update unit.

Operand fetch

A maximum of two data memory operands are fetched at any time. The resulting packet contains the opcode, operands, result address, and signal address. Then, the packet is routed to the proper queue for execution. All executable instructions, supplied by the scheduling unit, are independent. If the scheduling unit supplies instructions continuously, a triple-port data memory will be used efficiently, where two ports are used for reading operands and the third port is used for writing the result.

Single and multi queue

The execution units are divided into two types: single and multi cycle units. The receiving packet is routed to either the single or the multi queue according to its opcode. Independent instructions may overlap in time during execution. The mutli queue is dedicated to queue the multi cycle multiply instruction packets through a simple distribution bus. Other instructions are queued in the single cycle queue for ALU execution and nonlinear function evaluation.

Multipliers

The fixed point arithmetic multiplication can be implemented by a series of shift and add operations. The period of execution may vary according to the operand data value in order to exploit the bit parallelism of the multiplication operation. A set of asynchronous multipliers are used in order to reduce the multi queue waiting time. They inherently represent a pipelined multiplier. The queued multiply packet is routed to a multiplier that is not busy through a simple distribution bus. It is assumed that the maximum period of multiply execution is m cycles. Therefore, m asynchronous multipliers guarantee a minimum service time for multiplication. Once the multiplication result is ready, it is written into the data memory through an arbitration bus.

ALU

All other instructions such as Add, Sub and IF are executed within the ALU. Dedicated digital circuitry is used to implement the sigmoid function (see Section 2.3).

Result store unit

This unit writes the result into data memory. The result may be taken from different execution units through the arbitration bus. While the result is stored in the data memory, the signal address is written into the signal queue. This guarantees the availability of the result for consecutive instructions.

The processor architecture is flexible for sending and receiving messages from other processors (Figure 4.5). A message typically contains a data value, an address of the data memory, and a signal address of the instruction memory of the target processor. Once the message is received by a processor, the data value is first stored in the data memory, then the signal address is supplied to the signal queue of the processor.

4.4 Conclusion

In this chapter, The motivations for building dataflow systems to neural computing have been presented. A set of dedicated DFG primitives have been selected based on neural network computation requirements. A new neural dataflow architecture based on argument-fetching principles has been proposed. Different levels of parallelism are exposed through the use of software pipelining.

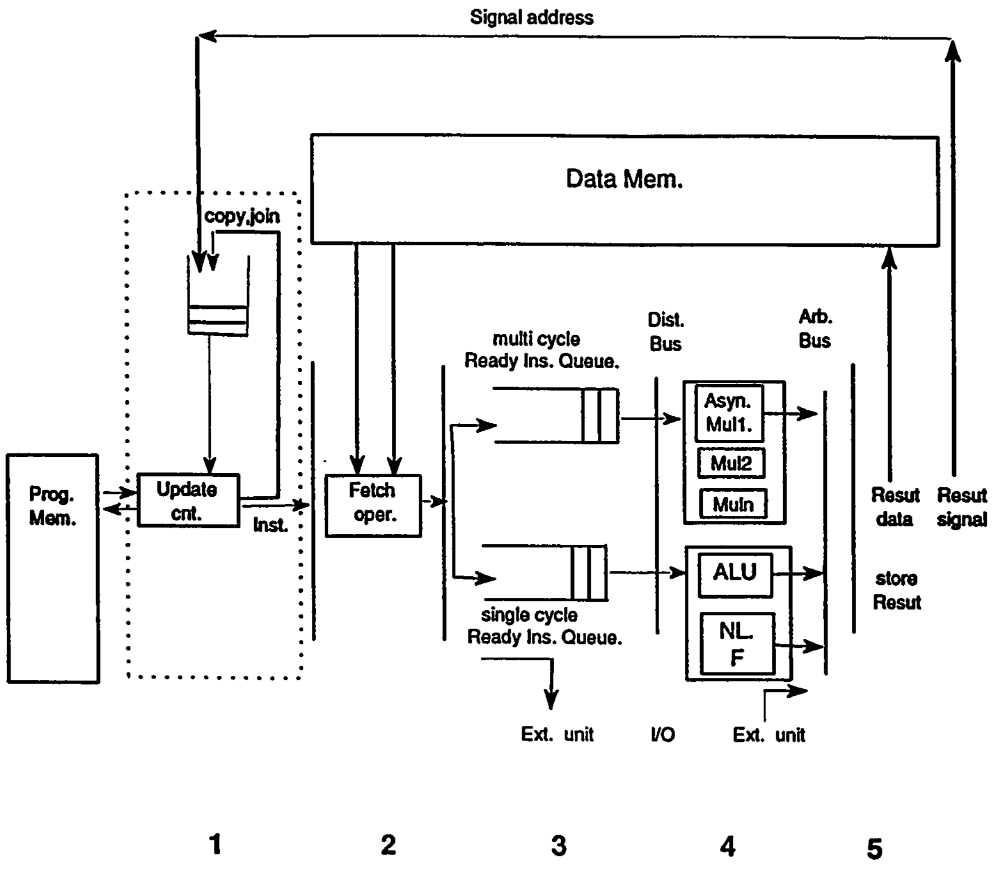


Figure 4.5: Detailed block diagram of the proposed DFP architecture.

Chapter 5

Simulation and Performance Evaluation

The architecture should be modeled and studied to see how the performance is affected. In this chapter, the simulation model is described and simulation results of the back propagation and the Hopfield networks are presented. These are widely used as hardware benchmark networks [32].

5.1 Simulation model

The system is not simple enough for analytical evaluation due to the nature of neural computations. In this case, we resort to simulation. Simulation is used as an alternative when analytical solutions are not possible. In simulation, a computer is used to evaluate a model numerically, and data are calculated in order to estimate specific characteristics of the model [28].

The system simulation model adopts a discrete-event view. It evolves in time by a representation in which the system state variables change instantaneously at a fixed-increment time interval (Δt). The simulation clock is advanced every Δt time interval. The time interval is a characteristic of the common pipeline stage delay. It is a reasonable assumption since all events occur at Δt, based on real hardware realizations of the architecture's pipeline stages.

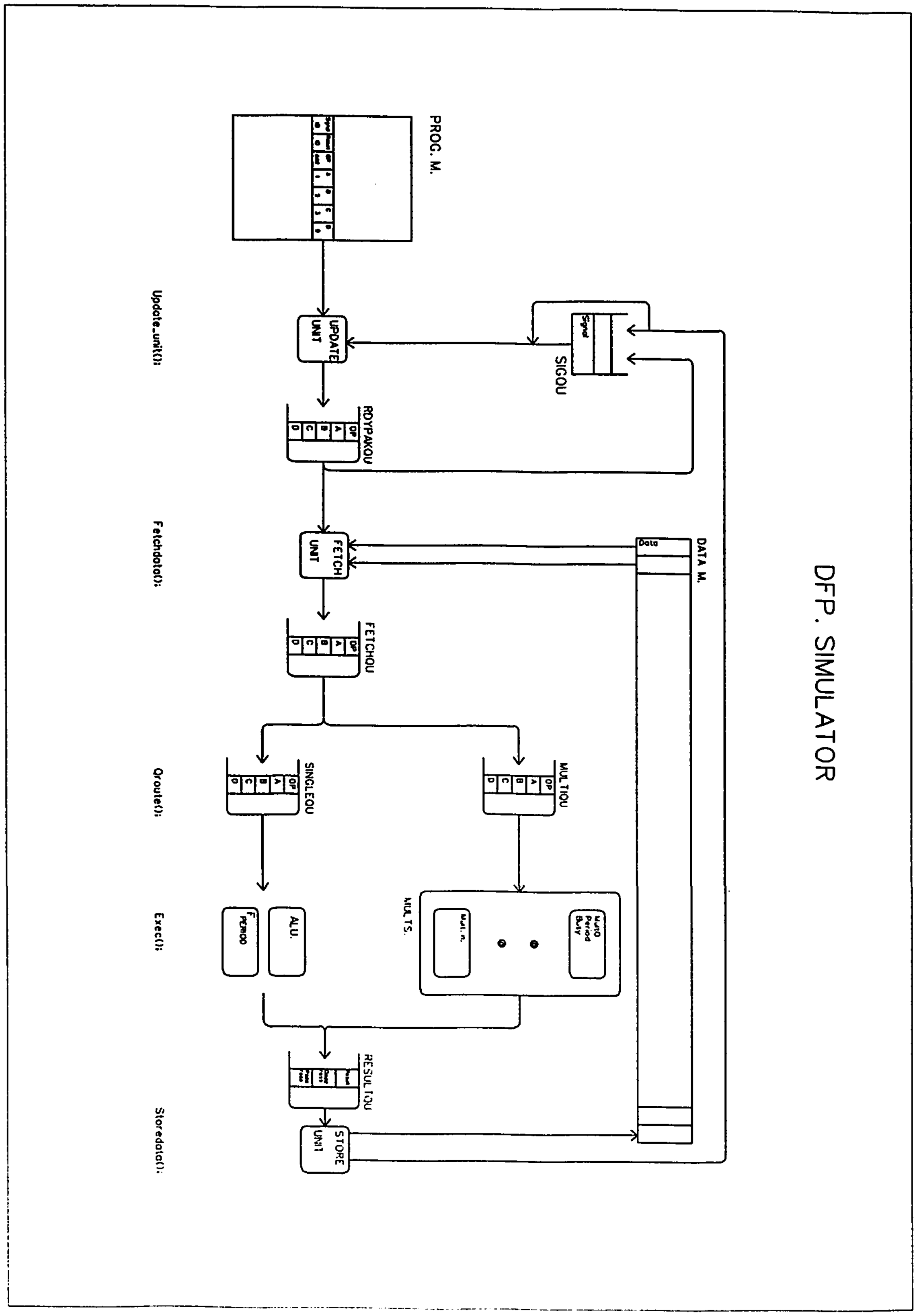
The simulation model is deterministic as no probabilisctic components are employed. True program instructions are mapped on the proposed architecture and simulation results are reported.

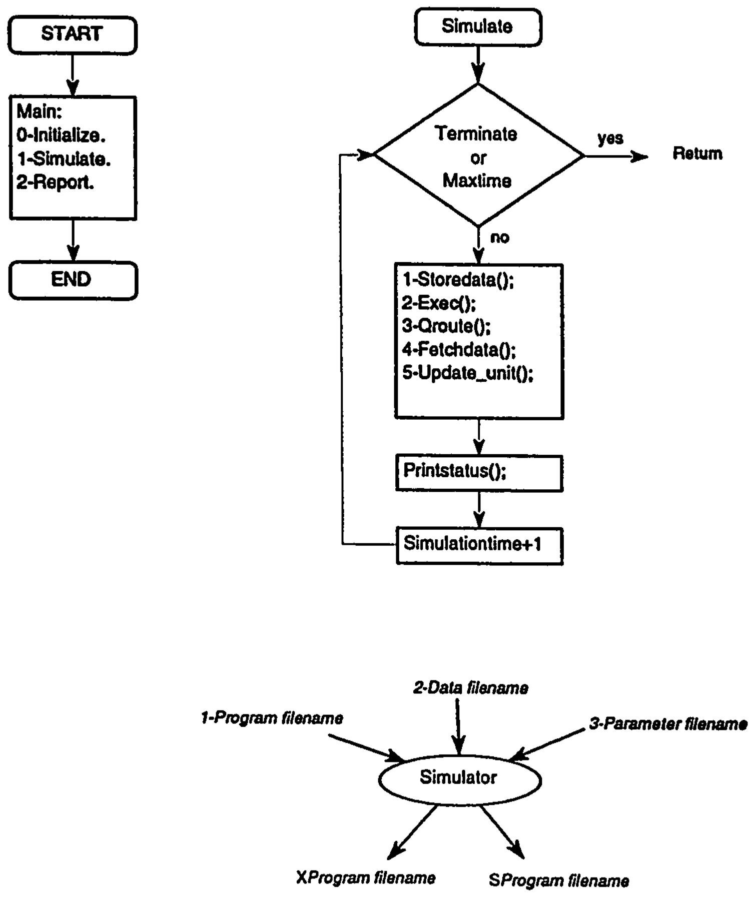
The high level language *C* has been chosen for the simulation of the dataflow processor architecture. Its popularity and portability make it a convenient tool.

5.1.1 Components of the system

The proposed processor architecture is characterized by five pipelined procedures: Update_Unit, Fetchdata, Qroute, Exec and Storedata. The logical relationship between components or blocks of the system is shown in Figure 5.1. Each block has its own characteristics in terms of functional representation and time delay. Once the simulation clock is incremented, a check is made of all blocks to determine the events that should occur at the end of the time interval. Then, the system states are updated accordingly. The main procedure of the simulator is shown in Figure 5.2. The scheduled events in the next cycle in different blocks may overlap in time.

Additional queues are added to the simulation model to make the model flexible for future development. These queues are: Rdypakqu, Fetchqu, and Resultqu.


DFP. SIMULATOR
PROG. M.
UPDATE UNIT
SIGQU
Signal
RDYPAKQU
OP
A
B
C
D
DATA M.
Data
FETCH UNIT
FETCHQU
MULTIQU
SINGLEQU
MULTS.
ALU.
FLPROC
RESULTQU
STORE UNIT
Update_unit();
Fetchdata();
Qroute();
Exec();
Storedata();



Examples :-

1-hopf6pro.dat	hopf3pro.dat
2-hopf6.dat	hopf3.dat
3-parmhop6.da	parmhop3.dat

Figure 5.2: Simulation algorithm.

5.1.2 Performance measures

The purpose of simulation is to simulate real neural network programs on the machine and analyze their impact on overall performance. System utilization and throughput measures are computed.

Several parameters that determine the machine configuration are varied each time a simulation test is conducted. These parameters are:

- U: number of signal updates allowed per time cycle in the update unit.
- M: number of asynchronous multipliers.
- m (multiplication period): number of cycles required for multiplication execution through shift and add operations. Each shift and add operation is equivalent to a 1-bit multiplication. When $M = m$, the set of asynchronous multipliers implements inherently a multiplication with a period of 1 cycle. In test cases, the value of m is chosen to be 8 cycles. This is based on the minimum precision of the multiply operands (9 bits). When $M \geq m$, the performance measures are independent of m.
- Multiplication mode: the multipliers can optionally be data dependent.
- Nonlinear function period: number of clock cycles required to evaluate the nonlinear function.

Other user parameters such as initial simulation clock, maximum simulation time (T) and observed intervals are set for each simulation run.

The simulation study is conducted by running different sizes of back propagation and Hopfield networks on different machine configurations in which mainly M and U are varied. The test case networks are transformed into DFG instructions and simulated on the proposed architecture.

Three main simulation outputs are considered:

- S: speed measure in connections per clock cycle (CPC). This refers to the numbers of connections evaluated during one clock cycle:

$$S(CPC) = \frac{F \times W}{T} \tag{5.1}$$

  where:

  F: number of nonlinear function evaluations (same as number of neurons in the recall phase).

  W: average number of weights per neuron.

  T: total execution time in clock cycles.

- r: rate of executable instructions supplied to the fetch data unit. This is an important parameter that refers to how frequently the scheduling unit supplies instructions.

- U_t: processor utilization. It indicates how long the DFP is busy with executable packets. The availability of one or more executable instructions in the execution units indicates that the processor is busy during that cycle.

Other performance measures are found in [1].

5.2 Back propagation networks

Back propagation (BP) algorithm is a supervised learning method in which the output error signal is fed back through the network as described in Section 1.2. This algorithm is the most common neural network learning algorithm. It is used to illustrate the principles and operations needed for the majority of neural networks.

5.2.1 XOR network

Recall phase

The recall phase of the XOR back propagation network is transformed into fine grain DFG instructions. Two transformations of the recall phase are presented. The first is when only one pattern per iteration is allowed (XOR1, Figure 5.3). This case corresponds to a strictly static dataflow transformation.

A more elaborate way is to allow successive waves of patterns per iteration using software pipelining. In this case, two pipelined patterns per iteration are allowed (XOR2, Figure 4.2).

Figures 5.4 and 5.5 show two output measures of XOR network simulation. The throughput of the network is increased by 55% when two successive waves of pipeline patterns are allowed. Moreover, the utilization is increased by nearly the same percentage. The performance measures for the second transformation are listed in Table 5.1. Due to the small size of the XOR network, the parallelism is limited. The output measures are independent of U when $M \geq 4$. The main purpose of this XOR test case is to study the effect of software pipelining.

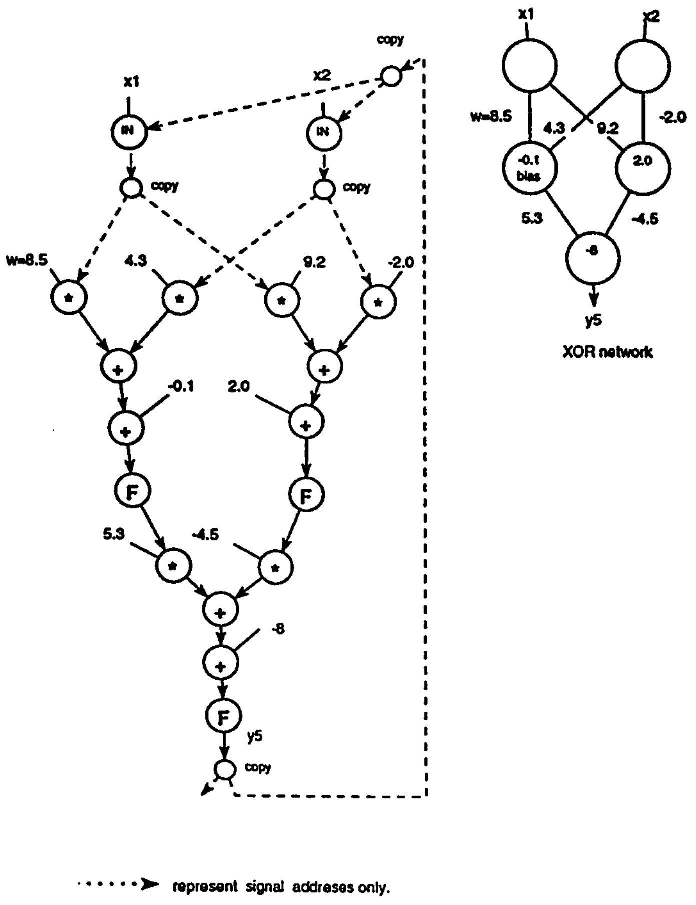


Figure 5.3: DFG of the recall phase of XOR1 network.

Table 5.1: The output measures for XOR2 network, T=30000.

U	1			2			3			4		
M	F	r	U_t	F	r	U_t	F	r	U_t	F	r	U_t
1	1527	28.9	77.9	1638	31.0	84.3	1638	31.0	84.3	1638	31.0	84.3
2	1903	36.0	75.7	2133	40.3	81.3	2162	40.9	82.0	2162	40.9	82.0
3	1967	37.2	74.8	2232	42.2	79.1	2303	43.5	79.0	2303	43.5	79.0
4	2043	38.6	73.2	2428	45.9	73.7	2516	47.7	73.2	2516	47.6	73.2
5	2043	38.6	73.2	2428	45.9	73.7	2516	47.7	73.2	2516	47.7	73.2
6	2043	38.6	73.2	2428	45.9	73.7	2516	47.6	73.2	2516	47.6	73.2
7	2043	38.6	732	2428	45.9	73.7	2516	47.6	73.2	2516	47.6	73.2
8	2043	38.6	73.2	2428	45.9	73.7	2516	47.6	73.2	2516	47.6	73.2

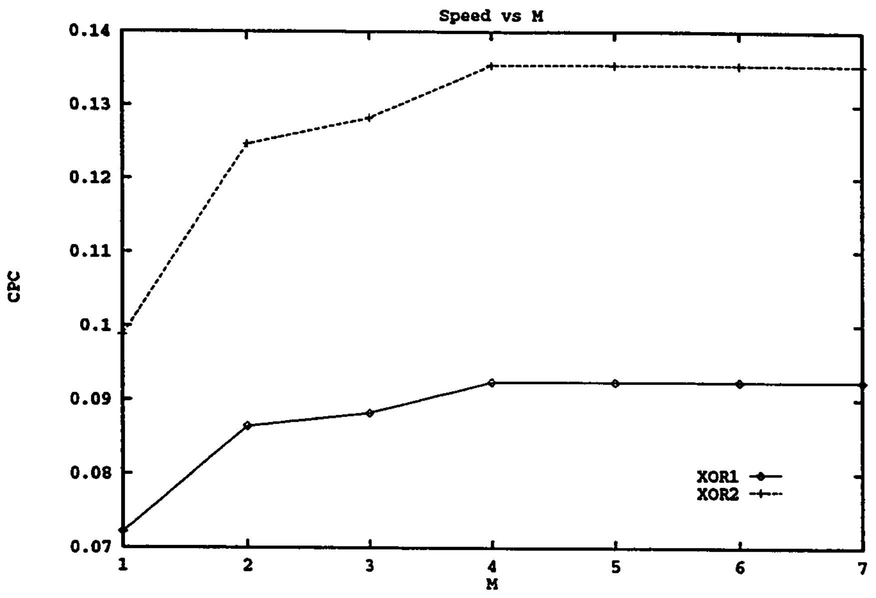


Figure 5.4: Speed (S) for XOR network.

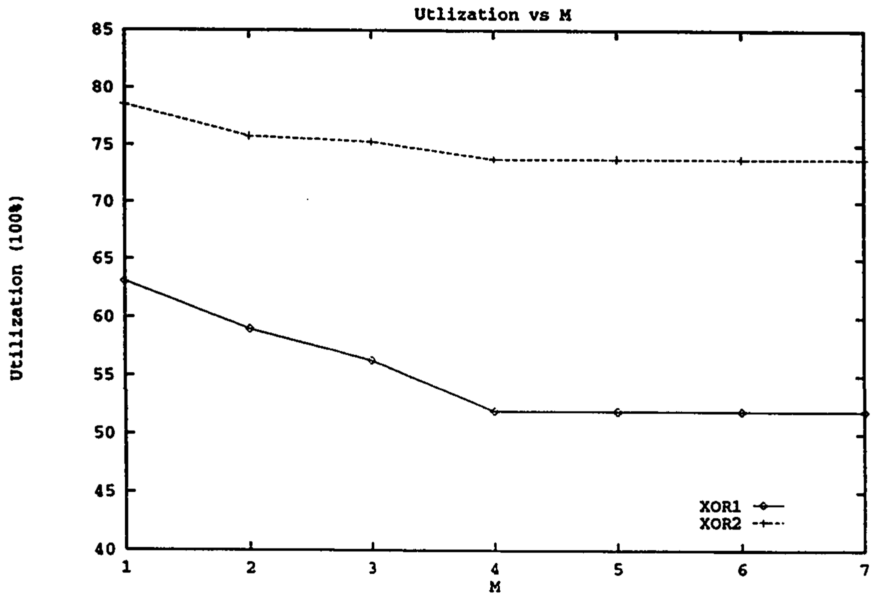


Figure 5.5: Utilization (U_t) for XOR network.

Learning phase

The learning phase of the network is transformed into DFG instructions (Figure 5.6). The speed is expected to be lower than the speed of the recall phase due to the weight data dependency as well as the critical path of the computations. The speed is found to be equal to 0.053 CUPC with $U = 1$, and $M = 8$. Learning of one pattern is assumed. The speed can be improved by allowing more than one training pattern to be used concurrently. The weight increments produced by a set of training pattern are added. Then weights are updated. In addition, more than one training session can be active at a time. In the case of four XOR networks with one training pattern each, the speed is increased to 0.1765 CUPC with $U = 2$. This gives about 4.4 MCUPS if 25 MHz clock frequency is assumed.

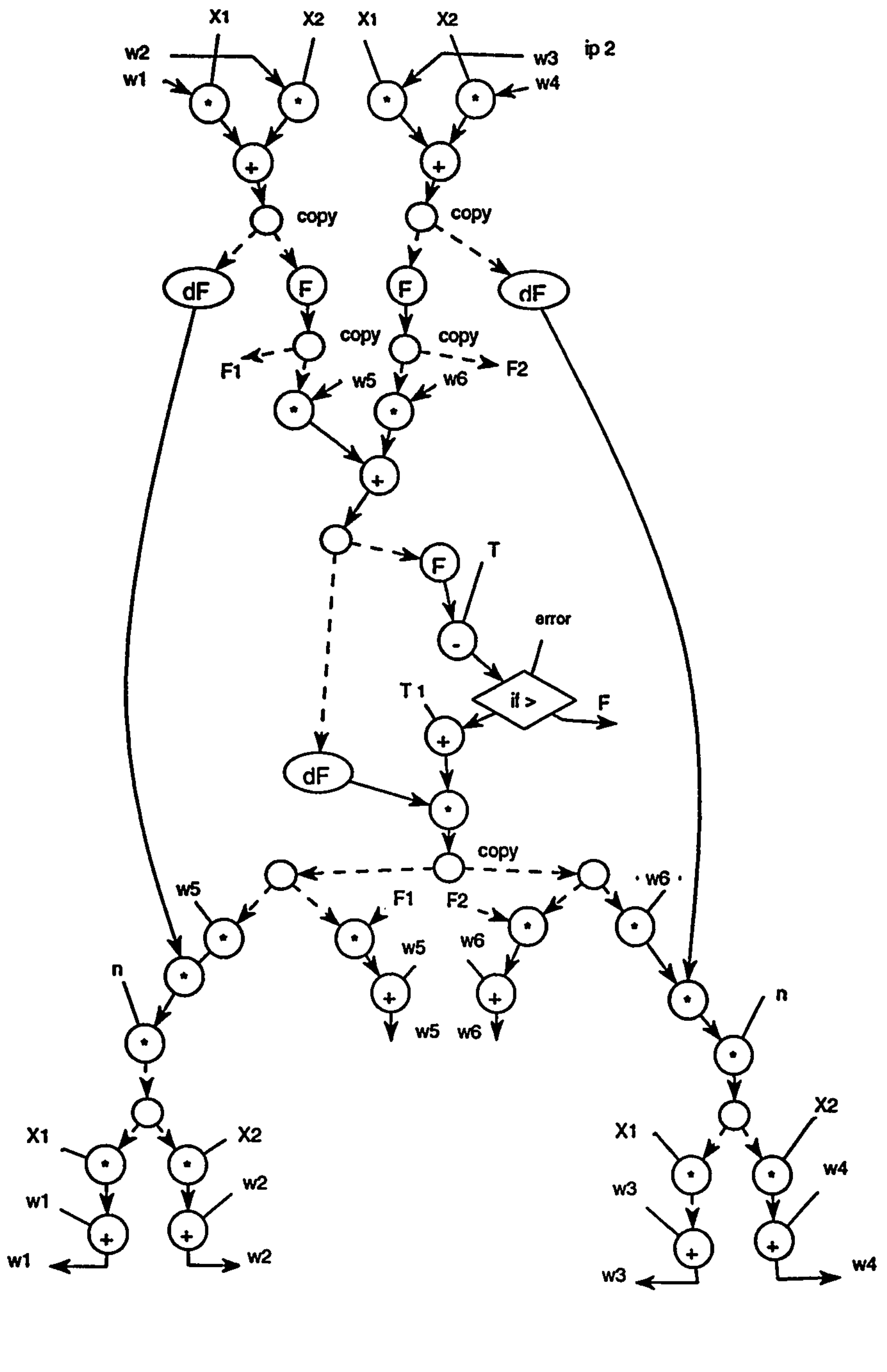


Figure 5.6: DFG of the learning phase of XOR network.

5.2.2 $BP_{4\times4\times4\times4\times4}$ network

In order to obtain realistic speed and utilization measures that reflect the true performance of the architecture, larger networks with more parallelism should be simulated. Three networks have been chosen for this purpose, one back propagation (BP) network with five layers $BP_{4\times4\times4\times4\times4}$, an image filter, and a Hopfield network of six neurons.

The purpose of the $BP_{4\times4\times4\times4\times4}$ test case is to obtain more realistic performance measures of the architecture. This network consists of five layers, including the input layer, with four neurons each. Three pipelined patterns per iteration are allowed through the use of fan-in points (Figure 5.7). The recall phase representation shows the capability for simulating different feed-forward networks with various connectivity patterns. The simulation results are shown in Table 5.2. Figures 5.8 and 5.9 show an alternative representation of the simulation results.

For different values of U, the speed (S) and the rate of supplied instructions (r) saturate with the maximum number of multipliers ($M = m = 8$). The speed (S) increases by about 5% when U is changed from 2 to 4 with $M = 8$.

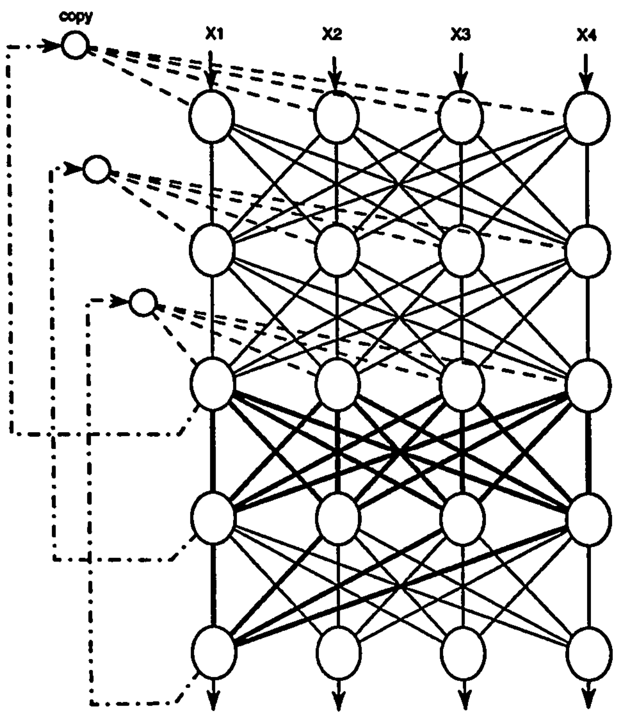


Figure 5.7: The recall phase of $BP_{4\times4\times4\times4\times4}$ network.

Table 5.2: The output measures for $BP_{4\times4\times4\times4\times4}$ network, T=30000.

U	1			2			3			4		
M	F	r	U_t	F	r	U_t	F	r	U_t	F	r	U_t
1	924	28.6	97.0	944	29.3	99.0	952	29.5	99.5	956	29.6	99.7
2	1732	53.5	95.0	1672	51.7	95.4	1712	52.9	97.3	1723	53.2	97.8
3	1808	55.8	83.0	2428	74.9	95.2	2364	73.0	95.5	2347	72.5	96.0
4	1808	55.9	73.8	2544	78.6	85.8	2611	80.6	87.8	2648	81.8	89.2
5	1808	55.9	72.7	2664	82.2	86.0	2746	84.8	88.0	2784	86.0	89.1
6	1808	55.8	71.1	2688	82.9	84.3	2784	85.9	86.2	2840	87.6	87.9
7	1808	55.8	69.6	2744	84.8	86.5	2840	87.7	87.7	2880	88.9	88.9
8	1808	55.8	69.5	2768	85.5	87.2	2888	89.1	89.1	2920	90.2	90.2
9	1808	558	69.5	2768	85.5	87.2	2888	89.1	89.1	2920	90.2	90.2

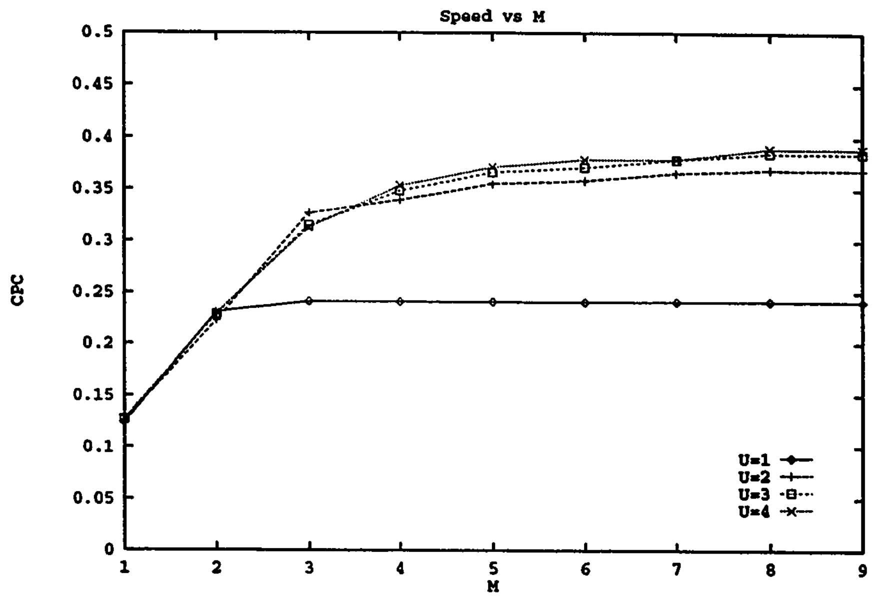


Figure 5.8: Speed (S) for $BP_{4\times4\times4\times4\times4}$ network.

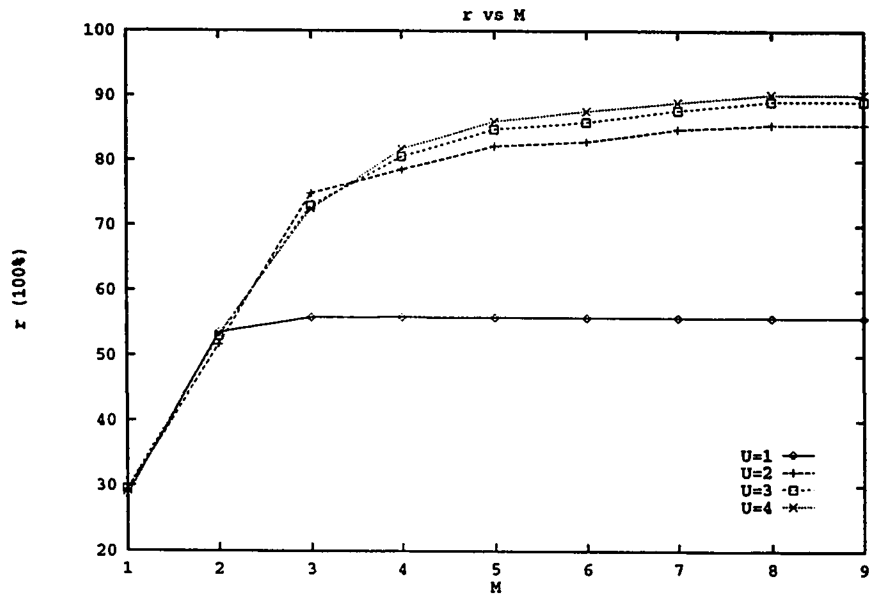


Figure 5.9: Rate of instructions (r) for $BP_{4\times4\times4\times4\times4}$ network.

5.2.3 An image filter network

Back propagation learns Marr's operator is a neural network model of the retinal responses to stimuli whose architecture is inspired by neurophysiological data [25]. With this model, a feed-forward network of an image filter that consists of four sub-networks, each having three layers ($16 \times 4 \times 1$) is simulated (Figure 5.10). The performance measures are listed in Table 5.3. Figures 5.11 and 5.12 show the effect of U and M on the performance. The size of the network is relatively large, consequently sufficient parallelism is offered and high processor utilization is achieved. The distributed parallel sub-networks reduce the cost on the scheduling unit ($U = 2$).

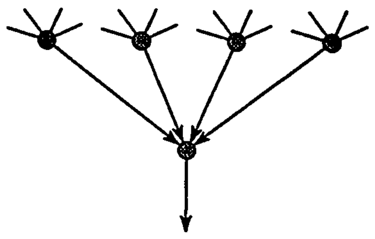


Figure 5.10: Connectivity of a single set of the image filter.

Table 5.3: The output measures for image filter network, T=30000.

U	1			2			3		
M	F	r	U_t	F	r	U_t	F	r	U_t
1	1053	39.1	99.8	1039	39.2	99.8	1039	39.2	99.9
2	1585	59.6	91.6	2064	77.5	99.9	2065	77.6	99.9
3	1585	59.6	77.6	2663	99.8	99.9	2665	99.8	99.9
4	1586	59.6	77.6	2664	99.9	99.9	2640	98.9	98.9
5	1586	59.6	77.5	2661	99.8	99.8	2657	99.7	99.7
6	1586	59.6	76.9	2661	99.8	99.9	2662	99.7	99.8
7	1586	59.6	76.1	2661	99.8	99.9	2662	99.7	99.8
8	1586	59.6	76.1	2661	99.8	99.9	2662	99.7	99.8
9	1586	59.6	76.1	2661	99.8	99.9	2662	99.7	99.8

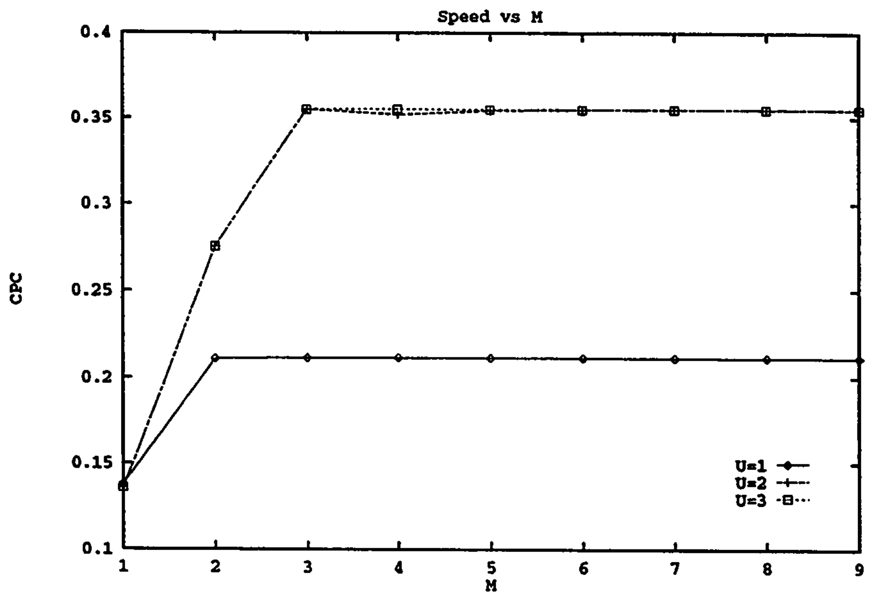


Figure 5.11: Speed (S) for image filter network.

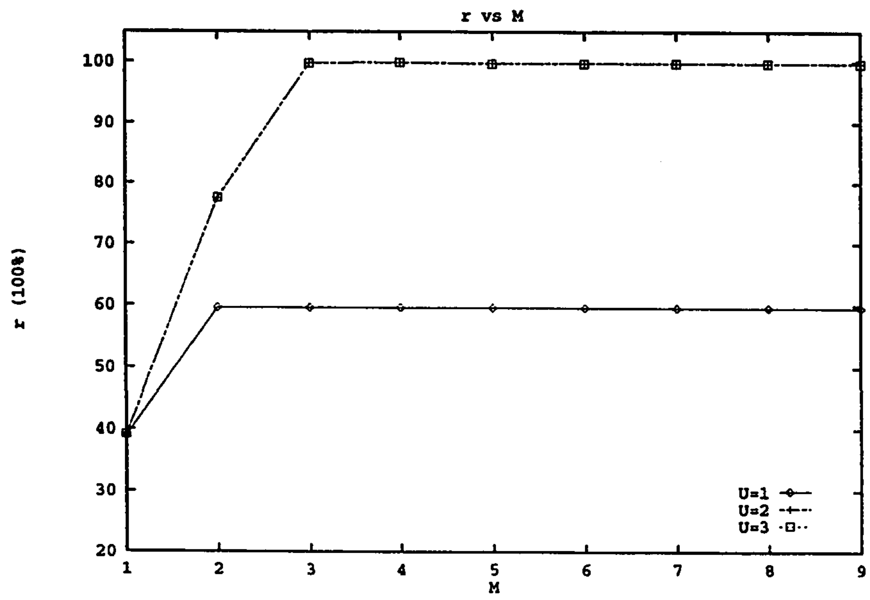


Figure 5.12: Rate of instructions (r) for image filter network.

5.3 Hopfield networks

Generality is an importatnt characteristic of the processor architecture. The architecture is flexible for different neural network models with various connectivity patterns. Hopfield networks are popular associative neural networks. In this section, the simulation of the parallel model of Hopfield networks is presented.

5.3.1 3-neuron Hopfield network

The parallel model of 3-neuron network is transformed into DFG instructions (Figure 5.13). The network simulation results are listed in Table 5.5 and shown in Figure 5.14 and Figure 5.15. The small size of the network does not reflect the actual performance of the architecture. Therefore, a larger Hopfield network of six neurons, as described in the next section is used.

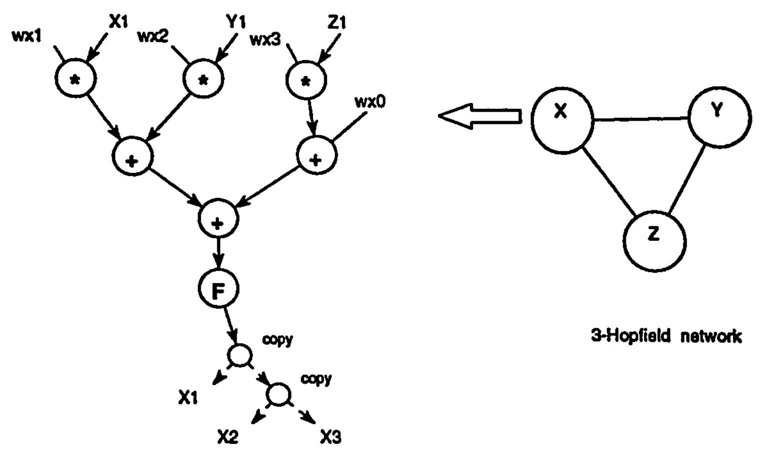


Figure 5.13: DFG of the 3-neuron Hopfield network.

Table 5.4: The output measures for 3-neuron Hopfield network, T=30000.

U	1			2			3			4		
M	F	r	U_t	F	r	U_t	F	r	U_t	F	r	U_t
1	1182	30.2	96.0	1199	30.7	97.3	1199	30.7	97.3	1199	30.7	97.3
2	1698	43.4	86.8	2043	52.3	81.8	1764	45.1	90.2	1764	45.1	90.2
3	1998	54.7	82.2	2043	52.3	81.8	1764	45.1	90.2	2091	53.5	53.5
4	2142	54.7	83.3	2250	57.5	82.5	2367	60.5	81.6	2367	60.5	81.6
5	2142	54.7	83.3	2430	62.2	83.8	2430	62.2	83.8	2430	62.2	83.8
6	2142	54.7	83.3	2499	63.8	83.3	2367	60.5	84.2	2367	62.2	83.8
7	2142	54.7	83.3	2398	61.3	82.7	2367	60.5	84.2	2367	62.5	84.2
8	2142	54.8	83.3	2398	61.3	82.7	2367	60.5	84.2	2367	62.5	84.2
9	2142	54.8	83.3	2398	61.3	82.7	2367	60.5	84.2	2367	62.5	84.2

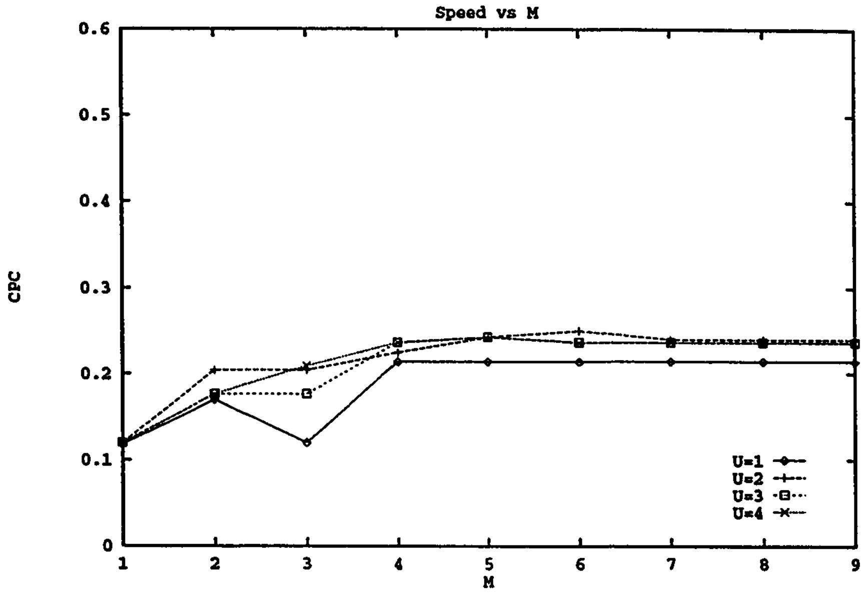


Figure 5.14: Speed (S) for 3-neuron Hopfield network.

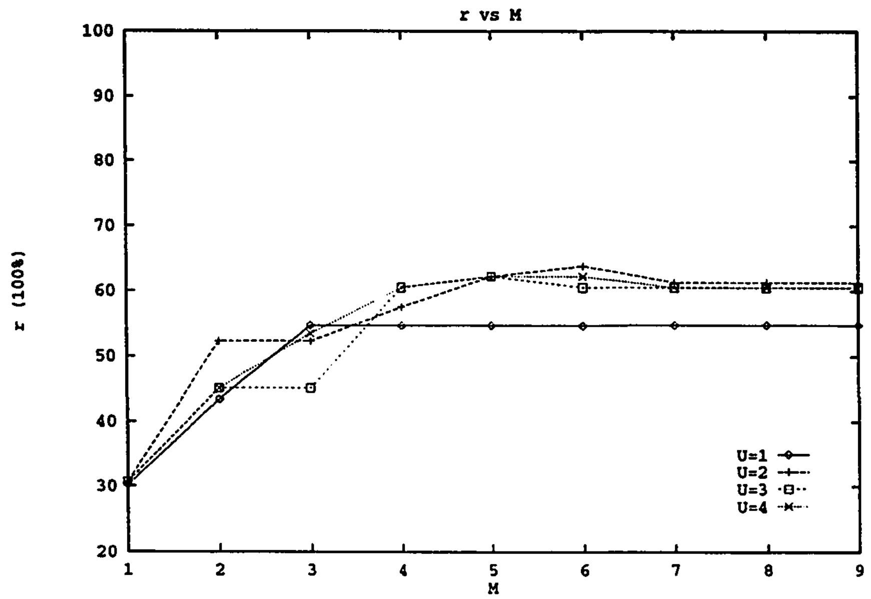


Figure 5.15: Rate of instructions (r) for 3-neuron Hopfield network.

5.3.2 6-neuron Hopfield network

This network is similar to the previous one but it has six neurons. As described earlier, the purpose is to simulate relatively larger networks in order to obtain true performance measures. The performance measures are listed in Table 5.5. Figures 5.16 and 5.17 show the effect of U and M on the performance.

The speed (S) and the rate of instructions (r) saturate with $M = 5$ when $U = 2$. The performance measures increase slightly when $U > 2$. This is due to the overhead effect of the control instructions on the update unit. The update unit with capacity of $U > 2$ is expected to supply more instructions, consequently r increases.

Table 5.5: The output measures for 6-neuron Hopfield network, T=30000.

U	1			2			3			4		
M	F	r	U_t	F	r	U_t	F	r	U_t	F	r	U_t
1	624	27.2	100	624	27.2	100	624	27.2	100	624	27.2	100
2	1248	54.1	99.8	1170	50.8	99.3	1170	50.8	50.8	1176	51.1	100
3	1302	56.5	89.7	1728	75.0	99.0	1551	67.4	99.1	1566	67.9	99.1
4	1302	56.5	82.9	1860	80.7	97.9	1884	81.9	97.2	1866	80.9	99.5
5	1302	56.5	82.1	1998	86.7	91.1	1963	85.3	91.8	1932	83.9	93.0
6	1302	56.5	83.9	1986	86.2	90.6	2071	87.2	87.2	1998	86.7	86.7
7	1302	56.5	83.9	1962	85.3	91.2	2052	89.2	89.1	2064	89.7	89.7
8	1302	56.5	83.9	1974	85.7	92.3	2112	91.8	91.8	2142	92.86	92.86
9	1302	56.5	83.9	1974	85.7	92.3	2112	91.8	91.8	2142	92.86	92.86

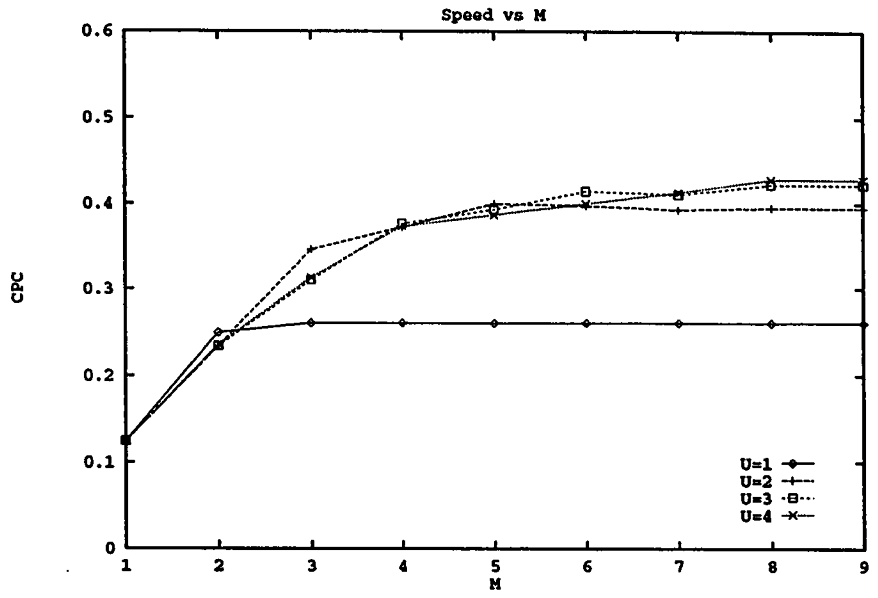


Figure 5.16: Speed (S) for 6-neuron Hopfield network.

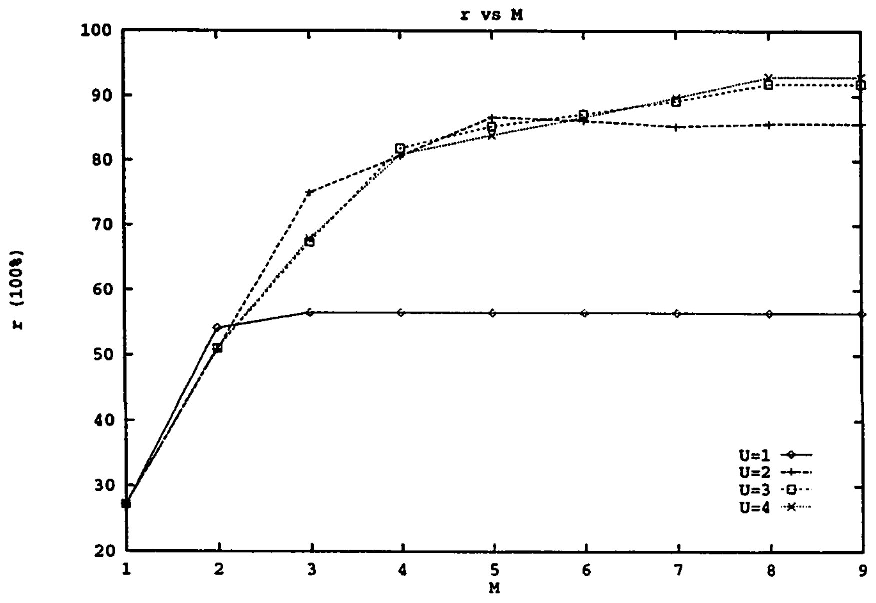


Figure 5.17: Rate of instructions (r) for 6-neuron Hopfield network.

5.4 Effect of asynchronous multiplication

In all test cases, the multiplication mode is varied in order to study the effect of the data-dependent asynchronous multipliers. The speed of the XOR learning phase increases by 8% when asynchronous multiplication is assumed over the fixed period multiplication.

However, the speed does not increase more than 1% with all other test cases as shown in Table 5.6. In such cases, the multiplication results are added to form a final linear summation (sum of products). Each add operation waits for the slowest of its two operands before execution. Thus, the time saved by the use of asynchronous multiplication is wasted in the consecutive add operations.

Table 5.6: The effect of asynchronous multipliers on the speed performance.

Test cases	Speed (F) Fixed mult. mode	Speed (F) Async. mult. mode
XOR (learning)	696	830
XOR (recall)	2021	2043
$BP_{4\times4\times4\times4}$ (recall)	1796	1804
6-Hopfield	1302	1302

5.5 Theoretical analysis

This analysis is based on relatively large neural networks, namely the $BP_{4\times4\times4\times4\times4}$, the 6-neuron Hopfield, and an image filter network.

Number of updates allowed per cycle (U):

The scheduling unit supplies a single stream of executable instructions at a rate of r. This rate (r) depends heavily on the capacity of the scheduling unit that is determined by the number of signal updates (U) allowed per cycle in the update unit. An important output of the study is to find out what is the minimum value of U that keeps the scheduling unit supplying executable instructions to the processing unit continuously or at a high rate.

From Figures 5.9, 5.12, and 5.17, it is seen that r increases considerably when U is changed from 1 to 2. The average signal density for program instruction (n) is defined as [16]:

$$S = \frac{total\ required\ signals}{total\ executed\ instructions} = \frac{\sum_{i=1}^{n}(a_1 + b_1)_i}{\sum_{i=1}^{n} exec(i)} \tag{5.2}$$

where,

a_1,b_1: two bits that represent the *reset* bits of the instruction.

$$exec(i) = \begin{cases} 1 & for\ executable\ instruction \\ 0 & else \end{cases} \tag{5.3}$$

The average signal update capacity for the test cases is less than two ($S \leq 2$) as shown in Table 5.7. Therefore, a scheduling unit with $U = 2$ theoretically should be sufficient.

Table 5.7: Average signal density for several networks.

Test cases	Average signal density (S)
XOR	32/17
$BP_{4\times4\times4\times4\times4}$	14/8
6-Hopfield	23/12
Image filter	91/56

Although the computations of the these networks are of the same type (sum of products), the networks differ in *connectivity*. As a result, the degree of parallelism offered by these networks varies. The image filter test case contains larger parallelism, therefore the performance measures such as r saturate with $U = 2$. There is no benefit in increasing U.

However, r increases by a small amount with $U = 3$, and 4. The special sequence of control instructions (copy and join) that are shown in the DFG's is a major cause of this increment. Since each copy instruction produces two signal addresses that are queued in the signal queue, the scheduling does not generate any executable instruction during this time. The copy instructions add an overhead to the update unit.

From Figures 5.8, 5.11, and 5.16, the parameter U has the same performance impact on the speed. The processor utilization (U_t) decreases as M increases. This is due to the increase of the service rate. However, the value of U_t never drops below the value of r as shown in Table 5.2 and Table 5.5. Full utilization can be achieved although parallelism is applied with $U = 2$ as shown in Table 5.3. Consequently, the parameter U plays a decisive factor in performance. An overhead to the update unit is introduced by the use of control instructions (copy and join).

Number of asynchronous multipliers (M):

As shown in Section 5.4, the effect of data-dependent asynchronous multipliers on these networks is negligible. *Connectivity* of the network has a major impact on the parameter M.

When a layer in the $BP_{4\times4\times4\times4\times4}$ is acknowledged, the acknowledgment signal fans out to 16 multiplications. The performance measures for the $BP_{4\times4\times4\times4\times4}$ require the maximum number of multipliers $M = m = 8$. For the 6-neuron Hopfield network, the neuron's output fans out to 6 multiplications letting the performance measures saturate with $M = 5$. On the other hand, three multipliers ($M = 3$) are enough to saturate the performance measures for the image filter test case. This is due to the special *connectivity* of the network. For different network's *connectivity* and different sizes of offered parallelism, the number of required multipliers M for a maximum multiplication service rate is determined by $M = m$. A fast multiplier can be a suitable alternative for the recall phase of back propagation and Hopfield networks.

5.5.1 Comparison to other machines

The simulation has shown a speed performance of 0.3948 CPC for the 6-neuron Hopfield network with $U = 2$. If it is assumed that the processor runs at 25 MHz (40ns clock cycle), the speed in connections per second is 0.3948 $CPC \times 25\ MHz$ = 9.87 $MCPS$. An approximate comparison of speed measure of neural network simulations on other dataflow architectures is presented in Table 5.8.

Table 5.8: Speed performance of neural network simulation on several dataflow architectures.

DataFlow machines	Speed
Qv-1 (16 PE)	11 MCUPS
Monsoon (1 PE)	2-3 MCPS
Proposed DFP (1 PE)	10 MCPS
MIT- tagged dataflow machine	?

The simulation of the 6-neuron Hopfield network on the proposed architecture is approximately 15 times faster than the network simulation on a Sun 10 machine. As mentioned earlier in the literature review, there are recent neural network hardware simulators that achieved a speed of several hundreds of $MCPS$ as shown in Table 5.9 [8]. The processor of such a system is powerful for neural network simulation. However, most of these systems are not highly scalable. An alternative approach is to simulate neural networks on highly and massively parallel systems with a large number of physical processors. One of the strong characteristics of the dataflow approach is the support for building highly and massively parallel systems. As described before, Goash and Hwang [17] have developed an asynchronous, message-passing model for highly and massively parallel systems in order to estimate the volume of interprocessor communication based on neural network demands. The study has been performed on different processor interconnection networks such as hypernets, hypercubes, and toruses.

Table 5.9: Speed performance for several non-datflow neural systems [8].

weights (bits)	activities (bits)	performance (10^9 CPS)	system - reference
1	1	5	[10]
4-8	low resol.	5	ETANN chip [20] Intel 80170NX
6	3	5	ANNA chip [14]
16	1	.01	Micro Devices MD1220 chip [15]
16	16	.8	Siemens MA16 chip [34]
8	9	1.9	48-chip wafer [29]
16	8	.1	SPERT [39]

5.6 Conclusion

In the proposed architecture, the scheduling unit supplies a single stream of fine grain scalar executable instructions to the processing unit at a high rate and low cost. Full processor utilization has been reached with low cost on the scheduling unit ($U = 2$) as shown in Figure 5.18. A scheduling unit capacity of $U = 2$ can be achieved by mapping the program, especially concurrent instructions into two or more memory blocks, each has an update rate of $U = 1$.

For the learning phase of back propagation networks, the effect of using asynchronous multipliers in the processing unit varies according to the nature of the program computations and the values of the data operands. However, the effect of data-dependent asynchronous multipliers on other test cases is negligible. The number of required multipliers M for a maximum multiplication service rate is determined by $M = m$.

Software pipelining has been used to expose more parallelism of neural networks through the use of the control instructions. The special connectivity, such as fan-in connections, found in neural networks ease software pipelining in exploiting different levels of parallelism, e.g. layer level.

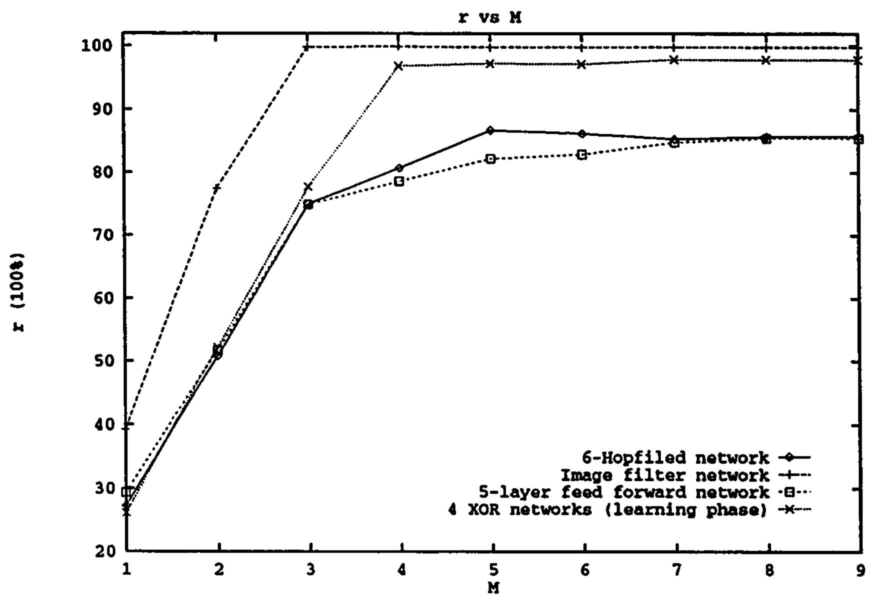


Figure 5.18: Rate of instructions (r) for different neural networks, $U = 2$.

Chapter 6

Conclusion and Recommendations

6.1 Conclusion

In this thesis, the computational and hardware requirements of neural networks have been studied. Review of current neural network implementations have been presented. The motivations for building dataflow systems for neural computing have been addressed: (1) neural networks offer *parallelism* at different levels for example layer and pattern levels, (2) they are suitable for *parallel* and *distributed systems* (a) neural networks can be mapped into a large number of processors and (b) *lo-*

cality of computations should be maintained and communication *latency* should be minimized, (3) the dataflow approach permits *generality* and simulation efficiency for a variety of neural network models including: (a) *Sparsely* and *fully connected* networks and (b) neural networks with *sparse activations.*

A new neural dataflow architecture based on argument-fetching principles has been proposed. Different levels of parallelism are exposed through the use of software pipelining. The architecture has been extensively studied and tested through simulation using several neural network examples with various parameters of the architecture. The simulation of proposed architecture shows good performance results. In the proposed architecture, the scheduling unit supplies a single stream of fine grain scalar executable instructions to the processing unit at a high rate and low cost. Full processor utilization has been reached with low cost on the scheduling unit ($U = 2$).

6.2 Recommendations

Two recommendations are presented in this section. The first is concerned with eliminating the overhead introduced by the control instructions on the scheduling unit. Next, The extension of the dataflow architecture to a hybrid architecture is discussed.

The signal addresses in the signal queue are fetched by the update unit in order to update the ***signal bits*** of both executable and control instructions. A sequence of copy instructions is used to signal a list of executable instructions; therefore they add an overhead to the update unit. This reduces the rate of supplying executable instructions to the processing unit. By eliminating the overhead introduced by the control instructions, the simulation of the test cases with $U = 2$ has achieved the same performance measures that were obtained previously with $U = 4$.

As a result, the control instructions should be stored in a separate block of memory, called control instruction memory, and handled separately. Basically, the signal addresses are divided into control and executable signal addresses (Figure 6.1). The control signal address is forwarded to a controller that deals with the control instruction memory, while the executable signal address is forwarded to the update unit which deals with the executable instructions, stored in the executable instruction memory. The first address in the signal queue is forwarded to either the update unit or the controller of the control instruction memory. When a controller receives an address for a copy instruction, it reads a list of executable signal addresses from the control instruction memory. Then, it supplies a list of executable signal addresses to the update unit at a rate of one executable signal address per cycle. The formats of these instructions can be modified as shown in Figure 6.2.

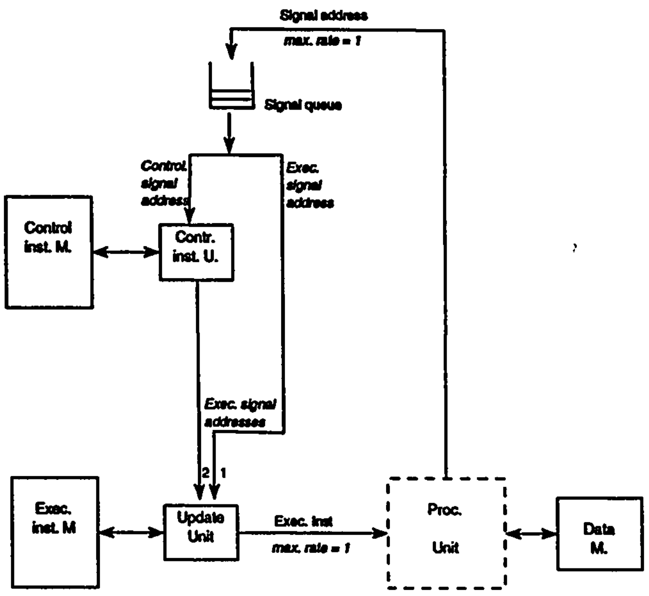


Figure 6.1: Recommended separation between control and executable instructions.

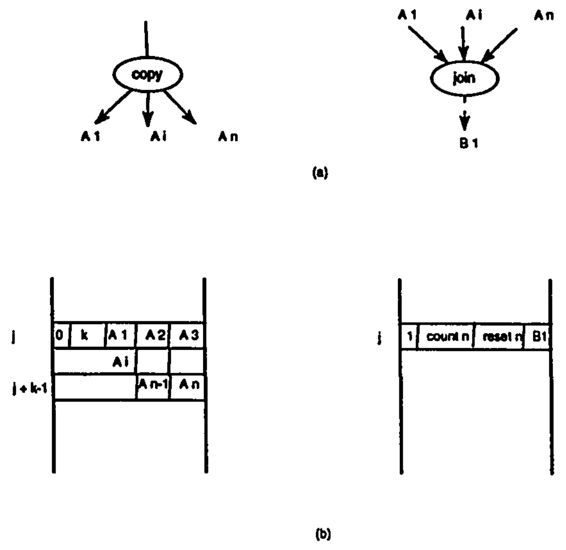


Figure 6.2: Recommended dataflow format of control instructions (a) and their corresponding machine format (b).

The scheduling unit supplies a pure fine grain instruction to the processing unit. The instruction can be considered as a *thread* to a fine grain instruction. In this thesis, the study emphasizes exploiting different levels of parallelism of neural networks, thus the architecture is pure dataflow. *Argument-fetching* dataflow models can be easily extended to hybrid models [16]. In a hybrid architecture, the *thread* may reference a sequence of instructions. Once the thread receives its n signals that determine the availability of n data operands, it is fired and the sequence of instructions are supplied to the processing unit in a pipeline manner and executed sequentially. The set of instructions can be considered as a macro dataflow graph node.

For example, a typical dataflow graph node found commonly in neural networks is shown in Figure 6.3. It represents a set of computations that fan-in to a final computation of the neuron output evaluation. There are two main advantages of executing sequentially such example in the hybrid architecture. First, large intermediate values that are introduced by the pure dataflow architecture are eliminated in the extended hybrid architecture. In this macro dataflow node example, 2^{n-1} intermediate values are eliminated. In addition, the number of signals required to execute the set of instructions in the macro dataflow node is $\frac{n}{2}$ instead of $\frac{3n}{2} + 1$ in the pure dataflow architecture. Consequently, the scheduling unit deals with less signals. The average signal ratio for an instruction in this example is reduced to

$S = 0.5$ instead of $S = 1.5$. The disadvantage of this approach is that no concurrency is allowed among the set of instructions within the dataflow graph node. Therefore, the extended architecture seems more suitable to neural networks.

Since the proposed architecture is a pure dataflow, large intermediate values have been introduced. Moreover, the static architecture limits the parallelism. The extension of the pure dataflow architecture to a hybrid one has been discussed. The hybrid architecture is efficient for fan-in computations. Such computations are commonly found in neuron evaluations.

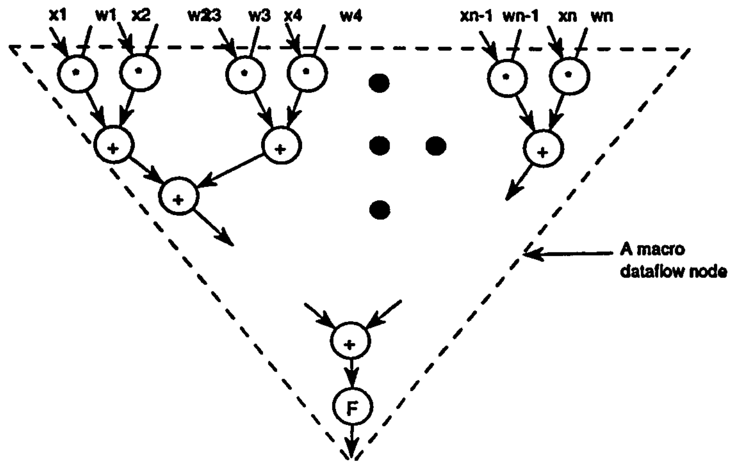


Figure 6.3: A typical macro dataflow graph node for neural computations.

6.3 Future work

The following points can be subject of future research:

- Extend the pure dataflow architecture model to a hybrid model, based on *argument-fetching principles.* NNs are naturally macro-dataflow graphs. The hybrid model is expected to show better performance with low cost of the scheduling unit.
- Explore dynamic dataflow architectures in order to maximize the parallelism offered by NNs.
- Extend the study of neural network simulation on highly and massively distributed dataflow parallel multiprocessors with large number of processors.
- Study other simulation aspects.

References

[1] Maher Abu-Mutlaq. *Simulator of a Neural Dataflow Processor: Technical Manual.* Computer Engineering Department, King Fahd University of Petroleum and Minerals, Dhahran, Saudia Arabia, 1995.

[2] A. Alhaj and H. Terada. A Data-driven Implementation of Back Propagation Learning Algorithm. In *Proceedings of the Intl. Conf. on AI.*, pages 588–593, 1992.

[3] A. Alhaj and H. Terada. Parallel Implementations of Back Propagation Networks on a Dynamic Data-Driven Multiprocessor. *IEICE Transactions on Information and Systems*, pages 579–588, 1994.

[4] Arvind and R. Iannucci. A Critique of Multiprocessing von Neumann Style. In *Proceedings of the Intl. Conf. on Computer Architecture ACM*, pages 426–436, 1983.

[5] P. Bessiere, A. Chams, A. Guerin, J. Herualt, C. Jutten, and J. C. Lawson. From Hardware To Software Designing A"Neurostatio". In *VLSI Design of Neural Networks*, page 314. Kluwer Academic Publishers, 1991.

[6] R. Braham. VLSI for Neural Computing. In *Proceedings of the 5th Intl. Conf. on Microelectronics*, pages 103–106, Dhahran, Saudia Arabia, 1993.

[7] R. Braham. Neural Computing Systems. *Les Annales Maghrebines de l'Ingenieur*, pages 49–56, April 1994.

[8] R. Braham. Performance and Precision of VLSI Neurocomputers. In *Proceedings of the first International Conference on Electronics, Circuits and Systems*, pages 582–585, Cairo, Egypt, 1994.

[9] R. Braham and J. Hamblen. The Design of a Neural Network with a Biologically Motivated Architecture. *IEEE Transactions on Neural Networks*, pages 251–261, September 1990.

[10] U. Cilingiroglu. A Purely Capacitive Synaptic Matrix for Fixed-weight Neural Networks. *IEEE Trans. Circuits and Systems*, pages 210–217, February 1991.

[11] S. Dasgupta. *Computer Architecture: A Modern Synthesis*, volume 2: Advanced topics. Wiley, 1989.

[12] J. B. Dennis and G. R. Gao. An Efficient Pipelined Dataflow Architecture. In *Proceedings SUPERCOMPUTING'88*, pages 368–373, 1988.

[13] R. C. Eberthart. Standardization of Neural Network Terminology. *IEEE Transactions on Neural Networks*, pages 244–245, June 1990.

[14] B.E. Boser et al. Hardware Requirements for Neural Network Pattern Classifiers. *IEEE Micro*, pages 32–40, February 1991.

[15] J. Yestrebsky et al. Neural Bit Slice Computing Element. In *Proceedings of Summer Comp. Sim. Conf.*, pages 787–790, 1990.

[16] J. Gaudiot and L. Bic. *Advanced Topics In Data-Flow Computing.* Prentice Hall, 1991.

[17] J. Ghosh and K. Hwang. Mapping Neural Networks onto Message-Passing Multicomputers. *Journal of Parallel and Distributed Computing*, pages 291–330, 1989.

[18] D. Hammerstrom. Neural Networks at Work. *IEEE spectrum*, pages 26–32, June 1993.

[19] R. Hecht-Nielson. *Neurocomputing.* Addison Wesely Pub. Co., 1990.

[20] M. Holler, S. Tam, H. Castro, and R. Benson. An Electrically Trainable Artificial Neural Network (ETANN) with 10240 "Floating Gat" Synapses. In *Proceedings of the Intl. Joint Conf. on Neural Networks*, pages 191–196, 1989.

[21] J. J. Hopfield. Neural Networks and Physical Systems with Emergent Collective Computational Abilities. In *Proceedings of the National Academy of Science*, pages 2554–2558, 1982.

[22] J. J. Hopfield. Neurons with Graded Response Have Collective Computational Properties like Those of Two-state Neurons. In *Proceedings of the National Academy of Science*, pages 3088–2092, 1984.

[23] K. Hwang and F. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, 1985.

[24] M. James and D. Hoang. Design of Low-cost, Real-Time Simulation Systems for Large Neural Networks. *Journal of Parallel and Distributed Computing*, pages 221–235, 1992.

[25] A. Joshi and C. Lee. Backpropagation Learns Marr's Operator. *Biological Cybernetics*, pages 65–73, 1993.

[26] S. T. Kim, K. Suwunboriruska, S. Herath, A. Jayasumana, and J. Herath. Algorithmic Transformation for Neural Computing and Performance of Supervised Learning on a Dataflow Machine. *IEEE Transactions on Software Engineering*, pages 613–623, July 1992.

[27] S. Y. Kung. *Digital Neural Networks*. Prentice Hall, 1993.

[28] A. M. Law and W. D. Kelton. *Simulation Modelling and Analysis*. McGraw-Hill, 1991.

[29] A. Masaki, M. Yamada, and Y. Hiraai. Neural Networks in CMOS: A Case Study. *IEEE Circuits and Devices*, pages 12–17, July 1990.

[30] M. Melton, T. Phan, D. Reeves, and V. D. Bout. TInMANN VLSI Chip. *IEEE Transactions on Neural Networks*, pages 375–384, May 1992.

[31] D. J. Myers and R. A. Huntchinson. Efficient Implementation of Piecewise Linear Activation Function For Digital VLSI Neural Networks. *Electronics Letters*, pages 24–25, November 1989.

[32] T. Nordstrom and B. Svensson. Using and Designing Massively Parallel Computers for Artificial Neural Networks. *Journal of Parallel and Distributed Computing*, pages 260–285, 1992.

[33] U. Ramacher, J. Beichter, W. Reeb, J. Anlauf, N., Bruls, U. Hachmann, and M. Wesseling. Design of A 1st Generation Neurocomputers. In *VLSI Design of Neural Networks*, pages 272–317. Kluwer Academic Publishers, 1991.

[34] U. Ramacher, U. Hacmann, W. Raeb, R. Bruls, J. Anlaut, M. Webeling, J. Beichter, and E. Sichender. Multiprocessor and Memory Architecture of the Neurocomputer Synapse-1. In *Proceedings of the 5th Intl. Conf. on Microelectronics*, pages 98–112, Dhahran, Saudia Arabia, 1993.

[35] U. Ramacher and U. Rucket. Guide Lines to VLSI Design of Neural Nets. In *VLSI Design of Neural Networks*, pages 1–17. Kluwer Academic Publishers, 1991.

[36] D. E. Rumelhart, J. E. Hinton, and R. J. Williams. Learning Internal Representations by Error Propagation. In *Parallel Distributed Processing: Explorations in the Microstructures of Cognitions.* MIT Press, 1986.

[37] I. G. Smotroff. Data Flow Architectures: Flexible Platforms for Neural Network Simulation. In D. S. Touretzky, editor, *Advances in Neural Information Processing*, pages 818–823. Morgan Kaufmann, 1990.

[38] J. M. Vincent. Finite Word length, Integer Arithmetic Multilayer Perceptron Modelling for Hardware Realization. In R. Linggard, D. Mayers, and C. Nightingale, editors, *Neural networks for Vision, Speech and Natural Languages*, pages 293–311. Chapman & Hall, 1992.

[39] J. Wawrzynek, K. Asnovic, and N. Morgan. The Design of a Neuro-Microprocessor. *IEEE Transactions on Neural Networks*, pages 394–398, May 1993.

Vita

- Maher Hamdan Abu-Mutlaq
- Born in 1970 in Riyadh, Saudi Arabia.
- Received the Bachelor of Science degree in Computer Engineering in 1992 from King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia.
- Received the Master of Science degree in Computer Science in 1995 from King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia.