A HANDEL-C IMPLEMENTATION OF THE BACK-PROPAGATION ALGORITHM ON FIELD PROGRAMMABLE GATE ARRAYS

A Thesis

Presented to

The Faculty of Graduate Studies

of

The University of Guelph

by

VIJAY PANDYA

In partial fulfilment of requirements

for the degree of

Masters of Science

August, 2005

ABSTRACT

A HANDEL-C IMPLEMENTATION OF THE BACK-PROPAGATION ALGORITHM ON FIELD PROGRAMMABLE GATE ARRAYS

Vijay Pandya	Advisor:
University of Guelph, 2005	Dr.Medhat Moussa & Dr. Shawki Areibi

The Back-Propagation (BP) algorithm for building an Artificial Neural Network (ANN) has been increasingly popular since its advent in the late 80's. The regular structure and broad field of application of the BP algorithm have drawn researchers' attention at attempting a time-efficient implementation. General Purpose Processors (GPP) and Application Specific Integrated Circuits (ASICs) are the common computing platforms to build an ANN based on the BP algorithm. However such computing machines suffer from the constant need of establishing a trade-off between flexibility and performance. In the last decade or so there has been significant progress in the development of special kind of hardware, a reconfigurable platform based on Field Programmable Arrays (FPGAs). FPGAs are shown to exhibit excellent flexibility in terms of reprogramming the same hardware and at the same time achieving good performance by allowing parallel computation.

The research described in this thesis proposes several partially parallel architectures and a fully parallel architecture to realize the BP algorithm on an FPGA. The proposed designs are coded in Handel-C and verified for its functionality by synthesizing the system on Virtex2000e FPGA chip. The validation of the designs are carried out on several benchmarks. The partially parallel architectures and the fully parallel architecture are found to be 2.25 and 4 times faster than the software implementation respectively.

Acknowledgements

I would like to take this opportunity to express my sincere gratitude to my advisers Dr. Medhat Moussa and Dr. Shawki Areibi for providing me with invaluable guidance throughout the research. Without their patience, constructive criticism and help this work would have not been possible.

Special thanks goes to my family, sister and my wife who have not only shown great patience but supported and stood by me in difficult times.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

An Artificial Neural Network (ANN) is an information-processing system based on generalization of the human-brain biology. A human brain is an extremely powerful biological computer that is capable of processing huge amount of information in parallel. This can be inferred from observing the ease by which a human brain performs tasks such as reading a hand-written note, recognizing a face and listening to music, all simultaneously. A human brain achieves such multi-tasking by having a large number of biological processing elements, called neurons. Neurons are interconnected through connection-links called synapses to transmit and receive signals. A typical human brain consists of 100 billion neurons [http] and an average 1000 to 2000 synapses per neuron. Such biological architecture in a human brain makes up a complex neural network (NN).

An ANN is a natural attempt at realizing multi-processor system similar to the biological neural network. However, the human brain is order of magnitude more complex than any ANN implemented so far. The building block of an ANN is the artificial neuron or processing element. The network is composed of highly interconnected artificial neurons by synaptic weights and it performs useful computations through a process of learning. Such an architecture is usually implemented using electronic components or simulated in software on a digital computer. The learning process involves adjusting the magnitude of signals passing through synapses. Such process, also called training, gives ANN an 'experience' by storing and processing information. One of the most common methods of training an ANN is the Error-Backpropagation Algorithm [Rume86].

The importance of the ANN can be realized from its remarkable ability to derive meaningful information from complicated or imprecise data and extract patterns and detect trends. A trained ANN can be considered an 'expert' in the category of information it has been given to analyze. Some of the basic advantages include:

- Adaptive learning: An ability to learn based on 'experience'.
- Self-Organization: An ANN can create its own representation of the information.
- Parallelism: An ANN is an inherently parallel processing system and hence special hardware devices can be designed to exploit this characteristic.

An important limitation of using ANNs in various applications is the lack of clear methodology to determine the network parameters before training starts [Nich03]. Also, the speed by which training can be carried out is slow for the Back-Propagation algorithm [Hayk99]. Hence, for many real-time applications, ANNs would not be a suitable computing platform. The lack of performance of ANNs in terms of speed is largely attributed to its implementation on conventional computers. A conventional computer is a sequential computing machine and hence is unable to exploit the inherent parallelism of ANNs. All these limitations will set up a foundation for the motivation behind the research work carried out in this thesis.

1.1 Motivation and Objectives

ANNs have been successfully used in a wide variety of applications. Some of the applications include: speech recognition, image processing, signal processing, communication financial prediction and control system design. The Broad field of application demands a computing platform which can efficiently compute ANNs and at the same time retain flexibility to accommodate various topologies. In other words, there is a constant need of establishing a trade-off between performance and flexibility in computation for ANNs. The emergence of reconfigurable hardware, Field Programmable Gate Array (FPGA), has promised to attend such need and has been establishing a cost-effective computing platform for many applications.

1.1.1 Reconfigurable Computing

Currently a computing machine can be implemented on one of two platforms: General Purpose Processor (GPP) and Application Specific Integrated Circuits (ASICs). GPP can execute any set of instructions to perform a computation. By changing the software instructions we can implement different algorithms without changing the hardware. This imparts excellent flexibility for various applications. For example, personal computers can run a range of applications like word processing, spreadsheet data application, graphics and image related application as well as highly computational intensive CAD tool applications all using the same processor. The downside of this flexibility is that performance must be tuned for a target set of programs and as such may not be optimum for any single program. The second platform ASICs are custom-made computing solutions for specific applications. They are very fast and efficient when executing the exact computation for which they are designed [Comp00]. Examples include custom designed graphic processors to speed up image processing applications, high speed wireless networking chips and power ICs to name a few. ASIC design for hardware realization of any kind of circuit has good performance and proves cost-effective when fabricated in large volume. The main disadvantage of this approach is that minor design changes require costly rebuilding of new chips. Thus the resources needed to design and synthesize the circuit would not be justified in case of small production volume of ASIC chips. ASIC is custom designed for target application and it has excellent performance but lacks flexibility due to the aforementioned problem. Hence it can be said that GPPs have higher flexibility but poor performance for specific application whereas ASICs have higher performance for target application but poor flexibility for general-purpose applications.

There is also another problem associated with GPP as well as with some ASICs having own instructions set. They exhibit sequential nature in processing applications. This is because a processor has to fetch each instruction from memory, decode it's meaning and execute it to perform a task. This serial execution of operation doesn't efficiently utilize the whole hardware and hence an individual operation has high overheads.

To circumvent these problems, researchers in hardware design started to focus more on special kind of platform called "Reconfigurable Hardware" a decade ago to implement computational operation. Reconfigurable hardware refers to large scale Complex Programmable Logic Device (CPLD) or Field Programmable Gate Array (FPGA) chips. Reconfigurable hardware promises an intermediate trade-off between flexibility and performance. Reconfigurable computing utilizes hardware that can be customized at run-time to facilitate greater flexibility without compromising performance [Bond00].

An FPGA provides a means to implement general-purpose applications on customized hardware in a cost-effective way. This will ultimately achieve potentially higher performance than software while maintaining a higher level of flexibility than hardware. Another striking characteristic of an FPGA is its ability to exploit the parallelism that any application may have. This is because FPGAs can perform computations in parallel using many active computing elements simultaneously, rather than sequentially reusing a small number of computing elements, it achieves higher throughput [Deho00]. The preceding discussion about performance and flexibility of various computing machines can be shown in Figure 1.1.

1.1.2 Objectives

The objective of this thesis is to investigate a trade-off between performance and flexibility in implementing the Back-Propagation (BP) algorithm of ANN on reconfigurable hardware FPGA. Since an FPGA is capable of exploiting the inherent parallelism of ANNs, the main objective will be to design a fully-parallel architecture and validate it for various benchmarks. To achieve this goal the following

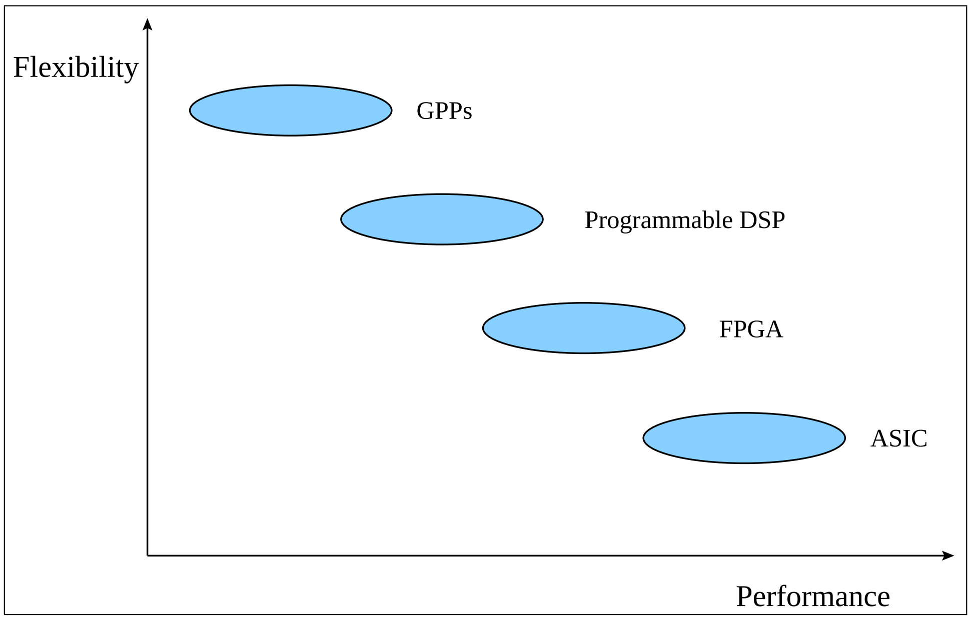


Figure 1.1: Performance Vs. Flexibility for various computing machines

objectives are identified:

- Perform detail review of literature and analyze various approaches for the implementation of ANNs on hardware.
- Get acquainted with Hardware Programming Language (Handel-C) and reconfigurable hardware platform such as RC1000 board with Virtex1000/Virtex2000e FPGA chip.
- Generate sequential architectures of ANNs to be implemented on FPGAs and evaluate/verify the architectures using several benchmarks.
- Generate partially parallel and fully-parallel architectures of ANNs and evaluate the performance in terms of area and performance.

Handel-C is a high level language based on ANSI C for the implementation of algorithms on hardware [Celo03]. It allows rapid development of multi-million gate FPGA designs and system-on-chip solution compare to high development time required in VHDL. Handel-C allows software engineers to design hardware without much training. They can quickly translate a software algorithm into hardware without having to learn about FPGA in detail [Mart02]. For example, a designer of Handel-C doesn't need to explicitly design state machines to synchronize the sequence of operations. Also, Handel-C automatically deals with clock which abstracts away much of the complexity of hardware design. The compilation and debugging time is very low in Handel-C compared to VHDL. However [SCoe04] has reported that Handel-C design uses more resources on an FPGA and takes more time to execute than a VHDL design. Figure 1.2 represents the overall design approach followed in this research.

1.2 Contributions

The following are the architectures developed to achieve the goal of implementing the BP algorithm on FPGA for the research work.

- Two Serial Designs: SIPEX and SIPOB[1] developed, based on the storage of the initial parameters
- Three Partially Parallel Designs: PARIO, PAROO and PARIO developed, based on the variety in the computation of Multiply-Accumulate for each

[1]Acronyms are described in Appendix A

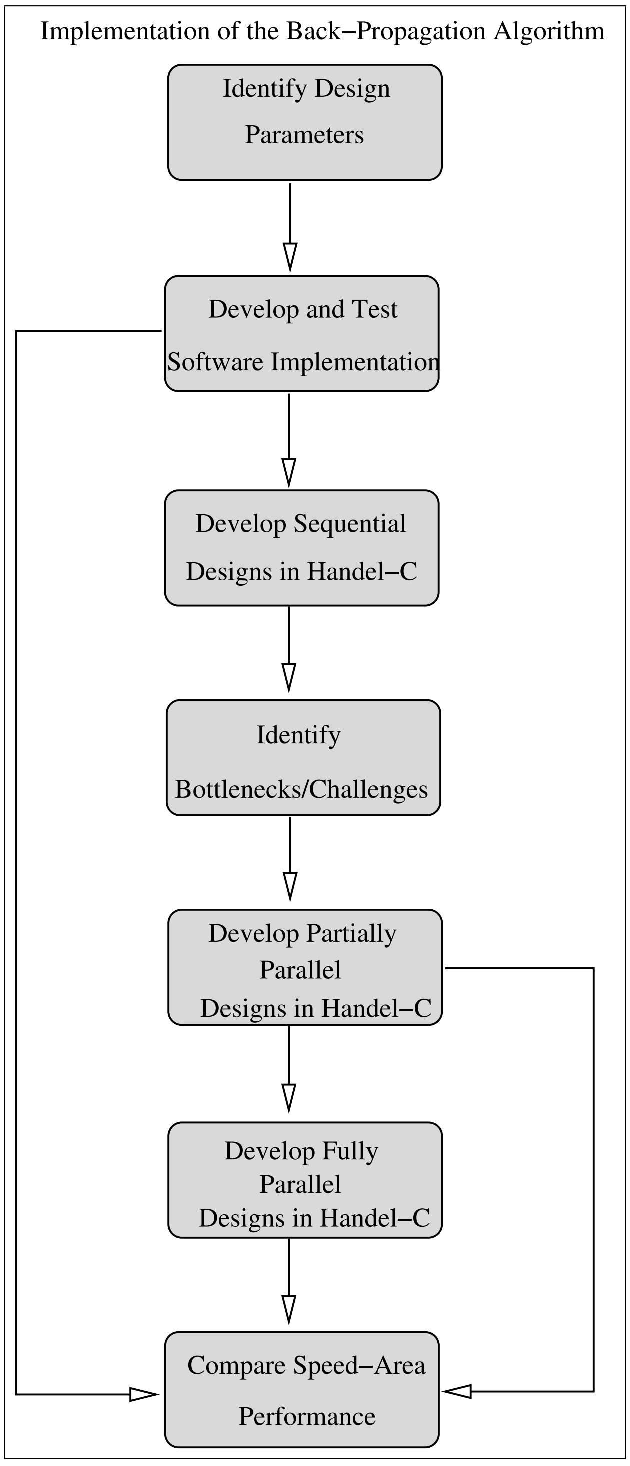

Implementation of the Back–Propagation Algorithm
Identify Design Parameters
Develop and Test Software Implementation
Develop Sequential Designs in Handel–C
Identify Bottlenecks/Challenges
Develop Partially Parallel Designs in Handel–C
Develop Fully Parallel Designs in Handel–C
Compare Speed–Area Performance

Figure 1.2: Overall Design Approach

stage of the algorithm

- A Fully Parallel Design: FPAR was developed to achieve high performance
- Publication of the above mentioned research in the International Conference on Reconfigurable Computing and FPGAs, ReConFig'05 [VP05]

All of the above architectures are validated for three benchmarks:

- XOR data set: A smaller benchmark (toy problem) with network size of (3-3-1)
- Iris data set: A comparatively medium benchmark (toy problem) with network size of (5-3-3)
- Cancer data set: A larger benchmark (Real-World problem) with network size of (10-11-2)

The FPAR architecture was not validated for the Cancer data set because of the large hardware resources required due to synthesis. One of the important contribution of the research work is the development of clear methodology for designing partially and fully parallel architectures. Such methodology will provide basic foundation on which future work can be based to further explore implementations of ANNs.

Various results are obtained by successfully implementing the above mentioned architecture on Virtex2000E FPGA chip. The results are analyzed for performance based on the Weight Updates per Second (WUPS), the speed and the gate counts. The software runs for the BP algorithm for the same benchmarks are carried out on Dual-Processor PIII 800MHz PC.

1.3 Thesis Outline

The rest of the thesis is organized as following:

Background describes the basics of reconfigurable computing and such platform consisting of FPGA. It also describes the basics of Artificial Neural Networks (ANN) and the BP algorithm.

Literature Review reviews the various implementations of ANNs on hardware. The chapter is divided into the ASIC and FPGA implementations of ANNs. The section on FPGA implementation of ANNs is further divided to focus more on the BP algorithm realization.

Experimental Setup describes various system specifications related to hardware and software designs. It also describes the benchmarks used to validate the hardware and software implementations of the BP algorithm.

Serial Architectures proposes several serial architectures for the BP algorithm. The results obtained by validating the architectures for various benchmarks are analyzed in detail. It also explains the Lookup Table (LT) approach for the implementation of the sigmoid function.

Parallel Architectures proposes several partially parallel and a fully parallel architectures for the BP algorithm. It introduces the concept of Branch-In and Branch-Out mode of multiply-accumulate. A detail analysis is given for the results obtained by validating the benchmarks.

Conclusion and Future Work concludes the thesis by describing the contributions made through the research work and some directions on the future work.

Chapter 2

Background

This chapter will shed some light on the background related to Artificial Neural Networks (ANNs) and reconfigurable computing platform in the form of Field Programmable Gate Arrays (FPGAs). The basics of the Back-Propagation algorithm will also be described in this chapter and the mathematical formula will be discussed in detail.

2.1 Field Programmable Gate Array

FPGAs are integrated circuits that consist of arrays of configuration logic blocks (CLBs) with an interconnection network of wires. Both the logic blocks and the interconnection networks are configurable [Bond01]. These devices can be configured and re-configured by the system designer through CAD tools, rather than by the chip manufacturer [Xili93].

Evolution of FPGAs resembles much more with memory structures. The fuse based FPGAs came to existence first and it was analogous to PROM structure. Other fuse based structure followed were Programmable Logic Array (PLA) and Programmable Array Logic (PAL). Generally all these fuse based structures are classified under the common term of Programmable Logic Device (PLD). Some of the limitations associated with PLDs caused an emergence of RAM-based (volatile) FPGAs. Currently available FPGAs are of this type in which programming a RAM bits of CLB, a configuration can be achieved. FPGAs started to become available in the late 80's and in early 90's. Since then researchers/designers began to see reconfigurable computing engines enabled by FPGAs. While reconfigurable architectures have only recently begun to show significant application viability, the basic ideas have been around almost as long as the idea of programmable general-purpose computing. In 1966 Jon Von Neuman, who is generally credited with developing the conventional model for serial, programmable computing, also envisioned spatial computing automata- a grid of simple cellular, building blocks which could be configured to perform computational tasks [Deho96]. Minnick in 1971 reported a programmable cellular array using flip-flops to hold configuration context. The turning point for reconfigurable hardware realized into an FPGA appeared when it was possible to place hundreds of programmable elements on a single chip [Deho96]. As Moore's law states: "the number of transistors on an integrated circuit would double approximately every 18 months", today it is possible to see FPGA with more than 10 million gates on a single chip.

2.1.1 Structure of FPGA

Most current FPGAs are SRAM-based. This means that SRAM bits are connected to the configuration points in the FPGA, and programming the SRAM bits will tend to configure the FPGA. A typical FPGA has a logic block with one 4-input LUT (Look up table), an optional D flip flop, and some form of fast carry logic. The LUTs allow any function to be implemented, providing generic logic. The flip-flop can be used for pipelining, registers, state holding functions for finite state machines, or any other situation where clocking is required [Comp00]. Figure 2.1 shows the basic structure of a logic block of an FPGA.

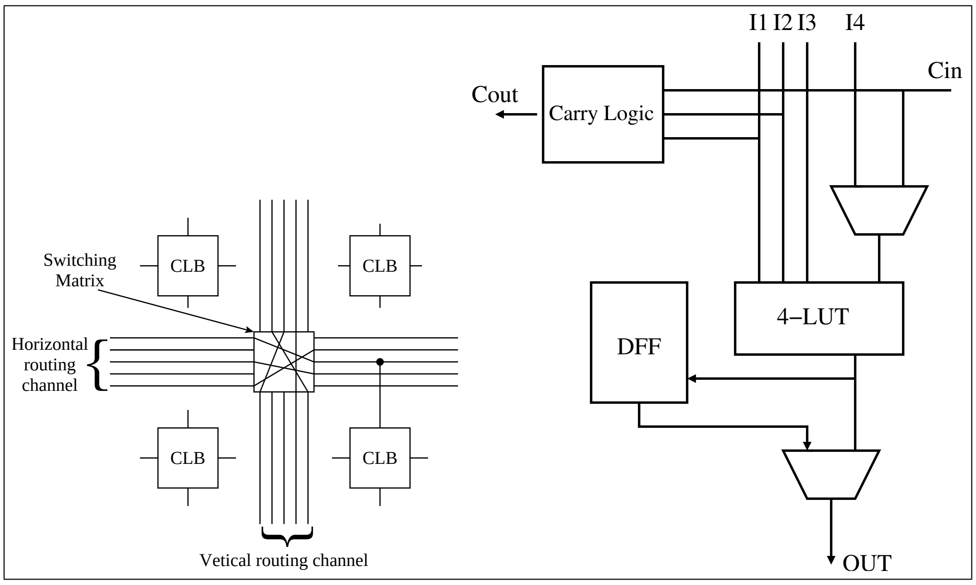


Figure 2.1: Interconnection network in FPGA and a basic logic block, with 4-input LUT, carry chain and D-type flip-flop

2.1.2 Coupling and configuration approaches for FPGA

Reconfigurable systems are usually formed with a combination of reconfigurable logic and a general-purpose microprocessor. The processor performs the operations that cannot be done efficiently in the reconfigurable logic, such as loops, branches, and possibly memory accesses [Comp00]. There are three different approaches for FPGA to function in combination with the host processor. In the first approach, a reconfigurable unit may be used as a coprocessor. In this case the coprocessor performs computations without the constant supervision of a host processor. The host processor initializes the reconfigurable hardware and sends necessary data for computation and coprocessor works on it independently [Comp00]. Such integration allows in most case for both host and reconfigurable hardware to function simultaneously due to less overhead of communication. This approach based on combining a reconfigurable hardware to the host processor is called 'tightly' coupled reconfigurable system. In the second approach, a reconfigurable unit can be used as an attached processing unit. This is a less tighter configuration compared to the first one. In this approach, host processor's memory cache is not visible to the attached reconfigurable unit. There is, therefore, a higher delay in communication between the host processor and the reconfigurable hardware, such as when communicating configuration information, input data and results [Comp00]. However such configuration allows for higher independence for computation as more CPU bound operations can be shifted to the attached hardware. The third approach, the most 'loosely' coupled form of combination, is an external stand alone reconfigurable hardware. This type of hardware communicates infrequently with the host processor. Each of these approaches has its advantages and disadvantages. The tighter the integration of reconfigurable hardware with the host processor the more it can be used in application with less overhead of communication. However it can not operate independently for significant amount of time without an interruption from the host. In addition, the amount of reconfigurable logic available in 'tightly' coupled form of hardware is often quite small. The more loosely coupled styles allow for greater parallelism in program execution, but suffer from higher communication overhead [Comp00]. This form of coupling is well suited for application in which a portion of computation can be done in reconfigurable hardware for long period of time without a great deal of communication. For a system in which there is a tight coupling between host and reconfigurable hardware, it is extremely necessary to divide the tasks to be performed in software and hardware. Such method of implementation falls into hardware-software co-design for computation.

There are two different approaches for implementing reconfigurable logic on FPGA chips. These are widely known as Compile Time Reconfiguration (CTR) and Run Time Reconfiguration (RTR). CTR is a static implementation strategy where each application consists of a single configuration. RTR is a dynamic implementation strategy where each application consists of multiple co-operating configuration [Deho00]. In CTR, once an operation commences the configuration doesn't change during processing. This approach is very similar to implementing an application in ASIC. Whereas RTR uses dynamic allocation scheme that re-allocates hardware during the execution of an application [Hutc95]. Each application consists of multiple time-exclusive configurations for the FPGA. These configurations are loaded during run-time of an application on the FPGA chip successively. This causes realization of new hardware for each configuration for particular portion of an application. There are two models of RTR: Global and Local. Global RTR allocates the whole FPGA resources (entire chip) for each configuration step. This results in new hardware realization on FPGA during each configuration at run-time. The designer's task in implementing an application in global RTR is to divide the application into roughly equal sized time-exclusive partitions to efficiently use the resources on FPGA [Hutc95]. This is also called temporal partitioning of an application. In local RTR, an application locally reconfigures subsets of the logic as the application executes. It may configure any percentage of the reconfigurable resources at any time. The organization of local RTR applications is based more on a functional division of labor than the phased partitioning used by global RTR applications [Deho00]. Once manual temporal partition is decided then global RTR can be easily implemented. The local RTR implementation approach is presently not available in CAD tools.

2.2 Artificial Neural Networks

For long Neurophysiologists, Neurosurgeons and psychiatrists all have put tremendous effort to understand the structure of the human brain and its inherent ability to build up its own rules through learning. Still much is unknown about how brain trains itself to learn the information and store this knowledge as generally can be referred as an "experience".

In the human brain typical 'neurons', which are the basic processing elements collect signal from the nervous system of the body through a host of fine structures called ‘dendrites’. The neurons send out spikes of electrical activities through a long, thin wire known as an ‘axon’ which splits into thousands of branches. At the end of each branch, a structure called ’synapse’ converts and imposes these activities as ‘excitation’ or ‘inhibition’ on connected neurons. Consequently these neurons propagate this information down to its axon. Learning occurs by changing the effectiveness of the synapses so that the influence of a neuron on another changes. Thus the brain, which is a neural network, is highly complex, nonlinear and parallel computer. It has the capability of organizing neurons and modifying the scale of electrical activities in synapses to perform certain computations. Figure 2.2 shows the structure of a biological neuron.

Typically, neurons are five to six orders of magnitude slower than silicon gates; events in a silicon chip occur in the nanosecond range, whereas biological neural events occur in the millisecond range [Hayk99]. However the brain compensates for the slow performance by having a staggering numbers of neurons and massive parallelism of activities between them. This has made the brain an extremely powerful biological computer to outclass the fastest digital computer so far in performing certain computations. Thus in the last two decades many artificial intelligence researchers have shifted focus to realize brain like artificial neural network architecture implementation on GPP or ASIC. The motto behind this move is the constrained capability of the digital computer to operate the task sequentially. Conventional computers use an algorithmic approach i.e. the computer follows a set of instructions in order to solve a problem. This algorithm/program is controlled by a single, complex central processing unit, which carries out execution of instructions one at a time, and store information at specific locations in memory. This is considered

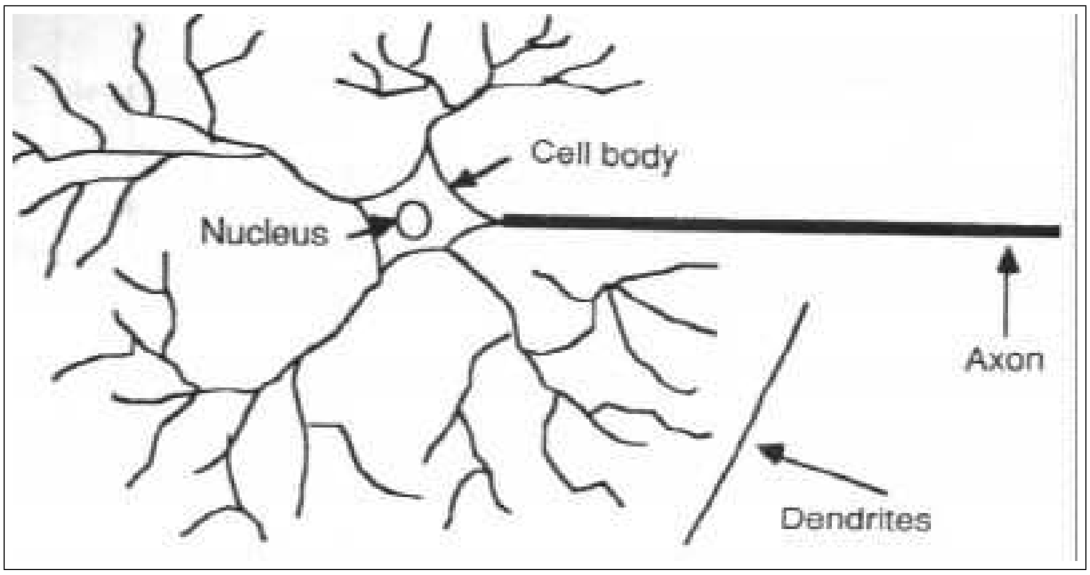


Figure 2.2: Neuron

quite insufficient in performing certain tasks.

Artificial neural network on the other hand processes the information in a similar way the human brain does. Thus we can adopt the following definition for artificial neural network viewed as an adaptive machine,
"An artificial neural network is a massively parallel distributed processor that has natural propensity for storing experimental knowledge and making it available for use. It resembles the human brain in two aspects:

- Knowledge is acquired by the network through a learning process.
- Inter-neuron connection strengths known as synaptic weights are used to store the knowledge." [Hayk99].

The process used to perform the learning is called a learning algorithm, the function of which is to modify the synaptic weights of the network in an orderly fashion so as to attend a desired design objective.

ANNs have been used in many tasks such as speech, vision and knowledge processing. ANNs also demonstrated their superior capability for classification and function approximation problems. It also has great potential for solving complex problems such as systems control, data compression, optimization problems and pattern recognition.

2.2.1 Learning in Artificial Neural Network

There are several algorithms that have been devised to teach an ANN and accumulate the experience through training. These include Error-back-propagation algorithm [Rume86], Radial basis function network [Broo88], boltzman learning algorithm to name just a few. The basic fundamental idea behind all these algorithms is the representation of knowledge in the form of training patterns to make network understand and organize its synaptic weights. This knowledge or training is the collection of several records of the data we wish to analyze, the network will run through them by feeding into neurons and learn how the inputs of each record may be related to the result. The very nature of each record or input depends on the kind of application the user is using. The network then continually refines itself by changing synaptic weights connecting various neurons until it can produce an accurate response when given those particular inputs. After training several cases, the network begins to organize itself to fit the data, much like a human brain learns from example. If there exists any consistent relationship between the inputs and result of each record, the network should be able to create internal mappings of weights. Thus the change in synaptic weights drives the network to such a state of 'experience' that later on it would be able to produce the correct output when different input patterns (not encountered during learning) are applied. This characteristic is widely known as generalization.

The Learning of a network can be achieved in 'supervised' or 'unsupervised' manner. Supervised learning involves applying training examples consisting of input signals and corresponding desired responses. Whereas unsupervised learning involves applying input signals only. The set of training patterns applied to a network is called an "Epoch". In training mode several epochs of patterns are applied to a network. Generally patterns are randomized for every presentation of an epoch to the network.

2.2.2 The Back-Propagation(BP) Algorithm

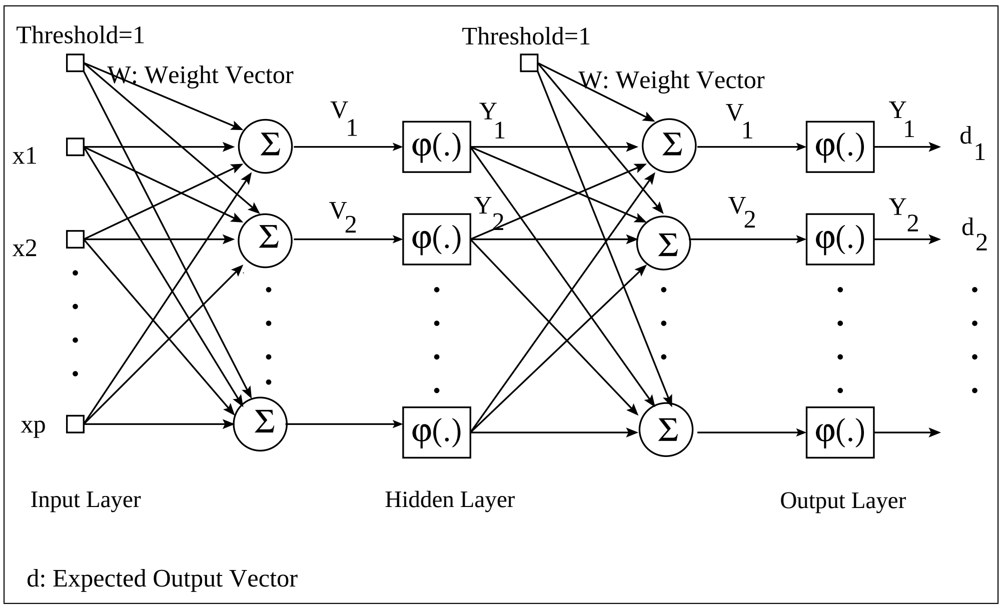


Figure 2.3: Architecture graph of Feed-Forward 2-layer network

A Multilayer feed forward network consists of an input layer, hidden layer and an output layer. This is also commonly known as multilayer perceptrons (MLPs). MLPs have been applied successfully to solve some difficult and diverse problems by training them in supervised manner with the "Error Back-propagation" [Rume86].

MLPs have distinctive characteristics of nonlinearity and high degree of connectivity between different layers of network, determined by weights. Nonlinearity at each layer of the network is satisfied by nonlinear transfer function. This activation function has limiting value between 0 and 1.

Figure 2.3 shows the multilayer perceptrons for the Back-Propagation algorithm. The Error back-propagation process consists of two passes through the different layers of the network: a forward pass and backward pass. The synaptic weights, weight vector [W], are initialized to store random values. In forward pass a training pattern, a input vector [x], is applied to the input layer of MLPs. Input vector is multiplied with synaptic weights and all multiplied values for each neuron are accumulated. This generates neuron output or multiply-accumulate vector [V]. It is passed through activation function generator to generate output within range of 0 and 1 which also serves as an input to the hidden layer. Finally a set of output is produced in response to the input at the output layer. During this pass all weights remain fixed. Error is calculated by subtracting actual output response from the desired output vector. In backward pass this error signal is back propagated through output layer first and then hidden layer to generate gradients for each layer. In weight update stage of the algorithm, gradient values along with learning rate are used to generate change in weight values for the pattern. Once all synaptic weights are updated a pattern is applied again and all the stages are repeated again.

Figure 2.4 shows a structure of a neuron in hardware connected to three synapses.

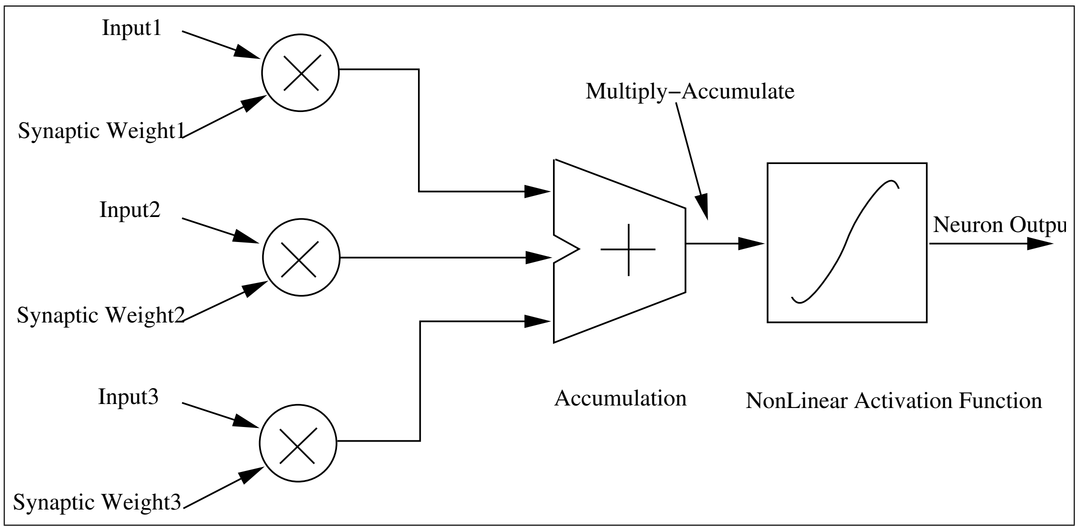


Figure 2.4: Basic structure of a hardware neuron connected to three synapses

2.2.3 The Back-Propagation Parameter Setting

There are couples of parameters, which are important in implementing error back-propagation: rate of learning and momentum term. The error back-propagation algorithm changes the synaptic weights for different connections between neurons in different layers. Steepest descent is the method to find the trajectory in weight space starting from some random initial solution. The smaller the learning rate parameter η, the smaller the change in weights in each iteration and smoother the trajectory in weight space. Hence a network will learn slowly but search more weight space meticulously. If learning rate parameter is too high, it will speed up the learning of network resulting in large change in synaptic weights. This can sometimes lead to an oscillatory behavior of network. A simple method of increasing the rate of learning and avoiding instability is to include a momentum term to weight correction rule. In general weight correction rule can be given as [Hayk99],

$$\triangle W_{ij}(n) = \alpha \triangle W_{ij}(n-1) + \eta\delta(n)y(n) \tag{2.1}$$

where, $\triangle W_{ij}$: change in weights for connection between neuron i and j in different layer
α: Momentum term
η: Learning rate parameter (LR)
δ: Local gradient in weight space
y: Input to connection
n: Input pattern number
Momentum term is usually a positive number. It controls the feedback loop acting around $\triangle W_{ij}$. For adjustment of weight function to be convergent momentum constant must be restricted to the range $0 < \alpha < 1$. Inclusion of momentum in the back-propagation algorithm tends to accelerate descent in weight space in steady downhill directions. This indirectly has stabilizing effect in case of oscillation around minima. Momentum and learning rate parameters are typically adjusted (usually decreased) as the number of iteration increases.
Hence, from the preceeding discussion and the Figure 2.3, the back-propagation algorithm can be summarized as follows:

1. **Initialization:** Choose the topology of the network and randomly select all weights and thresholds for connections in range of [-1,1].

2. **Presentation of training examples:** Present the network with an epoch

of training examples. For each example in the set ordered in some fashion, perform the forward and backward computation.

(a) **Forward computation:** Let a training example in an epoch be denoted by [X,d], with input vector X applied to the input layer of nodes and 'd' is the expected output vector. The net internal activity of neurons of the hidden layer can be written as:

$$V_j(n) = \sum_{i=1}^{p} (W_{ji} X_i + Threshold) \tag{2.2}$$

where i is the previous layer of network to layer j and p is the number of nodes in layer i. These outputs are passed through a sigmoidal function. Hence the output of the hidden layer which also becomes input for output layer can be given as:

$$Y_j(n) = \frac{1}{(1 + e^{-V_j(n)})} \tag{2.3}$$

As this is propagated to the next layer, the output of which can be calculated from equation 2.2 and 2.3. For every node in the output layer, the error is calculated as,

$$e_j = d_j(n) - Y_j(n) \tag{2.4}$$

where j is the node in output layer

(b) **Backward computation:** By proceeding backward compute the local gradients.

(i) For output layer,

$$\delta_j = e_j(n)Y_j(n)[1 - Y_j(n)] \tag{2.5}$$

where j is the node in output layer and Y_j is the actual output of node j in output layer.

(ii) For hidden layer gradient can be given as,

$$\delta_p = Y_p(n)(1 - Y_p(n)) \sum_k \delta_k(n) W_{kp}(n) \tag{2.6}$$

where p and k are the nodes in the hidden and the output layer respectively.

(c) **Adjust the weights:** Weights are adjusted as per the following formula,

$$\begin{aligned} \Delta W_{ji}(n) &= \eta\delta(n)y(n) \\ W_{ji}(n+1) &= W_{ji}(n) + \Delta W_{ji}(n) \end{aligned} \tag{2.7}$$

where j is the node in next layer to i.

(d) **Iteration:** Iterate the whole process by presenting epoches of training patterns until the average squared error of whole network over the entire training set reaches an acceptably small value.

2.2.4 Transfer Function

Computational elements or nodes in ANNs are characterized mainly by the type of non-linearity they implement. In multi-layer perceptrons usually this non-linearity is imparted by using an activation/Transfer function at output of each neuron. Common examples of activation functions include Threshold (hard-limiter or step function), pseudo-linear (ramp) functions, hyperbolic tangent functions and sigmoid functions [Hert91]. A nonlinear activation function commonly used in (MLPs) is the sigmoidal nonlinearity, a particular form of which is defined by the *logistic function*

$$f(x) = \frac{1}{1 + e^{-\beta x}} \tag{2.8}$$

where, $-\infty < x < \infty$

The sigmoid function outputs a value that is close to zero for a low total input value and close to one for a high input value. β is a threshold (scale) value that can make the slope between zero and one steeper or shallow. Equations 2.3 and 5.2 are same when $\beta = 1$.

Figure 2.5 shows a plot of the logistic function.

The derivative of this function is

$$f'(x) = f(x) \times (1 - f(x)) \;\; with \;\; \beta = 1$$

The non-linearity of the sigmoid function is important, since it helps make output reachable to desired value. If the transfer function used is linear, each of the neuron inputs would get multiplied by the same proportion during training. This could

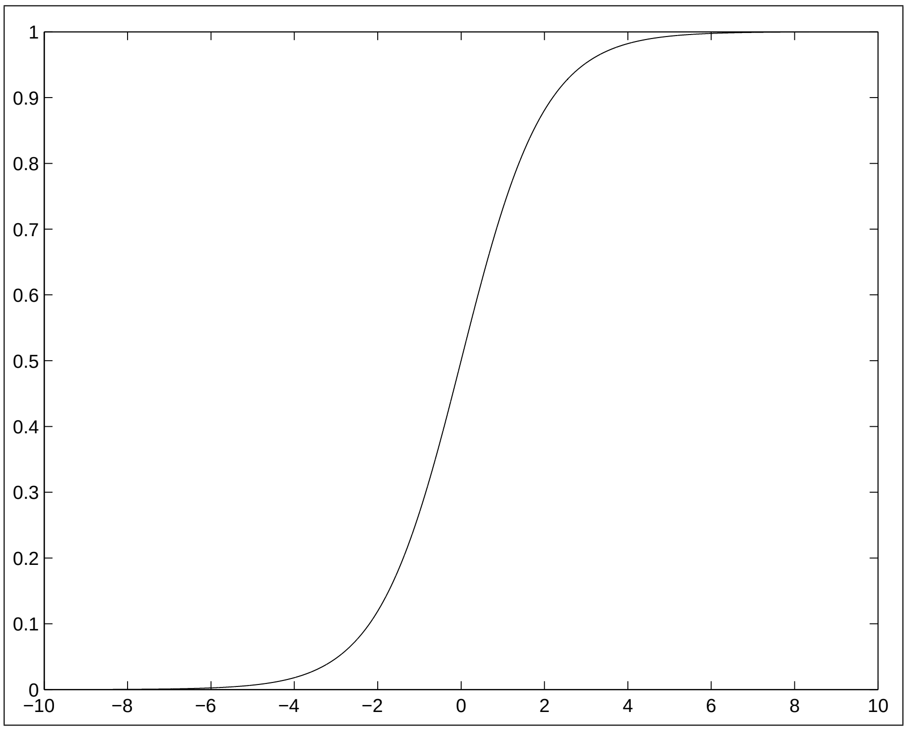


Figure 2.5: Logistic Function

cause an entire system to drift in weight-error space during training.

The hyperbolic function counterpart to the sigmoid is the hyperbolic tangent(tanh) function. The hyperbolic tangent produces output in -1 to 1 range symmetric to on both sides of zero. It can be described as following:

$$f(x) = \tanh \beta x = \frac{1 - e^{-\beta x}}{1 + e^{-\beta x}} \tag{2.9}$$

The derivative of tanh function is

$$f'(x) = (1 - f^2(x)) \;\; with \;\; \beta = 1$$

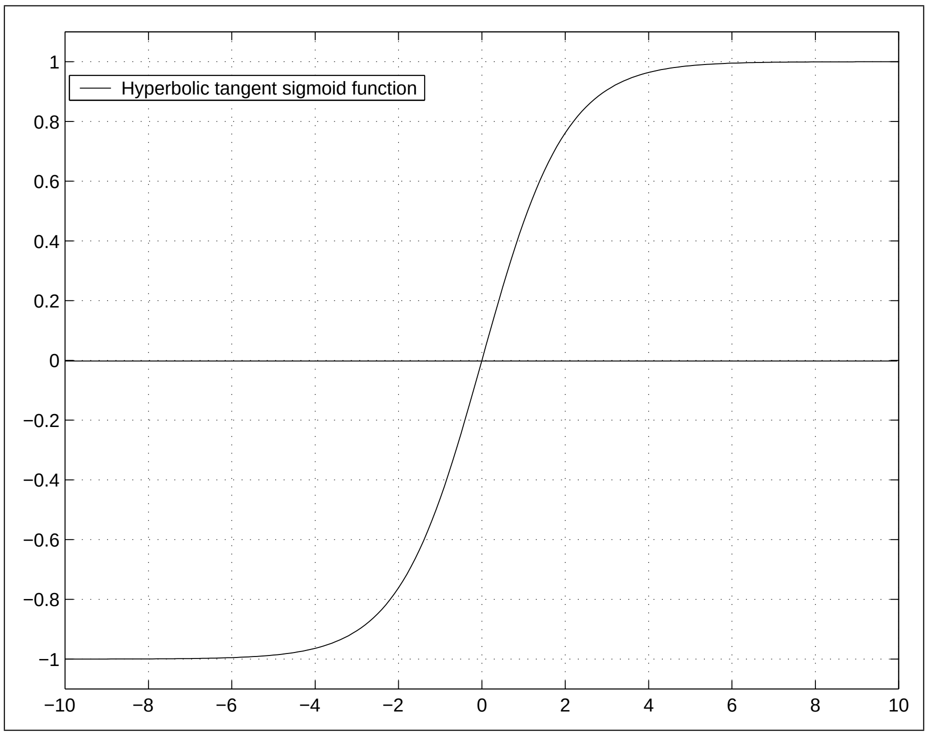


Figure 2.6: Hyperbolic Tangent Function

For most modeling tasks of the back-propagation algorithm, the sigmoid function is considered to be a baseline transfer function to measure results. The sigmoid function will generally produce the most accurate model however it is slower in achieving convergence [Chan04]. In certain models the use of tanh exhibits faster learning but not so accurate results. Most of the researchers have chosen the sigmoid function as transfer function for their implementation of the back-propagation algorithm on hardware.

2.3 Summary

In this chapter the background related to the Field Programmable Gate Arrays and Artificial Neural Networks was described. Various configuration approaches of an FPGA were explained in detail. The basics of Artificial Neural Networks was described and detail formula for the back-propagation algorithm was explained in later part of the chapter.

Chapter 3

Literature Review

There has been several attempts to implement an Artificial Neural network (ANN) on hardware. In the next few sections, we will examine different approaches used for realizing an ANN on hardware. In section 3.1 we will discuss various implementations of an ANN on ASICs. Section 3.2.1 will review the implementation of some of the algorithms of an ANN on the FPGAs. Section 3.3 will discuss the implementation and performance related issues of the back-propagation algorithm on FPGAs.

3.1 Neural Hardware

Over the past few years, the development of dedicated neural hardware for various applications has been carried out to speed-up ANN algorithms. The goal was to achieve better performance over general purpose system by processing ANN algorithms in parallel. Researchers have interchangeably described the term neural

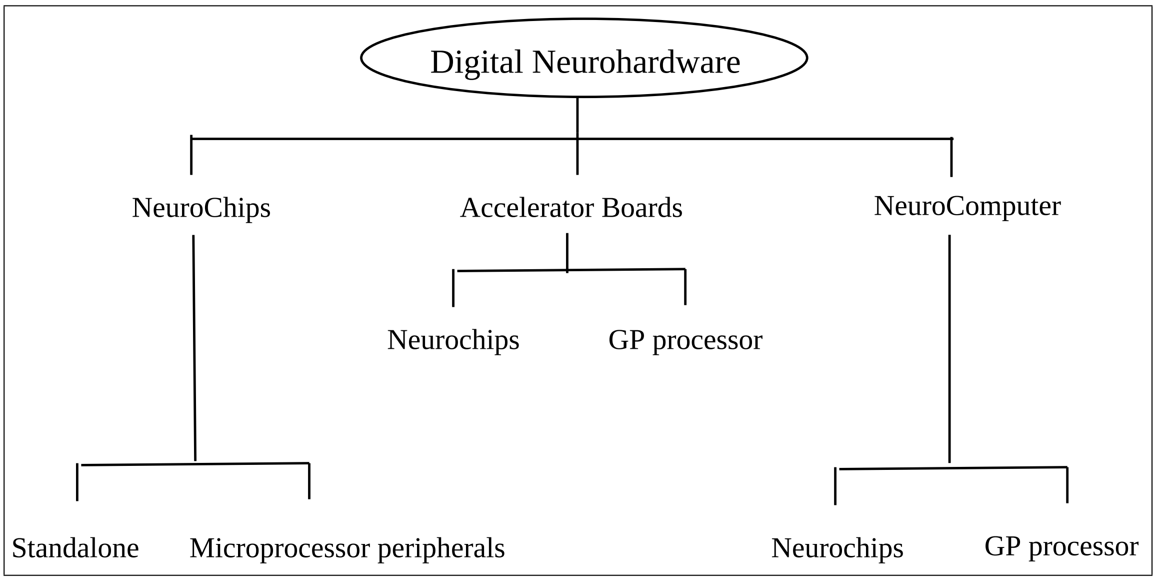


Figure 3.1: Taxonomy of digital neurohardware

hardware as neurohardware in various literature.

Today, there is a wide range of commercially available neural network hardware in the market. There is a huge diversity between the designs of these neurohardwares. The designs differ in various aspects such as architecture, type of the applications, numerical representation etc. Hence, for the better evaluation of the existing neurohardware some kind of taxonomy is required. The categorization of neurohardware is subjective and differs widely in existing literature. Our interest of study is based on the taxonomy of digital neurohardware proposed by [Scho98]. Digital neurohardware can be classified by: system architecture, degree of parallelism, neural network partition per processor, inter-processor communication and numerical representation. The basic classification of neurohardware can be given by Figure 3.1.

The neurohardware system architectures range from single stand-alone neurochip to full-fledged microcomputers [Scho98]. The heart of the neurohardware system is neurochip which can serve as a stand-alone or microprocessor peripheral. Accelerator boards are custom neuroprocessor boards generally connected to a PC or workstation. Generally, accelerator boards are equipped with neurochip and general processor. Neurocomputers are dedicated machines developed solely for implementing neural functions. In the 80's the basic idea of neurocomputer design is to connect many general processors in parallel to allow for fast processing. The emergence of the neurochips for implementing neural network algorithms made it possible to design neurocomputers consisting of only neurochips.

Various architectures also differ in flexibility with which it can implement parallelism. The higher number of processing elements provides higher degree of parallelism but at the cost of expensive chip-area. Depending on the number of processing elements employed in the system, neurohardware can be classified from coarse grained with fewer number of processing elements to fine-grained with almost one-to-one implementation of processing nodes [Scho98].

Digital neurohardware also vary in numerical representation of arithmetic operations. Floating point implementations give higher precision but it is area consuming and complex in implementation. Fixed point arithmetic is less complex and less area consuming but it has lower precision. The fixed-point arithmetic can prove beneficial when an ANN algorithm does not require high precision for implementing certain applications. Other numerical representations such as pulse stream coding are also used. There are several ways to map an ANN on processing elements. This can be done by mapping neuron (n-parallel), synapses (s-parallel) or whole network in parallel to each processing element. The key concepts of an efficient mapping are load balancing, minimizing inter-PE communications and less requirement of synchronization between the PEs (Processing Elements) [Scho98]. In general, there exist different types of neurohardware for an ANN algorithm implementation. The choice of architecture depends on the application and an intended ANN algorithm. In the following sections we will look into several examples of well-known neurochips, accelerator boards and neurocomputers.

3.1.1 Neurocomputers build from digital neurochips

Digital neural ASICs are the most powerful and mature neurochips. Digital implementation offers high precision, reliability and high programmability compare to its analog counterparts. Following are the few examples of neurcomputers build from the neurochips of some well-known vendors.

CNAPS: [Scho98]

CNAPS (Connected Network of Adaptive Processor) is a very well known neurocomputer from Adaptive Solutions INC. Basic building block of this neurocomputer is N6400 neurochip which consists of 64 PEs. These PEs are connected in cascade by a broadcast bus in a SIMD (Single Instruction Multiple Data) mode. PEs employ fixed point arithmetic for adders and multipliers. The standard system consists of four N6400 chip on a board controlled by a sequencer chip. Each PE is equipped with 4Kbytes of on chip SRAM to hold weights. There are also available two 8 bit buses to allow input and output data broadcasting. The ANN is mapped n-parallel on this chip to efficiently use limited capacity of data buses. One of the main features of this architecture is its capability to implement various algorithms including backpropagation, Kohonen self-organizing maps as well as image processing algorithms. Another advantage of this architecture is the scalability as additional N6400 chips can be easily added. Performance in implementation of an ANN algorithm is generally measured in Connections Per Second (CPS). This architecture is claimed to perform 1.6 GCPS (back-propagation) for 8 and 16 bit weights and 12.8 GCPS for 1 bit weights. The developers claimed in 1994 that the CNAPS was the world's fastest learning machine.

SYNAPSE-1 [Scho98]

The basic building block of SYNAPSE-1 (Synthesis of Neural Algorithms on a parallel systolic engine) is Siemens' MA-16 neurochip. It consists of eight MA-16 chips connected in two parallel rings controlled by Motorola MC68040 processors. This forms 2-dimensional systolic array of MA-16 chips arranged in two rows with four columns. The weights are stored in off chip 128Mb DRAM. The neuron transfer functions are calculated off-chip using look-up tables. An ANN algorithm can be implemented with 16 bit fixed point numerical representation. Several algorithms including backpropagation and Hopfield network have been successfully implemented on the SYNAPSES-1.

Hitachi WSI [Heem95]

This is a wafer scale integration architecture aimed at implementing back-propagation learning. On each silicon wafer (0.8μm CMOS technology) 144 neurons are integrated. Eight silicon wafers are connected by a time-sharing hierarchical bus structure. Weights are stored with 16 bit precision, while activation values have a fixed 9-bit precision. Each neuron is mapped to a PE that contains one multiplier, several adders and local on chip memory. Interconnection between neurons is achieved by a global data bus while synchronization of processing units is achieved by a global control bus. NETtalk benchmark was run on Hitachi WSI with 203 inputs, 64 hidden units and 26 output units, resulting in a performance of 150MCUPS.

3.1.2 Accelerator boards built from digital neurochips

Accelerator boards are widely used neurohardware because they are cheap, simple to connect to PC or workstation and easy to use. Accelerator boards generally supplied with user friendly software tools. It can achieve speed one order of magnitude higher than sequential implementation. Generally accelerator boards work in conjunction with PC as it can reside in the expansion slot (compatible with ISA or PCI) of the PC or workstation. We will now review some commercially available accelerator boards.

Ni1000 Recognition Accelerator Hardware: [hinT]

Ni1000 is a high speed classification engine for pattern recognition developed by Nestor/Intel Corp. This recognition accelerator is a PC add-on board that has been specifically developed for optical character recognition applications. Its architecture consists of classifier, 16-bit microcontroller (Harvard architecture) and bus interface to the host PC. The chip is compatible with commonly used RBF algorithm like Restricted Coulomb Energy (RCE) as well as back-propagation algorithm. The microcontroller has got 4k x 16 programmable flash memory to store controller program, 256 x 16 general RAM and 32 bit timer. The Classifier consists of math unit, distance calculation units (512) and Prototype Array (PA). The math unit operates on 16-bit floating point (10-bit mantissa and 6-bit exponent) numbers. The Prototype Array (PA) of classifier stores upto 1000 prototypical vectors. The distance calculation units to calculate distance between input vectors and prototype array of vectors. The accelerator can accept input vectors with maximum of 256 feature dimensions (5 bits each) and outputs up to 64 classes. Due to the tight coupling between the PA and distance calculation unit, Ni1000 can classify 33,000 patterns per second. Ni100 is claimed to perform 8.2 billion CPS in pipelined operating mode with 33 MHz clock.

NT6000 [Heem95]

NT6000 accelerator card is a neural network plug-in PC card that fulfill the need for intelligent data acquisition [Heem95]. They are generally equipped with DSP processor to speed up neural processing to 2 MCPS. The implemented neural network algorithms on this system are back-propagation and Kohonen maps.

IBM ZISC [Heem95]

ZISC (Zero Instruction Set Computer) is a neurochip developed by IBM. It is mainly used for pattern recognition applications. A single ZISC chip holds 36 neurons or prototypes to implement an RBF network trained with RCE (or ROI) algorithm. Currently ZISC ISA and PCI cards are available to work as plug-ins to PC. The ISA card holds 16 ZISC chips with 576 prototypes. The PCI card can hold 19 chips with 684 prototypes. PCI cards can process 165,000 patterns per second, where patterns are 64 8-bit element vectors.

In summary, we have reviewed neural hardware on custom built computers that could be either neurochip or accelerator board. The main goal of building such computer as can be observed from the data is to achieve high performance. Following table summarizes various neural hardware implementations with respect to the speed, the algorithm implemented and the type of the computer.

Neural Hardware	Type	Algorithm	Performance	No. of Processing Elements/chip
CNAPS	Neuro-Chip	Back-Propagation Self-organizing Map	1.6 GCUPS Max.	64
SYNAPSE-1	Neuro-Chip	Back-Propagation Hopfield Network	33 MCUPS	Not mentioned
Hitachi WSI	Neuro-Chip	Back-Propagation	150 MCUPS	144
Ni1000	Accelerator Board	Back-Propagation RBF network	8.2 GCUPS Max.	Not mentioned
NT6000	Accelerator Board	Back-Propagation Kohonen map	2 MCPS	Not mentioned
IBM ZISC	Accelerator	RBF network	Not mentioned	36

Table 3.1: Summary of various neural hardware

3.2 ANN implementations on FPGAs

The ongoing revolutionary progress of microelectronics is the driving force behind the constant development of new technical products. FPGAs are such kind of innovation in the microelectronics for the computing applications. In the last decade it has become common to realize computationally expensive algorithms on the FPGAs. The continuous growth in the FPGA's density has made possible to build system-on-chip designs with a complexity of more than a million gates and an internal RAM. Therefore, more recently the FPGAs have also proved an attractive hardware platform for an area consuming ANN algorithms. Another advantage of this device is its ability to combine programmability with the increased speed of operation associated with parallel hardware solutions [Lysa94]. Various advantages of implementing the ANN algorithms on the FPGAs than ASICs were discussed in the previous chapter. In the next section we will review various implementations of ANN algorithms like SOM, RBF, associative memory on the FPGAs. Section 2.2 will describe various implementation of the error back-propagation algorithm on the FPGAs.

3.2.1 Mapping Algorithms onto FPGAs

With respect to functionality an ANN can be distinguished for association, classification and approximation [Poor02]. [Poor02] implemented an algorithm on FPGAs for each functionality. They used dynamically reconfigurable hardware accelerator, RAPTOR2000, to implement algorithms that are required for special application. The RAPTOR2000 system consists of a motherboard and up to six application specific modules. The motherboard provides communication infrastructure for the modules to host via PCI bus. The six modules are connected in ring topology for inter-communication. Each module consists of Xilinx Virtex XCV1000 FPGA chip and 256Mbytes of SDRAM. All modules are connected to common local bus for communication with other devices or modules and for communication with the host via PCI bridge. The PCI bus bridge can work in slave serial mode in which host configures the modules initially by downloading bitstream into the FPGAs. An additional broadcast bus can be used for simultaneous communication between the modules. For fast memory access of the host system, a dual port SRAM is used which can be accessed by all modules. The reconfiguration of the module on the RAPTOR2000 can be started by the host computer. The RAPTOR2000 is claimed to have reconfiguration time of 20ms for each module.

[Poor02] chosen SOM (Kohonen maps) for classification task, NAM (Neural Associative Memory) for association task and RBF for function approximation task to implement on the RAPTOR2000 system. For implementation of SOM five modules are applied. Four modules are used for implementing a matrix of PEs while the fifth module is used as a matrix controller. The dual port SRAM is used to store the input vectors. Using XCV1000 devices total 64 processing elements can be achieved on a single module. In this case [Poor02] achieved the performance of 11.3GCPS which is far better than 80MCPS achieved on a personal computer (AMD athelon, 800MHz). [Poor02] also implemented the SOM algorithm on RAPTOR2000 system using different Xilinx Virtex devices. Arithmetic operations are carried out in 16-bit precision fixed point representation. For the implementation of NAM, RAPTOR2000 system is equipped with six modules. In this case each module allows to realize 512 neurons. The neuron unit consists of a control unit, neural processing unit, priority encoder and decoder and 128Mbytes SRAM. Synthesis results showed a space consumption of 3200 CLBs for 512 neurons. The neurons in this implementation can work at 50MHz and requires 5.4 μs for one association.

There has been an FPGA implementation of novel kind of neural network model that dynamically changes its size called FAST (Flexible Adaptable Size Topology) by [PU96]. The main problems with most neural network models are the lack of knowledge in determining number of layers, the number of neurons per layer and how they will be interconnected [PU96]. The ontogenic neural networks aim at overcoming this problem by offering the possibility of dynamically changing the topology. The ART (Adaptive Resonance Theory) and GAR (Grow And Represent) are this kind of networks and the FAST has been derived from its concepts. The FAST network consists of feed-forward fully connected two layers and learning is carried out in unsupervised fashion. This network is suitable for clustering or categorizing the input data. A neuron in output layer is added when sufficiently distinct input vector, determined from sensitivity region, is encountered. The sensitivity region is determined by the distance between the input vectors and the weight reference vectors. In the pruning stage depending on the overlap of sensitivity regions between neighboring neurons, a neuron in the output layer is deleted. The FAST neural network architecture is composed of 68331 micro-controller and four Xilinx XC4013 FPGA chips. On these chips FAST neurons, sequencer and I/O mapping registers are implemented. The FAST neuron is composed of three different blocks for the execution of an algorithm: distance computation, learning and pruning. Each neuron includes nine 8-bit adders and a single 8-bit shift-add multiplier. It also includes random number generator for pruning process. The sequencer is a finite state machine which controls the execution of the algorithm in three different stages. The FAST neural network model is applied to a color learning and recognition problem for digital images. This requires clustering of image pixels by chromatic similarity properties. In this experiment pixels coordinates of an image are randomly presented to the network, which results in the segregation of the image into four clusters, one per output neuron. It is found that every input vector can be categorized in 8μs. The result obtained with hardware implementation closely resembles that obtained from software simulation.

[DAbr98] applied the Hopfield neural network on the FPGA to solve classical constraint satisfaction problem: the N-queen problem. The goal of the N-queen problem is to place N queens on an N x N chessboard in mutually non-attacking positions. This involves certain constraint to be satisfied to place the queen safely on the chessboard. [DAbr98] mentioned that the implementation of the Hopfield network on FPGA for this problem differs from other implementations in several ways. First, the weights are small and can be represented using integers. This significantly reduces the size of arithmetic units. Second, the neuron values are restricted to 0 or 1. This removes the need for multiplier units as vector products merely become conditional equations based on 0 or 1. The architecture of this implementation consists of 16 XC4010 FPGA devices connected by 4 programmable switches (FPICs) on an Aptix AP4 reconfigurable board. Using this configuration it is possible to download a design consisting of up to 160,000 gates. The neurons and their interconnections are specified in VHDL which is synthesized in Exempler Logic's Galileo system tools. The performance of this system is compared with two other software simulations in 'C' for 4,6 and 8 queens problem. The result indicates that it is possible to gain between 2 to 3 orders of magnitude speedup with hardware realization. [DAbr98] mentioned that constraints on the N-queen problem are simpler than those found in real world scheduling applications.

There has also been an attempt to implement probabilistic neural network on the FPGAs for classification problem. The automatic classification of space borne multispectral images is quite a computationally expensive process. The multispectral images are obtained from sophisticated Earth Observing System (EOS) satellites launched by NASA to understand the earth's environment. One of the purposes of the classification of these images is to divide the earth's terrain into different classes like urban, agricultural, forest land, barren etc. [Figu98] implemented a probabilistic neural network (PNN) on FPGA to classify multispectral images captured from LANDSAT-2 EOS satellite. The multispectral images are formed by scanners and represent set of images each corresponding to one spectral band. A multispectral's image pixel is represented by a vector of size equal to the number of bands [Figu98]. The PNN classifier algorithm involves computing the probability of each pixel to fit into one of the classes. The equation for checking the probability requires complex calculations for subtraction, multiplication and the exponential of image vectors. The architecture of the PNN classifier consists of two XC 4013E FPGAs (termed as XFPGA and YFPGA), each with 13000 gates, on X213 hardware accelerator board. Due to the limited numbers of gates, fixed-point arithmetic is used for the calculations. The width of the fixed-point data path was determined by simulating variable bit operations in C. This way the weights values of 10 bits are found to be satisfactory. The weight memory is mapped on SRAM of the YFPGA. The YFPGA is also composed of subtraction unit, square unit and band accumulator unit. The XFPGA unit consists of an exponential unit, a multiplier unit and a class accumulator unit. Due to lack of space on the XFPGA, a class comparison unit was moved to the host computer. A look-up table is used to calculate the negative exponential. Both FPGAs can interface with the host through PCI bus. This enables host to configure the data initially. The performance of this system is compared with two software simulations carried out on DEC and Pentium machine. The time required for the simulation of algorithm written in 'C' on DEC machine running at 200MHz and Pentium machine running at 166MHz are found to be 22 and 30 minutes respectively. By using X213 accelerator board with two FPGA devices the time for implementation is reduced to 77 seconds yielding higher than one order of magnitude speed up.

3.3 Back-Propagation Algorithm on FPGAs

This section will describe various implementations of BP algorithm on single FPGA or the custom card built of one or more FPGAs. We will discuss the features of each architecture very briefly depending on the description provided in the referred papers. We will also focus on the type of arithmetic schemes used by various researchers for the computation of BP algorithm on FPGA. We will also elaborate on various available approaches for the implementation of the non-linear activation function. At last, a section will take on the issues pertaining to the performance/cost of already implemented designs.

The backpropagation(BP) algorithm is widely used for training multi-layer feed-forward neural network. It has been a popular choice for implementing an ANN on the FPGAs for its regular structure and the broad field of applications.

[Eldr94] implemented the BP algorithm on Xilinx XC3090 FPGA using custom built Run-Time Reconfiguration Artificial Neural Network (RRANN) architecture. The RRANN architecture divides the BP algorithm into the sequential execution of three stages known as Feed-Forward, Back-Propagation and Weight-Update. The RRANN contains one global controller on one FPGA and neural processing units on rest of the available FPGAs. The global controller is responsible for sequencing the execution of various stages and providing the initial parameters to the neural processors. Each neural processor consists of 6 hardware neurons and local hardware subroutines implemented with state machines [Eldr94]. The datapath of the architecture mainly consists of a broadcast bus, error bus and control bus. The broadcast bus is a time-multiplexed data bus on which global controller places one input value at a time. The neural processors read this value, do the weight multiplication for neurons in the first hidden layer. After the multiplication is complete, the product is accumulated with previous input iterations, and the next input is broadcasted over the bus. This is basically a partially parallel architecture in which each hardware neuron requires one multiplier only. The backpropagation stage is carried out by broadcasting error values back through the network in a manner suitable to the availability of weight values. This avoids the requirement of weight duplication, had the BP stage been carried out in a similar fashion as feed-forward stage. The weight update stage executes in a similar manner to the feed forward stage.

[Beuc98] designed RENCO (REconfigurable Network COmputer) to implement the BP algorithm on four Flex 10K130 FPGAs and Motorola microprocessor(68360). The processor bus is connected to all four FPGAs, each has its own memories (SRAM and DRAM). The processor is directly connected to an EPROM storing an operating system, as well as a DRAM and a Flash memory. [Beuc98] chose the Hand-written character recognition, a classification problem, to implement using BP algorithm on FPGAs. The hardware implementation of the design employs partially parallel architecture. [Beuc98] pointed out the difficulties in developing fully-parallel network as waste of hardware resources and scalability since such a hardware would require a multiplier for each connection (synapses). Hence, time-multiplexed interconnected scheme was developed which provides a good trade-off between speed and scalability/resources. The BP algorithm is divided into five sequentially executed stage for reconfiguration purpose. The network initialization and error computation are two extra stages in this approach from the one used by [Eldr94]. A design based on a fine-grained FPGA implementation of an ANN by [Lysa94] is a novel approach to combine previously reported design ideas. The design is implemented on Atmel AT6005 FPGAs, the only reconfigurable platform available at that time. Each layer in the design consists of maximum of four neurons. Pulse-stream arithmetic is used to provide an efficient mapping of the network on FPGA. This technique is discussed in more detail in next section. The neuron activation function is a simple binary step function rather than the sigmoid function which is more complex to implement on hardware [Lysa94]. Synaptic weights have a resolution of four bits. [Lysa94] implemented reconfiguration of the network for each layer. Hence, after processing of a network layer is complete, the neuron outputs are latched and the FPGA is reconfigured to load the next layer.

A hardware-software co-design approach to implement an ANN on reconfigurable devices and GPP, a flexible design platform LIRMM, was developed by [Molz00]. It is based on a signal processer to implement the software part of a given application and two reconfigurable devices (XC4013E FPGA - Xilinx) to implement the hardware part of the same application. The software part is described using the C programming language and the hardware part is described using VHDL. The hardware part is responsible for propagation of the input patterns which essentially covers Feed-forward and Back-Propagation stage. Whereas the software part is responsible for updating the synaptic weights and communication with host. Author has not described in detail about the purpose behind such kind of partition in hardware and software part. The design features a static topology with two neurons in the input layer, three in the hidden layer and one in the output layer(2-3-1 structure). The activation function is implemented by means of lookup tables.

The input values are represented in a 4-bit fixed-point format, where 1 bit is used for sign and three bits for the fractional part. The values of the synaptic weights are represented by a 10-bit fixed-point number, where 1 bit represents sign, 4 bits integer and 5 bits fraction. A typical application in the processing image area was used as a validation benchmark. The task consists in locating letters of a given image, pre-processed to 16 gray levels to fit in input codification. Authors worked on different bit sizes for fractional part of the weights and found that 5 bits are a good trade-off between area and correctness of classification of the image. The results obtained in this paper are not compared with software performance. [Li04] also implemented the BP algorithm on FPGA with hardware/software co-design approach. [Li04] built the fixed-point library in VHDL for numeric computations.

[Osso96] also attempted at implementing error-backpropagation algorithm on user-programmable Xilinx FPGAs. Authors described the possibilities of processing in a layered neural network can be Node level parallelism and Link (Synapses) level parallelism. Node parallelism requires one multiplier per neuron, as all neurons in a layer work in parallel and accumulate partial sum at a time. Link parallelism is the highest degree of parallelism that can be reached in a neural network. All connections of a neuron to other neurons are calculated at the same time by having same no.of multiplier as the synapses. The authors used node level parallelism to implement 3-3-1 backpropagation network in four Xilinx 4013 and one Xilinx 4005 FPGAs. The data transmission between layers of the neural network are based on the broadcasting principle in which neuron from previous layer send its activation values on a common data bus to all neurons in next layer. This approach of time-multiplex bus is similar to the one used by [Eldr94][Beuc98]. The data format chosen for the network is 8 bits for Input signals and synaptic weights as [Osso96] found it sufficient for simple benchmarks used. [Osso96] used the nonlinear sigmoid function implemented on look-up tables.

[Sima93] has modified the back propagation algorithm to work without any multiplication and to tolerate computations with a reasonable low resolution. Authors pointed out the need of fast multipliers requiring large hardware being a bottleneck in a design. Many researchers tried to reduce the size of a circuit by limiting the resolution of computation. [Whit92] tried to train networks where the weights are limited to powers of two. In this way all multiplication can be reduced to mere shift operations. However restricting the weight values severely impact the performance of a network and it is tricky to make the learning procedure converge [Sima93]. Authors felt the absence of general theory at that time on the requirements for enough resolution and presumed that it depends on size and architecture of a network. In this paper they present an algorithm that instead of reducing width of weights, reduces the resolution of states (signals), gradients and learning rates. The activation function for the network is simplified to a saturated power of two function. This function gives performance comparable to the sigmoid or the saturated ramp function. Also the gradients are discretized to be power of two values. This simplification of values to power of two functions eliminates the need for multipliers as it can be carried through mere shift operations. [Sima93] ran an experiment with large network consisting of 60,000 training patterns for handwritten character recognition, a classification problem. The results obtained are very similar to an algorithm with full precision gradients and sigmoid values. However, authors did not implement this algorithm on any hardware platform but suggested that such algorithm is well suited for integrating large network on a single chip. They also pointed out that such network also achieves high speed up since shifting operation is faster than multiplication.

An on-line arithmetic based reprogrammable hardware implementation of error-backpropagation algorithm is a novel approach developed by [Gira96]. The arithmetic representation is handled within a serial arithmetic and allows very fast operations. [Erce77] first developed the on-line method of computation through redundant number representation. Authors used Xilinx series XC4025 with 32MB RAM capacity to implement the algorithm. They chose 16 binary bits to present the weights and state(signal) values. In case of weights, 8 bits are used for both integer and fraction part while the activation values is presented with whole 16 bits for fraction ($-1 \leq \tanh \leq 1$). No information is provided in this paper on nature of the application used by the authors. [Skrb99] also developed an on-line arithmetic based reconfigurable neuroprocessor. However in this short paper no details are given on the architecture or the implementation/results of the algorithm on FPGA. Since multiplication and addition are the two most frequent arithmetic operations in a neural network application, the amount of resource and time to execute it becomes an important factor for choosing a mode of arithmetic computation. A neural network contains numerous connections performing arithmetic operations, thus requiring a substantial amount of wires. As parallel arithmetic requires large buses, it is not well suited for such implementation from the resource point of view. Bit-Serial arithmetic seem to be an effective solution in this scenario. The on-line arithmetic is a bit-serial operation with transmission starting from the most significant digits first (MSDF). Conventional bit-serial algorithms employ transmission in

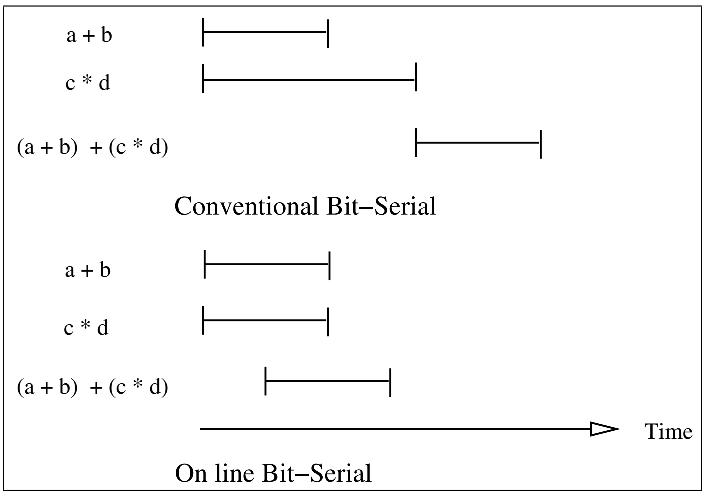


Figure 3.2: Computing (a + b) + (c * d)

least significant digits first (LSDF), a natural “paper-pencil” approach to perform computation. [Skrb99] pointed out that since on-line arithmetic allows overlapping of computation because of MSDF transmission, it can lead to a lower processing time. This is depicted in Figure 3.2.

[Skrb99] developed an implementation of neural network on hardware based on shift-add neural arithmetics. The shift-add arithmetics provides a complete set of complex functions like multiplication, square, square root etc. linearly approximated to reach sufficient simplicity. This can be achieved through only adders and barrel shifters. The details of neural arithmetic based on this approach is explained later in subsection. The implementation of perceptrons was verified on a card called ECX card. The card contains two Xilinx FPGAs of the XC3000 family. The first FPGA (X1) acts as a neural processing element and the second FPGA (X2) acts as a controller. The processing element accepts the weights and inputs in their logarithm form. Since multiplication can be performed as an inverse of sum of logarithms of operands, values in logarithmic forms are passed through LOGADD (Addition) and then EXP (inverse log) blocks. The weight memory is fast static RAM of 32K x 32 bits and the input/output memory is a fast static RAM of 8K x 16 words. No time multiplexing required to transfer data from RAMs to FPGA (X1) as both input/output and weight memory use separate data bus. The controller (X2) chip addresses weight and input/output memories and ensures data transfers. [Skrb99] also carried out an on-chip implementation on $0.7\mu m$ standard cell technology by the CADENCE design system. The ECX card FPGA platform can also implement Radial Basis Function (RBF) neural network and was validated using pattern recognition application such as parity problem, digit recognition, inside-outside test and sonar recognition.

[Ferr94][Mart94] developed a custom platform, called Adaptive Connectionist Model Emulator (ACME) which consists of 14 Xilinx XC4010 FPGAs. The ACME platform was successfully validated for XOR problem using 3 input, 3 hidden and 1 output nodes. The ACME platform uses 8 bit fixed-point to converge XOR problem using back-propagation learning, due to the availability of limited FPGA resources at the time of development.

3.4 Sigmoid Function implementation

Several approaches exist to implement the sigmoid function in hardware [Murt92].

Taylor Series expansion : This is done by approximating the sigmoid function using a Taylor series up to four or five terms. [Murt92] show following Taylor series formula for the sigmoid function which could be approximated in the range of -1 to 1.

$$y = \frac{1}{2} + \frac{3}{4}x - \frac{1}{4}x^3 \tag{3.1}$$

Look-Up Table (LUT) : This is a simpler approach in which values of the sigmoid function are stored in Off-chip or On-chip RAM or BlockRam on FPGA. Appropriate values are read back from the stored values by placing proper address, which can be generated with a simple circuit consisting of comparators and shifters. The benefit of this approach is the minimum logic cost while the drawback is the large silicon area required to map LUT on memory.

Co-ordinate rotation digital computer (CORDIC) The CORDIC scheme is an iterative method based on bit-level shift and add arithmetic operations. This approach also requires large silicon area[Murt92][Depr84].

PieceWise Linear(PWL) approximation : This is done by approximating the sigmoid function using a piecewise linear break up of the output. This approach has been proven to provide a good choice on the trade-off between the hardware requirements and the function precision [Zhan96]. PWL approximation can further be divided into linear and higher order approximation. [Alip91][Myer89] were the early presenters of the scheme for realizing the sigmoid function through PWL approximation. [Alip91] did lot of ground breaking work on simplifying the sigmoid function in order to obtain a realistic design capable of being implemented in VLSI technology. Since then, few function approximations have been presented largely based on the work done by [Alip91].

[Alip91] described PWL approximation of the sigmoid function by selecting an integer set of breakpoints for 'x' values and correspondingly setting 'y' values as *power of two* numbers. Hence considering a single approximating segment for each integer interval, the function can be linearized and described as following:

For Positive Axis,

$$y = 1 + \frac{1}{2^{|(x)|}}(\frac{\hat{x}}{4} - \frac{1}{2}) \tag{3.2}$$

where,

$\hat{x}$: Decimal part of x with its own sign

$|(x)|$: Integer part of x

With the same notation as above, for negative axis,

$$y = (\frac{1}{2^{|(x)|}}(\frac{\hat{x}}{4} + \frac{1}{2}) \tag{3.3}$$

Above simplified version can be easily implemented on hardware with a shift register and a counter to control it. This eliminates the need for a dedicated multiplier in the circuit. [Alip91] derived the above formula based on a linear break up of a sigmoid curve into 15 segments in the range of -8 to 8. The approximation proposed by [Myer89] is a *modified curve* based on the principles of A-law for Pulse Code Modulation (PCM) systems. The modified curve consisting of 7 segments was used to approximate the function. [Alip91][Myer89] also presented the formula for the calculation of derivation of the sigmoid function based on the linear segmentation of the derivation curve.

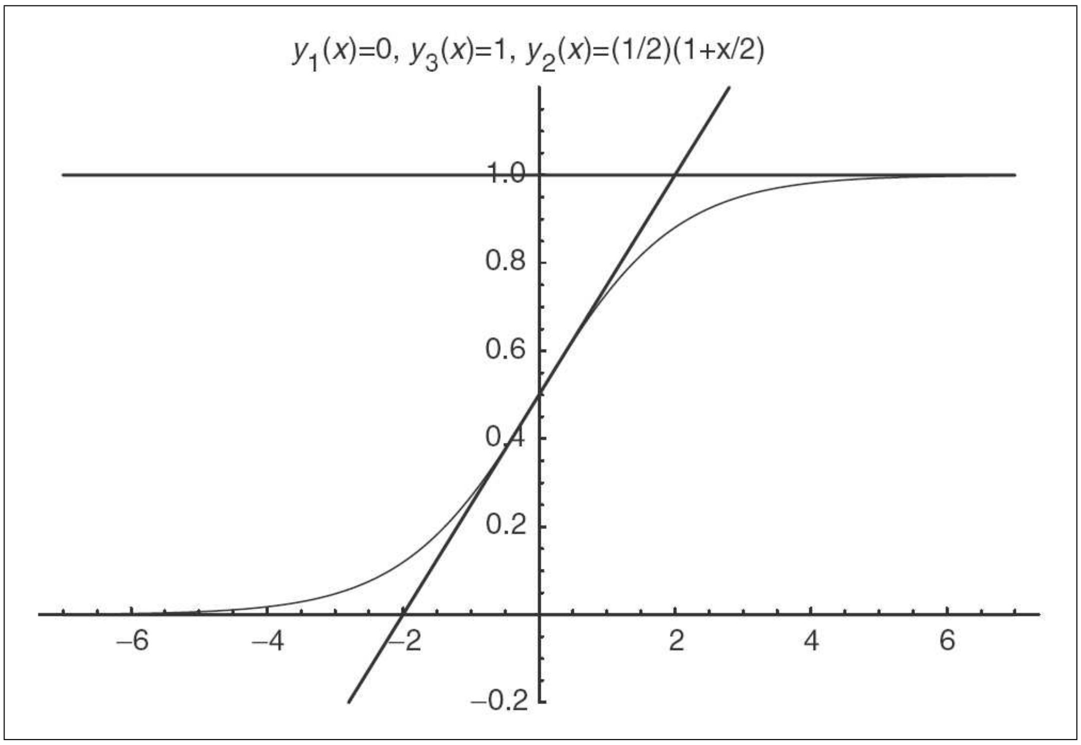


Figure 3.3: Initial minimum structure of Sigmoid for PWL through CRI

[Bast04] proposed a Centered Linear Approximation (CRI) scheme to the efficient generation of an optimized approximation to the sigmoid function and its derivative. CRI is a recursive computational scheme for the generation of PWL functions. Authors claim that the PWL approximation based on the CRI scheme can be applied to any *non-linear function*. However not much detail was given in this paper about the CRI algorithm and how functions can be smoothed to fit through PWL approximation. Figure 3.3 shows the initial minimum structure when CRI is applied to the approximation of the sigmoid function. This essentially a 3 segment approximation, where two segments represent the saturation at high and low input values. Whereas intermediate segment is a tangent to the reference sigmoid function at $x = 0$. The equation for this simplified approximation can be described as below:

$$\begin{aligned} y_1(x) &= 0 \\ y_2(x) &= \tfrac{1}{2}(1 + \tfrac{x}{2}) \\ y_3(x) &= 1 \end{aligned} \tag{3.4}$$

However it can be seen from the above equation and Figure 3.3 that 3 segment linearization is a very rough approximate of the function since error seems to increase for values close to the end of the curve before saturation. [Bast04] has noted that quantity of the segments can be increased and can be calculated for each interpolation level as follows:

$$Numberofsegments \quad = \quad 2^{q+1} + 1$$

The number of segments can be increased by adding pair of tangent on both side of y axis at some value of $|x|$. So the number of segments increases to 5, 9, 17 and 33 for $q = 1, 2, 3 \ \ and \ \ 4 \ \ respectively.$

The approach presented by [Samm91] advocated the use of fast multiplier and adder circuit for the approximation of the sigmoid function through linearization. The Sigmoid curve is broken into four segments, positive saturated, positive unsaturated, negative unsaturated and negative saturated. The non-linear part between the saturation limits is approximated using a least-squares *polynomial of degree two.*

$$\begin{aligned}
&Y = 1 \quad if\ x \geq highlimit\\
&Y = 1 - gain(highlimit - x)^2 \quad if\ x \geq 0 \sim x < highlimit\\
&Y = 0 + gain(highlimit + x)^2 \quad if\ x < 0 \sim x \geq lowlimit\\
&Y = 0 \quad if\ x < lowlimit
\end{aligned} \tag{3.5}$$

where the *lowlimit* and the *highlimit* values can be obtained from the saturation limits of the sigmoid curve and

$$gain = \frac{1}{2(highlimit)^2}$$

The scheme presented by [Samm91] is essentially a second-order approximation of the sigmoid function. [Zhan96] also proposed such scheme which improves the approximation of the function over its linear first order counterpart. In particular it reduces the average and maximum error of the approximation and requires only one multiplication. In this scheme the sigmoid function inputs are divided into segments and a second order approximation is applied for the computation of each segment [Samm91]. For a segment $[\alpha, \beta]$, if the input $\imath$ is in the interval $\imath \in [\alpha, \beta]$ and the approximated output is $H(u)$, in general a second order approximation is given by

$$H(u) = A + C * (\imath + B)^2 \quad where \ \ |C| = 2^{-n} \tag{3.6}$$

Since C is a *power of two*, the multiplication with C can be computed with shifting operation. An algorithm in detail is presented by [Samm91] in this paper to evaluate the values of A, B and C. Authors derived the formula for the approximation of the function divided into two segments of (-4,0) and (0,4) and computed as follows:

$$H(u) = \begin{cases} 2^{-1} \times (1 - |2^{-2} \times \imath|)^2 & -4 < \imath < 0 \\ 1 - 2^{-1} \times (1 - |2^{-2} \times \imath|)^2 & 0 \leq \imath < 4 \end{cases} \tag{3.7}$$

The scheme above can be used with sign-magnitude and 2's complement notation as well which are two commonly used fixed point number presentations. The fixed point number used in this scheme employs 4 bits for integer with 1 sign bit and 10 bits for the fraction. The average error produced by approximating the function with this method is in the order of 10^{-3}, reduced from the average of 10^{-2} for the first order PWL approximation.

[Beiu94] pointed out in their paper that even if approximation techniques for the simplification of the sigmoid function exist, the involved computation is still complex. Authors introduced a special kind of function, **A Particular Sigmoid Function**. [Beiu94] showed in this paper that a particular sigmoid function is equivalent with the classical sigmoid function if the gain is changed by a constant. Such function is described as follows:

$$f^*(n) = \frac{1}{1 + 2^{-n}}$$

The difference between the above equation and the classical sigmoid function is the constant scaling factor K, given by, $e^{-z} = 2^{\frac{-z}{\ln 2}}$ While implementing the function, the integer is scaled by this constant factor and then added to the value obtained from the above formula for the particular sigmoid function. [34] carried out the experiments for the inputs limited to the [-8,8] interval with 4 bit integer and 8 bit fraction representation. The computed error, the difference between the continuous and quantized function, is in [-0.16, 0.16] interval. The high error, in order of 10^{-1} compare to some of the other approximation described above, is attributed to the low precision used in addition to the effect of the approximation. Authors also proposed simplified versions of functions such as the hyperbolic tangent and the fast sigmoid. [Beiu94] also derived an algorithm for the derivative of the sigmoid function and extended to evaluate the formulas for other activation functions mentioned above.

From the above discussion on the approximation of the sigmoid function, we can summarize the classical solutions either investigated or implemented by researchers as follows:

- Look-Up table in RAM or ROM
- PieceWise Linear Approximation
- Sum of Steps evaluation (High order approximation) or Taylor Series expansion
- Other dedicated approximation such as CORDIC

The following sub-section gives an overview of various kind of arithmetic schemes used by researchers to realize the neural network algorithm on FPGAs.

3.5 Neural Arithmetics/Data representation

The goal of this section is to classify the neural arithmetics employed on FPGAs and examine the effect of it on the resources in terms of FPGA area. Since the execution of back-propagation algorithm requires extensive arithmetic operations, the choice for the selection of arithmetic scheme and number presentation plays a major role while targeting hardware. The performance of ANN on hardware is highly dependant on range and precision of numbers used to represent the signals. These signals include initial parameters like input, output, weight data and intermediate values like activation at each neuron, derivative of activation and error values. The purpose of developing an arithmetic scheme for hardware implementation is to present signals with sufficient range and precision which would make network convergent while keeping the area-cost within available FPGA resources. Various kinds of data presentation and arithmetic schemes used in architectures mentioned in earlier section are described as follow:

Pulse Stream Arithmetic: Pulse Frequency Modulation (PFM) is a coding scheme where circuit values are represented by the frequency of narrow constant-width pulses[Lysa94]. Signals encoded in this manner can be multiplied and summed at each node using simple logic gates. This technique is known as pulse-stream arithmetic and maps well onto fine-grained FPGAs. Figure 3.4 shows the fractional value of 7/16 presented in pulse stream.

[Lysa94] established an architecture of pulse stream neuron based on above arithmetic principle. Input (Activation value) to each neuron is a constant stream of narrow pulses while synaptic weights are constructed as gating func-

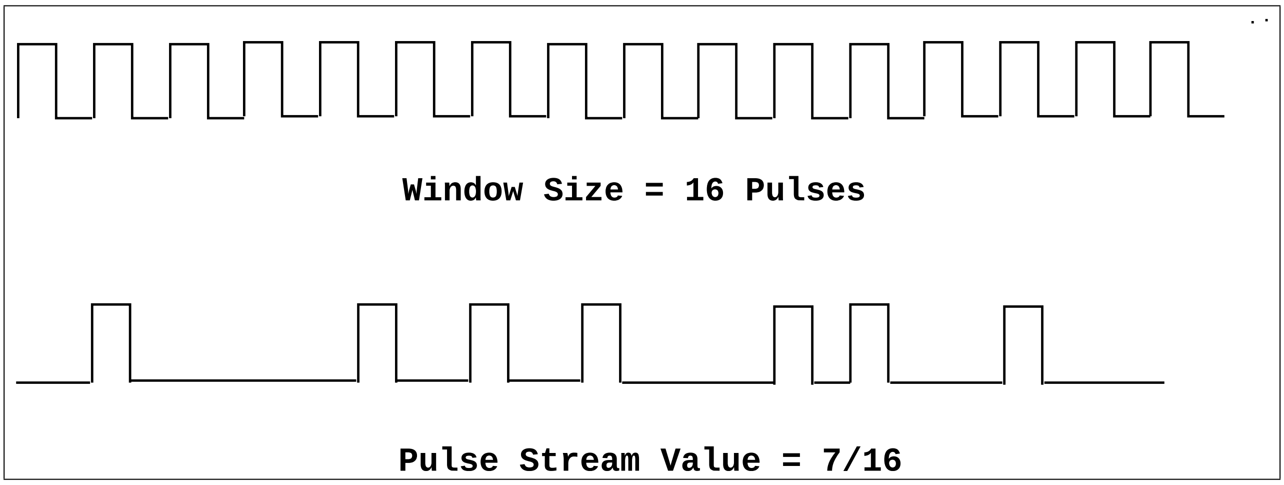


Figure 3.4: Example of a Pulse Frequency Modulated signal

tion by selectively ORing a series of *chopping clocks*[Murr89]. The chopping clocks are synchronous, non-overlapping binary clock with duty cycle of 1/2, 1/4 and so on. [Lysa94] used a 4 bit chopping clock generator which can be used to construct weights in the range of 0 to 15/16. Multiplication of the input pulse-stream by the weight values can be achieved simply ANDing the input and gating function.

Floating Point: This representation of numbers is very common for ANN implementation on general purpose computer. The floating point representation for hardware implementation of ANN provides high range and precision at the expense of valuable circuit area on FPGA. Hence in the past, most of the researchers avoided using an arithmetic based on floating point presentation due to unavailability of high capacity FPGAs. [Sima93] used floating point representation for several values in implementing Back-Propagation algorithm. Authors used 1 bit mantissa and 4 bit exponent for learning rate, 1 bit mantissa and 3 bit exponent for state (activation) values and 1 bit mantissa and 5 bit for gradient values. Though this network was validated on software platform first, authors claim with several hypothesis that it can easily be implemented on a chip.

[Clou96] built a *Virtual Image Processor* (VIP) which uses 5 Altera EPF81500 FPGAs with 1.5MB static RAM to validate neural network algorithms, image-processing and pattern recognition applications. In this paper authors showed to use only state signals (activation) as floating point number with 1 bit mantissa and 2 bit exponent. [Sahi00] checked the feasibility of floating-point arithmetic in reconfigurable systems. They pointed out in this paper that recent advances in FPGA technology offer the user more hardware resources on a single device and thus the greater potential to develop complex reconfigurable computing systems. [Sahi00] built three floating point modules: vector addition, subtraction and multiplication, modeled using VHDL and mapped to a Xilinx XC4044XL FPGA device. These modules are found to have speed up of 5 over the software model running on Pentium II 300MHz computer. However these modules are not mapped for any specific ANN application, image-processing or multimedia applications.

Fixed Point: The fixed point number representation is by far the most popular choice for implementing ANN algorithms on hardware. Fixed point arithmetic has traditionally been more area effective than floating point. Since many image processing and ANN applications are found to be working satisfactory with low range and precision and floating point representation costing high silicon area, the researchers widely used fixed point arithmetic for such applications. It is an ideal trade-off between area and range-precision requirement for FPGA based ANN architectures.

In Fixed-Point arithmetic, there are various possibilities with which basic computation for the BP algorithm such as multiplication, addition and derivation of activation function can be calculated. Following are some of the arithmetic techniques used by researchers for realizing Fixed-point hardware.

Bit-Serial Arithmetic This kind of technic calculates one bit at a time, whereas parallel arithmetic calculates all bits simultaneously. The bit-serial arithmetic consumes only small chip area but is very slow. This method has been seen in many ANN implementation on FPGA. The classic RRANN [Eldr94] and RENCO[Beuc98] architectures, well known implementations of ANN on FPGAs, used bit-serial data representation. This approach also helps fit more processing elements in small FPGA resources. Also the communication lines for arithmetic operation require only one bit wide, and bit serial adders require only one Xilinx Combinational Logic Block [Eldr94].

Shift operation/No multiplication This technique requires the presentation of numbers in ***"power of two"*** format. Hence multiplication is implemented with shifting operation. It was first introduced by [Sima93]. The drawback of such method is to find adequate numbers such that the output error is minimized. [Molz00] also used this approach to present input and activation function output in 4-bit fixed point format.

On-Line Arithmetic This method uses a redundant number presentation which allows very fast arithmetic operations. This representation is handled within a serial arithmetic. Therefore it exhibits all the benefits typical to bit-serial arithmetic mentioned above with faster speed. It computes an arithmetic operation in MSDF mode, transmitting Most Significant Digit First, unlike the conventional serial arithmetic which transmits data with Least Significant Digit First , an LSDF approach. [Gira96] explained in detail algorithms for addition, multiplication and tanh function using MSDF transmission (On-Line arithmetic) with redundant number presentation. The redundant number presentation expresses number in *radix-2* format. Such a format defined as borrow-save contains digit set -1, 0, 1. In borrow-save presentation each digit a_i of a number a is represented by two bits, a_i^+ and a_i^-, such that $a_i = a_i^+ - a_i^-$. For instance, digit 1 is presented by (1,0) while digit 0 has two possible representations namely (0,0) and (1,1).

Linearly approximated Functions [Skrb99] proposed so-called shift-add neural arithmetics which provides a complete set of linearly approximated functions suitable for the implementation of MLP on FPGA. Fundamental stones of the approach are $2^x, \log_2 x$. These functions are based on the shift operation in combination with the linear approximation. The linearly approximated 2^x and $\log_2 x$ can be defined as:

$$2^x = 2^{int(x)} \times (1 + frac(x))$$

$$\log_2 x = int(\log_2 x) + \frac{1}{2^{int(\log_2 x)}} - 1$$

[Skrb99] has shown that all other complex functions like multiplication, tanh, square and square root can be implemented using above simplified function and require only adders and barrel shifters on hardware. However such linearly approximated functions induce error to the result. The maximum error of all significant functions is found to be less than 6.5% in analysis tabulated by [Skrb99] in the paper.

3.6 Performance and Issues

The purpose of this section is to look into how the performance of various implementations has been evaluated by researchers the issues pertaining to it are addressed. This will provide a cursory idea when it is required to evaluate the performance of our architectures realized on FPGA. In this section we will focus on the performance of architectures implemented on one or more FPGAs for the BP algorithm only.

It should be noted here, on having done careful investigation, that there is not found any specific baseline or performance unit researchers relied on to evaluate their designs. Some used Weight Updates per Second(WUPS) [Eldr94] or Connections Update per Second (CUPS) while others used the frequency/speed with which design can compute as their unit of performance. Whereas some researchers were merely exploring the feasibility of FPGA implementation of the BP algorithm without going into much detail about the performance [Beuc98]. CUPS or WUPS can be defined as follows:

$$CUPS = N_w \star ClockRate/Cycles\ \ per\ \ Iteration \tag{3.8}$$

where, N_w: Total Synaptic Weights in Network

[Eldr94] performed the experiment, running at 14MHz, with number of FPGAs ranging from one to as high as 23 and showed in a graph that with multiple FPGAs, the performance can reach up to 2×10^5 WUPS. However [Eldr94] also noted that the reconfiguration of each stage on FPGA uses *extra time* that could otherwise be spent in executing the algorithm. The results obtained by [Gira96] from on-line arithmetic based implementation on several XC4025 FPGAs, running at 33MHz, estimated the performance to 5.2×10^6 WUPS. [Lysa94] implemented the BP algorithm on single ATMEL 6005 FPGA with complete reconfiguration for binary XOR function problem. Considering the reconfiguration overhead, the three-layer ANN produced results for the XOR problem at mere 24KHz that corresponds to network performance of 0.77×10^6CUPS. Authors noted that reported network is considerably slower than "static" FPGA based ANNs such as GANGLION, which is reported to operate at 4.48×10^9CUPS. However this impressive performance is achieved at considerable expense, using an array of more than 30 Xilinx FPGAs. [Ferr94][Mart94] developed the ACME board consisting of 14 Xilinx FPGA and validated the XOR problem on this platform. For this problem, the card running at 10MHZ performed at 1.64×10^6CUPS.

[Molz00] highlighted the results obtained in their experiments as their contribution in defining the number of bits required to code the synaptic weights, using

Signal-to-Noise ratio analysis. The implementation of hand-written character recognition problem by [Clou96] on Virtual Image Processor (VIP) board consisting of 2 x 2 matrix of Altera FPGAs performed 16 times faster than the software implementation on SPARC-10 workstation. Also the convolution process, performed on a binary image, was implemented on the same board and found to be more than 24 times faster than 90MHz Pentium general purpose processor. However [Osso96] concluded from their experiments involving four Xilinx 4013 and 4005H that Xilinx FPGA can be used for a fast prototype, but don't offer speed up that justifies a specialized hardware solution compared to a software solution.

It should also be noted here that quite a few implementations analyzed here, irrespective of the algorithm implemented, used multiple FPGAs to accommodate more processing elements and at the same time to distribute the computation task to ease burden on a single FPGA unit.

Tables 3.2 and 3.3 summarize the various BP algorithm implementations analyzed in this section with respect to the size of network, number of FPGAs used, the performance and the type of neural arithmetic considered.

3.7 Summary

In this chapter a detail survey of various implementation of ANNs on neural hardware was conducted. The neural hardware can loosely be classified into Neurocomputers built from neuro-chips, accelerator boards and reconfigurable hardware such as FPGAs. Our main focus in this chapter was to analyze various schemes for realizing ANN algorithm, specifically the BP algorithm, on FPGAs. The de-

Architecture Author and Year	Application	Topology	Arithmetics	Number Presentation
RRANN(Eldredge, Hutchings)1994	Not mentioned	Not mentioned	Bit-Serial Arithmetics	Fixed-Point (5-16)bit
RENCO(Beuchat, Sanchez)1998	Hand-Written character Recog.	Not mentioned	Bit-Serial Arithmetic	Fixed-Point (16bit)
LIRMM(molz, Engel et al)2000	Image Processing	2-3-1	Add-Shift Arithmetic	Fixed-Point (3-9 bits)
(Girau, Tisse-rand)1996	Not mentioned	Not mentioned	On-Line Arithmetics	Redundant number (Radix-2,16 bits)
ACME(Ferrucci, Martin)1994	XOR Function	3-3-1	Bit-Serial Arithmetic	Fixed-Point (8-bit)
(Ossoinig, Reisinger, et al)1996	Not mentioned	Not mentioned	Parallel Add-Mult.	Integer data (8-bit)
(Lysaght, Stockwood, Law et al)1994	XOR Function	3-3-1	Pulse-Stream Arithmetics	Input (16 bit) Weights(4 bit)
ECX(Skerbek 1999	XOR and Sonar	60-140-10 (Maximum)	Add-Shift Arithmetics	Fixed-Point (8-16 bit)

Table 3.2: Summary of various BP algorithm implementation on FPGA

tail review of such schemes provides a guideline for developing architectures for our research and help identify challenges already faced by researchers in this area. Some of the crucial features associated with ANNs include the choice of arithmetic scheme, the type of implementation of activation function, the number representation schemes and performance.

Some of the short comings of the approaches reviewed in the literature are mainly technology related. The FPGAs used in the approaches are few thousand gates in size. This restricted the size of the network that were implemented. CAD tools were not advanced enough to synthesize large designs in a reasonable amount of time. Some of the implementations were merely to verify the feasibility of realiz-

Architecture Author and Year	No. of FPGAs	Type of FPGA	System Clock MHz	Performance WUPS/Speed
RRANN(Eldredge, Hutchings)1994	Multi-FPGA Max. 60	Xilinx XC3090	10-14	12×10^6 Maximum
RENCO(Beuchat, Sanchez)1998	Multi-FPGA Max. 4	Flex 10K130	25	Not mentioned (16bit)
LIRMM(molz, Engel et al)2000	Multi-FPGA 2 Max.	Xilinx XC4013E	20	Not mentioned
(Girau, Tisse-rand)1996	Multi-FPGA	Xilinx XC4025	33	5.2×10^6
ACME(Ferrucci, Martin)1994	Multi-FPGA 14 Max.	Xilinx XC4010	10	1.64×10^6
(Ossoinig, Reisinger, et al)1996	Multi-FPGA 4 Max.	Xilinx XC4013	Not mentioned	Inferior to S/W
(Lysaght, Stockwood, Law et al)1994	Single-FPGA	ATMEL AT6005	20	0.77×10^6
ECX(Skerbek 1999	Multi-FPGA 2 Max.	Xilinx XC4010	Not mentioned	3.5×10^6

Table 3.3: Summary of various BP algorithm implementation on FPGA Cont'd

ing ANNs on FPGA by introducing variation in some of the aspects/parameters of an architecture such as number codification scheme, arithmetic computation and number of FPGAs from earlier implementations. It can also be noted from the performance data available that in most cases researchers considered the unit of performance as WUPS or CUPS. **Not much stress has been given on comparing the hardware implementation with its software counterpart**. In fact, all ASIC implementations fairly ignored any requirement of comparing the performance with a software implementation.

These short-comings have led to the development of ANN architectures for FPGAs utilizing the advancement in CAD technology. The work carried out in this research will also attempt at addressing the performance issues for various aspects such as speed and chip-area. The research work will focus on developing clear methodology for developing an ANN architecture rather than just investigating the feasibility.

Chapter 4

Experimental Setup

This chapter describes the basic setup and specifications regarding various experiments carried out in our research. Section 4.1 describes various system specifications for hardware and software implementations of the Back-Propagation (BP) algorithm. It also describes the development path of the hardware architectures. Several data sets were chosen to validate the developed architectures. Section 4.2 introduces the benchmarks used in our experiments. Section 4.3 will focus on various performance criteria and how the final result, average error, is computed.

4.1 System Specifications

As a result of the analysis carried out in Chapter 3, specifications for the ANN architectures can be defined as following:

1. **Learning algorithm**: A large field of applications and regular structure

makes the Back-Propagation algorithm a popular choice.

2. **Arithmetic Format**: 16 bit Fixed number arithmetic format will be used. It provides a reasonable trade-off between area and range-precision requirement.

3. **Activation Function**: Non-linearity will be imparted by using the sigmoid function. Various possibilities of efficient implementation will be investigated while developing an architecture.

4. **Performance**: Performance will be observed on Weight Update Per Seconds (WUPS) or Connections Update Per Second (CUPS) unit. An alternative approach will tend to compare the hardware architectures with its counterpart a software implementation.

The BP algorithm is first implemented on a software platform and then architectures are developed for the implementation on an FPGA. Three benchmarks are used to validate the software and hardware designs in the form of XOR logic gate problem, Iris flower data-set and the Cancer data set.

4.1.1 Software Specifications

The following is the software platform used to carry out the experiments.

- The Software implementation is based on:
  - Programming Language: C++
  - Hardware: Dual Processor Pentium III 800MHz
  - OS: Windows XP/2000

  - Compiler: Visual C++ 6.0

The software version incorporates training and testing phase of the BP algorithm. Some of the striking features where software version differs from the hardware architectures are:

- Use of 32 bit floating point number for weights, inputs, error, error-gradient and all intermediate calculations
- Exact calculation of Sigmoid function due to the availability of *pow()* in C++
- Use of rand() function to select the random patterns for the representation
- Use of floating point multipliers, adders and subtractors

The pseudo-code in Figure 4.1 illustrates the software implementation of the algorithm.

The Pattern mode of learning is carried out in training phase of the algorithm. The stopping criteria used in the software version is based on number of epochs or error.

4.1.2 Hardware Specifications

The following is the hardware platform used to carry out the experiments.

- The Hardware implementation is based on:
  - Reconfigurable Platform: Virtex2000e/Virtex1000 FPGA chip on RC1000 Reconfigurable board

```
Start Timer;
Read Topology file;
Read Input and Output data;
Generate random Weights;
for(j=0; j < Max_Epochs; j++)
{
  for(k=0; k < Train_Patt; k++);
  {
    Perform Forward computation;
    calculate Inst. Sum. Squared Error
    and store in an array;
    Perform Backward computation;
    Perform Weight Update;
  }
  Check for the Desired Minimum Error;
}
Stop Timer, Calculate total time;
Perform Testing; //Forward Pass
```

Figure 4.1: Pseudo code for Back-Propagation in Software

- Hardware Description Language: Handel-C DK2 compiler
- Synthesis tool: Xilinx ISE 6.2

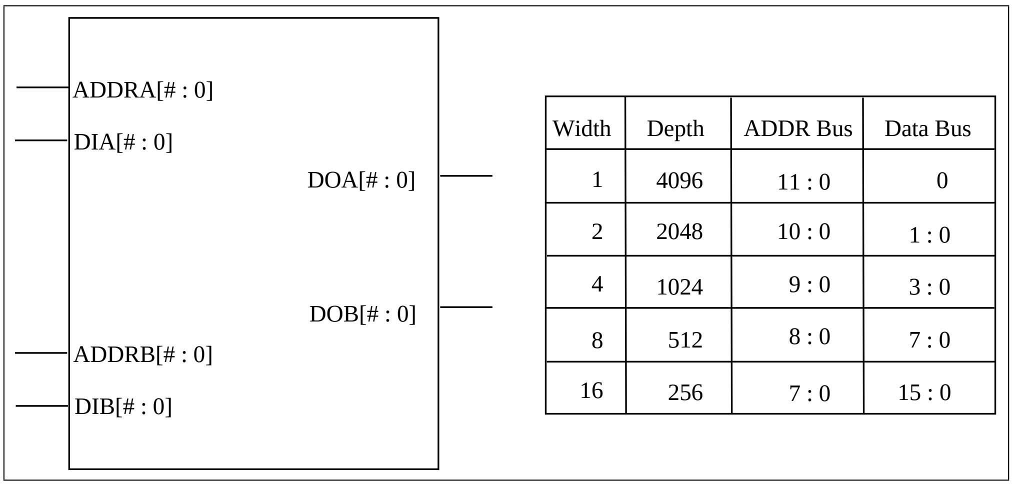
Basics of the RC1000 Reconfigurable board

This section will touch upon basic specifications of RC1000 [Supp01] board since the implementations were carried out on two RC1000 reconfigurable computing platforms supplied by Celoxica Inc. Further details of the RC1000 board is given in Appendix B.

An RC1000 board contains a Virtex1000 FPGA chip with 1 million gates, while the second board contains Virtex2000E FPGA chip with 2 million gates. The main feature of each board is a single FPGA and four Off-Chip asynchronous SRAMs, each of which is an 2M x 8 in size. The basic difference between a Virtex and VirtexE based FPGA chip is the size of incorporated BlockRam memory. The number of BlockRam memory modules in Virtex1000 and Virtex2000E chips are 32 and 160 respectively, each of which is fully synchronous dual-ported 4096 bit RAM. The structure of a BlockRam module in both chips remain the same as illustrated by Figure 4.2 with table showing depth and width aspect ratio.

From Figure 4.2 it can be seen that each BlockRam can be configured to operate as a 256 x 16 size of memory, which is required to accommodate the *width* of fixed point number representation in our designs. Hence the total size of the BlockRAMs in Virtex1000 and Virtex2000E will be set to 8k x 16 and 40k x 16 respectively.

As mentioned earlier, the RC1000 board contains four memory banks and has four 32-bit memory ports one for each memory bank. Each bank has separate



Width	Depth	ADDR Bus	Data Bus
1	4096	11 : 0	0
2	2048	10 : 0	1 : 0
4	1024	9 : 0	3 : 0
8	512	8 : 0	7 : 0
16	256	7 : 0	15 : 0

Figure 4.2: Dual-Port BlockSelectRAM in Virtex and VirtexE FPGAs

data, address and control signals. Therefore the FPGA on the board can access all four banks ***simultaneously and independently***. Data from each bank can be accessed as 8 bit or 32 bit. Details of the interaction between the FPGA and host are explained in Appendix C.

4.1.3 Development Path

The following are the six different architectures developed and tested in the research:

- Two Serial Designs: SIPEX and SIPOB[1]
- Three Partially Parallel Designs: PARIO, PAROO and PARIO
- A Fully Parallel Design: FPAR

The approach behind the development of the above mentioned architectures is shown in Figure 4.3

[1]Acronyms are described in details in Appendix A

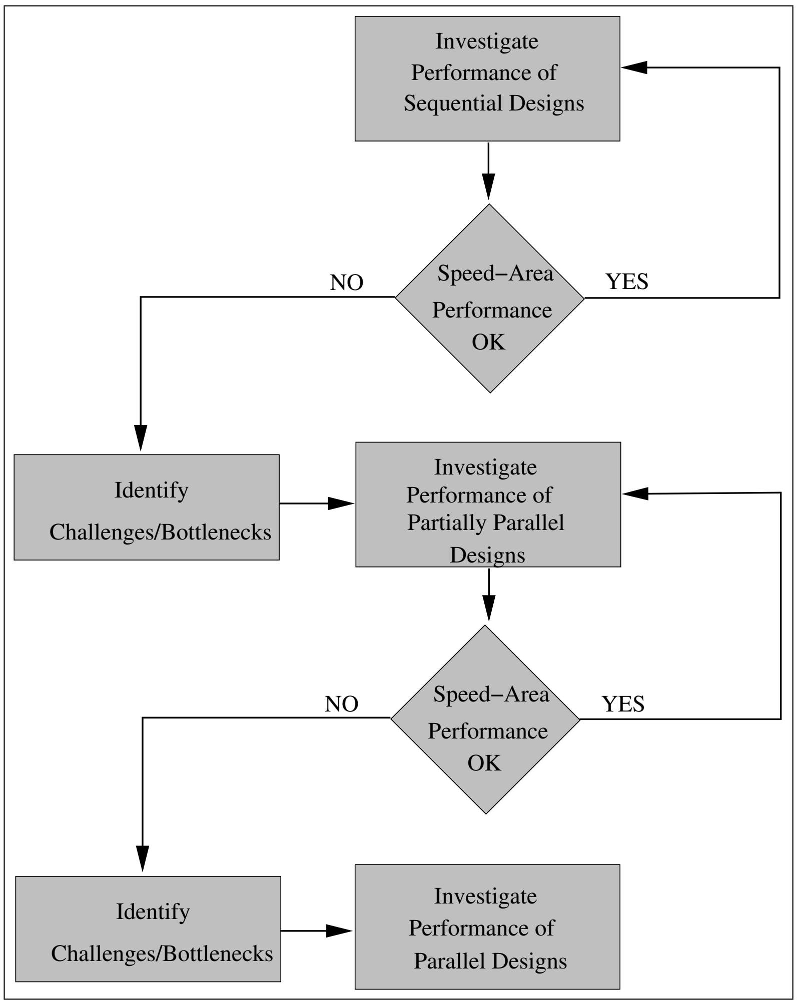


Figure 4.3: The path for the development of the architectures

As shown in Figure 4.3, during the course of research, first several serial architectures are developed and tested on the RC1000 FPGA board. This experimentation helped identify several issues related to Handel-C and the RC1000 board. The performance is measured in execution speed of the algorithm on the FPGA and chip-area consumed by the architectures. In second phase of the research, partially parallel architectures are developed and tested on the RC1000 board. The purpose of the development of such architectures is to establish a trade-off between speed and chip-area. Finally, a fully parallel architecture is developed to maximize the performance by exploiting the inherent parallelism of ANNs and validated for several benchmarks.

4.2 Benchmarks

Several benchmarks are used to validate the architectures developed for the hardware implementations. In this section we will describe the characteristics of the benchmarks that are used to evaluate performance of the developed architectures. ANNs are widely used to solve the problems in the field of image processing, pattern recognition and data mining. The nature of input and output to achieve desired task through ANNs varies for different applications. This in turn also categorizes the functionality of an each application that can be solved by ANNs. With respect to functionality, benchmarks for ANNs can be categorized for association, classification and function approximation problems [Poor02] as shown in Table 4.1.

Since the Back-propagation algorithm for ANN is suited for pattern classification problem, focus is given to such benchmarks. There are several sources available on

Input	Output	Task	Evaluation Criteria
discrete	discrete	association	storage efficiency
continuous	discrete	classification	error probability
continuous	continuous	function approximation	distance measure

Table 4.1: ANN application fields

the internet that provides benchmarks for ANNs. Since the application field of ANN is vast, it is observed that the benchmarks used by various researchers to implement ANNs on hardware vary significantly. Table 4.2 shows the benchmarks used for experiments described in this thesis.

Problem Title	Type	Network Structure	Error Goal	Applicability
XOR	Pattern Recognition	3-3-1	0.0022	"Toy" Problem
Iris	Pattern Recognition	5-3-3	0.03	"Toy" Problem
Cancer	Pattern Recognition	10-11-2	0.016	"Real World" Problem

Table 4.2: Benchmarks

4.2.1 XOR Data Set

The *Exclusive OR* benchmark is a common example of classification of input patterns that are not linearly separable. For patterns to be classified as linearly separable, they must be sufficiently apart from each other to ensure the decision surface consists of a single straight line in two-dimensional plane. This concept is illustrated in Figure 4.4. The Table 4.3 shows the truth table of the logical XOR function.

In the case of XOR we need to consider four corners of the unit square that

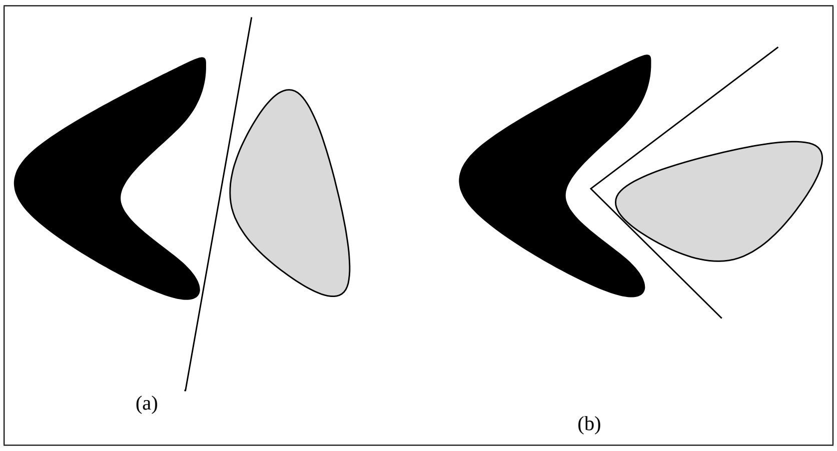


Figure 4.4: A pair of (a) linearly and (b) non-linearly separable patterns

X	Y	Output
0	0	0
0	1	1
1	0	1
1	1	0

Table 4.3: XOR function

correspond to the input patterns. Figure 4.5 shows the locations of the symbolic outputs of XOR function corresponding to four input patterns in X-Y plane.

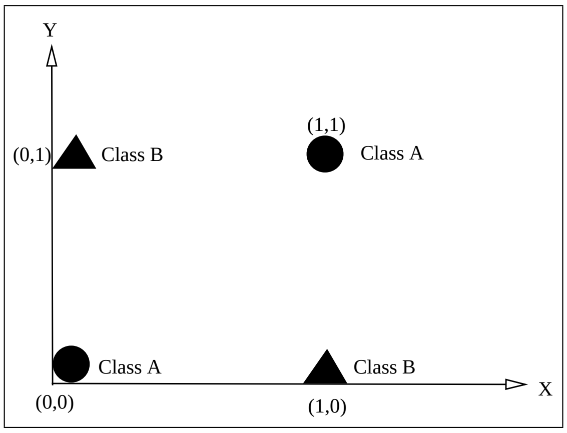


Figure 4.5: Non-linearity of XOR function

Polar opposite input pairs (0,0) and (1,1) require the same output in class A and also input patterns, lying in opposite corners, (0,1) and (1,0) require the same output in class B. It is obvious that there is no way to draw a single straight line to classify patterns in two classes so that the circles are on one side and the triangles are on the opposite side. Hence Multi-layer perceptrons are applied to solve the XOR function. The network used for this problem is 3-3-1 (including bias) with sigmoid function in each layer.

4.2.2 Iris Data Set

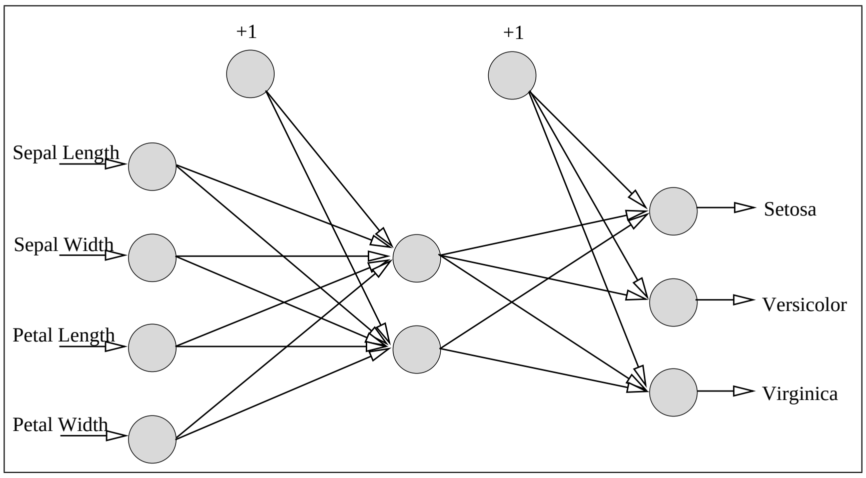


Figure 4.6: ANN for Iris Data Set

The Iris flower data set is a popular multivariate data that was introduced by [Fish36] as an example for discriminant analysis. Figure 4.6 shows a multilayer perceptrons network for the benchmark. The data reports on four characteristics of the three species of the Iris Flower, sepal length, sepal width, petal length, and petal width in centimeters. The data set shows each characteristic (attribute) scaled up ten times to represent all values in integers. The three species of the flower are *setosa*, *versicolor* and *virginica.* Figure 4.6 shows the multilayer perceptrons with 5-3-3 structure for Iris data set. The data set contains 150 instances for each of the four characteristics. The goal of using an ANN algorithm for this data set is to classify the Iris flower in one of the three species mentioned above. The network used for this problem is 5-3-3 (including bias input) with sigmoid function in each layer. The attributes are normalized to represent continuous values between 0 and 1 for network inputs as following:

$$\frac{N - N_{min}}{N_{max} - N_{min}} \tag{4.1}$$

where, N : Attribute value

N_{min} : Minimum value of each attribute

N_{max} : Maximum value of each attribute.

4.2.3 Cancer Data Set

The cancer data set problem is a realistic pattern recognition problem. The objective of the network is to classify tumor as either **benign** or **malignant** based on cell descriptions gathered by microscopic examination. Input attributes include clump thickness, uniformity of cell size and cell shape, the amount of marginal adhesion, and the frequency of bare nuclei. In this data set, each of these attributes is normalized by authors to represent continuous values in numbers between 1 to 10. The data set was originally generated at University of Wisconsin by Dr.Wolberg. The Cancer data sets are obtained from UCI learning repository databases [Blak98].

Group 1 of the data set contains 367 instances out of which first 300 instances are used for the validation. The network used for this problem is a 10-11-2 network (including bias input) with sigmoid function in each layer.

4.3 Evaluation Parameters

All the architectures described in this thesis are designed to execute the training phase of the BP algorithm only. Hence our goal is to achieve convergence of the algorithm at the end of the training. We will discuss various performance criteria considered in comparing the outcome of software and hardware implementations. We will also focus on various criteria used to stop the training of the network and how error is computed in determining the convergence.

4.3.1 Comparison Criteria

Hardware and software implementations of the BP algorithm are compared based on several criteria.

Speed: This performance criteria measures the time hardware and software implementations take to execute the BP algorithm. For hardware implementations, total time taken by the algorithm is the time lapsed between the start and end of the timer set by the C++ program running on the host as shown in flowchart in Appendix C.

Weight Updates Per Second: This performance criteria measures an architecture's ability to compute weights per second for hardware implementations.

Gate Count: Measurement of the equivalent gate counts on an FPGA evaluates the performance of the hardware implementation based on the area requirement.

4.3.2 The Error and the stopping criteria

The parameter of our interest or goal in achieving the convergence of the network is Average Sum Squared Error (ASE) per epoch. To calculate this, an Instantaneous Sum Squared Error (ISE) is obtained for every pattern in an epoch and averaged for number of patterns in the epoch. The ISE is obtained by calculating sum-squared error at neuron outputs for each iteration and thus can be written as,

$$ISE = \sum_{j \in C} e_j^2(n)$$

where, set C includes all the neurons in the output layer of a network and,
e_j is the error at node j of the output layer.

The ASE is obtained by summing and averaging ISE over N patterns and thus can be written as,

$$ASE = \frac{1}{N} \times \sum_{n=1}^{N} ISE$$

where, N is the number of patterns in an epoch.
For a given training set, ASE represents the *cost function* as the measure of learning performance. Hence the goal of the learning process is to minimize the cost function.

In the current implementations, ASE is not calculated by the FPGA but instead ISE is computed for every pattern during the learning phase and stored in the external RAM. At the end of the execution the host fetches all the results of ISE from the external RAM and calculates ASE. The reason behind such division of computation is twofold:

- ASE involves the addition of ISEs for whole epoch which in most cases will overflow for the "range" specified in the Fixed-Point representation. For example, in case of the Iris data set, the early stage of the training was found to generate ASE around 0.32 which would result from the addition of ISEs to 32 and then averaging it over 100 patterns. Since the largest positive integer our fixed-point number representation can accommodate is 15 so the result will overflow in this case.
- ASE also requires the division operation which is normally prohibitive to realize on hardware.

The drawback of such an approach is that it requires large memory for storing ISEs. Fortunately, each memory bank of the four available external asynchronous RAM banks is sufficiently large and therefore memory bank 3 is dedicated for storing ISEs.

Another point that requires an attention in the execution of the BP algorithm is the stopping criteria used to mark the convergence and stop execution. The hardware architectures presented in this thesis use number of epochs as their stopping criteria. ASE is not computed on the FPGA and therefore it is not possible to know if the desired goal has been achieved during the learning. Therefore, the algorithm is executed for specific number of epochs, the value of which is required to be set at the compile time of the code. The required number of epochs are determined from the software runs of the algorithm for different topologies of each benchmark. Hence the comparison of all our experimental results in this thesis is based on achieving desired ASE with a priori knowledge of required epochs from the software implementation.

4.3.3 Performance measurement

In this section we will describe how the performance, the time to execute the algorithm, is measured in software and hardware versions. The time is calculated in milliseconds and it includes following operations for each of the versions.

Software Version:

- Read topology and initial parameters (Input, Output and Weights): It involves memory access (read/write file operation).
- Start the execution of the algorithm and accumulate ISEs for each pattern in an epoch. calculate ASE and store it in an array (No file operation).
- Complete the execution of the algorithm after several number of epochs.
- Timer stops. Calculate the total time taken for the execution.
- Store ASE in the memory (file operation) to check the convergence.

Hardware Version:

- Host program sends a start signal to FPGA and timer starts
- The FPGA fetches initial parameters and topology from Off-Chip RAMs of the RC1000 board and transfer it to BlockRAMs

- Start the execution of the algorithm and store ISE for each pattern in Off-chip RAM.
- Complete the execution of the algorithm after several number of epochs
- FPGA sends an end signal to the host program and timer stops. Calculate the total time taken for the execution.
- The host program fetches ISEs from Off-chip RAMs and calculates ASE for each epoch to check the convergence.

The software and hardware version differ in the storage of ISEs. The hardware versions require access to off-chip memory on the RC1000 board to store the ISEs due the reasons mentioned in section 4.3.2. Whereas, the software version doesn't store ISE instead it accumulates for each pattern. Hence it can be summarized that,

Total time taken by the Hardware Versions = Time taken by FPGA to execute the algorithm + Off-chip RAM access time (Transfer of Initial parameters and Topology to BlockRAMs) + Off-Chip RAM access time (Storage of ISEs for each pattern)

Total time taken by the Software Version = Time taken by CPU to execute the algorithm + Hard-Drive access time (Transfer of Initial parameters and Topology)

4.4 Summary

In this chapter system specifications related to hardware and software implementations were described briefly. The basic design of the RC1000 platform was also described. Three benchmarks XOR data set, Iris data set and Cancer data set were introduced to validate the architectures developed in this research work. A brief introduction on various performance measurement units was given.

Chapter 5

Serial Architectures

This chapter describes serial architectures/designs developed for implementing the Back-Propagation (BP) algorithm for Artificial Neural Network(ANNs) on an FPGA. A detailed description of each architecture and results obtained is presented in the following sections. Section 5.1 describes the Fixed-point arithmetic scheme used throughout the research work. In order to randomize the pattern presentation to the network, some form of random numbers are required. Section 5.2 sheds some light on pseudo-random number generation based on Linear Feedback Registers. Section 5.3 explains the serial architectures in detail and finally 5.4 discusses the outcome of the implementation.

5.1 Fixed-point Number Representation

Fixed-point arithmetic is generally used when hardware cost, speed, or complexity is important [Nich03]. Finite-precision quantization issues usually arise in fixed-point systems. As described in Chapter 3 various researchers have used different formats in fixed-point number representation. A hardware designer has to deal with an issue associated with establishing a trade-off between hardware resources consumed by the datapath and the convergence time required to achieve performance. The choice of range-precision of a chosen fixed-point number directly affects the ability of a network in determining a compromise between chip-area and performance. However the trade-off between performance and hardware resources is application specific. [Holt91] studied this issue and showed that 16-bit fixed-point number provided the most optimal area vs. convergence rate trade-off for generic back-propagation algorithm, whose application is not known a priori. Hence the choice for the number of bits in our research is set to 16 bit-fixed point number representation.

Software runs of the BP algorithm were performed specifically to observe the "range" of the weight and all other intermediate signal values during the course of convergence. From these runs, a 5-bit length of integer part (including Sign bit) and 11-bit length of fraction part in a fixed-point number is found to be satisfactory. Such 16-bit fixed-point format of 1 sign bit, 4 integer bits and 11 fraction bits was adopted as illustrated by Figure 5.1.

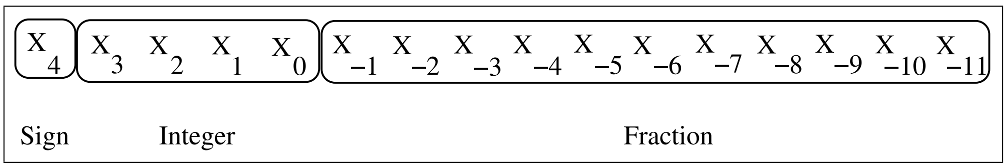

X_4
X_3 X_2 X_1 X_0
X_{-1} X_{-2} X_{-3} X_{-4} X_{-5} X_{-6} X_{-7} X_{-8} X_{-9} X_{-10} X_{-11}
Sign
Integer
Fraction

Figure 5.1: 16 bit Fixed-Point Number

The most significant bit X_4 represents the sign bit while $X_3 - X_0$ and $X_{-1} - X_{-11}$ represent the integer and fraction bits respectively. This number representation will be denoted as **1:4:11** in the rest of the thesis.

The logic to compute the integer equivalent of binary can be extended to computing a fixed-point number. The fixed-point number can therefore be computed as following:

$$(-1)^{X_4} \times (X_3 2^3 + X_2 2^2 + ... + X_0 2^0) + (X_{-1} 2^{-1} + X_{-2} 2^{-2} + ... + X_{-11} 2^{-11}) \quad (5.1)$$

Two of the most commonly used Fixed-point number notations are sign-magnitude and two's complement. The sign-magnitude notation is used throughout our research. There are few reasons behind the choice of the sign-magnitude notation. The arithmetic operations in sign-magnitude notation can be performed with **15-bit unsigned** fixed-point numbers as opposed to **16-bit signed** fixed-point numbers in two's complement notation. This might prove beneficial in terms of chip-area. A sign bit is treated separately in the sign-magnitude notation. Debugging is also easier in case of the sign-magnitude notation. In addition the fixed-point library for Handel-C had a serious glitch for signed arithmetic operations in early versions of DK compiler.

The 16-bit fixed-point Arithmetic format adopted here has the following characteristics:

- It produces a representation error of 2^{-11} which is in the order of 10^{-4}. The representation error is also called the resolution of the fixed-point number. Every increment/decrement in a fraction bit will increase/decrease the number by approximately 0.0005.

- It can produce a range of integer values from -15 to 15. The maximum value can be represented by the number system is $2^4 - 2^{-11} \approx 15.9995$.

- Examples of two numbers, a negative and a non-negative number, in sign-magnitude number system can be given as,

$$-11.6 \approx 11011.10011001100 \approx -11.5996$$

$$7.9 \approx 00111.11100110011 \approx 7.8999$$

The following section describes a random number generator popularly known as Linear Feedback Shift Register (LFSR). LFSR based design is widely used for generating pseudo random numbers. In our research we have used LFSR based design for randomized presentation of training patterns to the network.

5.2 Linear Feedback Shift Register(LFSR)

LFSR is a sequential shift register with combinational feedback logic around it that causes pseudo-random cycle through a sequence of values. LFSR is simple to realize on hardware and also consumes less resources. Typical applications include: Counters, Built-in Self Test (BIST), pseudo-random number generation, data encryption and decryption. Although LFSR tend to follow a uniform pattern in the generation of random numbers, the randomness provided in the presentation of patterns is found to be sufficient in our experiments.

An LFSR when clocked advances the signal through the register (flip-flop) from one bit to the next most significant bit. Some of the outputs of registers are combined in XOR configuration to form a feedback mechanism. This mechanism feeds back the outputs formed by the combination of one or more XOR into the input of one of the flip-flop. The selections points where outputs are selected for XORing are also termed as “taps”. The choice of taps determines how many values there are in a given sequence before the sequence is repeated. Certain tap settings yield maximal length sequence of $(2^n - 1)$. Figure 5.2 shows 3-bit LFSR with taps at register bits (1,2).

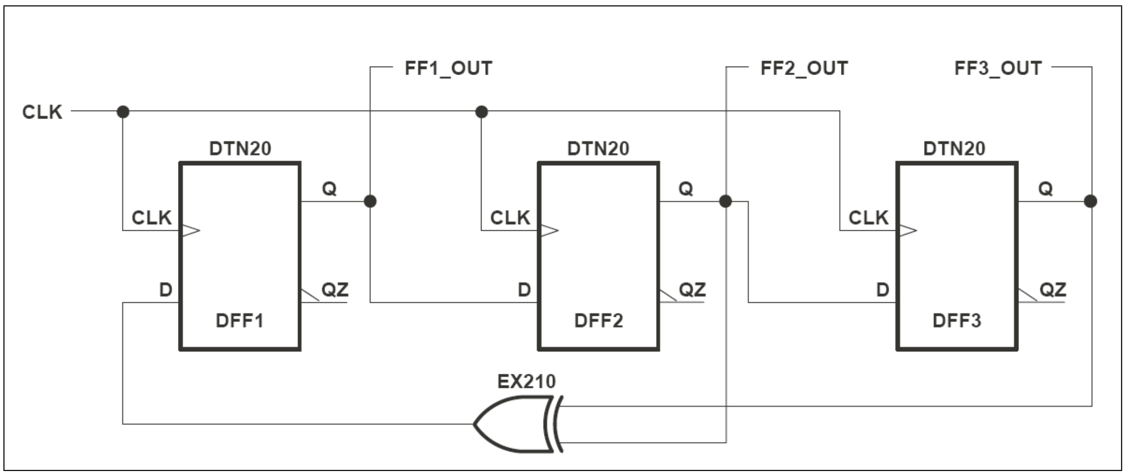


Figure 5.2: 3-bit LFSR

Table 5.1 illustrates the sequence of the pseudo-random values generated by the above 3-bit LFSR.

Clock Pulse	Random Sequence(FF1, FF2, FF3)	Comments
0	0, 1, 1 (3)	Seed Value
1	0, 0, 1 (1)	
2	1, 0, 0 (4)	
3	0, 1, 0 (2)	
4	1, 0, 1 (5)	
5	1, 1, 0 (6)	
6	1, 1, 1 (7)	
7	0, 1, 1 (3)	Starts to repeat

Table 5.1: Pseudo-Random sequence of 3-bit LFSR

From Table 5.1 it can be seen that the sequence is repeated after going through 7 ($2^n - 1$) different values. A sequence produced by 'n' length LFSR which has $2^n - 1$ different values is called a **PN-Sequence** (Pseudo-Noise sequence). For any given width LFSR there are many tap combinations that give maximal length sequences. Fortunately, these tap combinations for LFSRs ranging from 2 to 32 bits are easily found in various related literature. For example 7 and 8 bit LFSR, which will mainly be of our interest because of the number of the training patterns in the benchmarks, has taps at (0,6) and (1,2,3,7) respectively that yield the maximal length of sequence.

Another point that designers have to consider while designing LFSR is that all zeros in a sequence is not possible unless the seed value is chosen to be all zeros. However this is a prohibited state since the LFSR will continue to shift all zeros indefinitely using XOR operations. Therefore a designer has to make sure or provide a seed number which is not zero at power-up of the circuit.

Two ways of generating a random sequence through LFSR are tested in our experiments while implementing the BP algorithm on FPGA. These are:

- Generate a sequence for an epoch before its presentation or in other words before the start of the algorithm computation for an epoch
- Generate a random number for next pattern in parallel to the computation of the algorithm for previous pattern, **ideally** eliminating a need of any extra clock cycles for the random number generation

In our serial implementations, we have used the first approach because of its simplicity. Whereas all partially parallel and a fully parallel architectures use the second approach to generate random numbers.

5.3 Serial Implementations

This section describes two serial implementations of the BP algorithm carried out on the RC1000 board consisting of Virtex1000/Virtex2000e FPGA chip.

The goal of designing an initial architecture that executes the algorithm in a serial fashion was to lay the foundation for developing a parallel architecture. Some of the challenges faced in designing the serial architectures will attempt to serve as a guideline for enhancing the final parallel designs. Prior to the description of the architecture aspects pertaining to the sigmoid function will be explored.

An ANN model of Multi-layer Perceptrons (MLP) employing the back propagation algorithm requires the use of a non-linear activation function at the output of each neuron. For experiments carried out for the research in this thesis, we will consider the Sigmoid Function as the activation function in the implementation of MLP model. The Sigmoid function is given by,

$$f(x) = \frac{1}{1+e^{-x}} \tag{5.2}$$

where, $-\infty < x < \infty$

Efficient implementation of the sigmoid function on an FPGA is one of many difficult challenges faced by designers. This is because of the requirement of division operation and the calculation of exponent part in equation 5.2. Hence in most cases computationally simplified alternatives of sigmoid function are used.

One approach attempts to store pre-calculated output of the sigmoid function in LTs. In this approach, the output of the sigmoid function is precalculated for each possible value of input, which includes every increment of precision starting from the lowest number and ending with the highest number possible in a given number representation. The computed output values are then stored in LTs - mostly on/off-chip RAM or BlockSelectRAM of FPGAs. Although arithmetic operations in our implementation were based on 16 bit fixed-point numbers, the length of the LT for the sigmoid function can be curtailed to 8k. In this case 3 bits were omitted off the input to the sigmoid by considering only 13 bits without significant loss of range or precision as shown in Figure 5.3

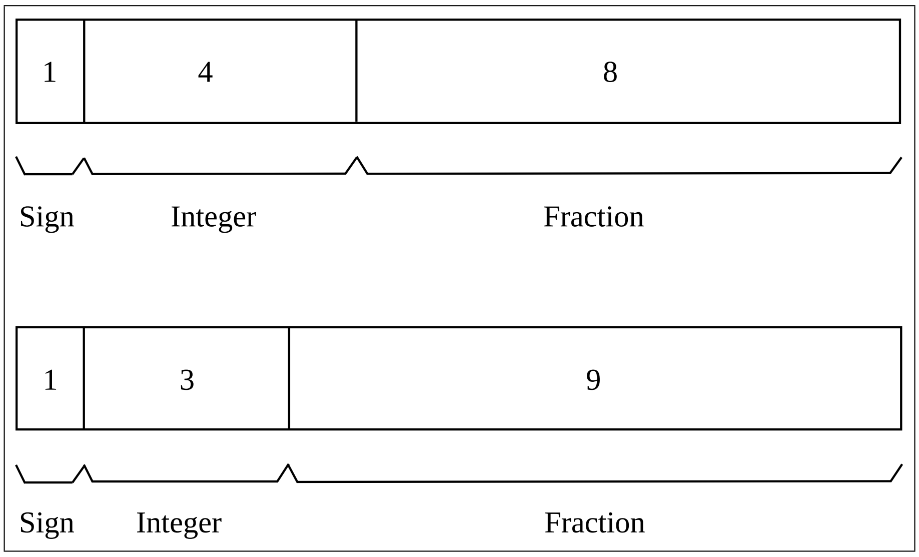


Figure 5.3: Reduced Fixed point number to Sigmoid input

As shown in Figure 5.3 one way to curtail the number is by omitting three bits in the fractional part to make it a 1-4-8 structure. Another way to decrease the size of the number is to omit two bit in fraction and one bit in integer resulting in a 1-3-9 structure. An omission of a bit in integer and thus reducing it to 3 bits will not affect the range as it can be seen from the Figure 2.5 that the function saturates to 1 or 0 as input approaches |7|, a limit well within the range for 3 bits. The benefit of the second way over the first approach is the higher precision.

The two serial implementations differ in nature by means of storing initial parameters and LTs. The difference are:

- **S**erial **I**mplementation with **P**aramertes/LT in **EX**ternalRAM (SIPEX)
- **S**erial **I**mplementation with **P**aramertes/LT in **O**n-chipRAM and **B**lockRAM (SIPOB)

Various types of possible memory creation on Virtex series FPGA chips are described in Appendix D. Some of the common features of both architectures:

- 16 bit Fixed-point number representation in the form of 1:4:11 (Sign:Integer:Fraction). Fixed-point library supplied with Handel-C compiler is used to encode the representation.
- Random number generation by employing LFSR
- Sigmoid function realization based on Look-Up Table technique
- Pattern Mode of learning

Figure 5.4 illustrates the method of computation common in both architectures.

The dark arrow in Figure 5.4 represents the direction of the computation and shows that synapses are being computed one at a time (serial). The approach used

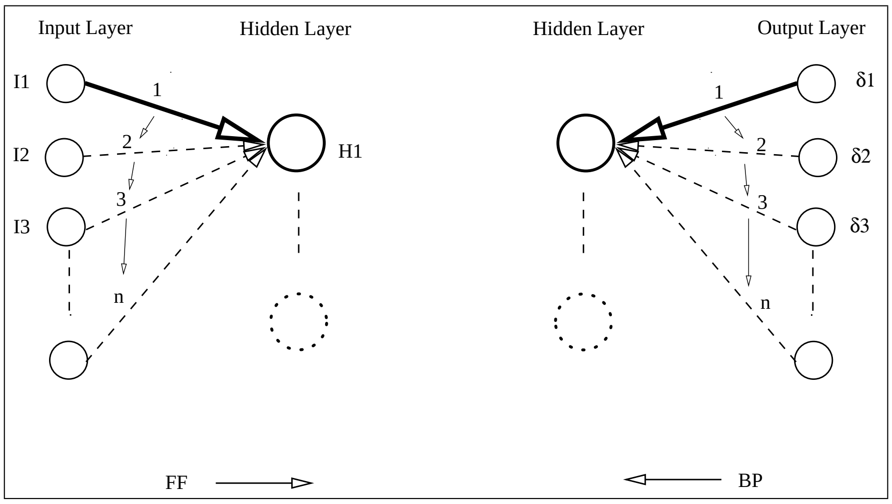


Figure 5.4: Computation of the BP algorithm in Serial Fashion

to encode the algorithm in programming language is to divide it in the following three time-exclusive stages. Each of the stages consist of several modules. Both SIPEX and SIPOB architectures have the following similar modules:

- Feed-Forward Stage
  - Multiply-Accumulate module facilitates computation for one layer
  - Error Calculation module
  - Sigmoid function module: address generation and data retrieval from LTs
- Back-Propagation Stage
  - Module for Output gradient calculation
  - Module for Hidden Layer gradient calculation
  - Module for calculating Derivative of sigmoid

- Weight Update Stage
    - Module for weight-update

These designs have a single arithmetic module consisting of a multiplier, adder and subtractor. This enables the designs to compute **one** synapse (multiplication) in one clock cycle. The serial designs proposed are not purely sequential in nature like GPP. For example, the multiplication process can be pipelined with the accumulation process. The start of the accumulation process is delayed by the time required to execute one multiplication-process. In other words, if one process is executing a multiplication on neuron $n+1$, another process executes an accumulation at neuron n in parallel. For the feed-forward stage, the combined multiplication-accumulation process can be further pipelined with a sigmoid function calculation process. If we denote || for a pipeline implementation, a rough depiction of computation at **a single neuron** in a layer in the feed-forward stage can be given by:

((Multiplication || Addition) || Sigmoid Calculation)

Similarly, for the Back-Propagation and weight-update stages, we can write:

((Multiplication || Addition) || Derivative of Sigmoid)

((Multiplication || Calculation of $\triangle W$ and Updated weights)

An Algorithmic State Machine (ASM) for the feed-forward stage in SIPEX and SIPOB architectures is illustrated in Figure 5.5.

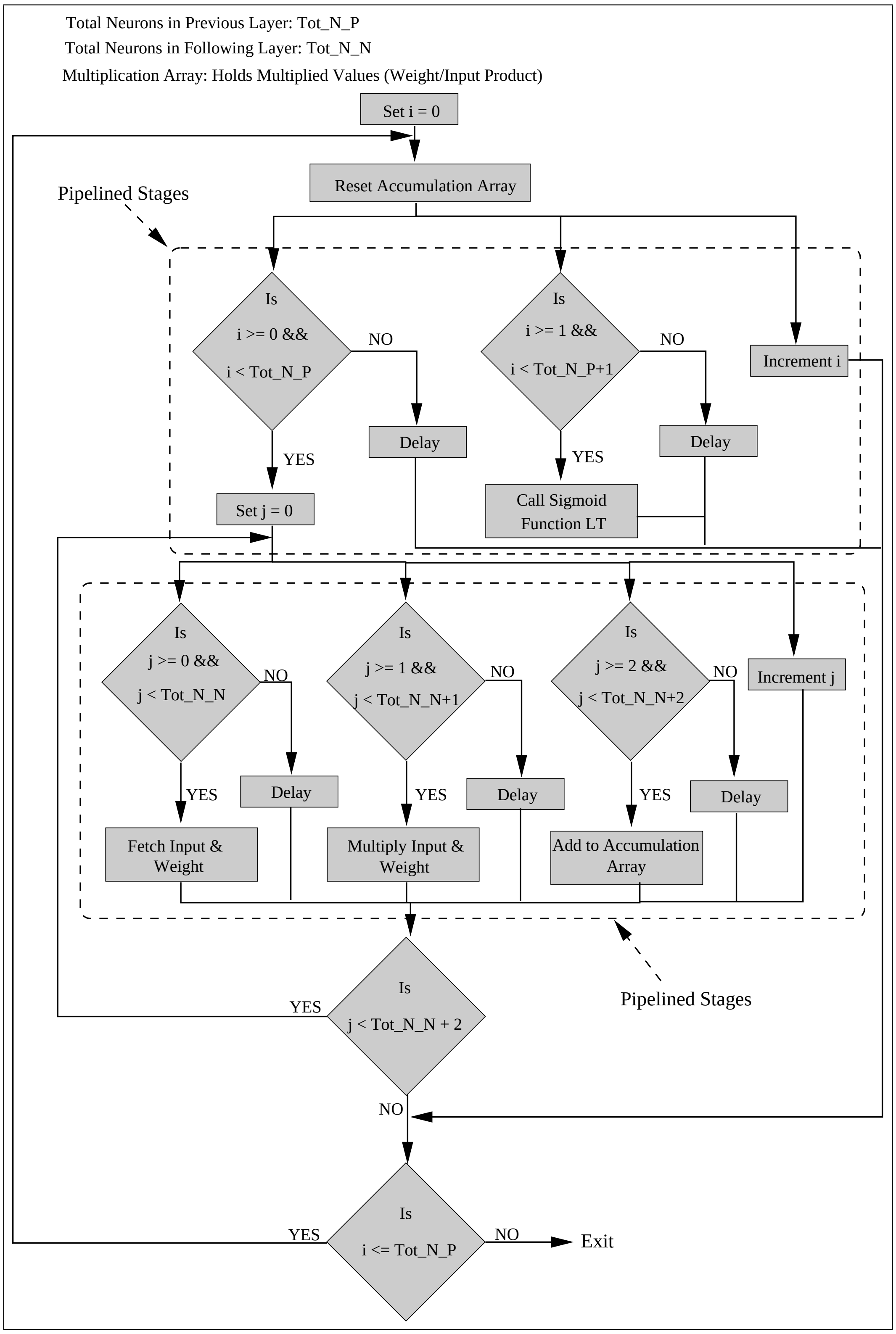


Figure 5.5: ASM for the Feed-Forward stage in SIPEX and SIPOB

5.3.1 SIPEX

This architecture is a result of an early stage of experimentation and aimed at understanding and evaluating some fundamental aspects of ANN implementation on the hardware platform available for this research.

In this architecture Lookup Table (LT) for the sigmoid function is stored in Off-Chip SRAMs. The module responsible for calculating sigmoid function will generate address to access Off-Chip SRAMs. For Virtex and VirtexE FPGA on RC1000 board, external clock, which is faster than the internal clock, is divided by four to time the Off-Chip SRAMs during read and write operations. The LT was implemented on Memory Bank 3 for 8k input values of the sigmoid function. The advantage of such an implementation is that the area utilized is minimal. However the drawback is that such implementation is inherently serial and not suitable for parallel scaling [Savi04]. Since each memory bank can be accessed once in a clock cycle, at the most four sigmoid functions can be calculated simultaneously if the same LT is copied to the available memory banks. Another drawback of realizing LT on Off-chip RAM is the need for the division of fast external clock. This scales down the frequency to $\frac{1}{4}$ of the external clock for the complete design.

The main feature of this architecture are:

- It stores initial parameters such as input, output and weight values and Look-Up table(LT) for Sigmoid function in Off-Chip asynchronous SRAMs on RC1000 board.
- As mentioned in Chapter 4 the RC1000 board consists of 4 memory banks, each of the size 2M x 8. The initial parameters are stored in memory bank 0

and LT for the sigmoid function is stored in memory bank 3. Memory bank 1 is reserved for storing the instantaneous sum squared error for each pattern. These values on completion of execution of the algorithm will be transferred to host to calculate Average Sum squared Error.

- Each memory bank can be accessed once in a clock cycle. Hence this is a serial transmission of data from/to bank to/from FPGA chip. However, all four banks can be accessed simultaneously.
- Maximum number of neurons that can be accommodated in each layer are set at 32. However this is not the maximum limit set by the FPGA chip on the board and can be further increased.

5.3.2 SIPOB

This architecture was developed following SIPEX to explore the possibilities of enhancing the network performance by using BlockRam to store LTs for the sigmoid function and On-chip RAMs to store initial parameters.

BlockRAMs are chosen to store LTs of the sigmoid function to circumvent some of the problems associated with the use of Off-Chip SRAMs as described in Section 5.3.1.

Figure 5.6 shows a block diagram of LT implementation of the sigmoid function in BlockRAMs. It uses the method of reducing the width of a sigmoid input/neuron output as described in Section 5.3. A comparator checks the sign and compares the input value to a maximum limit of 7.9980. On crossing the limit on either side of zero will generate saturated output of '0' or '1'. All other input values within the

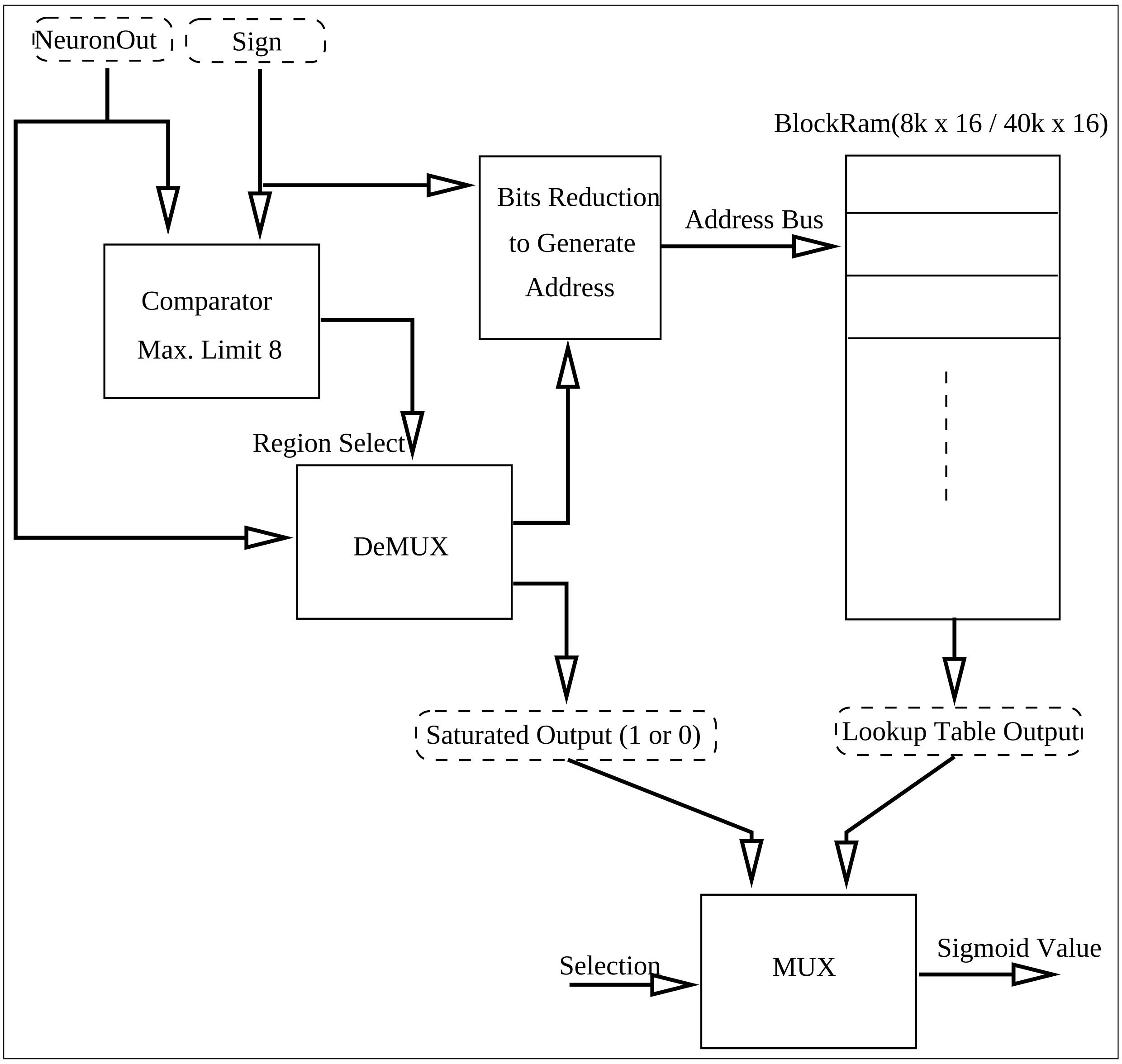


Figure 5.6: Look-Up table for Sigmoid function implemented in BlockRam

limit go to Address generator to generate an address for the BlockRAMs.

The problem arises in case of Virtex1000 FPGA chip which contains only 8k x 16 size BlockRAM, resulting in a complete usage for LTs and leaving no memory for the initial parameters. Here all BlockRam modules (32 in case of Virtex1000) are **combined** to form a single unit of memory for the LTs. It is found that such an approach, which is the only possible way of forming a large unit of memory, levied high logical cost and routing resources. This is because the extra chip-area is consumed by multiplexing units which are needed to cover address and data bus of each BlockRAM module in the unit. Consequently, it affects the speed with which design can execute the algorithm. It is observed through separate practice example that if BlockRam modules are used individually (256 x 16 bits) for read and write operations, it can attain maximum frequency possible, 100Mhz in Virtex1000 chip. Every addition of a BlockRAM module to form bigger unit of memory and accommodate more data further decreases the frequency with which read/write operations can be performed. This approach is also not suitable for parallel scaling since generating multiple copies of LT will not be area efficient.

The main features of this architectures can be summarized:

- It stores initial parameters in On-chip RAMs and LT for sigmoid function in BlockRAMs. This would eliminate the need of accessing the external asynchronous RAMs.
- Maximum number of neurons that can be accommodated in each layer are set at 12. The maximum number of patterns in a data set allowed is 255. These limits are imposed by the size of the FPGA chip.

5.4 Results and Analysis

In this section we will present the results obtained by the successful implementation of the serial architectures for several benchmarks. A detailed analysis of the results will be presented.

Tables 5.2, 5.3 and 5.4 show the time taken by SIPOB architecture to execute the BP algorithm for different benchmarks.

LR	Epochs	SIPOB Time(mS)	Software Implementation Time(mS)
0.25	7000	440	240
0.5	3300	220	125
0.75	2200	140	78
Frequency(MHz)		25	

Table 5.2: XOR data set experiments for 0.0022 ASE

LR	Epochs	SIPEX Time(mS)	SIPOB Time(mS)	Software Implementation Time(mS)
0.1	242	1000	500	250
0.15	160	675	340	170
0.2	123	531	270	140
0.25	101	438	210	100
0.3	83	375	180	90
Frequency(MHz)		25	25	

Table 5.3: IRIS data set experiments for 0.03 ASE

- It should be noted here that the software version of the algorithm are run for the same number of epochs as the hardware versions to compare timings.

LR	Epochs	SIPEX Time(mS)	SIPOB Time(mS)	Software Implementation Time(mS)
0.3	221	6910	3000	1100
0.4	166	5000	2300	780
0.5	135	4125	1900	630
Frequency(MHz)		25	25	

Table 5.4: Cancer data set experiments for 0.016 ASE

However the software versions are found to take 5% to 10% less epochs to reach the desired goal (ASE). Hence the network for the software version will be slightly over-trained.

- One of the difficult challenges faced while implementing the SIPEX on RC1000 board is that the Handel-C requires an internal clock of $\frac{1}{4}$ of the external frequency to time the Off-Chip RAMs. The problem with Handel-C is that it scales down the frequency of the whole design by 4. The RC1000 board allows maximum 100MHz external frequency to clock the FPGA chip. Hence the maximum internal frequency with which design can run will be 25MHz. This frequency corresponds to the delay of $\frac{1}{25\times10^6} = 40\ ns$ on hardware. Any delay in the routing + logic more than 40ns will cause the frequency to go down further. This bottleneck was encountered while implementing SIPEX, the first serial architecture, on the FPGA chip. This has led to exploring the possibility of realizing the designs in two clock domains in which the access to the Off-chip RAM is isolated from the main computation and runs at lower frequency.

- As illustrated from Tables 5.2, 5.3 and 5.4 the SIPOB architectures is inferior to the software version running on General Purpose Processor. SIPOB, the serial hardware architecture, running at 25MHz is slower than the software version running on dual processor PIII 850MHz. Also the high consumption of area/routing resources for the SIPOB implementation does not allow for higher clock frequency.

Table 5.5 shows the Weight Updates Per Second(WUPS) achieved by SIPOB.

	Weight Update/Second(WUPS)in million	
Benchmarks	SIPEX	SIPOB
XOR	-	0.6
Iris	0.45	0.9
Cancer	0.8	1.8

Table 5.5: Weight Updates per Second

It can be observed from Table 5.5 that for SIPEX and SIPOB architectures, WUPS increases with an increase in the size of the network. This behavior can be attributed to the fact that the serial designs are not completely sequential in nature like the general purpose processor. Although the computation nature of SIPOB dictates it to one synapses at a time, some of the operations associated with it can be pipelined. Figure 5.7 shows such pipeline for a neuron in FF stage and calculates the number of clock cycles for a layer in XOR and Iris.

As per the Figure 5.7, XOR takes 4.66 cycles/weight in FF while Iris takes 4 cycles/weight. Such calculation can be derived for rest of the stages too and will be found to result in less number of clock cycles per weight as the network grows. Hence the larger the network the better the WUPS for the same architecture.

Table 5.6 shows the are requirement in terms of the equivalent gate counts on

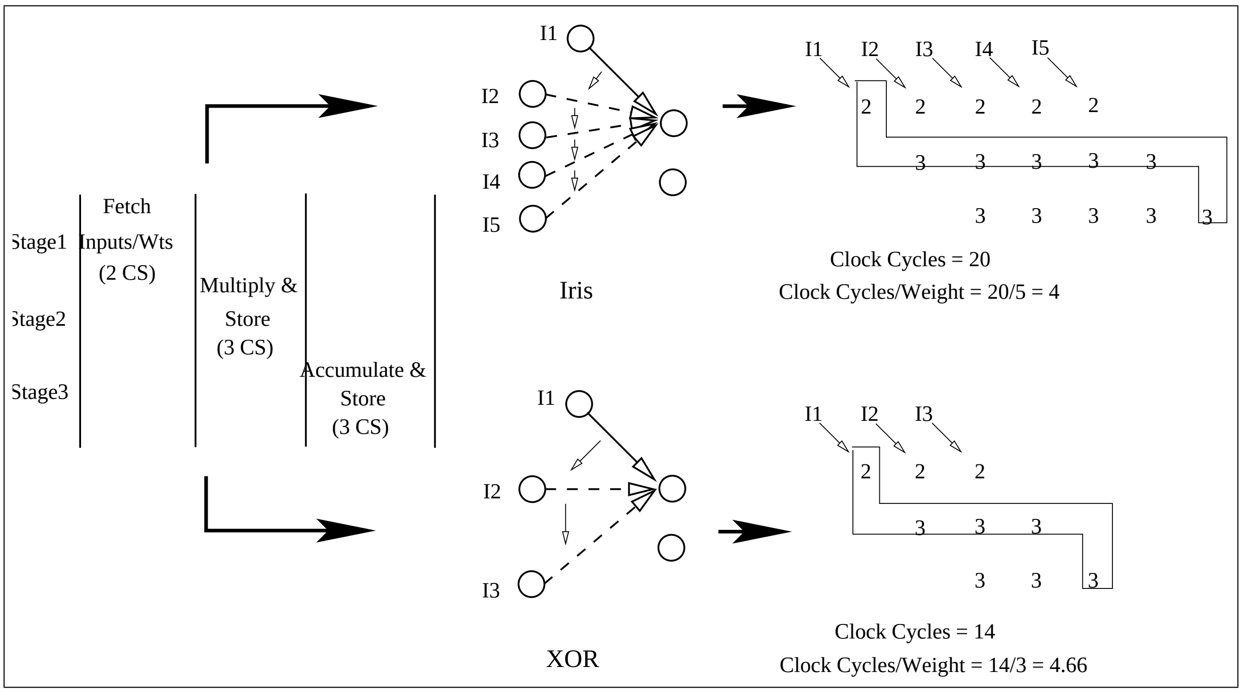


Figure 5.7: Comparison of Clock Cycles per weight in SIPOB

Viretex2000e FPGA.

Benchmarks	Gate Count(in million)	
	SIPEX	SIPOB
XOR	0.2	1.4
Iris	0.2	1.4
Cancer	0.2	1.4

Table 5.6: Gate Counts for Benchmarks

- SIPOB generates the hardware for 12-12-12 size network. Hence the gate count remains same for all benchmarks at 1.4 million.

- Although the execution is serial in nature, SIPOB requires high numbers of logic gates. The reason behind the intensive utilization of the hardware is the creation of large On-Chip RAMs for storing initial parameters specifically Input and Output. Since the maximum allowable number of nodes in a layer and total patterns in SIPOB are 12 and 255 respectively, it requires two On-chip RAMs, each of size $255 \times 12 \approx 3k(16bits)$, for storing Input and Output patterns. Such realization of the On-chip RAMs imposes high area requirement and thus restricts the size of the network that can be fit on the chip. The details of the area requirement imposed by the creation of various memories through Handel-C on Virtex chip is explained in Appendix D.

- Another reason for the high number of gate counts in SIPOB is the utilization of BlockRAMs for storing LTs of Sigmoid function. The size of the LT is $8k \times 16bits$ and that will require 32 BlockRAMs to be merged to act as a single memory module. Since the BlockRAMs are distributed on the FPGA chip, such merging not only demands high routing resources but requires the area for the logic needed to multiplex address and data bus from all 32 BlockRAMs.
- In contrast to SIPOB, SIPEX, the first serial approach, requires around 0.2 million gates because it does not use on-chip RAMs and BlockRAMs.

SIPEX architecture is the outcome of an early stage of the experimentation and not all the features of handel-C are exploited to optimize it for speed. For example, 'while' loop can be used instead of 'for' loop, output gradient can be calculated while the output error is being calculated and shifters can be used for power of two operations instead of multipliers. However SIPEX does help identify some bottlenecks and build a foundation for the development of SIPOB and the parallel architectures.

5.5 Summary

In this chapter two serial architectures SIPEX and SIPOB were proposed to implement the BP algorithm on the RC1000 board. These architectures differ in nature by means of storing LTs for the sigmoid function and initial parameters. However from SIPOB architecture it was found that LT approach falls short from the aim of having a good performance in a small area on the final chip (resource) [Beiu94]. The implementation of SIPEX architecture helped identify issues related with the access of Off-Chip RAMs and required division of the external clock. Hence the subsequent development of designs will be attempted at executing algorithm in two clock domains. The encoding of serial designs was a preliminary attempt at understanding the Hardware Descriptive Language Handel-C. It has helped understand the intricacy of the language and that knowledge can be further utilized in the development of parallel designs. Overall, the implementation of serial architectures provided the foundation for understanding the concepts of an ANN realization on an FPGA.

Chapter 6

Parallel Architectures

This chapter describes parallel architectures/designs developed for implementing the Back-Propagation (BP) algorithm for Artificial Neural Network(ANNs) on an FPGA. A detail description of each architecture and results obtained is presented in the following sections. Section 6.1 will describe the categorization of parallel architectures and introduces the concept of Branch-In and Branch-Out mode of multiply-accumulate. Section 6.2 will shed some light on piece wise linear approximation of sigmoid function. Section 6.4 proposes three partially parallel architectures and analyzes the outcome of the implementation on Virtex 2000E FPGA. Section 6.5 proposes a fully parallel architecture and analyzes the results of the implementation.

6.1 Categorization of Parallel Architectures

In ANNs parallelism can be achieved either at the node or layer level. Parallelism at the layer level indicates that the computation of all synapses values in a layer is performed in parallel. Such an approach requires an architecture with complete computing elements (multipliers) for all synapses in that layer. This configuration ensures the highest degree of parallelism and termed as a fully-connected parallel architecture. It leads to a very high level of network performance because it requires the same number of multipliers to be employed in a layer as the number of synapses(links). This is illustrated in the Figure 6.1 by giving an example of a network for XOR data-set (Only input and hidden layers of the network are shown).

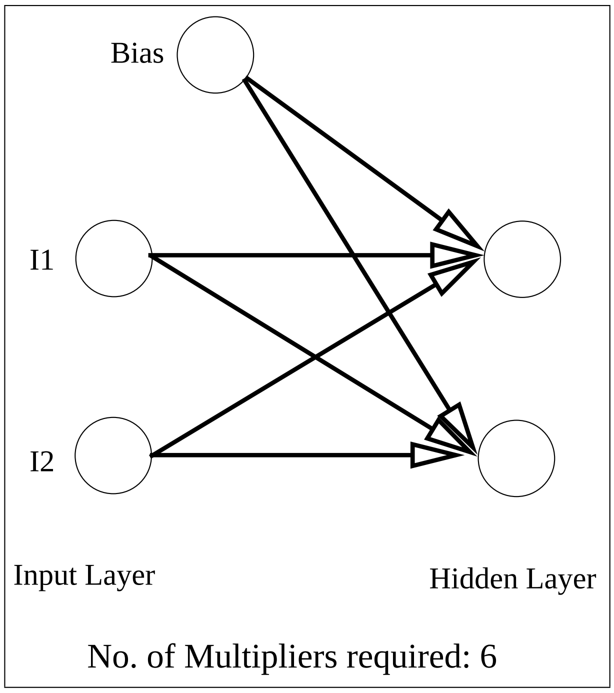


Figure 6.1: Link/Layer level parallelism

Parallelism at the node level means the computation of all synapses connected to a single node in a layer is performed in parallel. The node level parallelism can be achieved with a partially parallel architecture. This architectures compute synapses only branching in or out of a node at a time in parallel as illustrated in Figure 6.2 (Only input and hidden layers of the network are shown). In such designs computation is carried out with reference to nodes in previous/following layer. The notations "previous layer" and "following layer" refer to the position of a layer with respect to the other layer in the direction of computation as illustrated in 6.2. If the reference for calculation is based on the previous layer then the synapses in consideration are the ones that are **branching-out** of each neuron, one neuron at a time, from that layer. Hence the required number of multipliers in this case are set to the number of synapses leaving a neuron of the previous layer. The operation for such type of signal propagation is referred to as **"Branch-Out"**.

If the reference for calculation is with respect to the following layer then the synapses in consideration are the ones that are **branching-in** at each neuron. Hence the required number of multipliers in this case are the equivalent to the number of synapses approaching a neuron of the next layer. The operation of such type of signal propagation is referred to as **"Branch-In"**. It is evident that partially parallel designs require less resources than its fully parallel counterparts.

The Branch-In and Branch-Out method of computation form the very basis of partially-parallel designs for back-propagation algorithm. Since all three stages of the algorithm, Feed-Forward, Back-Propagation and Weight-Update, require basic Multiplication-Addition operations, any of the two methods (or a combination) can be utilized to perform computation for a particular stage. Figures 6.3 and

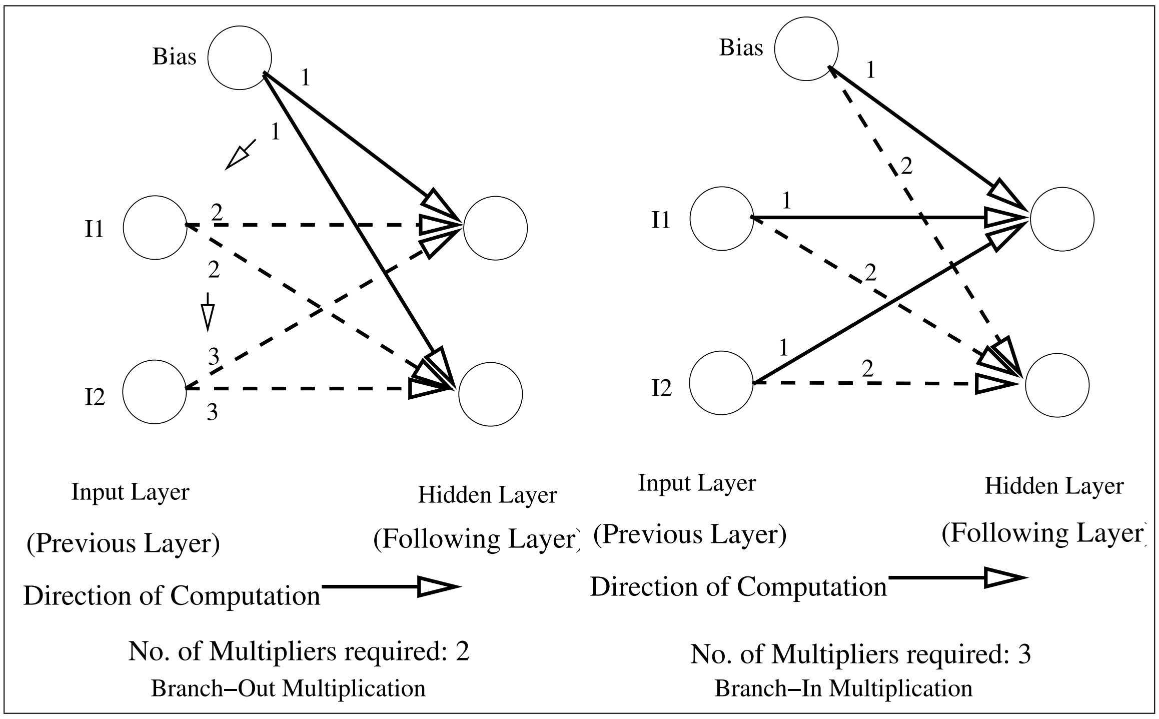


Figure 6.2: Branch-In and Branch-Out Multiplication

6.4 illustrate the concepts of multiplication-addition in Branch-In and Branch-Out modes of computations for Feed-Forward stage respectively.

A mathematical description for the Branch-Out multiply-accumulate as shown in Figure 6.4 can be given as:

W_i = Weight Vector for neuron I_i in input layer

$M_{i\_1} = I_1 \times W_1$

$M_{i\_2} = I_2 \times W_2$

Accumulation at Hidden node $H_i = \Sigma M_{i\_j}$,

where j = 1 to No. of hidden nodes

From the Figures 6.3 and 6.4 it can be observed that the addition of multiplied

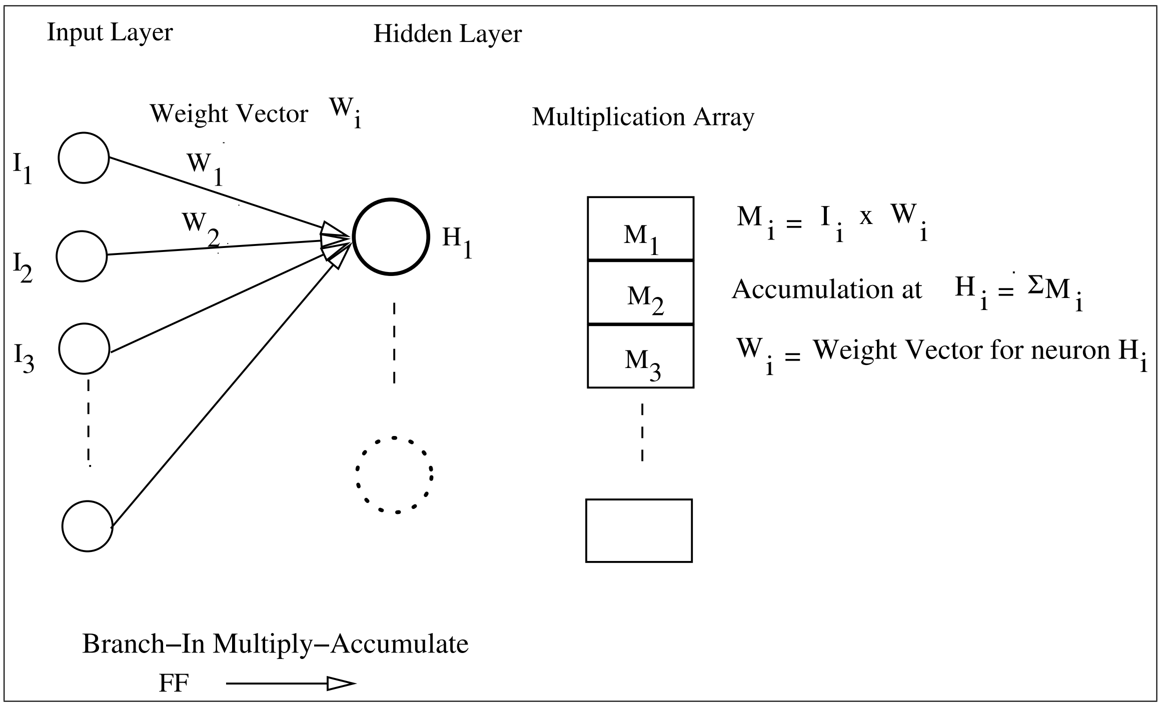


Figure 6.3: Branch-In Multiplication

values at each hidden neuron (or node at following layer) requires different approach for Branch-In and Branch-Out modes. In Branch-In mode, **for a** node in a following layer the required multiplied values are available in a multiplication array after **one instance** of parallel multiplication. Whereas in case of Branch-Out, only one multiplied value is available **for each** node in a following layer after one instance of parallel multiplication. Hence each neuron in next layer can be assigned with an adder in case of Branch-Out. An adder at each neuron will accumulate values after each instance of parallel multiplication and provide the partial sum as illustrated in Figure 6.4. In this way final sum at each neuron would become available once all synaptic weights in the following layer is computed. This simplified and efficient way of addition in the case of Branch-Out can be termed to assigning one "Multiplier-

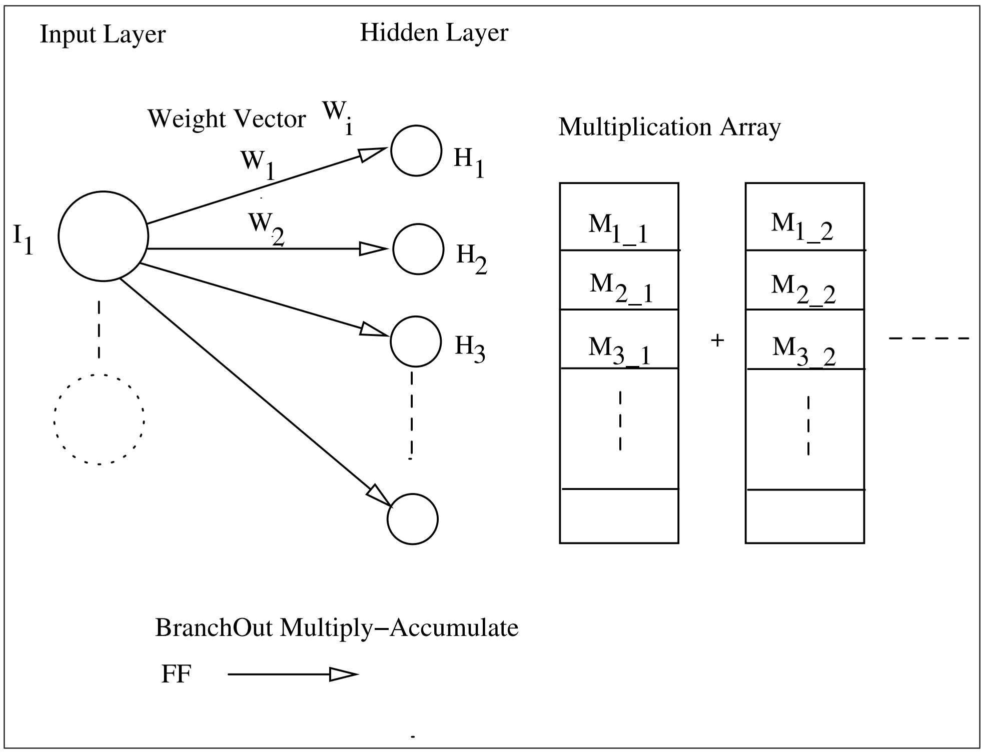


Figure 6.4: Branch-Out Multiplication

Adder unit per Neuron".

An addition in case of Branch-In can be performed in several ways depending on the number of adders provided. Since all required multiplied values become available for a node at every instance of parallel multiplication, there exists a question how efficiently addition can be computed with optimal number of adders. By providing a single adder, serial addition will perform computation in 'n' iteration, where 'n' being the number of synapses leaving from previous layer (Size of Multiplication Array). In most cases, this may not be efficient as the size of network grows larger with increasing number of neurons in each layer and thus requiring more iteration to complete addition in serial fashion. Designing a multi-input adder which assembles the same number of adders as the number of neurons less one in the next layer would provide the fastest, at least in terms of clock cycles, computation. However the feasibility of such an adder depends on the structure of custom arithmetic module or pre-built arithmetic library of the hardware programming language in use. Figure 6.5 shows such an adder module comprising of (n-1) adders, which computes all additions sequentially in one large clock cycle. The drawback of

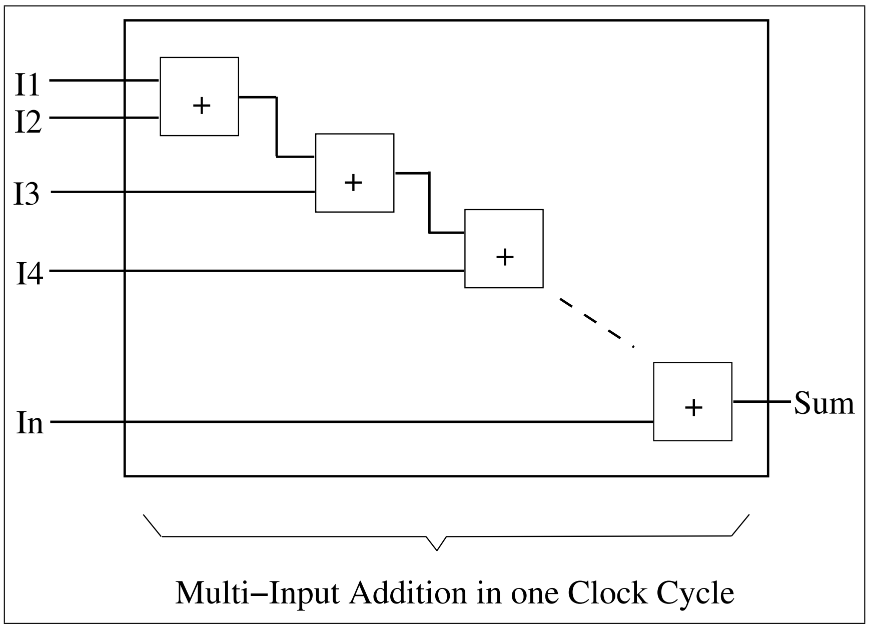


Figure 6.5: Multi-Input Adder module

multi-input adder module is it consumes a large clock width to compute additions. It might prove efficient for less number of additions in case of a small network. Since the maximum frequency with which a design can run on hardware is defined by the largest delay in a synchronous operation, the feasibility of accommodation of the multi-input adder remains to be determined from place and route data of synthesis tool.

Another way of performing an addition in case of Branch-In is shown in Figure 6.6. It starts with providing (n/2) adders in the first stage of computation and then continues computing at each successive stage with adders equal in number to half of the previous stage until final sum is reached.

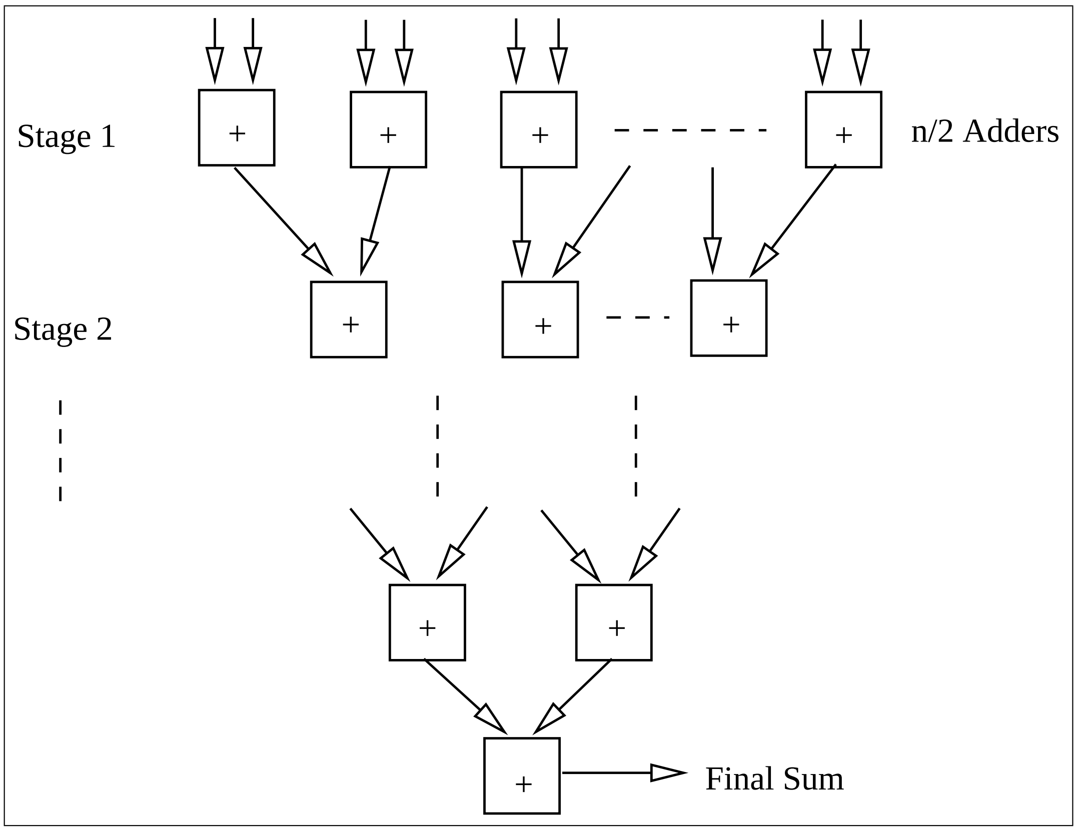


Figure 6.6: Branch-In Addition

The required number of stages = S, where $n \leq 2^S$

This approach requires several stages depending on the length of a multiplication array to complete the accumulation. Since all additions in a stage in this approach are performed in parallel, maximum logic delay in a circuit will be equivalent to one addition operation per stage. The maximum logic delay will be higher in case of a multi-input adder because of sequential computation of additions in a single stage. This results in a circuit requiring a larger clock cycle which in turn reduces the frequency of the design. Hence in all our parallel architectures, Branch-In addition as shown in the figure 6.6 will be used. Prior to the detail description of these architectures, the design aspects pertaining to the sigmoid function will be explored.

6.2 Sigmoid Function Implementation

In Chapter 5 the LT approach for the sigmoid function implementation was found to require either a large chip-area or access to Off-Chip RAMs, both of which have certain drawbacks.

An alternative approach to implementing the sigmoid function in digital VLSI technology is Piece-Wise Linear Approximation (PWL) which was described in detail in Chapter 3. The implementation of the sigmoid function was based on Piece-wise linear approximation for all parallel designs proposed in the thesis. The reasons behind this incorporation of PWL approach is the short-comings of LT approach observed through practical outcome of serial implementations and the benefits of efficient PWL approximation depicted in theoretical study of the literature review.

As described in Chapter 3, PWL approximations can be roughly divided into linear and higher order (mostly second-order) approximation. Though second-order approximations tend to produce better approximated results than linear schemes, we have focused on the first-order linear approximation because of its simplicity and comparatively cost-effective requirement of the resources. Such first-order PWL approximation is well described by [Bast04][Savi04]. As described in Chapter 4, our Fixed-point number representation is of the form 1-4-11 which essentially can accommodate positive and negative integer values in the range of 0 to 15. Looking at Figure 6.7 of the sigmoid function, the function for the positive and negative spectrum tend to saturate at 1 and 0 respectively for input values higher than |8|. This region of the graph can fairly be included by two segments, each on the positive and negative axis extending beyond the saturation limit. The central region can

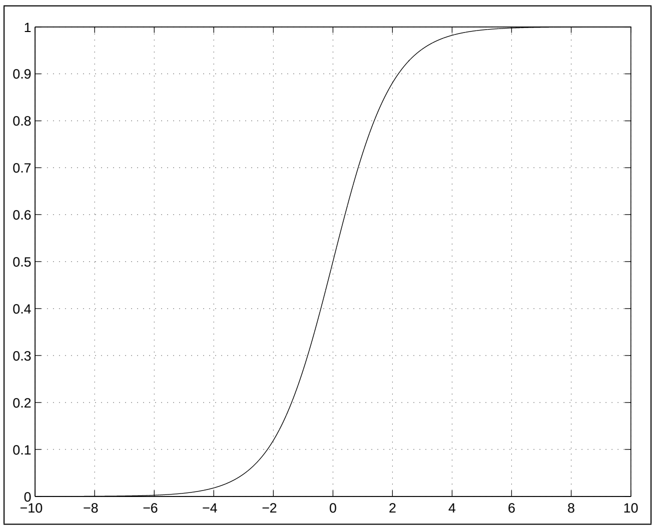


Figure 6.7: Sigmoid Function

be approximated as the linear region which is also the tangent to the function in $x = 0$. This region has a slope of 1/4 and input values are symmetric to $y = 0.5$. The slope is **"power of two"** nature hence calculation of output in this region will essentially be a shift operation followed by an addition of constant (0.5 in this case). The region close to the saturation on both side of the axis can be linearized with a slope of 1/64. This linear region starts from the saturation limit |8| and intersects the central linear region at $x = |1.6|$. Again, as mentioned earlier, the calculation in this region can also be carried out with shifters and adders due to the slope which is in **"power of two"** arithmetics. This region can be conceived as two linear segments on both sides of the x axis.

From the preceding discussion we can construct a 3-piece linear approximation of the sigmoid function as depicted in Figure 6.8 along with an error plot showing the difference in approximation and the actual value.

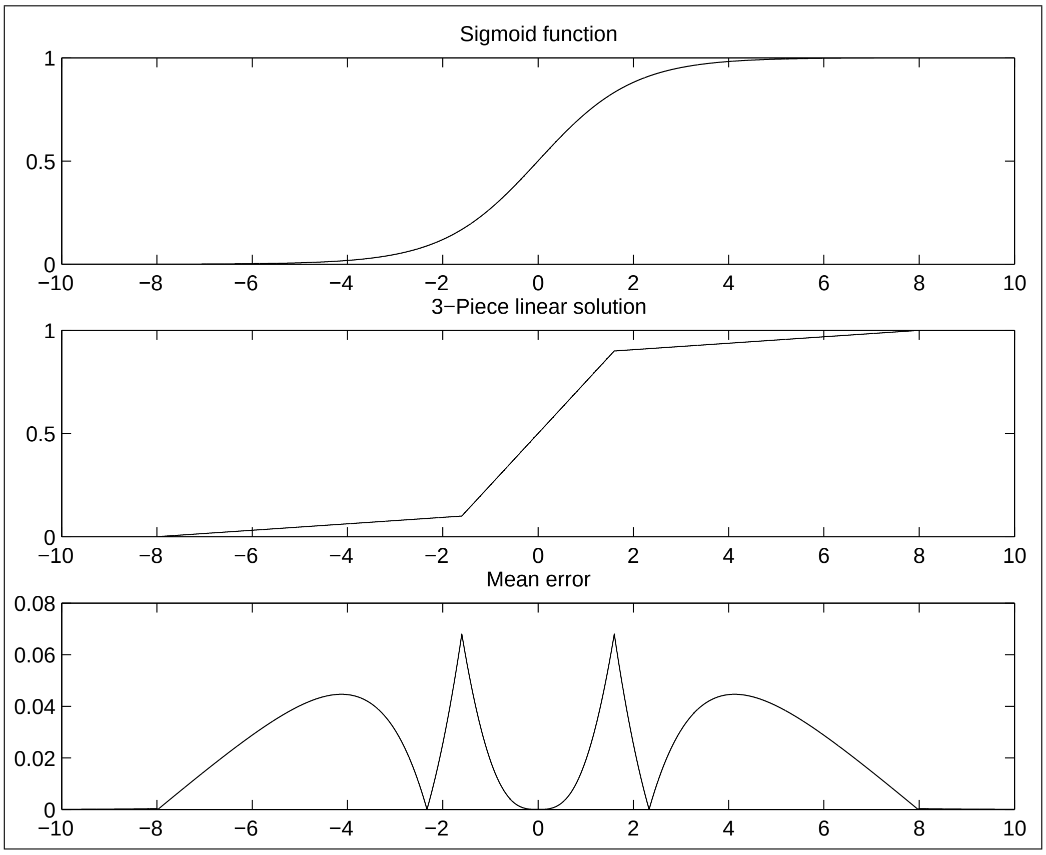


Figure 6.8: Approximated Sigmoid Function

The approximation shown in the Figure 6.8 can be described by the following equations by,

$$f(x) = \begin{cases} 1 - \frac{1}{8} \times (1 - \frac{x}{8}) & 1.6 \leq x < 8 \\ \frac{1}{8} \times (1 + \frac{x}{8}) & -1.6 \geq x > -8 \\ \frac{1}{2} \times (1 + \frac{x}{2}) & -1.6 < x < 1.6 \end{cases} \tag{6.1}$$

Linear regions approximated in equation 6.1 can be generated in hardware as shown in Figure 6.9.

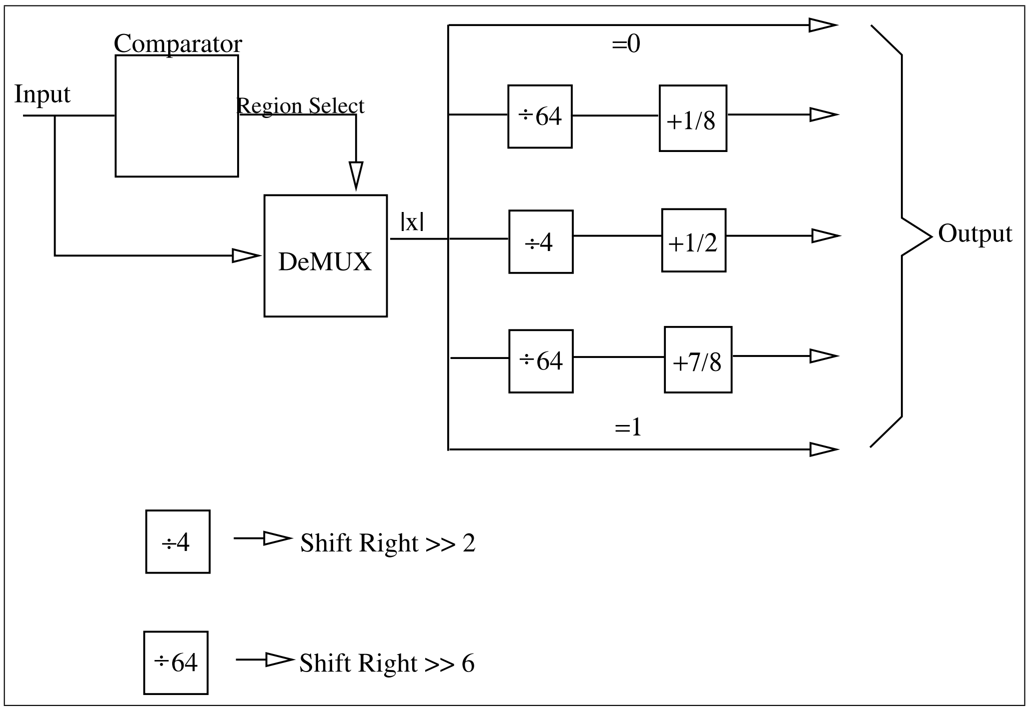


Figure 6.9: A Sigmoid Function module

A Comparator compares the input with breakpoints of linear regions defined in equation 6.1 and selects the appropriate region for an input. If the input values is above |8|, then it will saturate the output at 0 or 1. For other linear regions only shifters are required to divide the input by 64 or 4 and then appropriate constants are added to find the output. It can be seen from the Figure 6.8 that the maximum error is 7% and found at the intersection of the linear regions. Also the area below the error curve is higher in the region where sigmoid function approaches the limits. The reason behind this behavior is we have not taken tangent at any point of the curve in this region $(|1.6|, |8|)$ unlike the one taken in the central region at $x = 0$. [Bast04] recommended choosing a line which is also a tangent of the curve at some point in approximating a region with linear segments. However it will require more number of segments (probably two more) in our case to reach the limits -8 and 8.

The benefit of PWL approximated sigmoid function unit is that it involves basic shift and add operations which doesn't require significant chip-area. Hence it is practical to assign such unit to each processing node of a layer in a network. This will allow parallel computation of sigmoid functions for all nodes in that layer.

The back-propagation stage of the BP algorithm requires the derivative of the sigmoid function. Various researchers have derived the formulas based on PWL approximation for the derivative [Alip91][Myer89][Bast04][Beiu94]. It should be noted here that though the PWL approximated function described in preceding discussion is differentiable, we have used the derivative of the **actual** function, $f(x)\times(1-f(x))$. The main reason behind this is that by approximating the sigmoid function we are inducing a small error in the computation and it will deviate more if we used the approximated derivative function. Also the $(1 - f(x))$ part of the actual derivative is nothing but a two's complement of the function and hence can be easily constructed on hardware. The requirement of multipliers in computation of $f(x)\times(1-f(x))$ can be fulfilled by using the multipliers provided for the synaptic weight-input multiplications. However this will come at an extra price on routing, since using the same module consisting several multipliers, at different parts of the chip will require more routing resources.

Table 6.1 represents the area/speed performance of various sigmoid function implementations considered so far.

Sigmoid Implementations	Equivalent Gate Counts	4-Input LUTs on FPGA	Slice on FPGA	Clock Cycles per fetch
LT on Off-Chip RAMs	1800	118	87	2.5
LT on BlockRAMs	500K	149	134	2.5
Linear Approximation	2200	196	144	3.0

Table 6.1: Various Sigmoid Function Implementations

LT implementation of the sigmoid function on Off-chip RAMs requires less chip-area because of simplicity of the design. LT implementation of the sigmoid function on BlockRAMS requires large area. In this approach 32 BlockRAM modules are combined to form a single memory of 8k in size. An access to such a memory module requires a significant logic on the FPGA. PWL approximated sigmoid function requires higher clock cycles/generation than the other two approaches. However comparable low chip-area requirement of the PWL approach allows the generation of multiple sigmoid function in a circuit.

6.3 Structure of Data Storage in BlockRAMs

In Section 6.1 we highlighted the need for storing the initial parameters are stored in BlockRAMs. Hence before describing each of the architecture, we will describe memory map and show how the parameters are stored in several BlockRAMs. This aid the reader in understanding how the parameters and intermediate values are fetched and stored in parallel during processing. The purpose of storing the initial parameters such as input, output and weight vectors in BlockRAMs is to facilitate the parallel access of the information to various modules of the architectures. Each BlockRAM is set to a 256 x 16 bits as already shown in the Figure 4.2. Since each BlockRAM can be accessed once in a clock cycle, it is essential that the storage of each initial parameter be distributed among few BlockRAMs. For example, to store 200 (patterns) x 4 (vector size), 4 BlockRAMs are used. The first element of each BlockRAM stores a value from first input pattern vector, the second stores a value from the second input pattern vector and so forth. The same concept has been applied to store weights as shown in Figure 6.10.

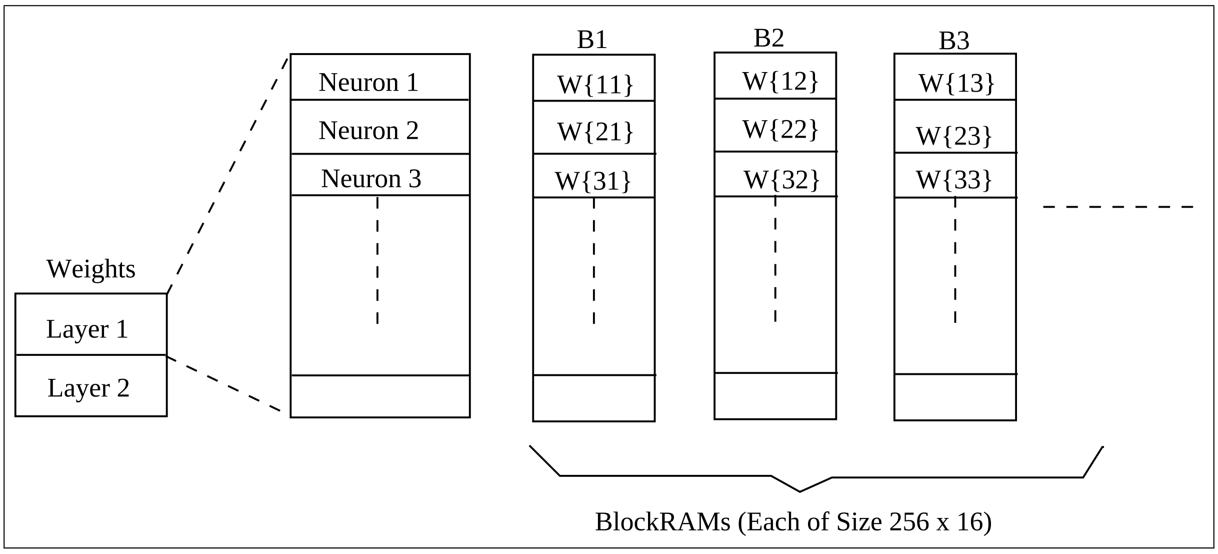


Figure 6.10: Weight Storage in BlockRAMs

It is clear from Figure 6.10 that all the weights associated with a particular neuron can be accessed in parallel.

6.4 Partially Parallel Implementations

As noted in chapter 3, achieving a high speed requires the exploitation of inherent parallelism in an ANN. As such we attempted at developing partially parallel architectures for the following reasons:

1. To achieve better performance over serial implementations
2. To evaluate different designs for its ability to accommodate different topologies of networks on Virtex2000e FPGA chip.

The chip-area (resources) and performance (speed of convergence) are vital two parameters that need to be checked for all architectures while training the networks.

6.4.1 Categorization of Partially Parallel Architectures

Some of the common features of the three architectures, though described earlier, worth mentioning here for completeness are:

- Using 16 bit Fixed-point number representation in the form of 1:4:11 (Sign:Integer:Fraction)
- Developing random number generators based on LFSR
- Realizing sigmoid function through PWL approximation by employing 3-piecewise linear function
- Storing initial parameters stored in BlockRAMs
- Pattern Mode of learning

Since Branch-In and Branch-Out methods can be applied to any of the three stages of the algorithm to accomplish the multiply-accumulate operation, there are various possibilities by which a partially parallel architecture can be built. These possibilities explore the various options of assigning the methods to three stages. For example, one such architecture can have a Branch-In mode of operation for the Feed-forward stage and a Branch-out for the back-propagation stage or vice versa. The weight-update stage always uses the same method of multiply-accumulate of the feed-forward stage. Hence one of the differences between these architectures is how the multiply-accumulate methods are assigned to the feed-forward and the back-propagation stage.

Yet another difference in the architectures is the number of sigmoid function units employed for each layer. In Branch-In mode of operation the final sum of weight/input product at each neuron becomes available in sequentially. Hence when Branch-In mode is applied, a single sigmoid function unit per layer is sufficient as the unit can be pipelined to be used by other neurons. Whereas in Branch-Out mode of operations the final sum of weight/input product at all neurons becomes available instantly after several stages of partial sums. Hence when Branch-Out mode of operation is applied, it is practical to employ the same number of sigmoid function unit as the number of nodes in a layer to achieve parallel computation.

Based on the preceding discussion, the different partially parallel architectures can be described as follows:

- **PAR**allel architecture with (Branch)**O**ut and (Branch)**I**n operation in Feed-Forward and Back-Propagation stages respectively is called **PAROI** and has the following features:

- Branch-Out mode in Feed-Forward stage
- Multiple Sigmoid Function units
- Branch-In mode in Back-Propagation stage
- For 5-3-3 topology of ANN for Iris data-set:
  - Multipliers: 3 (Maximum of Hidden and Output vectors)
  - Branch-Out Adders: 3 (Maximum of Hidden and Output vectors)
  - Branch-In Adders: 3 (Two stages of addition with adders in each stage are $2 \rightarrow 1$)
  - 3-piecewise linear sigmoid Units: 3 (Maximum of Hidden and Output vectors)

Figure 6.11 shows the method of multiply-accumulate for the PAROI architecture.

- **PAR**allel architecture with (Branch)**O**ut and (Branch)**O**ut operation in Feed-Forward and Back-Propagation stages respectively is called **PAROO** and has the following specifications:
  - Branch-Out mode in Feed-Forward stage
  - Multiple Sigmoid Function units
  - Branch-Out mode in Back-Propagation stage
  - For 5-3-3 topology of ANN for Iris data-set:
    - Multipliers: 3 (Maximum of Hidden and Output vectors)
    - Branch-Out Adders: 3

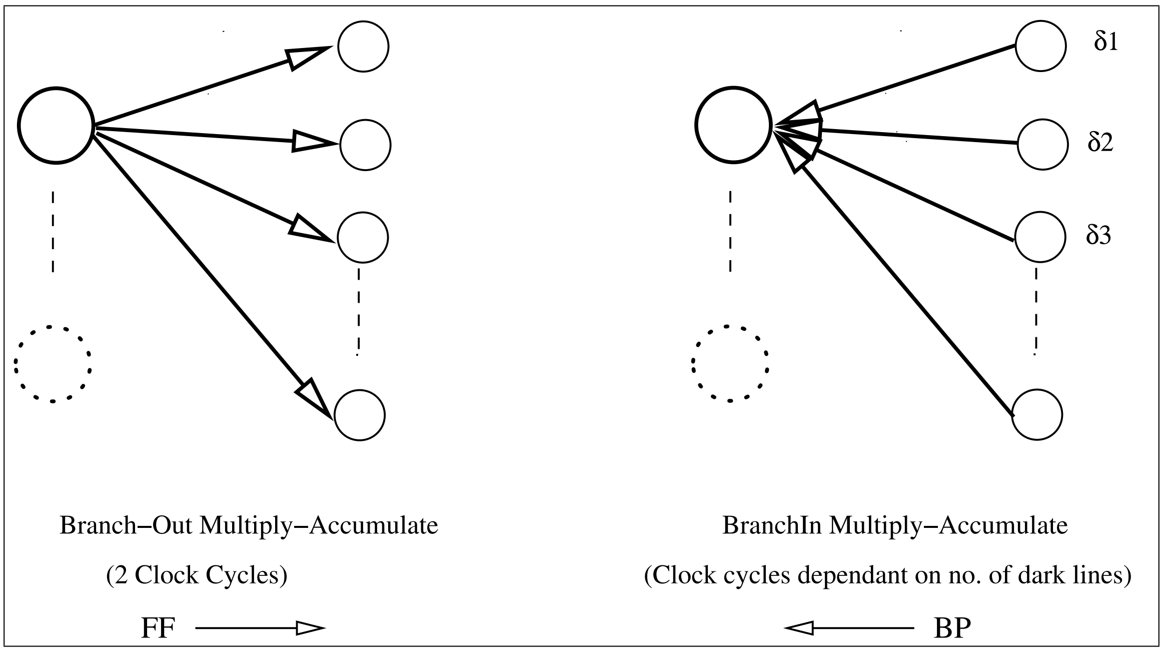

δ1
δ2
δ3
Branch–Out Multiply–Accumulate
(2 Clock Cycles)
FF
BranchIn Multiply–Accumulate
(Clock cycles dependant on no. of dark lines)
BP

Figure 6.11: PAROI

* 3-piecewise linear sigmoid Units: 3 (Maximum of Hidden and Output vectors)

Figure 6.12 shows the method of multiply-accumulate for the PAROO architecture.

- **PAR**allel architecture with (Branch)**I**n and (Branch)**O**ut operation in Feed-Forward and Back-Propagation stages respectively is called **PARIO** and has the following specifications:
  - Branch-In mode in Feed-Forward stage
  - Single Sigmoid Function unit
  - Branch-Out mode in Back-Propagation stage

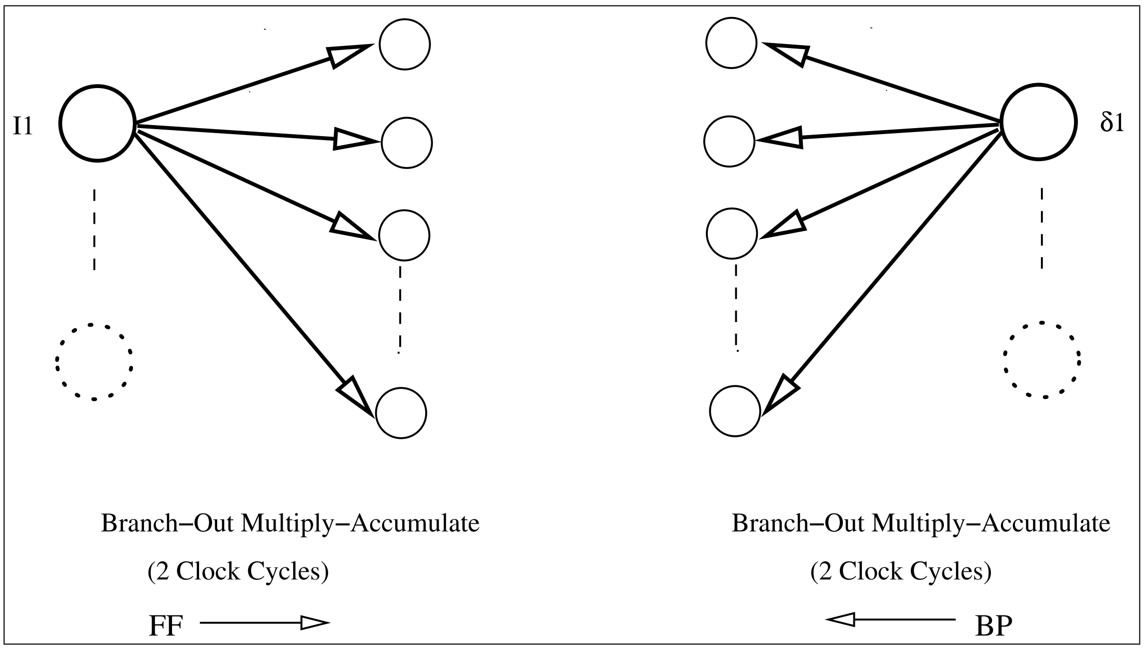

I1
δ1
Branch−Out Multiply−Accumulate
(2 Clock Cycles)
FF
Branch−Out Multiply−Accumulate
(2 Clock Cycles)
BP

Figure 6.12: PAROO

– For 5-3-3 topology of ANN for Iris data-set:
  * Multipliers: 5 (Maximum of Input and Hidden vectors)
  * Branch-In Adders: 6 (Three stages of addition with adders in each stage are $3 \rightarrow 2 \rightarrow 1$)
  * Branch-Out Adders: 5 (Maximum of Input, Hidden and Output vectors)

The required number of arithmetic units for 5-3-3 topology of ANN for Iris data set are summarized in Table 6.2.

Figure 6.13 shows the method of multiply-accumulate for the PARIO architecture.

Branch-In method performs additions in several stages and in most cases number of clock cycles required are higher than Branch-Out method of addition which

Arithmetic Units	PAROI	PAROO	PARIO
Multipliers	3	3	5
Branch-In Adders	3	-	6
Branch-Out Adders	3	3	5
Sigmoid Generators	3	3	1

Table 6.2: Arithmetic Units for 5-3-3 topology of Iris Data-Set

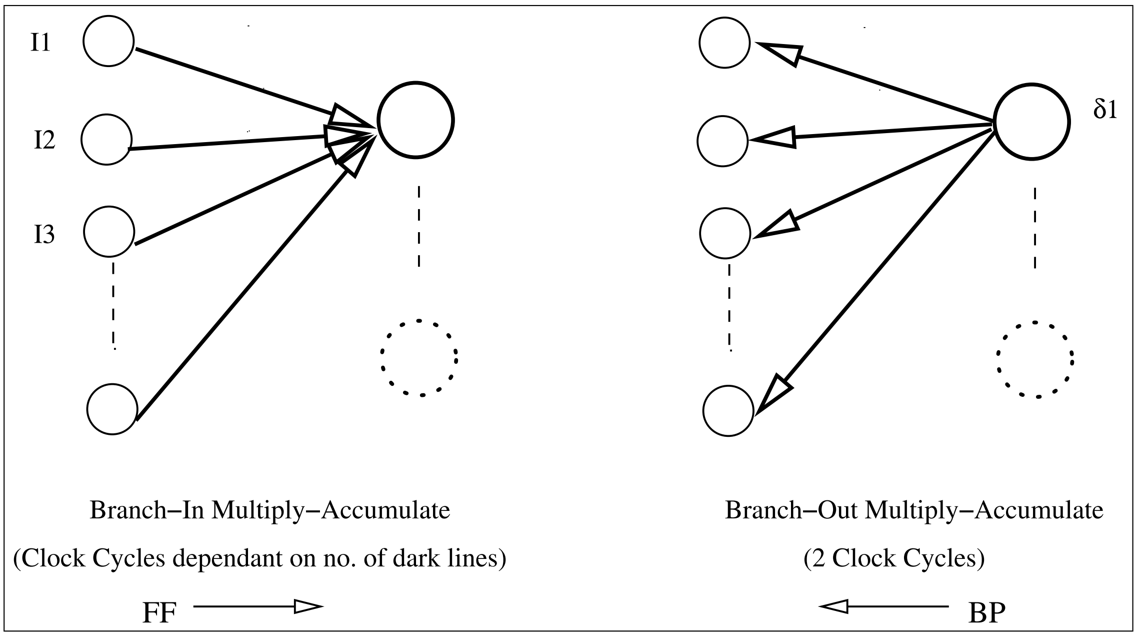

I1
I2
I3
δ1
Branch–In Multiply–Accumulate
(Clock Cycles dependant on no. of dark lines)
Branch–Out Multiply–Accumulate
(2 Clock Cycles)
FF
BP

Figure 6.13: PARIO

takes only 2 clock cycles. Hence no partially parallel architecture with Branch-In addition in both Feed-Forward and Back-Propagation stages was built (could have been termed as PARII).

6.4.2 Structure of Partially Parallel Architectures

The basic structure of the three ANN architectures consist of the following modules:

1. Main Processor Module: controls the execution of the three stages of the algorithm. The module also communicates with the host to start and end the execution of the algorithm on the FPGA.

2. Neuron Accumulation Module: performs multiply-accumulate computation for a layer in Feed-Forward stage.

3. Sigmoid Generation Module: performs computation of the Sigmoid function approximated by 3-piecewise linear segments. This module computes a single or multiple sigmoid functions depending on the choice of the architectures.

4. Error and Output Gradient Module: computes the output error in Feed-Forward stage and the gradient value for the output layer in Back-Propagation stage. This module also computes Instantaneous Sum-squared Error (ISE) and stores in either BlockRAMs or Off-chip RAMs.

5. Hidden Layer Gradient Module: performs computation of the gradient value for the hidden layer in Back-Propagation stage.

6. Weight Update Module: performs computation of the weights for a layer in Weight-Update stage.

7. Initialization Module: transfers initial parameters and the topology from Off-Chip RAMs to BlockRAMs. It also initializes several intermediate parameters of the algorithm that is required to be stored during the execution.

8. Random Generation Module: generates random numbers based on Linear Feedback Shift Register(LFSR). This module generates a random number in advance for next presentation in parallel to the execution of previous presentation.

The basic structure of all three architectures remain very similar and can be illustrated in Figure 6.14.

The functionality of a partially parallel architecture shown in Figure 6.14 can be described as following:

(a) The Processing starts by reading the control register(ReadControl) and requesting access to external RAM (RequestMemoryBank) from the host. The host accordingly downloads .bit files to the FPGA. It also downloads the initial parameters and the ANN topology to the Off-chip RAM Banks.

(b) The main Processor module issues the Start_Initial signal to The Initialization module to enable transfer of initial parameters and ANN topology from Off-chip RAM banks to BlockRAMs. The Initialization module also initializes intermediate parameters such as error-gradient, neuron output etc. On completion, the Initialization module issues End_Initial signal to the Main Processor module.

(c) The Main Processor module reads the topology from BlockRAM and set data registers such as LayerCount, NoEpochs, NumNeuron (No. of neurons in each layer) etc.

(d) The Main Processor module then issues a StartRan signal to Random Gen-

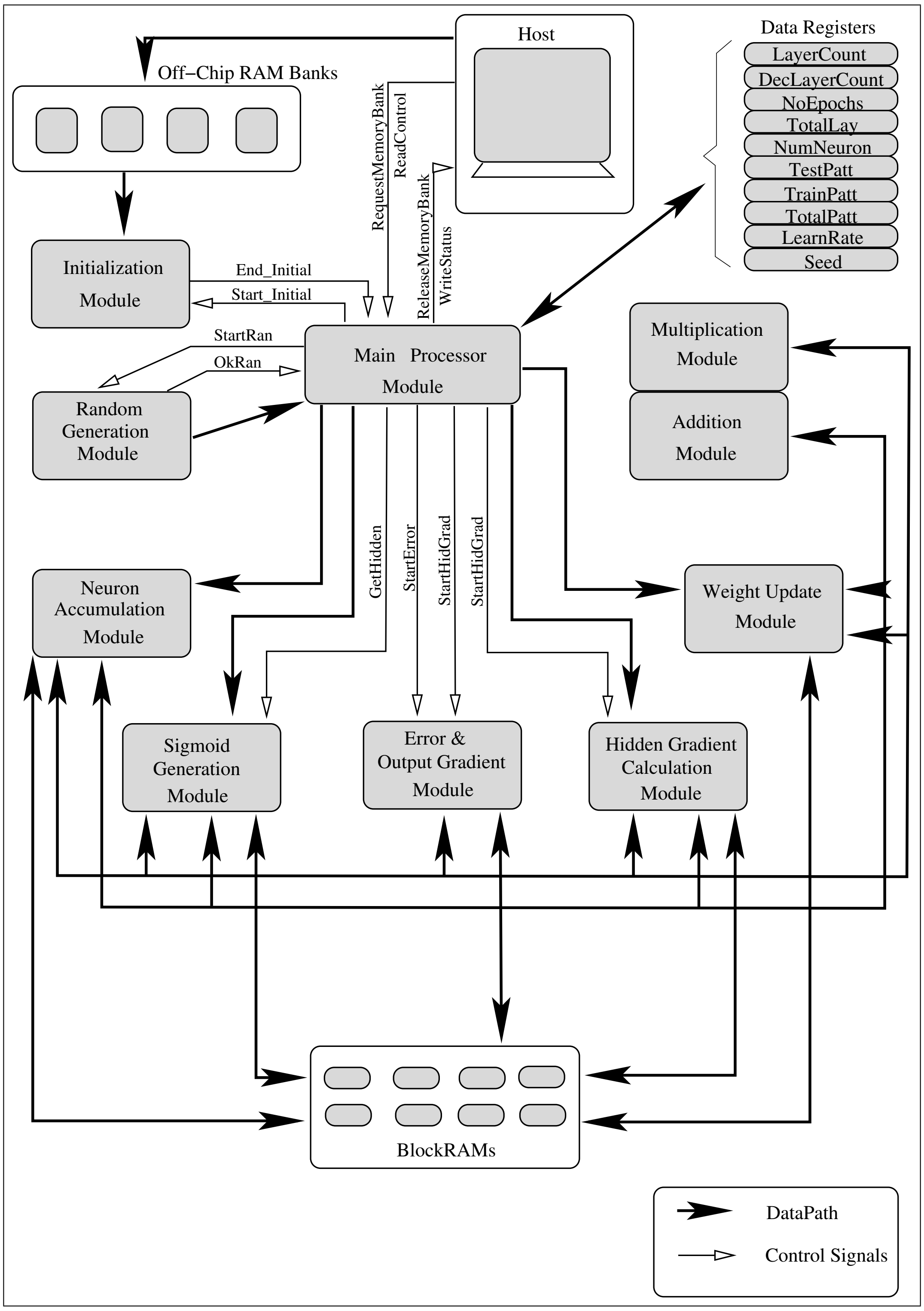


Figure 6.14: The basic structure of a partially parallel architecture

eration module to supply the first random number. The Random Generation module sends the random number on the GetRan channel and issues OkRan to mark the completion of transfer.

(e) A random seed value is supplied to the Random module for the generation of the numbers for the first epoch. For every successive epoch, the seed value is incremented by one.

(f) The Main Processor starts the execution of the algorithm by fetching an input pattern based on the random number generated in previous iteration. The Random Generation module is called to function in parallel to all three stages and generate a number for the next iteration.

(g) The Main Processor module calls Neuron Accumulation Module which performs multiply-accumulate operation (Branch-In or Branch-Out) for a single layer. Next, the Processor calls the sigmoid generation module which computes PWL approximated sigmoid function. This module also computes the derivative of the sigmoid function. In parallel, it also issues the GetHidden signal to fetch patterns for the next layer.

(h) The Error module computes error at each node in the output layer. For discrete values of desired output (1 or 0), **the error at each node will be either 2's complement or the negative of the actual output value**. The module also computes error-gradient for the output layer. Then it issues StartHidGrad signal to Hidden Gradient module to start the computation of error-gradient in the hidden layer, in parallel to the computation of **In-**

stantaneous Sum-squared Error(ISE). Error module stores the ISE in BlockRAM in case of Double-Domain design or in Off-chip RAM Bank3 in case of Single-Domain design. The concepts of single and double-domain designs are described in Appendix F.

(i) The Hidden Gradient module computes the error-gradient for the hidden layer. It fetches the derivative of neuron output already computed in sigmoid generation module from BlockRAMs.

(j) The Weight Update module computes the weight update for each layer. It issues GetHidden signal to fetch patterns for the following layer while updating the weights for previous layer in parallel.

(k) The Main Processor issues OkRan signal to the Random Generation module to mark the completion of the algorithm.

(l) The execution for each pattern is carried out in similar fashion described above until the epoch count equals the NoEpochs variable set by the topology.

(m) The Main Processor module generates control signals to send the ISE value to the Initialization module to store it back to external RAM in case of Double domain design.

(n) All of the modules described in preceding discussion use two arithmetic modules employing Fixed-Point multiplication and Branch-In and Branch-Out Addition as needed.

(o) The Main Processor module completes the execution by sending WriteStatus and ReleaseMemoryBank signals to the host.

(p) Once the Off-chip RAM banks are released by the main processor, the host acquires access to RAM and reads ISE back to the host memory.

6.4.3 PAROI

The Algorithmic State Machines (ASMs) of the PAROI architecture are represented by Figure 6.15 (Neuron-Accumulation module), Figure 6.16 (Hidden Layer Gradient module) and Figure 6.17 (Weight Update module).

6.4.4 PAROO

The ASMs for the Hidden layer gradient module of the PAROO and PARIO architectures is presented in Figure 6.18. The ASM shows the Branch-Out type sum-accumulate for the Back-Propagation stage. The Neuron accumulation and Weight update modules remain similar to PAROI as shown in Figures 6.15 and 6.17 respectively.

The implementation of the PAROO architecture encounters a typical problem related with parallel access of weights in both the Feed-Forward and Back-Propagation stages. This architecture employs Branch-Out mode of multiply-accumulate operation in both FF and BP stages of the algorithm. The weight storage for FF and BP stage **for a layer** is shown in Figure 6.19. It should be noted here that the storage for the Feed-Forward and Back-Propagation stages yields different matrices for a **same set** of weights. Actually, both matrices are

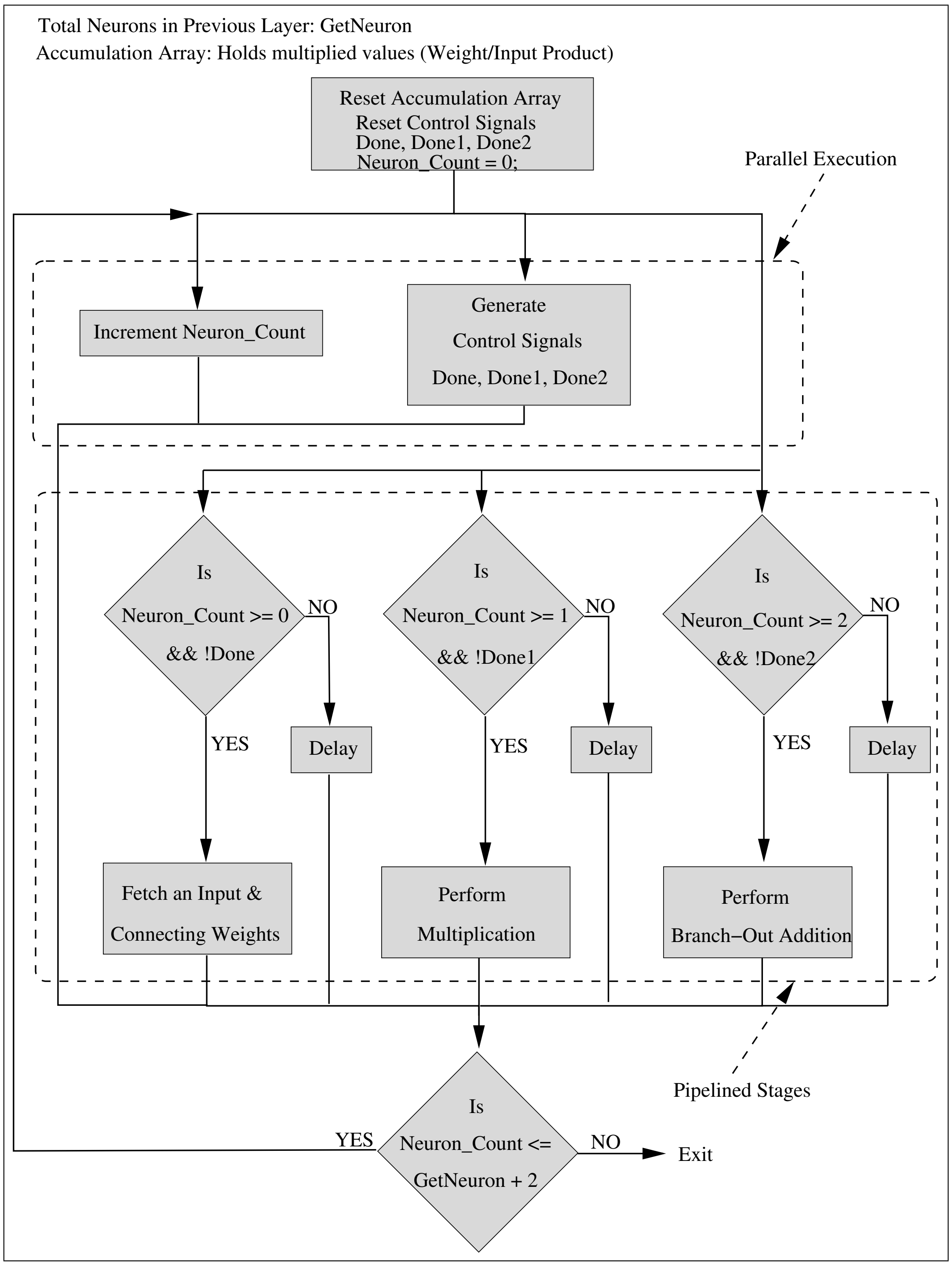


Figure 6.15: ASM for Neuron Accumulation module in PAROI and PAROO

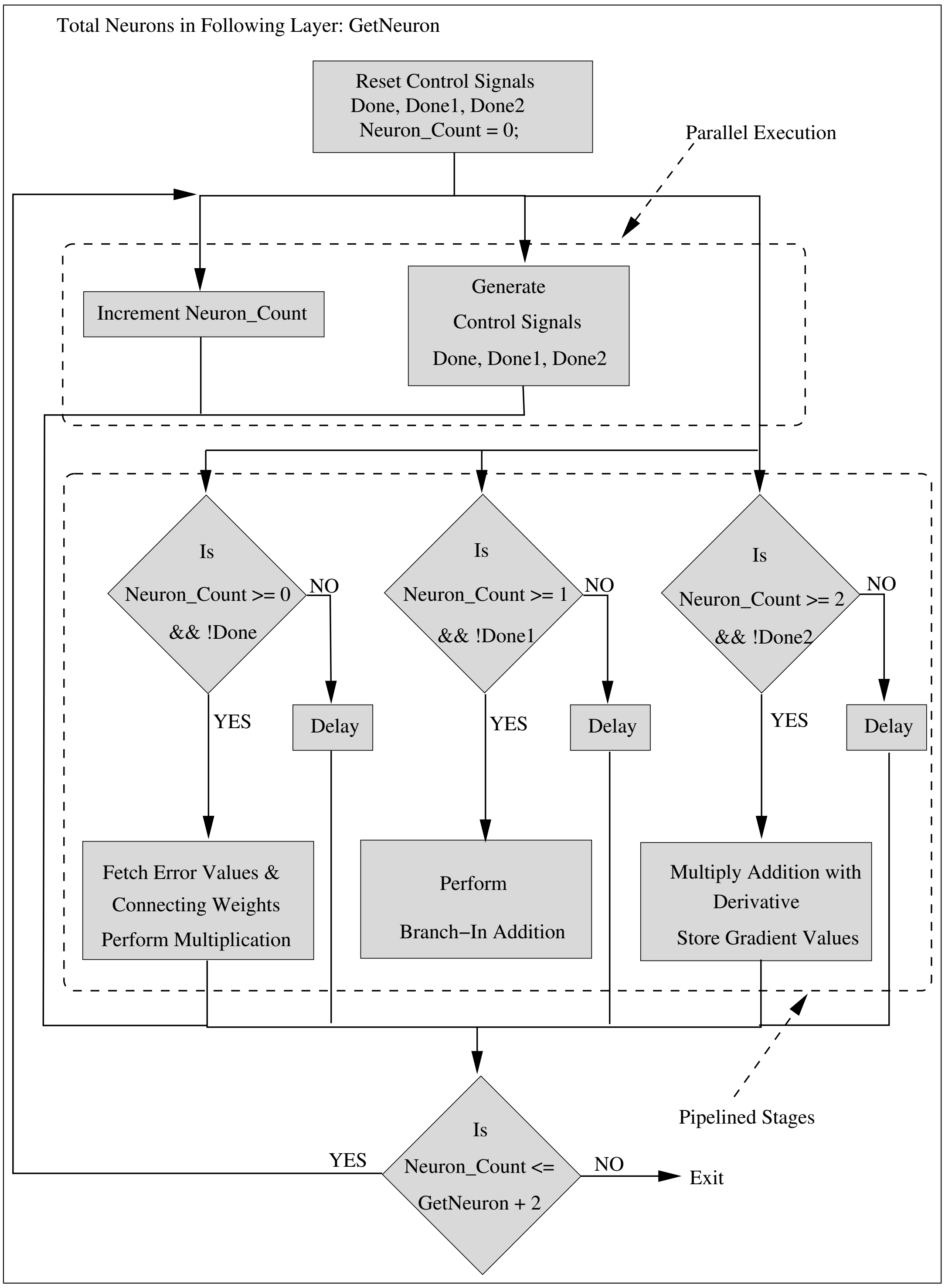


Figure 6.16: ASM for Hidden layer Gradient module in PAROI

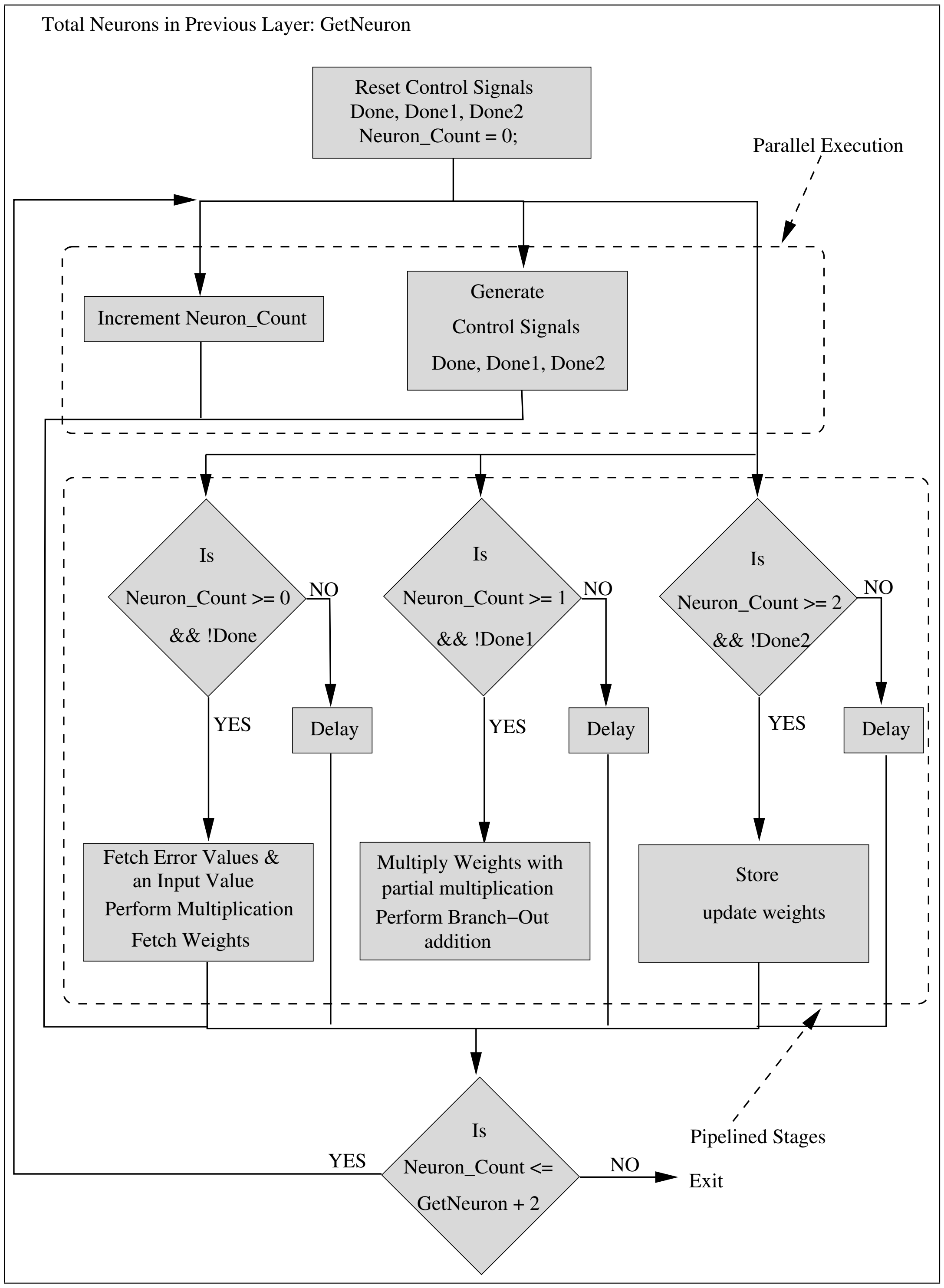


Figure 6.17: ASM for Weight update module in PAROI and PAROO

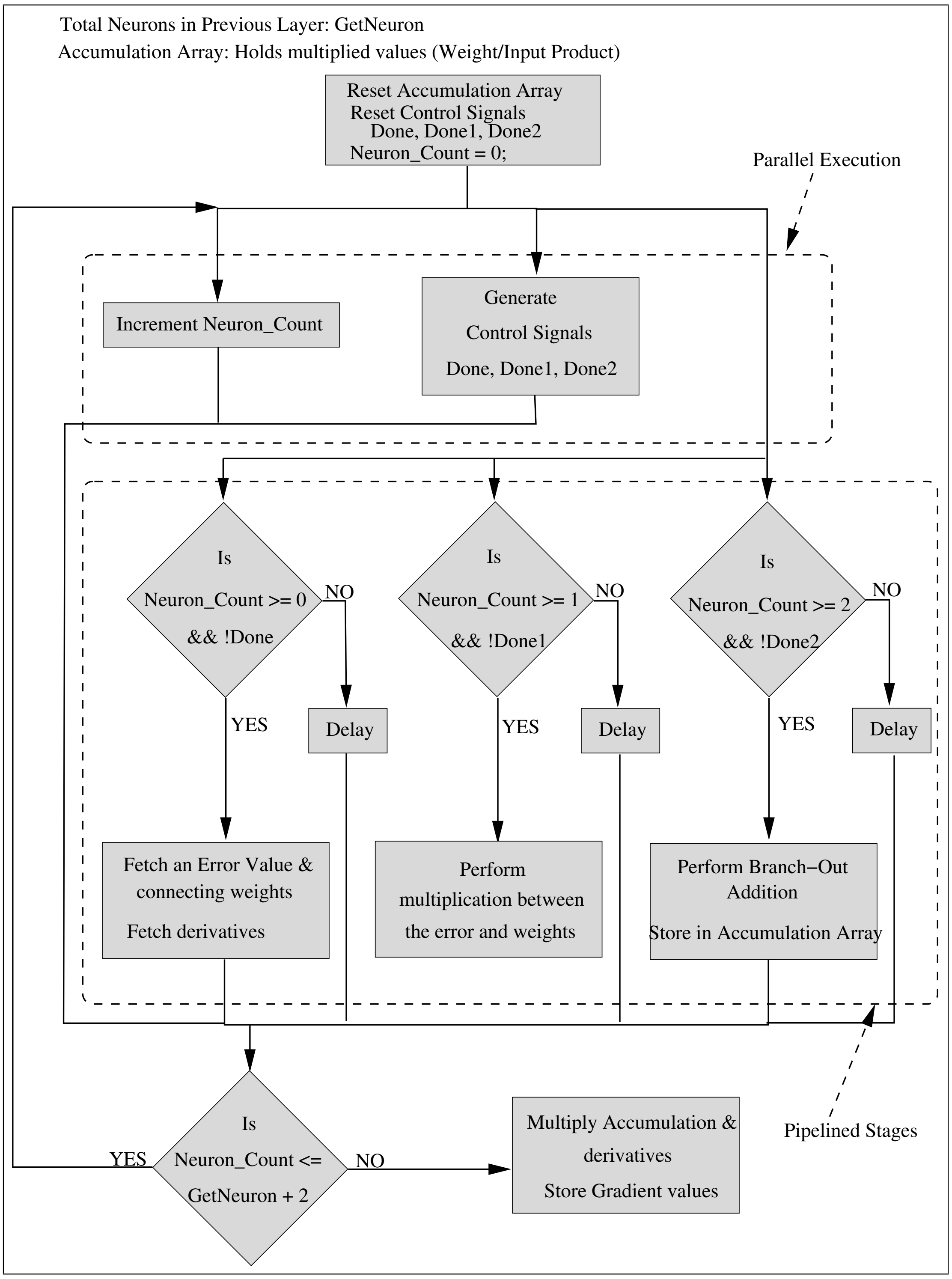


Figure 6.18: ASM for Hidden layer Gradient module in PAROO and PARIO

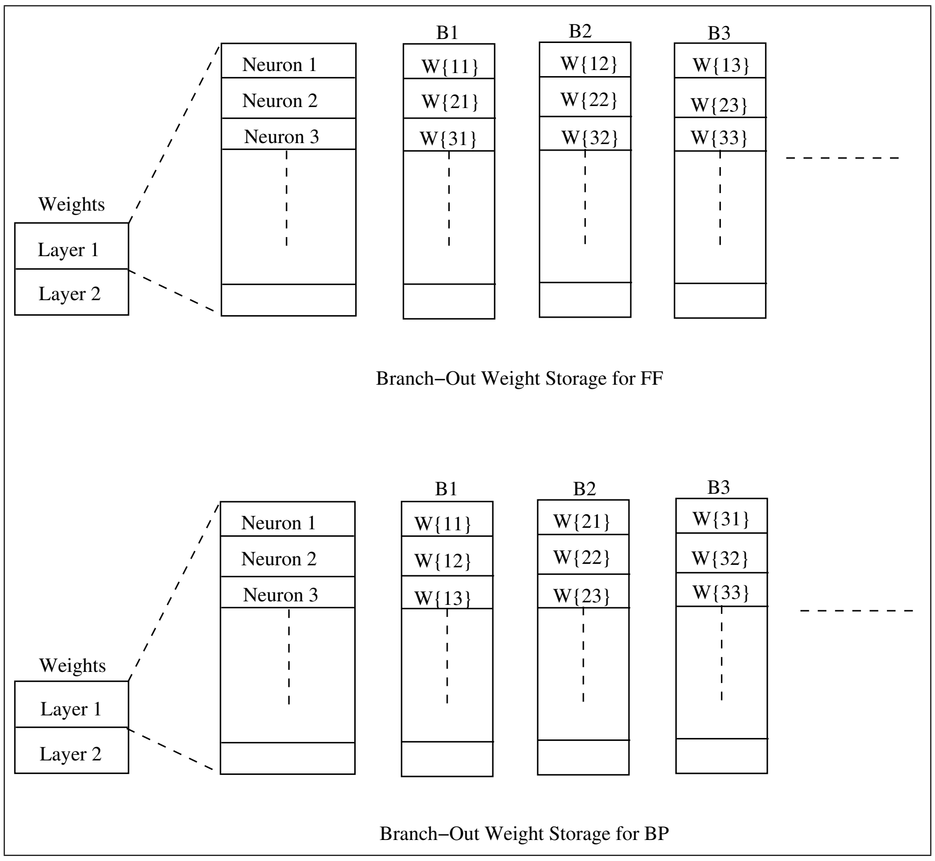


Figure 6.19: The need for the duplication of weights

complement of each other. A careful investigation will reveal that the weights in consideration in a layer for parallel access are different when the same type of multiply-accumulate mode is used in both the Feed-Forward and Back-Propagation stages. Hence to facilitate parallel access of weights in both stages, we need to provide two sets of BlockRAMs for storing one set of weights. However the problem with such duplication of weights is that it will require weights to be stored after every weight-update in both set of BlockRAMs. This may consume extra clock cycles in execution of the algorithm. However, the requirement for the duplication of weights can be limited to weights connected to the output layer only, since the

BP stage doesn't need hidden layer synapses at any point in its calculation of error values. This inherent behavior of the BP stage can be exploited to save the clock cycles from being consumed for the duplication of weights. Hence in the PAROO architecture the Weight-Update stage initiates updating weights from the output layer and then moves on to the hidden layer unlike the conventional Weight Update stage which computes in forward direction starting from the hidden layer. Once the weights in the output layers are updated, they can be duplicated to second set of BlockRAMs in parallel to the WU stage which moves on to update the weights in the hidden layer. This will ensure very little consumption of extra clock cycles provided the WU stage for the hidden layer is long enough.

6.4.5 PARIO

Figures 6.20 and 6.21 are the ASMs for neuron accumulation and weight update modules of the PARIO architecture. The ASMs show the Branch-In type sum-accumulate for both modules. The hidden layer gradient module remains similar to the PAROO as shown in Figure 6.18.

6.4.6 Results and Analysis

In this section we will tabulate the results obtained by the successful implementation of the partially parallel architectures for several benchmarks. We will also give detail analysis of the processing. Tables 6.3, 6.4 and 6.5 show the time taken by the partially parallel architectures to execute the BP algorithm.

- All the partially parallel designs running between 19 to 25MHz execute the

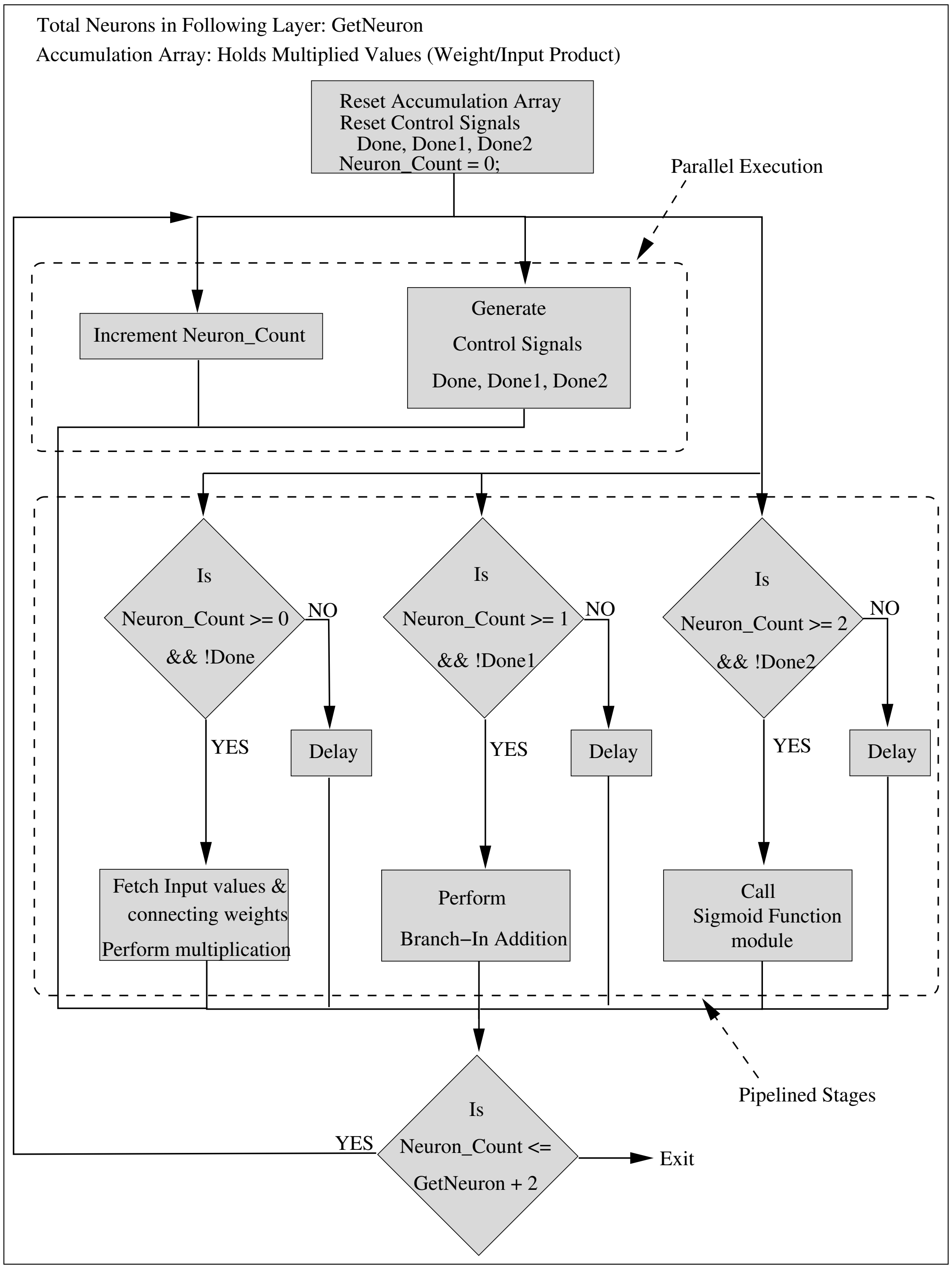


Figure 6.20: ASM for Neuron Accumulation module in PARIO

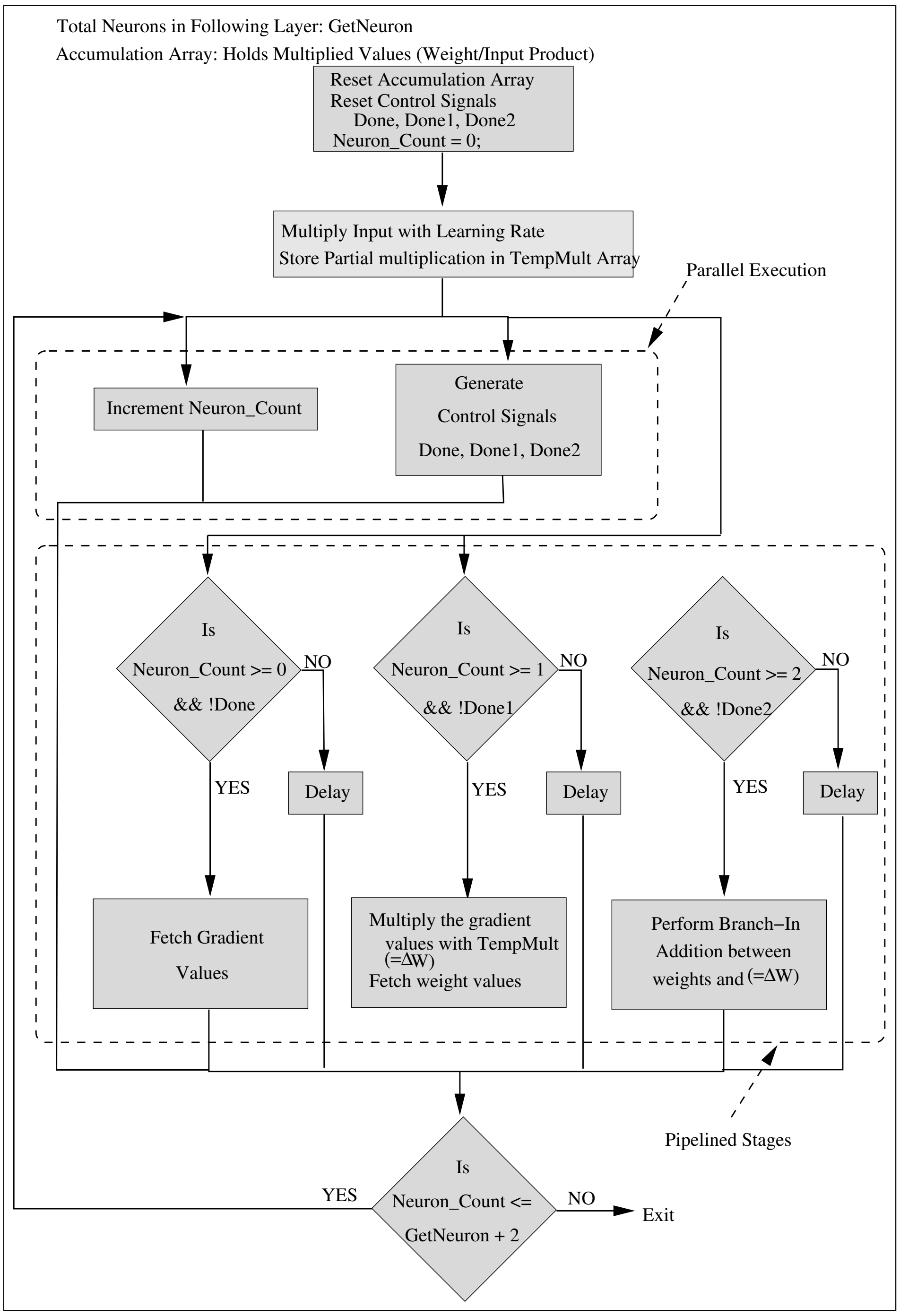

Total Neurons in Following Layer: GetNeuron
Accumulation Array: Holds Multiplied Values (Weight/Input Product)
Reset Accumulation Array
Reset Control Signals
Done, Done1, Done2
Neuron_Count = 0;
Multiply Input with Learning Rate
Store Partial multiplication in TempMult Array
Parallel Execution
Increment Neuron_Count
Generate Control Signals Done, Done1, Done2
Is Neuron_Count >= 0 && !Done
NO
YES
Delay
Is Neuron_Count >= 1 && !Done1
NO
YES
Delay
Is Neuron_Count >= 2 && !Done2
NO
YES
Delay
Fetch Gradient Values
Multiply the gradient values with TempMult (=ΔW)
Fetch weight values
Perform Branch−In Addition between weights and (=ΔW)
Pipelined Stages
Is Neuron_Count <= GetNeuron + 2
YES
NO
Exit

Figure 6.21: ASM for Weight Update module in PARIO

LR	Epochs	Partially Parallel Designs Time(mS)			Software Implementation Time(mS)
		PAROI	PAROO	PARIO	
0.25	7000	125	110	110	240
0.5	3300	60	50	62	125
0.75	2200	32	32	50	78
Frequency(MHz)		25	25	25	

Table 6.3: XOR data set experiments for 0.0022 Average Sum squared Error

LR	Epochs	Partially Parallel Designs Time(mS)			Software Implementation Time(mS)
		PAROI	PAROO	PARIO	
0.1	242	141	110	160	250
0.15	160	90	80	110	170
0.2	123	65	60	80	140
0.25	101	60	50	60	100
0.3	83	47	32	50	90
Frequency(MHz)		22.5	25	19	

Table 6.4: IRIS data set experiments for 0.03 Average Sum squared Error

LR	Epochs	Partially Parallel Designs Time(mS)			Software Implementation Time(mS)
		PAROI	PAROO	PARIO	
0.3	221	600	440	530	1100
0.4	166	440	330	400	780
0.5	135	360	270	330	630
Frequency(MHz)		19	20	18	

Table 6.5: Cancer data set experiments for 0.016 Average Sum squared Error

algorithm on average **2.25 faster** than the software version for all three benchmarks.

- PAROO produces the best result among all the partially parallel designs. This can be explained by the fact that PAROO doesn't use any Branch-In addition in any of the stages. Branch-In addition requires more clock cycles than Branch-Out addition. However a significant difference is not found in the results of the three partially parallel designs.

- The 3-piece linear approximation of the sigmoid function seems to have fairly worked for all the parallel designs. As mentioned in Section 6.2, such approximation introduces a 7% error in calculation of the sigmoid function. This error becomes significant in case of a smaller benchmark like XOR where it takes more iterations to converge than the LT approach. This is illustrated by Table 6.6.

LR	LT Approach SIPOB	Linear Approach All Parallel designs
	Epochs	Epochs
0.25	4600	7000
0.5	2100	3300
0.75	1400	2200

Table 6.6: Convergence Vs. Sigmoid function realization: XOR data set

Table 6.7 shows the Weight Updates per Second(WUPS) achieved by the partially parallel architectures.

Table 6.8 shows the area requirement in terms of the equivalent gate counts on the Virtex xcv2000e FPGA for the partially parallel architectures.

Benchmarks	Weight Update/Second(WUPS)in million		
	Partially Parallel Designs		
	PAROI	PAROO	PARIO
XOR	2.2	2.4	2.4
Iris	3.4	4.1	3
Cancer	9.1	12.2	10.1

Table 6.7: Weight Update per Second

Benchmarks	Gate Count(in million)		
	Partially Parallel Designs		
	PAROI	PAROO	PARIO
XOR	0.3	0.35	0.35
Iris	0.5	0.5	0.55
Cancer	1.2	1.3	1.3

Table 6.8: Gate Counts for Benchmarks

- The partially parallel design PAROO is able to achieve WUPS as high as 12.2 million for the cancer data set. This is higher than any of the architectures onto FPGA for the BP algorithm discussed in the literature review. PAROO also shows the best results among the partially parallel designs.

- WUPS and gate count in all three partially parallel designs are very similar. Hence we conclude that different combination of Branch-In and Branch-Out type of operations to the stages of the BP algorithm doesn't yield significant difference in the WUPS and gate count.

- As the size of the network increases, the area requirement on the FPGA chip also increases. However, due to increase in the circuit size, routing delays between them increases. Higher routing delays in a circuit cause increase in total delay which in turn limits the clock frequency with which the circuit can run. Placement And Route (PAR) of synthesis tool generates a timing analysis report indicating routing and logic delays. It was found for architectures which consumes high chip-area, that the percentage routing delay was quite higher than the logic delay. Hence relatively complex parallel designs run at lower frequency for Cancer data set.

- All partially parallel designs incorporated the facility of executing the algorithm in either single clock or two clock domain environment. The purpose of such incorporation was to check if the computation of the algorithm can be performed at a frequency higher than 25MHz on isolating Off-Chip SRAMs. Unfortunately, none of the architectures showed any improvement in frequency over their single-clock domain counterparts except SIPOB which showed some improvement. The obvious reason would be the longest path delay in the design exceeding 40ns and thus overriding the scaling-down effect of the Off-chip RAMs. All the results tabulated above are carried out in single-clock domain.

6.5 Fully Parallel Architecture

This section will explain the architecture of a fully parallel design. The section starts with describing the basics of such an architecture. Next, the main features of the developed design will be mentioned. Finally, various modules of the fully parallel architecture will also be described.

6.5.1 Motivation

A natural extension in the research after having developed partially parallel architectures is to build a fully parallel architecture. A fully parallel architecture employs a layer level parallelism. Such parallelism is achieved by the computation of all synapses in a layer in parallel. As mentioned earlier, this is accomplished by employing a number of multipliers equivalent to the total number of synapses in a layer. Such computation yields high level of performance and can be further augmented by assigning as many arithmetic and functional units as possible to the network. These units include Branch-In/Branch-Out adders and sigmoid function generators. However the high performance of a fully parallel architecture comes at a cost of high consumption of chip-area. Hence such an implementation was highly discouraged in the early stage when FPGA chip consisted of few thousand gates. Since the research conducted for this thesis has the facility of having an FPGA chip of more than a million gates, we have designed a fully parallel architecture to explore the possibility of enhancing the performance. However the maximum size of an ANN that can be fitted on the available hardware remains as a challenge for such designs. Hence the purpose of developing a fully parallel architecture is to investigate the possibility of successfully establishing a trade-off between chip-area and the performance for different size ANNs.
Figure 6.22 illustrates a fully parallel ANN where dark lines indicates parallel multiplication of all synapses with inputs in a layer.

Each node is designed with a separate Branch-In Adder module. This in place facilitates parallel computation of addition for all nodes in a layer. Such provi-

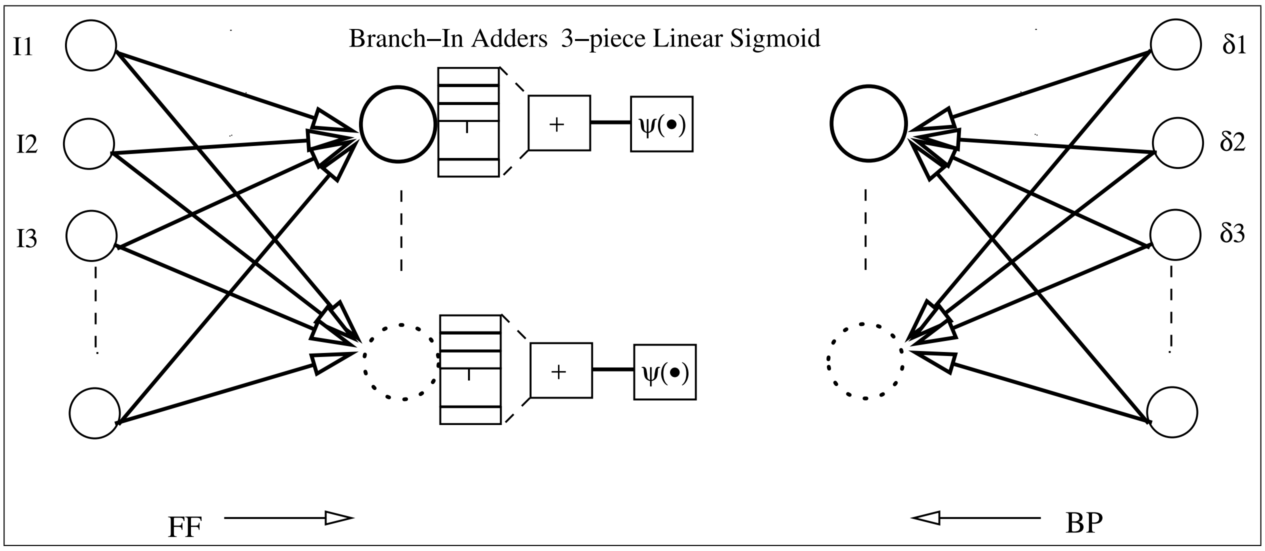


Figure 6.22: A Fully Parallel ANN

sion of Branch-In adder modules is necessary since all nodes in a layer will have corresponding multiplication values available simultaneously. Each node is also assigned a 3-piecewise linear sigmoid function to facilitate parallel computation of the function.

6.5.2 System Specification

The following are some of the features of the **F**ully **PAR**allel architecture **(FPAR)**.

- 16 bit Fixed-point number representation in the form of 1:4:11 (Sign:Integer:Fraction).
- Random number generation by employing LFSR
- Initial parameters stored in BlockRAMs
- "Pattern Mode" learning

- Number of multipliers are equal to the number of synapses of a layer maximum among the two layers. This will help reuse the same multipliers while computing a layer with less number of synapses.
- A separate Branch-In module for each node to perform addition. The number of required Branch-In modules is set to the maximum of the hidden/output layer neurons. This will help reuse the Branch-In adder modules while computing a layer with less number of neurons.
- Multiple 3-piecewise linear sigmoid functions.

6.5.3 FPAR modules

The basic structure of FPAR is illustrated by Figure 6.23. The functionality of various modules of the FPAR is explained in the following sections.

Initialization module

Figure 6.24 shows the Initialization module which is responsible for fetching initial parameters from Off-chip RAMs and storing it into BlockRAMs. The module is initiated by receiving Start_Initial signal from the main processor. The module also reads the topology file from the Off-chip RAMS and stores it to BlockRAMs. Since all synapses are computed in parallel in FPAR, all weight values of a layer are required to be accessed in parallel for efficient computation. Hence, every weight value of a layer is stored in a different BlockRAM as opposed to partially parallel architecture where only weight values connected to a single neuron are stored in

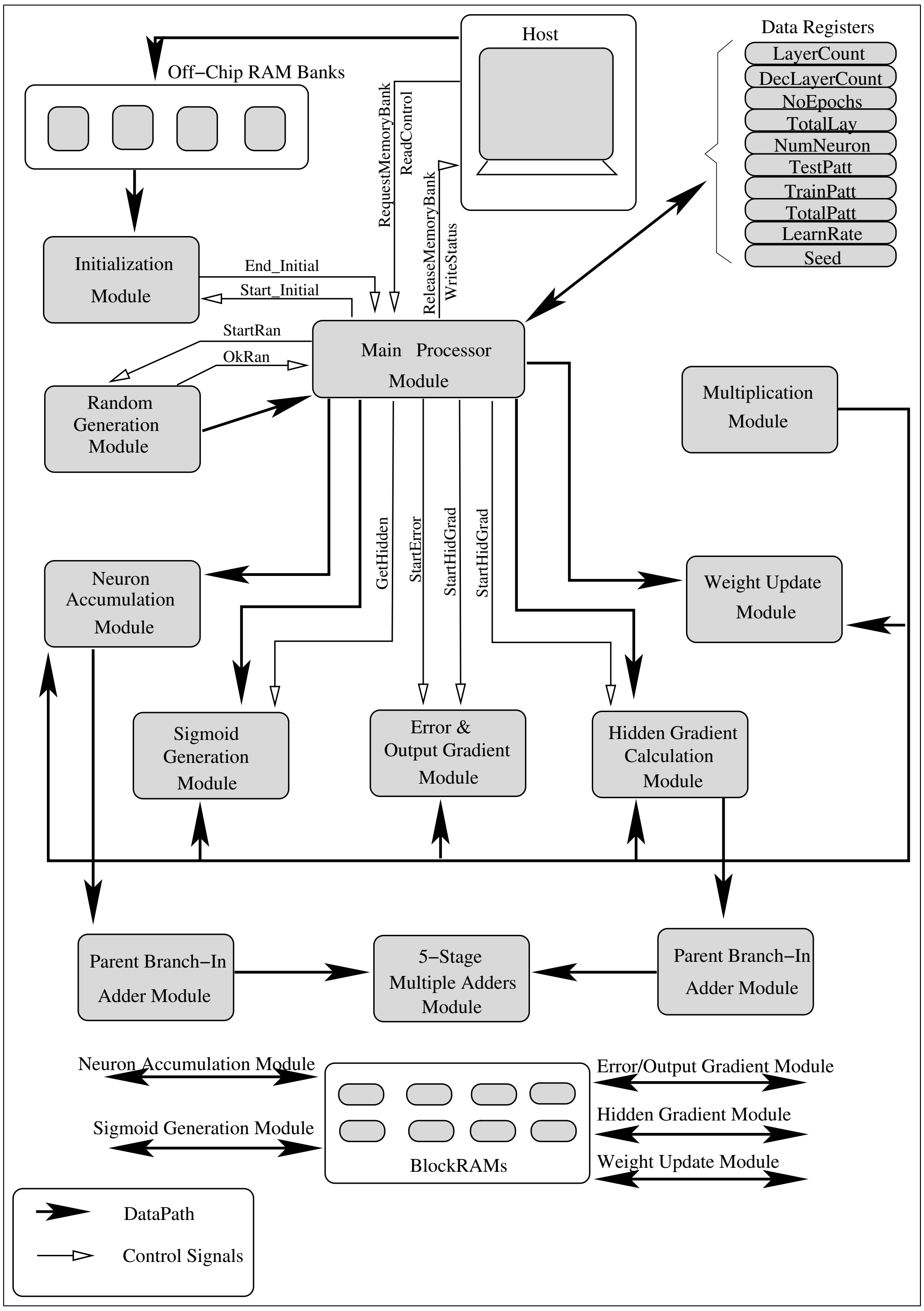


Figure 6.23: The basic structure of a fully parallel architecture

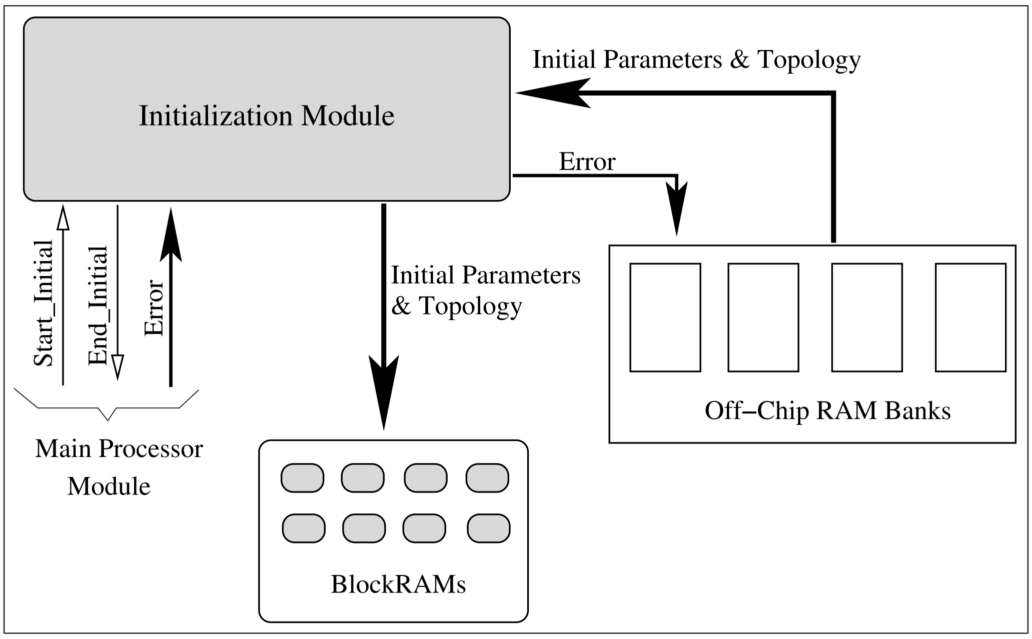
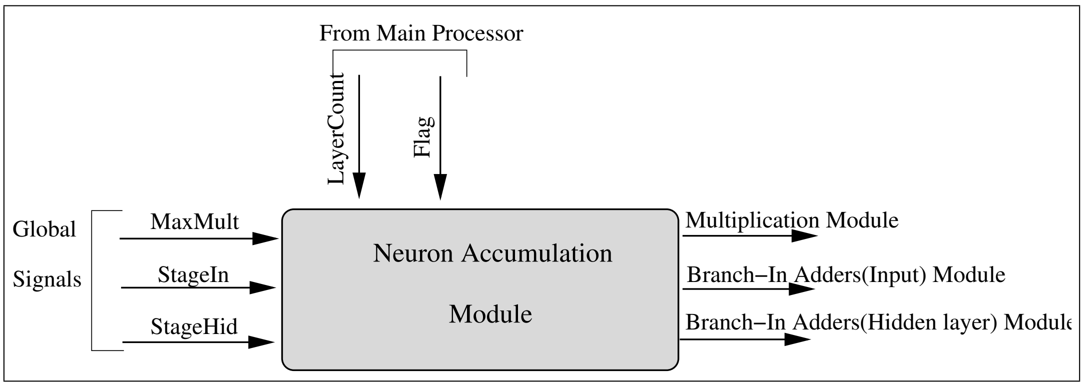


Figure 6.24: Initialization Module

different BlockRAMs. In Double clock domain designs, error value is stored to Off-chip RAMs via Initialization module.

Neuron Accumulation Module



From Main Processor
LayerCount
Flag
Global Signals
MaxMult
StageIn
StageHid
Neuron Accumulation Module
Multiplication Module
Branch–In Adders(Input) Module
Branch–In Adders(Hidden layer) Module

Figure 6.25: Neuron Accumulation Module

Figure 6.25 shows the Neuron Accumulation module which performs the multiply-accumulate operation for each layer. It fetches weight values for all synapses and input values of the layer and performs multiplication in parallel and stores in a multiplication array. It passes product values to the Branch-In adder modules to perform accumulation. StageIn and StageHid signals will provide the values for number of stages required for Branch-In addition in Input and hidden layers respectively. The Flag signal, depending on the layer, decides which Branch-In adder modules to be activated.

Sigmoid Function Module

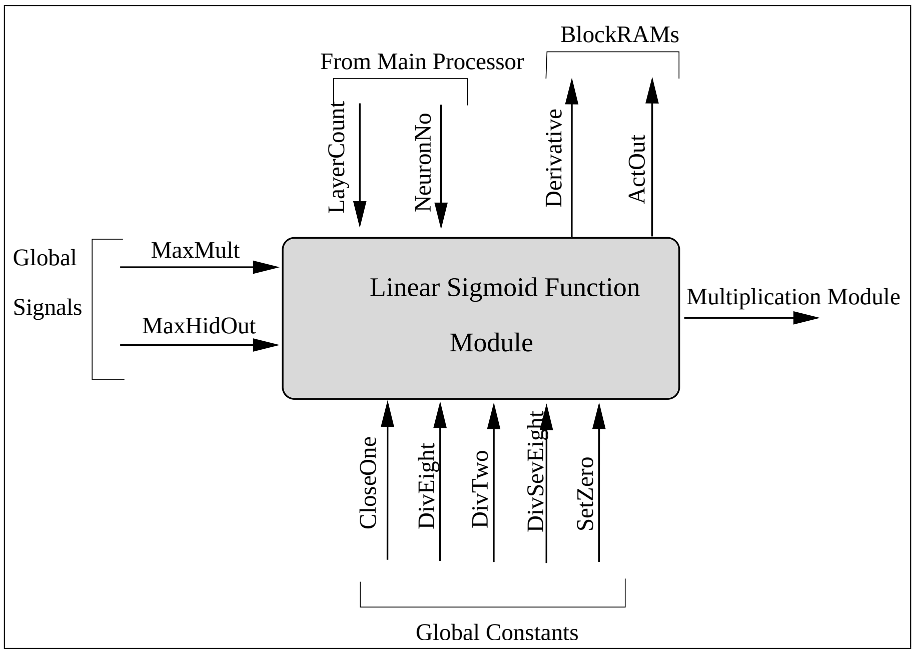


Figure 6.26: Linear Sigmoid Function Module

Figure 6.26 shows the linear sigmoid function module which computes the sigmoid functions for all nodes in a layer in parallel. The number of linear sigmoid function modules required is equal to the maximum of the hidden/output layer neurons. The module is supplied with several constant values to determine the break-points of the linear regions. This module implements equation 6.1, described earlier for the sigmoid function. The module also computes the derivative, $f(x)(1-f(x))$, of the sigmoid function. The module stores the sigmoid function values and derivative into ActOut and Derivative BlockRAMs. It also sets the GetHidden signal which allows the fetching of input values for the computation of the following layer.

Error and Output Gradient Module

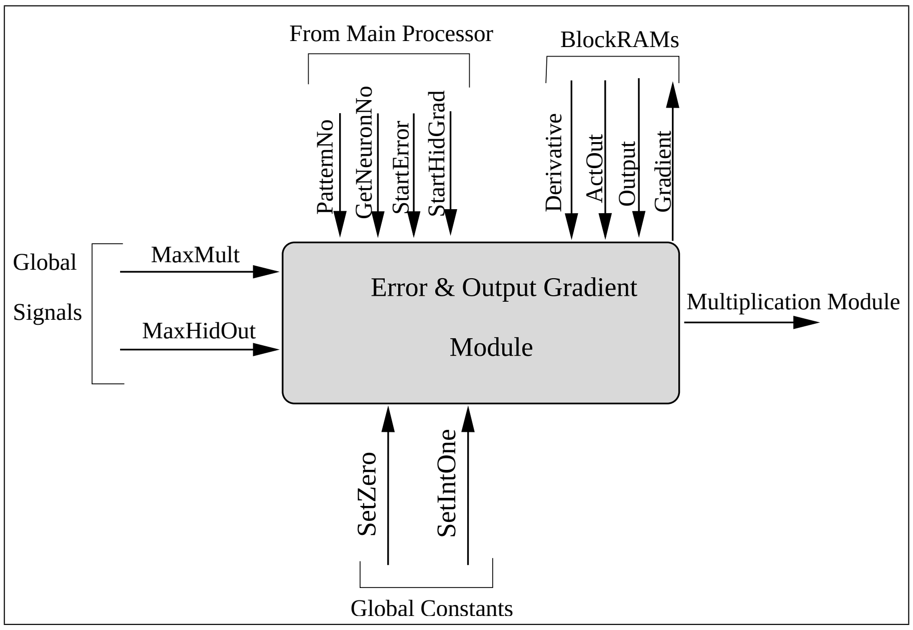


Figure 6.27: Error and Output Gradient Module

Figure 6.27 shows the error and output gradient module which computes error at each node in output layer. For discrete values of desired output (1 or 0), the error at each node will be either 2's complement or the negative of the actual output value. The module also computes error-gradient for the output layer. It issues

StartHidGrad signal to the Hidden layer gradient module to start the computation of error-gradient, in parallel to the computation of Instantaneous Sum squared Error(ISE). It then stores the Error in BlockRAM in case of Double-Domain design or in external RAM in case of Single-Domain design.

Hidden Layer Gradient Module

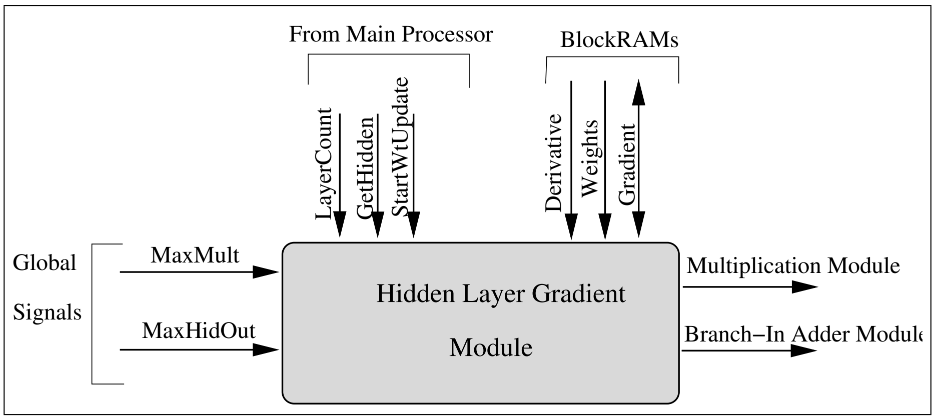


Figure 6.28: Hidden Layer Gradient Module

Figure 6.28 shows the Hidden layer gradient module which is responsible for computing error-gradient values in the hidden layer. The module computes all synapses in the output layer in parallel and issues a StartWtUpdate signal to the Weight Update module. The Back-Propagation and Weight-Update stages are not completely isolated in this architecture. The multiplication module becomes available once it has computed the synapses for hidden layer gradient module. The multiplication module can be used to compute $\triangle W$ in weight update module. Hence, following two computations can be performed in parallel without any clash of data or arithmetic units which helps save some clock cycles.

Hidden Layer Gradient Module: Branch-In addition and calculation of error-gradient of the hidden layer.

Weight Update Module Multiplication of error-gradient of the output layer, learning rate and the hidden layer output to generate $\triangle W$.

Weight Update Module

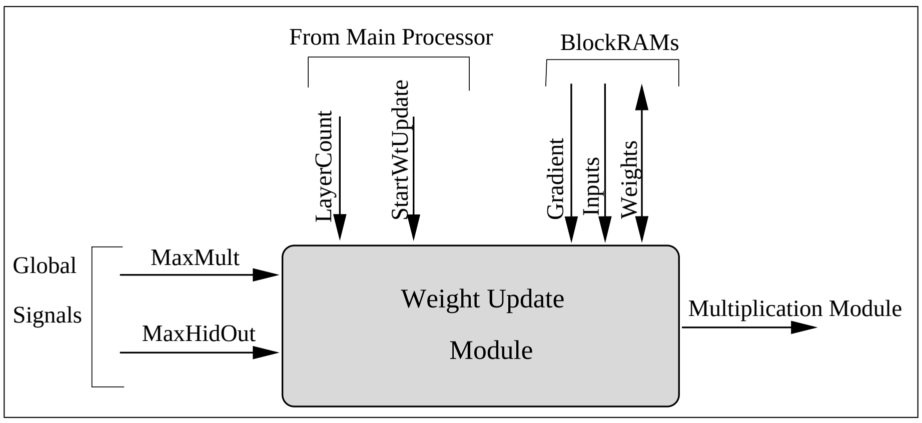


Figure 6.29: Weight Update Module

Figure 6.29 shows the Hidden layer gradient module which starts updating weights from the output layer and then moves to the hidden layer unlike the conventional forward movement from the hidden layer. The module starts updating weights in the output layer when it receives StartWtUpdate signal from the Hidden layer gradient module indicating that the multiplication module is available.

Multiplication and Addition Module

Figure 6.30 shows the multiplication and Branch-In Addition module which is responsible for computing all the parallel multiplication. It is supplied with two input arrays In1 and In2 and an Index to set the number of multipliers to be used.

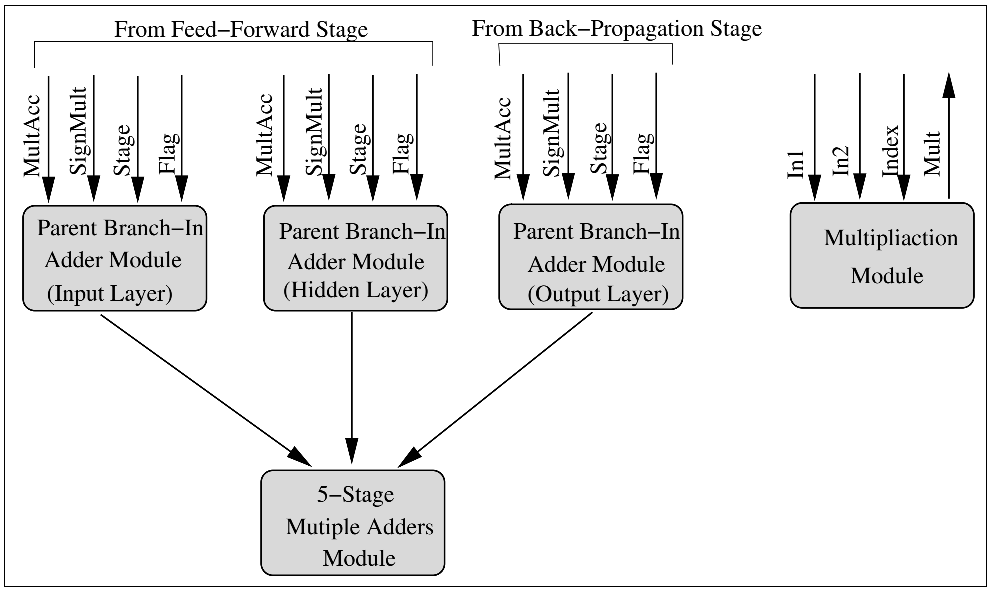

From Feed−Forward Stage
From Back−Propagation Stage
MultAcc
SignMult
Stage
Flag
MultAcc
SignMult
Stage
Flag
MultAcc
SignMult
Stage
Flag
In1
In2
Index
Mult
Parent Branch−In Adder Module (Input Layer)
Parent Branch−In Adder Module (Hidden Layer)
Parent Branch−In Adder Module (Output Layer)
Multipliaction Module
5−Stage Mutiple Adders Module

Figure 6.30: Multiplication and Branch-In Adder Module

Branch-In addition is performed by calling several modules. First the parent Branch-In module receives an array (MultAcc) of all multiplied values in a layer along with its sign (SignMult) and number of required stages (Stage). The parent module separates the values from MultAcc into a small array consisting of multiplied values for each node. These small arrays are passed to 5-stage multiple adders module. The 5-stage adders module can perform addition of maximum 32 multiplied values(stages: $16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$). Since every node is assigned with a 5-stage adder module, every stage of addition is performed simultaneously in all nodes. For example, if the input layer has 7 nodes and the hidden layer has 3 nodes, the design will require three 5-stage adder modules to accumulate seven multiplied values. Since seven values can be added in three stages (7: $4 \rightarrow 2 \rightarrow 1$), a 5-stage adder module will behave as a 3-stage adder module.

6.5.4 Results and Analysis

Tables 6.9, 6.10 and 6.11 show the time taken by different architectures and the software version of the design to execute the BP algorithm for the three benchmarks.

LR	Epochs	Serial Design Time(mS)	Partially Parallel Designs Time(mS)			Fully Parallel Design Time(mS)	Software Implementation Time(mS)
		SIPOB	PAROI	PAROO	PARIO	FPAR	
0.25	7000	440	125	110	110	**100**	240
0.5	3300	220	60	50	62	**62**	125
0.75	2200	140	32	32	50	**40**	78
Frequency(MHz)		25	25	25	25	25	

Table 6.9: XOR data set experiments for 0.0022 ASE

LR	Epochs	Serial Design Time(mS)	Partially Parallel Designs Time(mS)			Fully Parallel Design Time(mS)	Software Implementation Time(mS)
		SIPOB	PAROI	PAROO	PARIO	FPAR	
0.1	242	500	141	110	160	**50**	250
0.15	160	340	90	80	110	**40**	170
0.2	123	270	65	60	80	**30**	140
0.25	101	210	60	50	60	**25**	100
0.3	83	180	47	32	50	**16**	90
Frequency(MHz)		25	22.5	25	19	25	

Table 6.10: IRIS data set experiments for 0.03 ASE

- FPAR design running at 25Mhz for the Iris benchmark executes the algorithm on average **4.5 times faster** than the software version running on PIII dual processor 800MhZ PC.

LR	Epochs	Serial Design Time(mS)	Partially Parallel Designs Time(mS)			Fully Parallel Design Time(mS)	Software Implementation Time(mS)
		SIPOB	PAROI	PAROO	PARIO	FPAR	
0.3	221	3000	600	440	530	n/a	1100
0.4	166	2300	440	330	400	n/a	780
0.5	135	1900	360	270	330	n/a	630
Frequency(MHz)		25	19	20	18	n/a	

Table 6.11: Cancer data set experiments for 0.016 ASE

- FPAR takes approximately 60 clock cycles for the presentation of each pattern irrespective of the size of the network. Hence the implementation of a larger network will achieve more speed-up than smaller networks. This is the reason behind similar performance of FPAR to partially parallel design for the XOR benchmark.

- The Cancer benchmark is too large for FPAR to realize on a single Virtex xcv2000e chip. However, if the cancer data set were implemented on a hypothetical system consisting of more than one Virtex xcv2000e chip and assuming FPAR will run at 25MHz, the time it will take to execute 221 epochs at a learning rate of 0.3 can be given by:

$$\frac{Cycles\ per\ iteration \times No.\ of\ Patterns\ in\ an\ epoch \times No.\ of\ epochs}{Frequency}$$

$$= \frac{60 \times 200 \times 221}{25 \times 10^6} \approx 110ms$$

which is 10 times faster than the software version.

Table 6.12 shows the Weight Updates per Second (WUPS) achieved by the hardware designs.

- For partially parallel and fully parallel designs WUPS increases with an increase in the size of the network. This is due to the fact that the parallel architectures perform more ***multiplications*** in a same size clock cycle for the bigger benchmark than the smaller one.

- The FPAR validation based on the Iris data set also attains an improvement in the WUPS over the partially parallel designs. FPAR takes 60 clock cycles for the presentation of each pattern irrespective of the size of the network. Hence the larger the network the better the performance achieved by FPAR.

- It should be noted here that the most of the architectures on FPGA for the BP algorithm visited in the literature review contain multi-FPGA hardware. This facilitates the implementation of the desired size network and hence achieves high WUPS. The same will hold true if the cancer data set were implemented on a hypothetical system consisting of more than one Virtex xcv2000e chip. Assuming FPAR will run at 25MHz, the WUPS for the cancer data set can be achieved as high as 51 million as shown below:

  $$WUPS = N_w \star ClockRate/Cycles\ per\ Iteration = 122 \star 25 \times 10^6/60 \approx 51\ MWUPS$$

  where, N_w is the number of weights in the network

Table 6.13 shows the area requirement in terms of the equivalent gate counts on the Virtex xcv2000e FPGA.

Benchmarks	Weight Update/Second(WUPS)in million				
	Serial Design	Partially Parallel Designs			Fully Parallel Design
	SIPOB	PAROI	PAROO	PARIO	FPAR
XOR	0.6	2.2	2.4	2.4	**2.0**
Iris	0.9	3.4	4.1	3	**8.4**
Cancer	1.8	9.1	**12.2**	10.1	n/a

Table 6.12: Weight Update per Second

Benchmarks	Gate Count(in million)					
	Serial Designs		Partially Parallel Designs			Fully Parallel Design
	SIPEX	SIPOB	PAROI	PAROO	PARIO	FPAR
XOR	0.2	1.4	0.3	0.35	0.35	0.45
Iris	0.2	1.4	0.5	0.5	0.55	0.65
Cancer	0.2	1.4	1.2	1.3	1.3	n/a

Table 6.13: Gate Counts for Benchmarks

It should be noted here that the area requirement for FPAR is not significantly higher than the partially parallel designs. This is because FPAR doesn't need significant ***extra*** hardware for small benchmarks like XOR and Iris. However this will not hold true for the Cancer data set. This is due to the fact that the FPAR architecture for the Cancer data set (10-11-2) requires 121 multipliers and 10 Branch-In adder modules as opposed to 10 multipliers and a single Branch-In adder module required for the partially parallel designs.

Figures 6.31, 6.32 and 6.33 show the convergence graph of XOR, Iris and Cancer data sets for various learning rates. It should be noted here that in these figures not all data points are plotted for the purpose of clarity.

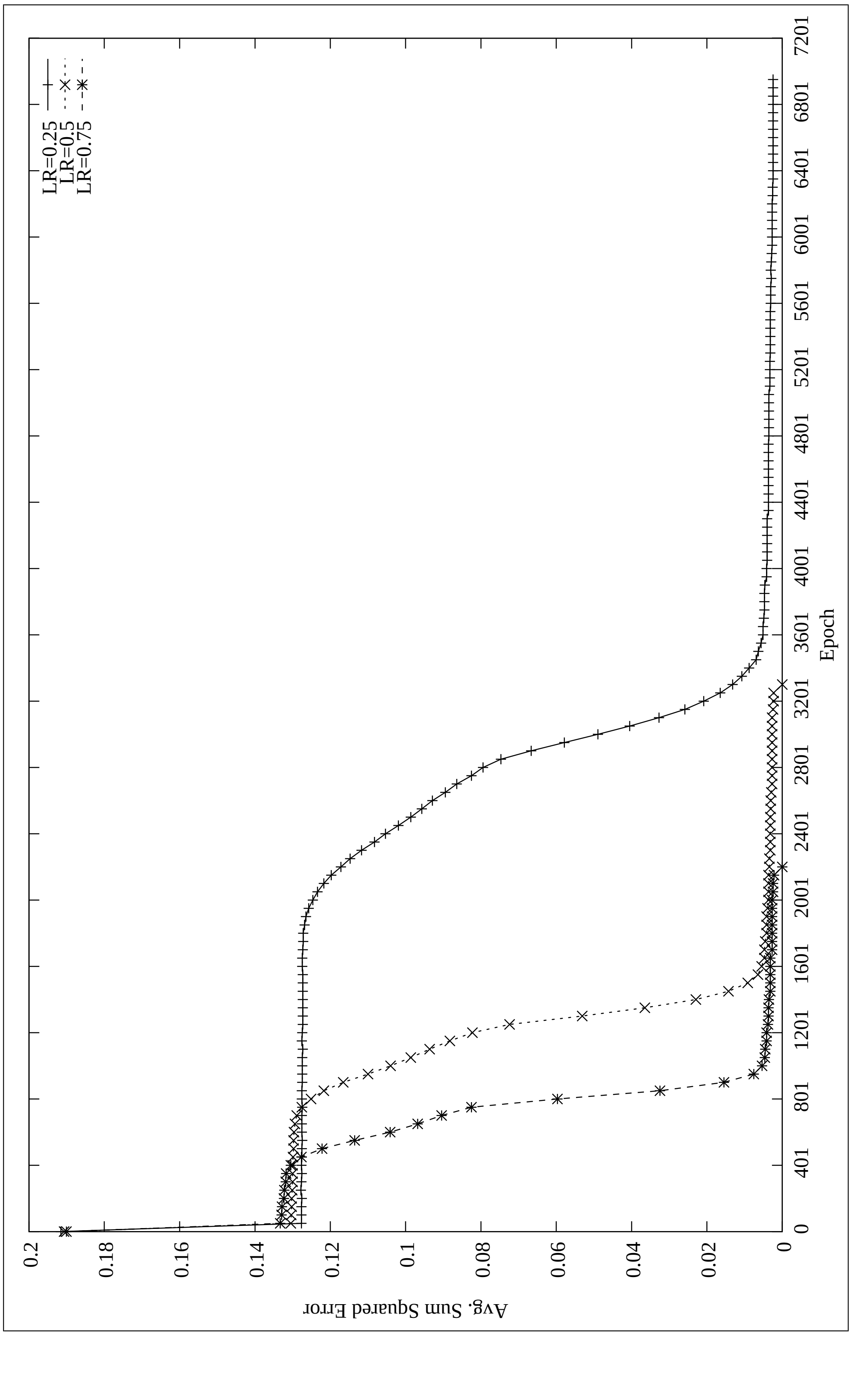


Figure 6.31: Convergence Graph for the XOR benchmark for various Learning Rates

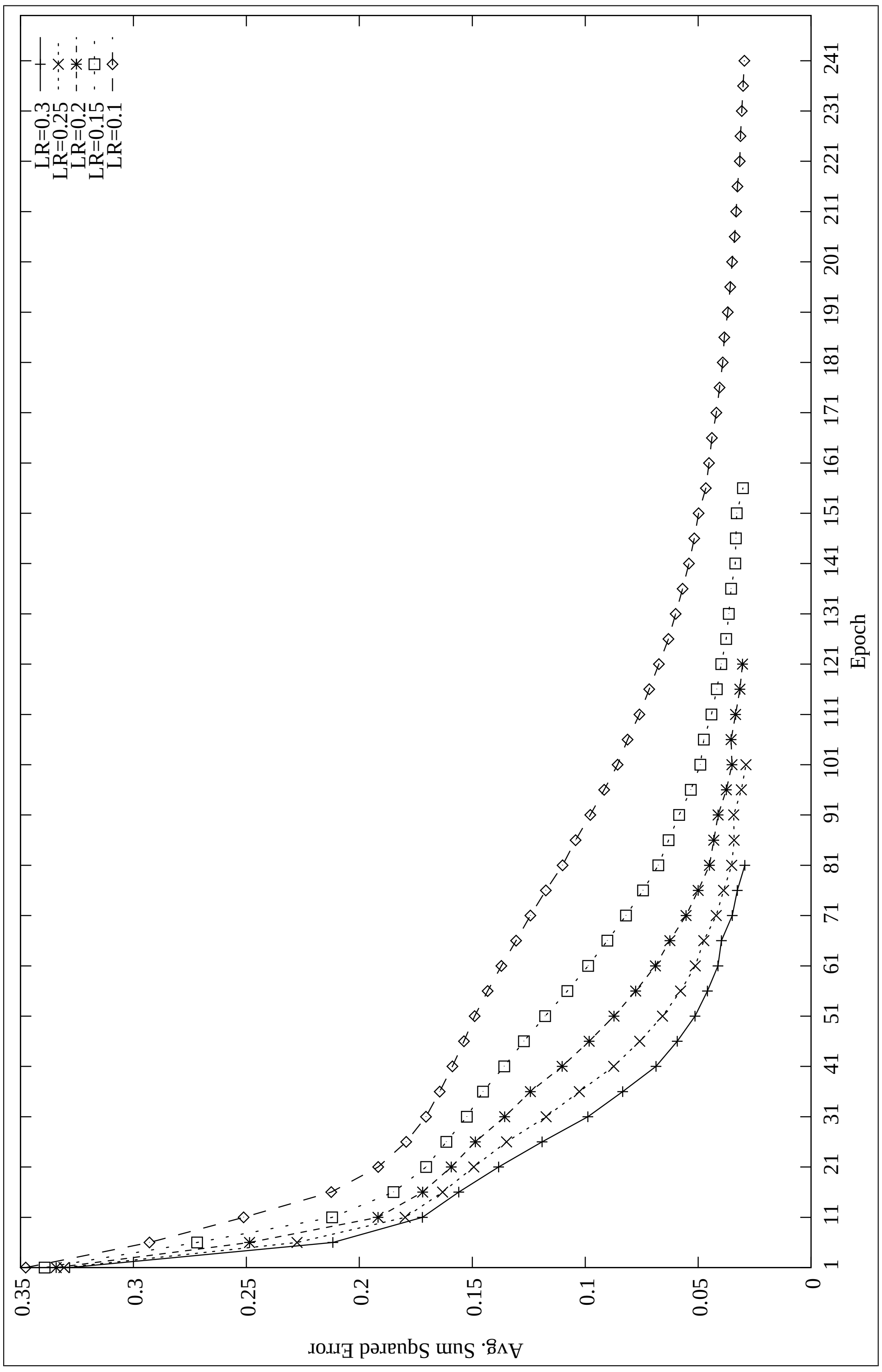


Figure 6.32: Convergence Graph for the Iris benchmark for various Learning Rates

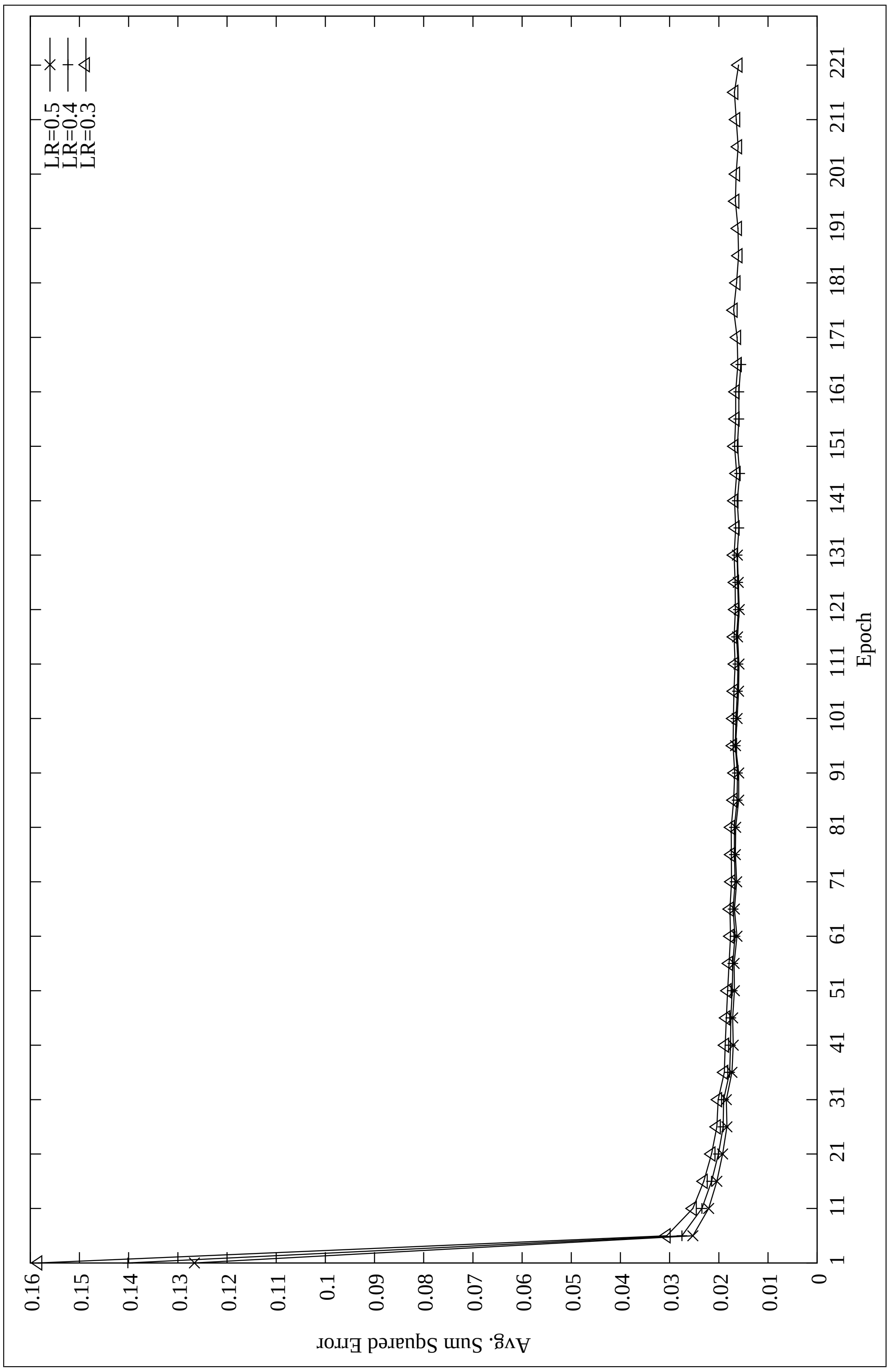


Figure 6.33: Convergence Graph for the Cancer benchmark for various Learning Rates

6.6 Summary

This chapter introduced the concepts of Branch-In and Branch-Out mode for an essential multiply-accumulate operation for the Back-Propagation algorithm. The details of implementing a 3-piece linearly approximated sigmoid function was also presented. Several architectures that enhance the functionality of the serial designs were presented including:

- PAROI, PAROO and PARIO: Partially parallel approach, differ in the assignment of Branch-In and Branch-Out mode operations to the stages of the BP algorithm
- FPAR: A fully parallel efficient design

Partially parallel architecture were designed to explore the trade-off between performance and area requirement. A 3-piecewise linear sigmoid function was introduced and validated the architectures for various benchmarks. Finally, a fully parallel architecture was developed which consisted of high number of arithmetic units to exploit the complete parallelism of a network.

Results obtained for the various architectures were tabulated and analyzed. The analysis was carried out based on performance in terms of gate count, speed and WUPS. The main highlights of the results are:

- FPAR's speed enhancement of 450% over the software version for the Iris data set.
- Partially parallel designs speed enhancement of 225% over the software version for all data sets.

- WUPS as high as 12.2×10^6 achieved and can be further augmented to 40×10^6 for FPAR design on multi-FPGA system.

Chapter 7

Conclusions and Future Work

Artificial Neural Networks (ANN) have a broad field of applications. The Back-Propagation (BP) algorithm has been popular among researchers working in the field of Artificial Intelligence as a learning algorithm. ANNs are derived from a biological structure of human brain which is a massive parallel network of neurons. Traditionally ANNs were implemented on General Purpose Processor (GPP) and ASICs. However these computing platforms suffer from the constant need of establishing a trade-off between performance and flexibility.

This thesis proposed several ANN architectures for the implementation of the BP algorithm on Virtex2000e FPGA. During the course of research initially two serial architectures were developed. The purpose of building serial architectures was to identify the bottlenecks and build the foundation for the development of parallel architectures. Next, three partially parallel architectures were developed. The purpose of these architectures was to explore the speed-up while accommodating various size networks. Some of the challenges faced in partially parallel architectures served as guidelines for developing a fully parallel architecture.

The multiply-accumulate is the basic arithmetic operation required in the execution of the BP algorithm. The concepts of Branch-In and Branch-Out mode of multiply-accumulate were introduced in detail. These two modes form the very basis of the parallel design development. The following architectures were proposed:

- SIPEX and SIPOB: Serial approach, differs in the storage of initial parameters and Lookup Table (LT) for the sigmoid function
- PAROI, PAROO and PARIO: Partially parallel approach, differ in the assignment of Branch-In and Branch-Out mode operations to the stages of the BP algorithm
- FPAR: A fully parallel design

Random number generation through LFSR and 3-piece linearly approximated sigmoid function were described in detail.

Three benchmarks XOR, Iris and Cancer, relatively small, medium and large respectively, are chosen to validate the architectures. The main contribution of this thesis is the development of the partially parallel and fully parallel designs which are shown to perform 2.25 and 4.5 times faster than the software version. The partially parallel designs attempt at establishing a trade-off between chip-area and the performance while still achieving WUPS as high as 12×10^6, higher than any of the architectures reviewed in the literature.

Yet another contribution worth mentioning here is the successful implementation of the Cancer benchmark which is a "real world" problem for all the architectures except FPAR. The validation of Cancer benchmark for FPAR architecture requires high number of arithmetic modules which will result in high chip-area consumption. It was shown through calculation for a hypothetical multi-FPGA system that the FPAR can achieve WUPS up to 51×10^6 and 10 times speed enhancement over the software version for the Cancer data set. Another contribution is the successful implementation of the XOR and the Iris benchmarks for FPAR architecture with a significantly low hardware requirement of 0.4 and 0.6 million gates on the Virtex xcv2000e FPGA chip respectively.

Throughout the research work, Handel-C was used as the hardware description language. General C like structure of Handel-C makes the development easier for a novice hardware engineer. Handel-C allows fast simulation of large designs as opposed to VHDL simulator Modelsim. The development time is considerably less than VHDL. However it was found that debugging in Handel-C was not easy for ANN architectures. We have used fixed-point arithmetic number representation. The problem with Handel-C compiler is that it can not display fixed point number in output text file instead it displays equivalent decimal values for integer and fractional part. Since the BP algorithm involves tedious arithmetic operations, debugging of various parameters in fixed-point numbers becomes complex. Another problem was associated with RC1000 reconfigurable board which requires the scaling down the frequency of the design to $\frac{1}{4}$ of the external frequency in case of Off-chip memory access.

7.1 Future Work

The run-time reconfiguration characteristic of an FPGA can also be utilized to implement large ANNs. The BP algorithm is well suited for such reconfiguration as it can be easily divided into three time-exclusive stages. However the time required to reconfigure the various stages on an FPGA can be a bottleneck in such designs. Also, whether an FPGA based system can have the reconfiguration ability varies among the vendors of such hardware.

Architectures described in this thesis execute three stages of the BP algorithm in a sequence. Although full parallelism has been applied at a layer level, significant speed improvement over the software version was not possible. Investigation can be carried out to pipeline three stages: Feed-Forward, Back-Propagation and Weight Update.

The possibility of combining the initial parameters and topology with .bit file and directly downloading it to the BlockRAMs of an FPGA should be investigated. Xilinx synthesis tool provides such utility for specific hardware platform. This will eliminate the requirement of accessing Off-chip RAM which can result in a faster execution of the algorithm.

Architectures based on the current work can be developed in other Hardware Descriptive Language such as VHDL and performance in terms of the area requirement and the speed can be compared.

Future work based on the FPAR architecture can be carried out to implement the cancer data set on a dedicated multi-FPGA hardware. Multi-FPGA systems are found to be very common in the literature review for executing the BP algorithm.

Multi-FPGA hardware not only accommodates large networks but also achieves high speed enhancement and WUPS.

Instead of using the ready-made fixed-point library supplied by the Handel-C DK compiler, a separate fixed-point library can be developed and the area-speed criteria can be investigated. for the fixed-point arithmetic operations.

Appendix A

Glossary

ANN	:	Artificial Neural Network
BP	:	Back-Propagation
FPGA	:	Field Programmable Gate Array
GPP	:	General Purpose Processor
ASIC	:	Application Specific Integrated Circuits
CTR	:	Compile Time Reconfiguration
RTF	:	Run Time Reconfiguration
WUPS	:	Weight Update Per Seconds
CUPS	:	Connections Update Per Seconds
MLP	:	Multi-Layer Perceptrons
LR	:	Learning Rate
LFSR	:	Linear Feedback Shift Register
VHDL	:	Very High Speed Integrated Circuit Hardware Description Language
LUT	:	Lookup Tables (On an FPGA chip)

VLSI	: Very Large Scale Integration
LT	: Lookup Table (Generalized)
PWL	: Piece-Wise Linear
SIPEX	: Serial Implementation with Parameters/LT in EXternalRAM
SIPOB	: Serial Implementation with Parameters/LT in On-chipRAM and BlockRAM
PAROI	: PARallel architecture with (Branch)Out and (Branch)In
PARIO	: PARallel architecture with (Branch)In and (Branch)Out
PAROO	: PARallel architecture with (Branch)Out and (Branch)Out
FPAR	: Fully PARallel
ISE	: Instantaneous Sum-squared Error
ASE	: Average Sum-squared Error
ASM	: Algorithmic State Machine

Appendix B

RC1000 Board

The RC1000 board [Supp01] is a reconfigurable platform consisting of one large Xilinx FPGA with four banks of memory for data operations. The RC1000 is a PCI bus plug-In card for PICS. It also has two PC sites for I/O with the outside world. Figure B.1 shows the block diagram of the board.

B.1 FPGA

The RC1000 has a single site for a Xilinx FPGA in a BG560 package. The allowable FPGAs are 4085XL, 40150XV, Virtex V1000 and Virtex 2000E. Two RC1000 boards are available for this research, one containing Virtex V1000 and the other containing Virtex 2000E FPGA chip. Virtex1000 and Virtex2000e chip contain 1 million and 2 million gates respectively. Virtex1000 contains 32 BlockRAM modules, each of which can be configured to operate as 256 x 16 memory block. Whereas,

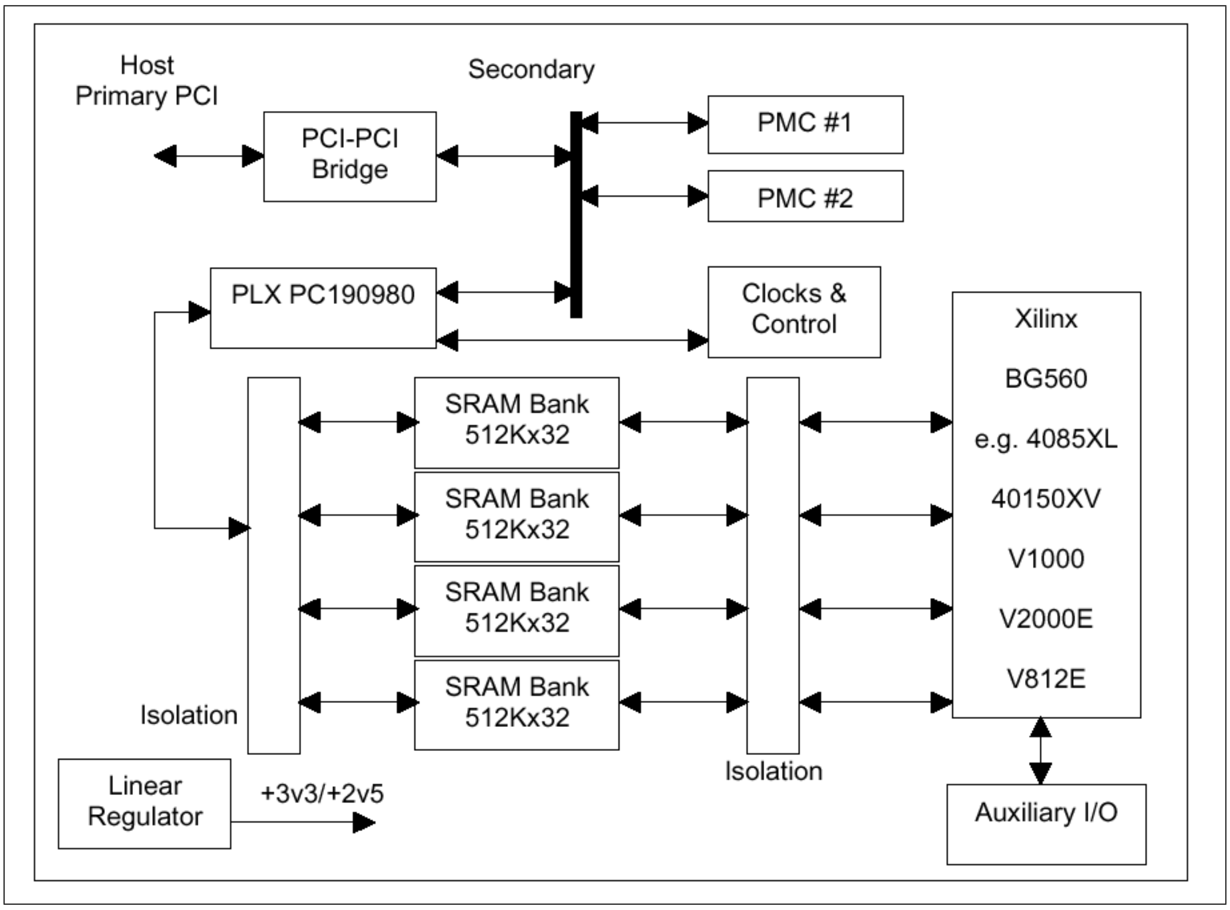


Figure B.1: The block diagram of RC1000 board

Virtex2000e contains 160 BlockRAM modules of the same structure.

The RC1000 FPGA can be programmed from the host PC over the PCI bus or Xilinx XChecker download cable, or on-board serial ROM. The internal state of the FPGA can be readback for debugging purposes either using the XChecker cable or over the PCI bus to the host.

B.2 Memory

A standard complement of four Asynchronous memory bank of 2Mbytes x 8 each is provided. All four memory banks are accessible by both the FPGA and any other device on the PCI bus.

B.3 Clocks

The FPGA has two of its pins connected to clocks. One pin is connected to either a programmable clock or an external clock, selected by a jumper. The other pin is connected to either a programmable clock or the PCI9080 local bus clock, also selectable by a jumper. The latter programmable clock can be a single step clock driven by the host. Table B.2 shows the jumper connection for setting the clock source to the designs on FPGA.

JP2 pins connected	Clock Source
1-2	External clock on J10
2-3(default)	Programmable MCLK

Table B.1: Jumper 2 settings for Clock

JP3 pins connected	Clock Source
1-2(default)	Programmable VCLK
2-3	PCI 9080 LCLK

Table B.2: Jumper 3 settings for Clock

The programmable clocks are programmed by the host PC, and have a **frequency range of 400KHz to 100MHz**.

B.4 Data Transfers

There are 3 methods of transferring data or communicating between the FPGA and any PCI device, via the PLX PCI9080 controller chip.

- Bulk data transfers between the FPGA and PCI bus are performed via the memory banks. Synchronization between the FPGA and the other device is done using one of the other communication methods. Only PCI devices have access to PLX registers, the FPGA does not. Hence only PCI devices can set up and initiate DMA transfers. This mode of transfer is used in transferring initial parameters and LUT in the architectures to Off-chip RAMs.

- There are two unidirectional 8-bit ports, called control and status, for direct communications between the FPGA and PCI bus. Semaphores indicate when data has been written or read. These ports are used in our designs to issue a start signal from the host to the FPGA in the beginning and end signal from the FPGA to the host in the end.

- The PLX user I/O pins USER1 and USER0 are both connected to the FPGA to provide for single-bit communications.

B.5 Memory Banks on RC1000 Board

There are four 32-bit memory banks external to the FPGA. Each bank is 2Mbytes of asynchronous SRAM made up from four 512k x 8 memory chips. Each memory bank has separate address, data and control signals. Hence the FPGA can access all four banks simultaneously and independently.

All four banks also appear in the PCI address space, so they can be accessed by both the host and other PCI devices via the PCI9080 PCI bridge chip. The host support software allows the mapping of the memory into the virtual address space of a host application. Only one bank can be accessed at a time by the host or another PCI device.

The arbitration between the FPGA and PCI9080 is controlled by on-board logic as described in the next section.

B.5.1 Arbitration

Each bank of SRAM on the RC1000 can be accessed by the host via the PLX PCI9080 or by the FPGA. The interface to each SRAM bank from both FPGA and PCI9080 is controlled using four switches as shown in the Figure B.2 The switches work as four independent multiplexors, one per bank, to allow each bank to be accessed by one of the FPGA and the PCI9080 at a time, but never both. This arrangement minimizes the risk of damage caused by contention that can occur due to poor design on FPGA. The switches add 1ns of delay to address, data and control paths. The arbitration logic is implemented in a COLD that connects to the PCI9080 local bus and the FPGA.

B.5.2 Memory Access

Handel-C provides support for interfacing Off-Chip using the *ram* or *rom* keywords. The usual technique for specifying timing in synchronous and asynchronous RAM is to have a fast external clock which is divided down to provide the Handel-C clock

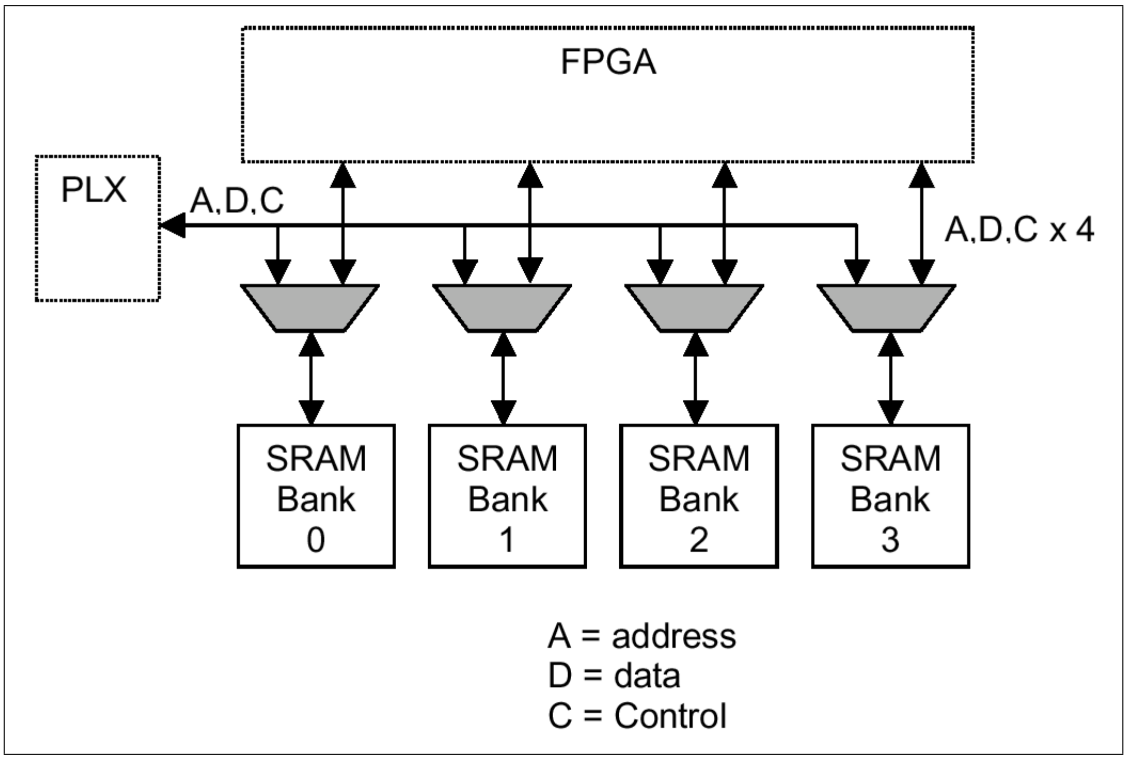


Figure B.2: Architecture of SRAM banks

and used directly to provide the pulses to the RAM.

The fast external clock uses the Handel-C *westart* and *welength* specifications to position the write strobe. This method of timing ASRAMs depends on having an external clock faster than the internal Handel-C clock. In this case the external clock is divided by specifying *westart* and *welength* parameters:

$$set\ Clock\ =\ external\_divide\ "pin\_number"\ 4$$

$$ram\ unsigned\ 6\ x[10]\ with\ \{westart = 2,\ welength = 1\ \}$$

The above example starts the pulse 2 whole external clock cycles into the Handel-

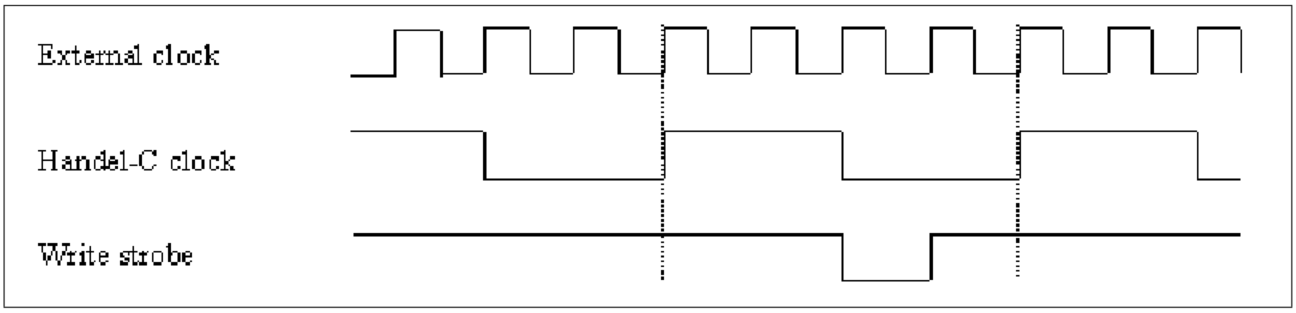


Figure B.3: Timing Diagram

C clock cycle and gives it a duration of 1 external clock cycle. Since the external clock is divided by a factor of 4, this is equivalent to a strobe that starts half way through the internal clock cycle and has a duration of one quarter of the internal clock cycle.

This timing allows half a clock cycle for the RAM setup time on the address and data lines and one quarter of a clock cycle for the RAM hold times.

Appendix C

Host Communication

Figure C.1 shows the very basic communication between the FPGA and memory banks to the host. The host is a PC running C++ program through which com-

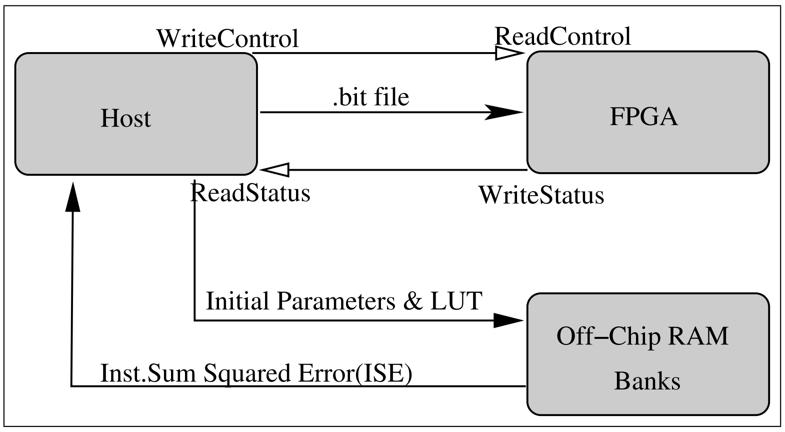


Figure C.1: Communication between the host and the RC1000 board

munication is established to the FPGA and the Off-chip RAM banks. The host support library is provided with the RC1000 board which contains several functions needed to achieve the communication. The host essentially performs the sequence of operations as shown in Figure C.2:

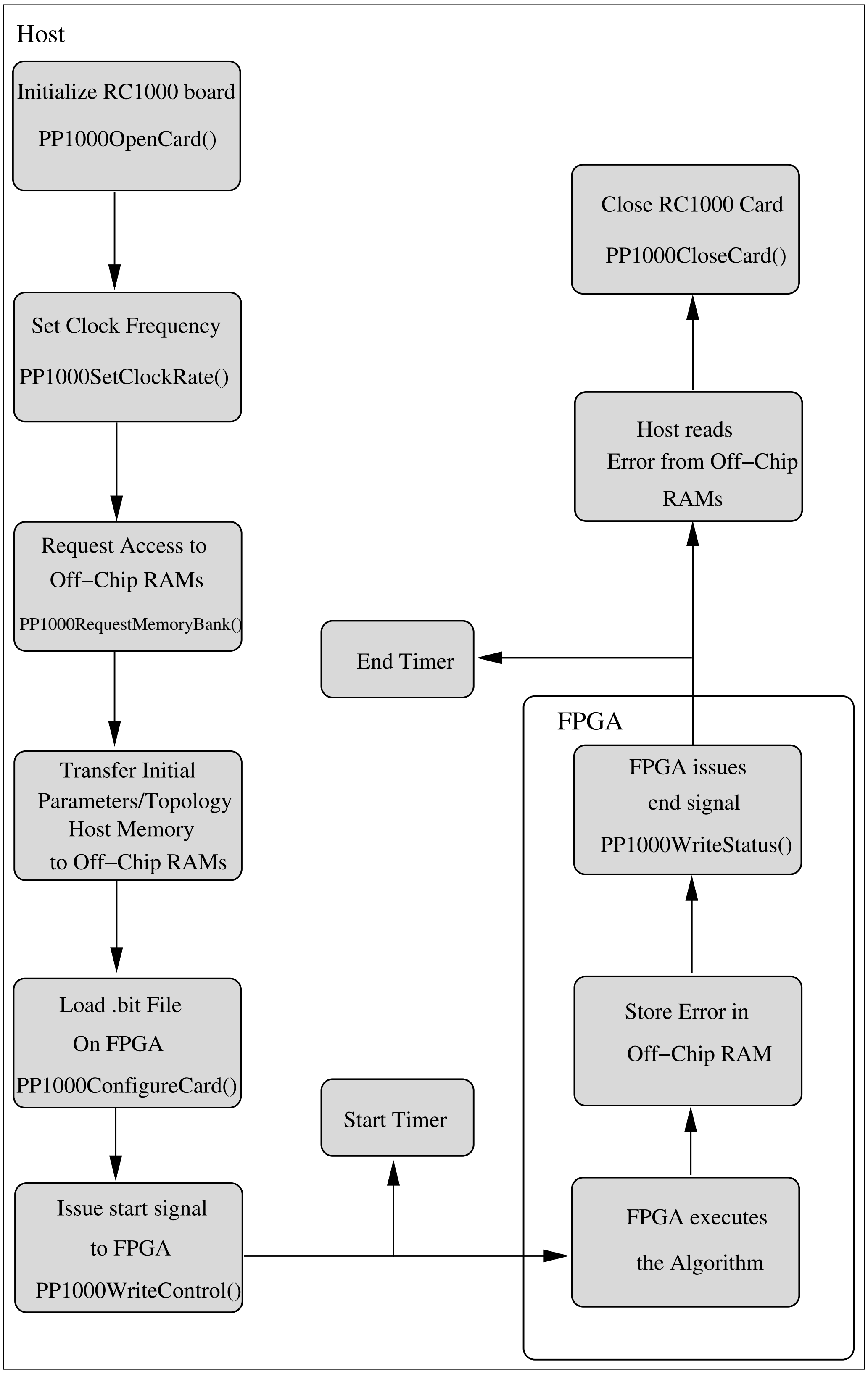


Figure C.2: Communication between Host and FPGA

Appendix D

Memory Creation on the Virtex FPGA chip and EDIF

There are various ways that handel-C allows to create a memory in Virtex and VirtexE series FPGAs. The type of the memory can be:

- Distributed RAM or On-Chip RAM, which is implemented by look-up tables in the logic blocks of FPGA
- BlockRAM, Fixed sized synchronous memory available on Virtex FPGAs
- Registers, which is implemented in flip-flops in FPGA

Xilinx Virtex and VirtexE FPGAs are made up of Configurable logic blocks (CLBs). Each CLB consists of 2 slice in these FPGAs and each slice contains two LUTs. Each LUT can be configured as a 16 by 1 bit synchronous RAM, which is called distributed RAM or On-chip RAM. The two LUTs within a slice can be combined to form a 16 by 2 bit or 32 by 1 bit synchronous RAM. Deeper and wider on-chip RAM can be created by consuming more LUTs of FPGA. However it introduces a significant delay because of the multiplexors and routing required between CLBs. This is an important issue a designer has to bear in mind while allocating available memory resources, on or off chip, to initial and intermediate parameters. It is observed during the experimentation that creating longer on-chip RAM not only consumes considerably large chip area but takes longer in routing the design. The maximum number of neurons a layer can accommodate in SIPOB architecture is set at 12. When experiments carried out with maximum 15 neurons in a layer it was found that placement and route tool (PAR) exhausted all the routing resources. This is due to the fact that length of On-Chip RAMs for storing Input and Output are directly proportional to the maximum neurons in a layer. On-chip RAM can be accessed only once in a clock cycle.

Registers are created on FPGA by flip-flops. They are normally used to define variables in handel-C program. Registers can be combined to form an "array" of memories. Various values within an array can be accessed simultaneously in one clock cycle. However this comes at a high logic cost if a variable is used to index the array. This is because the compiler will generate multiplexor to switch between the registers to index the array.

BlockRam memory blocks are organized in columns. All Virtex devices contain two such columns, one along each vertical edge and extend full height of the chip. Whereas VirtexE devices contains few vertical BlockRam columns spread throughout the chip and separated by vertical CLB columns. Each BlockRam column in VirtexE devices also extend to the full height of the chip. A BlockRam also includes dedicated routing to provide an efficient interface with both CLBs and other Block-

RAMs. Figure D.1 and Figure D.2 show the position of BlockRAMs on VirtexE and Virtex FPGAs.

D.1 EDIF

The (EDIF) is a format used to exchange design data between different CAD systems, and between CAD systems and Printed Circuit fabrication and assembly. The 'Electronic' refers to the type of data, i.e. design data for electronic systems and not the mechanism of interchange. An EDIF file is machine readable and may be interchanged electronically. Such CAD systems are often referred to as Electronic CAD (ECAD) systems or Electronic Design Automation (EDA) Systems.

The EDIF format is designed to be written and read by computer programs that are constituent parts of EDA systems or tools, or by software that is part of front end manufacturing systems (CAM stations). By their very nature, the EDIF standards are behind the scenes for most EDA users.

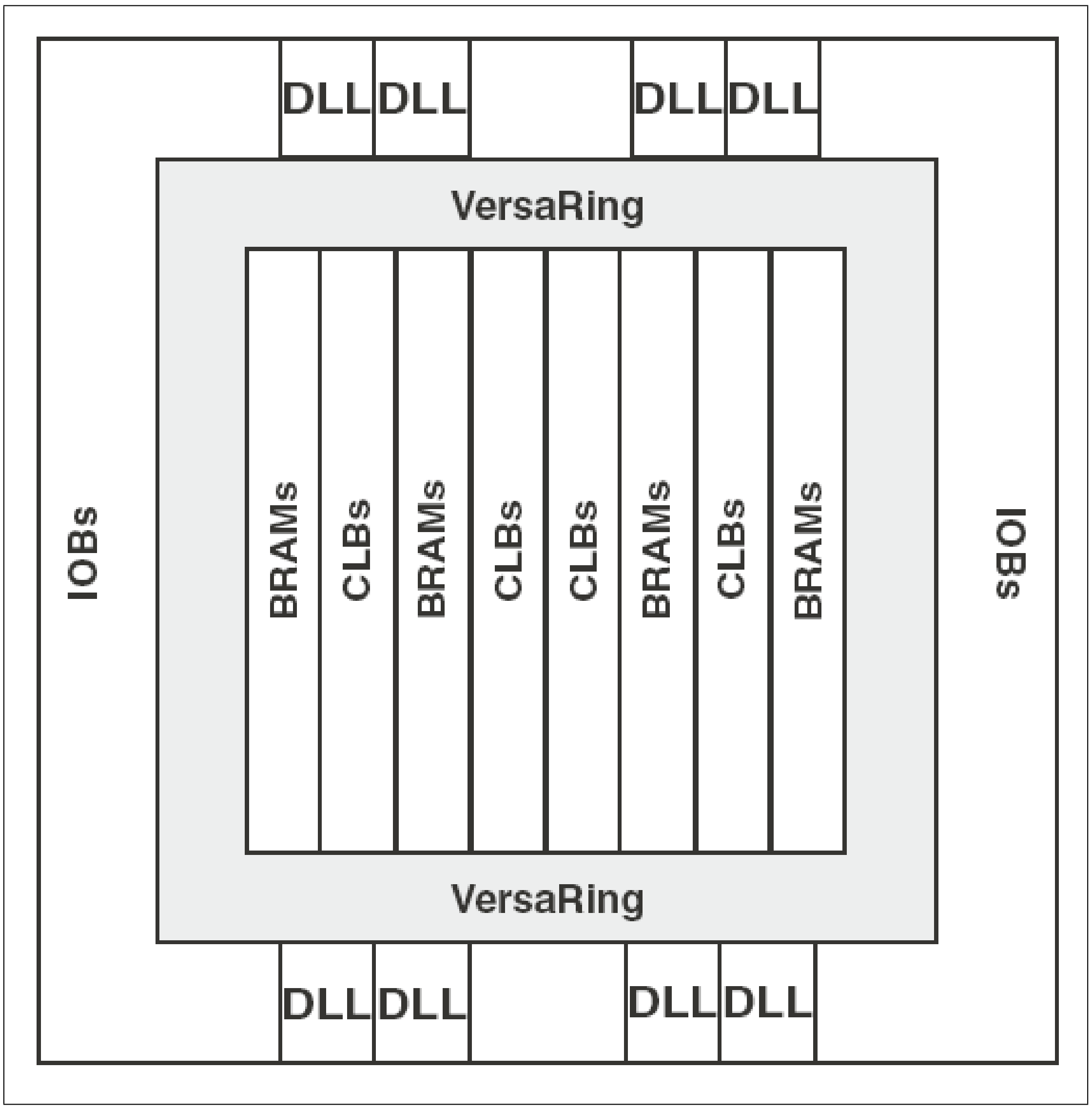


Figure D.1: BlockRAMs on VirtexE FPGA

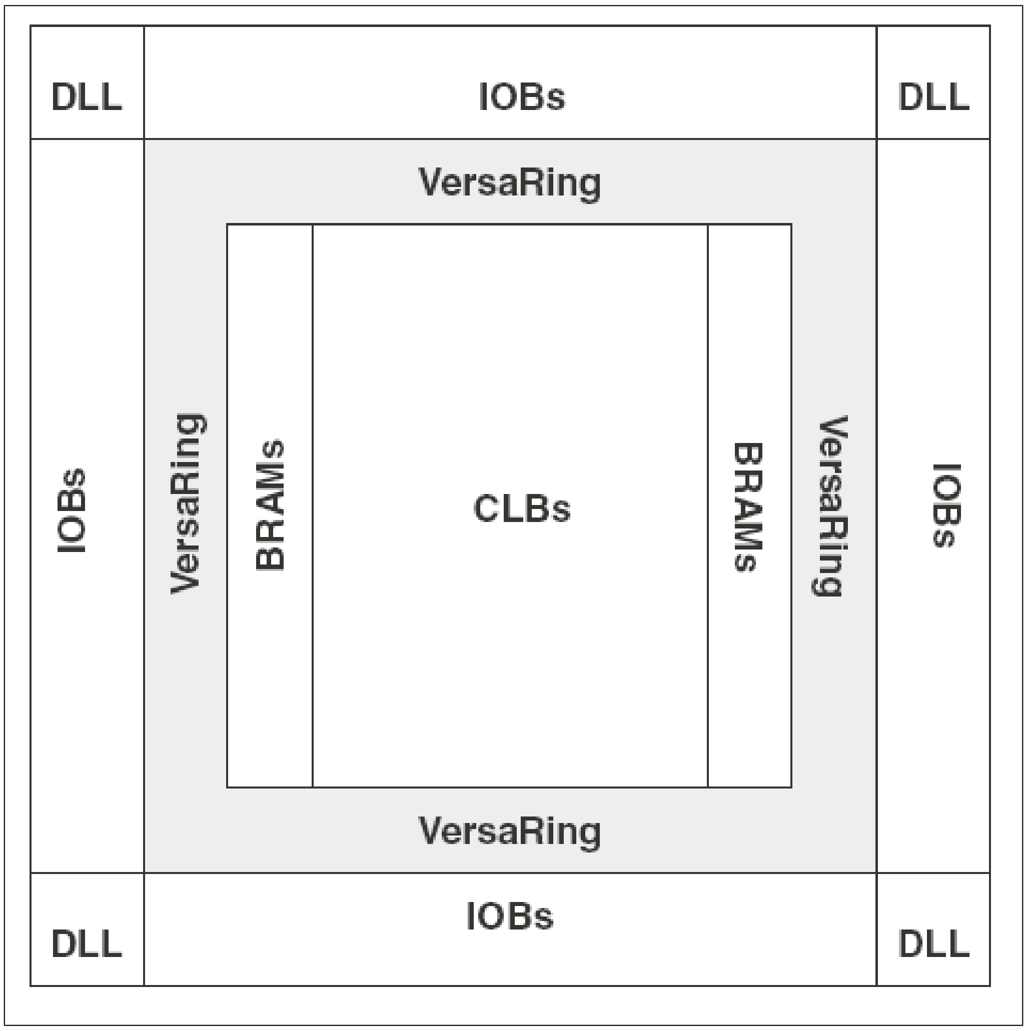


Figure D.2: BlockRAMs on Virtex FPGA

Appendix E

An Overview of the Simulation and Synthesis Process

In order to encode and then implement the design on an FPGA, it is important to understand the simulation and synthesis flow of the Handel-C language. There are two design flows in Handel-C: building for simulation and building for hardware. Figures E.2 and E.1 show the simulation flow and the synthesis flow for designs generated in DK compiler.

E.1 An overview of Simulation Process

For simulation of a design written in Handel-C, first all source files are compiled and linked to generate .dll files. Handel-C provides the facility of integrating C/C++ source files into a project along with Handel-C files. In such project DK compiler invokes C/C++ compiler in background to compile and then links with Handel-C

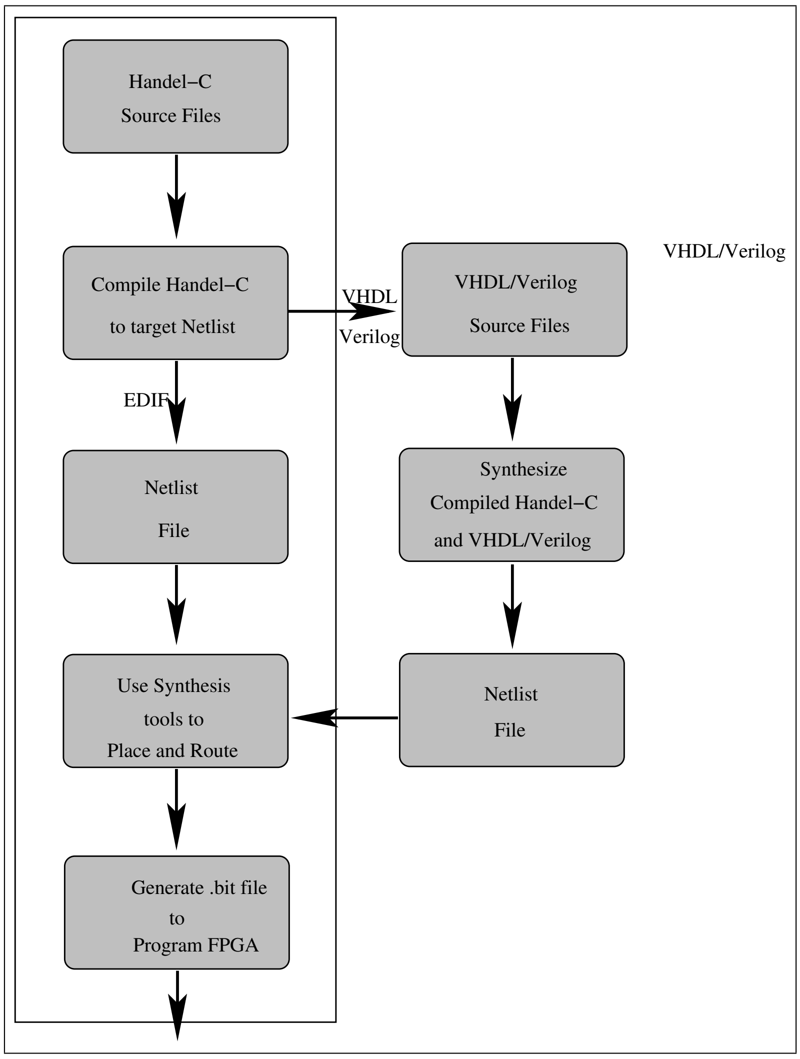

Handel–C Source Files
Compile Handel–C to target Netlist
VHDL
Verilog
VHDL/Verilog Source Files
VHDL/Verilog
EDIF
Netlist File
Synthesize Compiled Handel–C and VHDL/Verilog
Use Synthesis tools to Place and Route
Netlist File
Generate .bit file to Program FPGA

Figure E.1: Hardware Target Design Flow

files. Integration of C/C++ files allows certain functions to be computed in it to debug intermediate results. It can also be used to set up testbench files. However it should be noted that C/C++ files can be integrated only for the purpose of simulation and can not be used while synthesizing the project. In simulation outcome of the design is usually directed to the text files to compare with the expected output. Any discrepancies can be debugged and the necessary modification can be made in

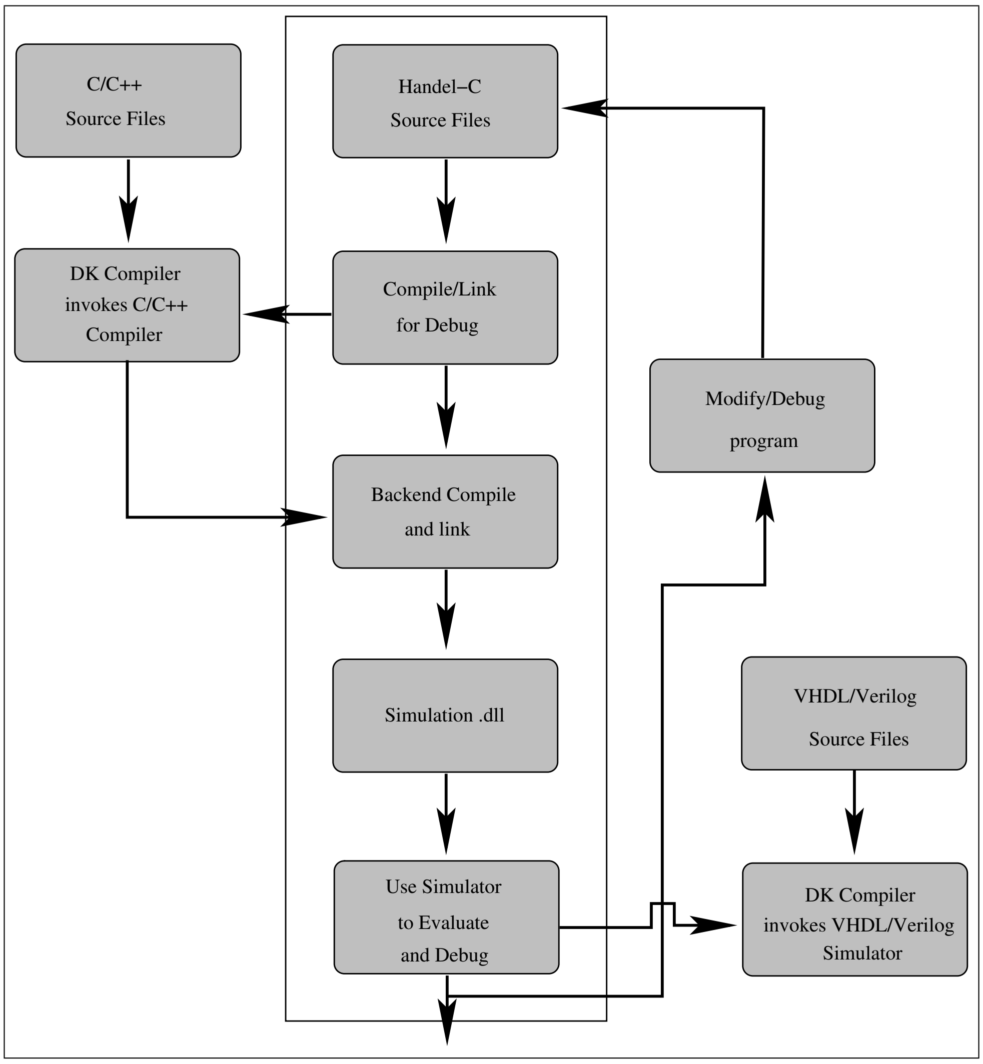

C/C++ Source Files
Handel−C Source Files
DK Compiler invokes C/C++ Compiler
Compile/Link for Debug
Modify/Debug program
Backend Compile and link
Simulation .dll
VHDL/Verilog Source Files
Use Simulator to Evaluate and Debug
DK Compiler invokes VHDL/Verilog Simulator

Figure E.2: Simulation Design Flow

source files. The Debugger also provides the facility of viewing various variables, setting breakpoints and sequencing through every clock cycle.

E.2 An overview of Synthesis Process

Synthesis process involves compiling and linking source files and generating a netlist. The netlist is in EDIF (Electronic Design Interchange Format) which creates the output file of .edf extension. In the following section a brief introduction of EDIF is given. Before generating netlist a necessary hardware interface is assigned to simulation files and all simulator input and output files are removed alongside C/C++ files. The hardware interface includes defining the clock pin and speed, the external RAM specifications and BlockRAMs. This can be accomplished in two ways as following:

- Using ready-made macros in Handel-C support software library supplied with DK compiler
- Defining manually in the code and assigning the pin numbers of the FPGA of the RC1000 board.

The generated netlist, .edf output file, is required to pass through mapping and placement and route(PAR) of a synthesis tool. The synthesis tool used in our experiment is Xilinx ISE 6.2 which finally generates a .bit file ready for programming FPGA chip. Simulation allows the integration of VHDL components as shown in the design flow. Synthesis can generate files in VHDL format also instead of EDIF as shown in the design flows. However, it should be noted here that in our experiments no VHDL/Verilog is used.

Appendix F

Clock Domains

A hardware architecture mapped on FPGA usually runs at one clock frequency. Handel-C allows designs to be associated with either a single clock domain or multiple clock domains. A Multiple (usually two) clock domain design contains two code segments running at different clock frequencies. The need of having two clock domains in a design arises when a section of code is slower than the other section of code because of certain hardware restrictions. For example, a code segment accessing a RAM with high response time can be slower than another code segment processing that data. In such a case Handel-C allows to isolate these two code segments and assign each a different clock frequency. This will help run certain code segments at a higher speed instead of having to reduce the speed of the whole code because of the slow code segments.

In Handel-C each clock domain is described by a separate main() function, and each main() function must be in a separate source code file. Communication between two clock domains in the same project is allowed through the use of channels and dual port BlockRAMs. Channels are used to ensure that data are transmitted and received accurately. Dual port BlockRAMs are used to transfer any data since sharing of variables, signals, interfaces and functions are not allowed between two clock domains. In Dual port BlockRAM, one port is used by one clock domain to write and the other port is used by the other clock domain to read data.

F.1 Two clock Domains in the implementations

We encoded all the hardware architectures, except SIPEX, in Handel-C to run either in single or double clock-domain. Figure F.1 shows the concepts behind the dual-clock domain designs.

An architecture based on single clock domain starts the execution of the BP algorithm by fetching initial parameters from Off-Chip RAMs and stores it to BlockRAMs. Next, it executes the algorithm on the FPGA and stores the resultant error back to the Off-Chip RAM which is read back by the host.

An architecture based on double clock domains isolate the communication with the Off-chip RAMs from the actual execution of the algorithm on the FPGA. The architecture starts the execution of the algorithm by fetching initial parameters and LT for the sigmoid function, if required, and writes it to BlockRAMs through one port in the first clock domain. In the second clock domain the architecture reads the data from BlockRAMs through the second port and executes the algorithm on the FPGA. The resultant Instantaneous sum squared error for every pattern is transferred back to the first clock domain via a BlockRAM and stored in Off-chip RAMs. Dual-port BlockRAMs serve as a channel between two clock-domains and

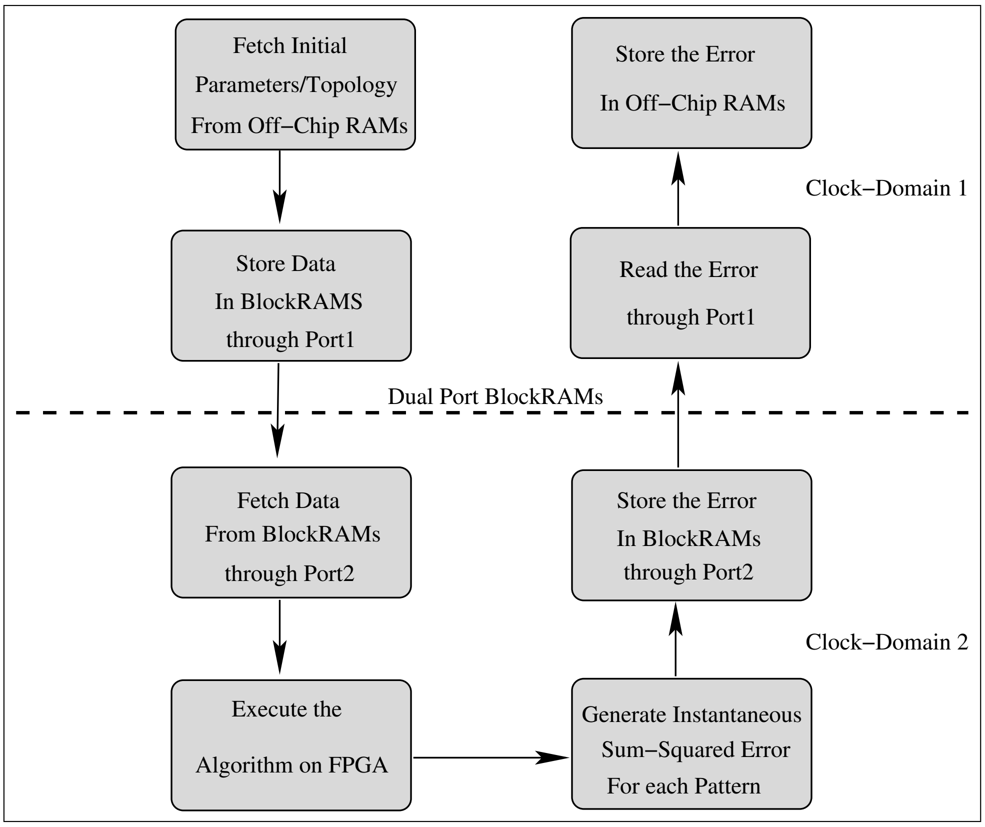


Figure F.1: Multiple Clock Domain Design

data can be transferred from one clock domain to the other clock-domain. The error is read back by the host. The following block diagram illustrates two clock domain designs with dual port BlockRAMs.

Bibliography

[Alip91] C. Alippi and G. Storti-Gajani, “Simple Approximation of sigmoidal functions: Realistic designs of digital neural networks capable of learning,” In *IEEE International Symposium on Circuits and Systems*, pp. 1505–1508, Singapore, 1991.

[Bast04] K. Basterretxea and J. Tarela, “Approximation of sigmoid function and the derivative for hardware implementation of artificial neurons,” In *IEE proceedings on Circuits, devices and Systems*, athens, February 2004.

[Beiu94] V. Beiu, J. Peperstraete, and J. Vandewalle, “Close approximation of sigmoid functions by sum of steps for VLSI implementation of Neural networks,” *The Scientific Annals, section: Informatics*, vol. 40, No. 1, , 1994.

[Beuc98] J. Beuchat, J. Haenni, and E.Sanchez, “Hardware Reconfigurable Neural Networks,” In *Parallel and Distributed Processing, IPPS/SPDP*, pp. 91–98, Springer-Verlag, 1998.

[Blak98] C. Blake and C. Merz, “UCI Repository of machine learning databases,” University of California, Irvine, Dept. of Information and Computer Sciences, 1998.

[Bond00] K. Bondalapati and V. Prasanna, “Reconfigurable Computing: Architectures, Models and Algorithms,” Technical Report, Department of Electrical Engineering, University of Southern California, Los Angeles, CA, USA, 2000.

[Bond01] K. Bondalpati, *Modelling and mapping for dynamically reconfigurable hybrid architecture* PhD thesis, Computer Engineering Department, University of South California, August 2001.

[Broo88] D. Broomhead and D. Lowe, “Multivariable functional interpolation and adaptive networks,” *Complex Systems*, vol. 2, pp. 321–355, 1988.

[Celo03] Celoxica, “Handel-C Language Reference Manual for DK 2.0,” 2003.

[Chan04] K. Chang and J. Liu, "Landslides Features Interpreted by Neural Network Method Using a High-Resolution Satellite Image," In *Geo-Imagery bridging continents, XXth ISPRS Congress, Commission 7*, pp. 574–579, Istanbul, Turky, 2004.

[Clou96] J. Cloutier, S. Pigeon, and F. Boyer, "VIP:An FPGA-based Processor for image processing and neural networks," In *5th international conference on Microelectronics for neural networks and fuzzy systems*, pp. 330–336, Switzerland, 1996.

[Comp00] K. Compton and S. Hauck, "An Introduction to Reconfigurable Computing," Technical Report, Northwestern University, Dept. of ECE, Evanston, IL USA, Department of Electrical and Computer Engineering Northwestern University, Evanston, IL USA, 2000.

[DAbr98] D.Abramson, K.Smith, P.Logothetis, and D. Duke, "FPGA based implementation of a Hopfield neural network for solving constraint satisfaction problem," In *Proceedings of 24th euromicro workshop on computational intelligence*, pp. 688–693, Sweden, August 1998.

[Deho00] A. Dehon, "The Density Advantage of Configurable Computing," *IEEE Computer*, vol. 33, No. 5, pp. 41–49, April 2000.

[Deho96] A. Dehon, "Reconfigurable architectures for general purpose computing," A.I technical report No. 1586, Massachusetts Institutes of Technology, October 1996.

[Depr84] E. Deprettere, P. Dewilde, and P. Udo, "Pipelined CORDIC architectures for fast VLSI filtering and array processing," In *International Conference on Accoustics, Speech and Signal processing*, pp. 41.A.6.1–41.A.6.4, 1984.

[Eldr94] J. G. Eldredge and B. L. Hutchings, "RRANN: A Hardware Implementation of the Backpropagation Algorithm Using Reconfigurable FPGAs," In *IEEE World Conference on Computational Intelligence*, Orlando, FL, 1994.

[Erce77] M. Ercegovac and K. Trivedi, "Online algorithms for division and multiplication," *IEEE transactions on Computers*, vol. C-26, No. 7, pp. 681–687, 1977.

[Ferr94] A. Ferrucci, *ACME: A Field-programmable Gate Array Implementation of a Self-adaptive and Scalable Connectionist Network* Master's thesis, University of California, Santa Cruz, 1994.

[Figu98] M. Figueiredo and C. Gloster, "Implementation of a probabilistic neural network for multispetral image classification on an FPGA based custom computing machine," In *5th Brazilian Symposium on Neural Networks*, Brazil, December 1998.

[Fish36] R. Fisher, "The use of multiple measurements in taxonomic problems," *Annual Eugenics*, vol. VII, No. II, pp. 179–188, 1936.

[Gira96] B. Girau and A. Tisserand, "OnLine Arithmetic-Based Reprogrammable Hardware Implementation of Multilayer Perceptron Back-Propagation," In *Proceedings of the Fifth International Conference on Microelectronics for Neural Networks and Fuzzy Systems*, pp. 168–175, IEEE Computer Society Press, 1996.

[Hayk99] S. Haykin, *Neural Networks : A comprehensive foundation*, Prentice-Hall, Englewood cliffs, New Jersey, 1999.

[Heem95] J. Heemskerk, *Overview of Neural Hardware. In: Neurocomputers for Brain-Style Processing. Design, Implementation and Application* PhD thesis, Unit of Experimental and Theoretical Psychology Leiden University, The Netherlands, 1995.

[Hert91] J. Hertz, A. Krogh, and R. Palmer, *Introduction to the theory of neural computation*, Perseus Books Group, 1991.

[hinT] http://www.warthman.com/projects-intel-ni1000 TS.htm, .

[Holt91] J. Holt and T. Baker, "Backpropagation simulations using limited precision calculations," In *In International joint conference on Neural Networks*, pp. 121–126, Seattle, WA, July 1991.

[http] http://hypertextbook.com/facts/2002/AniciaNdabahaliye2.shtml, .

[Hutc95] B. Hutchings and M. Writhlin, "Implementation approaches for reconfigurable logic applications," In *5th international workshop on FPGA*, Oxford, England, August 1995.

[Li04] X. Li and S. Areibi, "A Hardware/Software Co-design Approach for Face Recognition," In *16th International Conference on Microelectronics*, pp. 67–70, Tunis, Tunisia, December 2004.

[Lysa94] P. Lysaght, J. Stockwood, J. Law, and D. Girma, "Artificial Neural Network Implementation on a Fine-Grained FPGA," In R. Hartenstein and M. Z. Servit, editors, *Field-Programmable Logic: Architectures, Synthesis and Applications. 4th International Workshop on Field-Programmable Logic and Applications*, pp. 421–431, Springer-Verlag, Prague, Czech Republic, 1994.

[Mart02] P. Martin, "A Pipelined Hardware Implementation of Genetic Programming Using FPGAs and Handel-C," In *EuroGP '02: Proceedings of the 5th European Conference on Genetic Programming*, pp. 1–12, Springer-Verlag, London, UK, 2002.

[Mart94] M. Martin, *A reconfigurable hardware accelerator for back-propagation connectionist classifiers* Master's thesis, University of California, Santa Cruz, 1994.

[Molz00] R. Molz, P.Engel, F. Moraes, L. Torres, and M. Robert, "Codesign of fully parallel neural network for a classification problem," In *5th world multi-conference on systematics, cybernatics and informatics*, Orlando, Florida, 2000.

[Murr89] A. Murray, "Pulse Arithmetic In VLSI Neural Networks," *IEEE Micro Magazine*, vol. 9, No. 6, pp. 64–74, December 1989.

[Murt92] P. Murtagh and A. Tsoi, *Implementation issues of sigmoid function and its derivative for VLSI neural networks*, Volume 139, 3, May 1992.

[Myer89] D. Myers and R. Hutchison, "Efficient implementation of piecewise linear activation function for digital VLSI neural networks," *Electronic Letters*, vol. 25, pp. 1662–1663, 1989.

[Nich03] K. Nichols, *A Reconfigurable Computing architectures for Implementing Artifical Neural Networks on FPGA* Master's thesis, School of Engineering, University of Guelph, December 2003.

[Osso96] H. Ossoinig, E. Reisinger, C. Steger, and R. Weiss, "Design and FPGA implmentation of a neural network," In *Proc. 7th Int. Conf. on Signal Processing Applications and Technology*, pp. 939–943, Orlando, Florida, 1996.

[Poor02] M. Poormann, U. Witkowski, H. Kalte, and U. Ruckert, "Implementation of ANN on a reconfigurable hardware accelerator," In *Euromicro workshop on parallel, distributed and network based processing*, pp. 243–250, Spain, January 2002.

[PU96] A. Prez-Uribe and E. Sanchez, "FPGA Implementation of an Adaptable-Size Neural Network," In *VI international conference on ANN ICANN' 96*, pp. 383–388, Bochum, Germany, July 1996.

[Rume86] D. Rumelhart, J. McClelland, and PDP Research Group, *Parallel Distrubuted Processing: Explorations in the Microstructure of Cognition*, Volume 1: Foundations, MIT Press, Cambridge, Massachusetts, 1986.

[Sahi00] I. Sahin, C. Gloster, and C. Doss, "Feasibility of floating point arithmetic in reconfigurable computing systems," In *MAPLD International Conference*, DC, USA, 2000.

[Samm91] K. Sammut and S. Jones, "Implementing non-linear activation functions in neural network emulators," *Electronic Letters*, vol. 27, No. 12, , June 1991.

[Savi04] A. Savich, "Technical Report: Sigmoid Function Linearization," Technical Report, University of Guelph, Guelph, Canada, 2004.

[Scho98] T. Schonauer, A. Jahnke, U. Roth, and H. Klar, "Digital Neurohardware: Principles and Perspectives," In *Proceedings of Neuronale Netze in der Anwendung NN'98*, pp. 101–106, Magdeburg, Germany, 1998.

[SCoe04] S.Coe, *A Memetic Algorithm Implementation on an FPGA for VLSI Circuit Partitioning* Master's thesis, School of Engineering, University of Guelph, August 2004.

[Sima93] P. Simard and H. Graf, "Backpropagation without Multiplication," In *Advances in Neural Information Processing Systems*, pp. 232–239, Denver, USA, 1993.

[Skrb99] M. Skrbek, "Fast Neural Network Implementation," *Neural Network World*, Elsevier, vol. 9, No. 5, pp. 375–391, 1999.

[Supp01] Celoxica Customer Support, "RC1000 Hardware Reference Manual," 2001.

[VP05] S. Areibi V. Pandya and M. Moussa, "A Handel-C Implementation of the Back Propagation Algorithm on Field Programmable Gate Arrays," In *International Conference on Reconfigurable Computing and FPGAs, ReConFig'05*, pp. —, IEEE, Puebla, Mexico, Sep 2005.

[Whit92] B. White and M. Elmasry, "A digital Neocognition neural network model for VLSI," *IEEE transactions on Neural Networks*, pp. 73–85, 1992.

[Xili93] Xilinx, "Xilinx: The Programmable Gate Array Data Book," 1993.

[Zhan96] M. Zhang, S. Vassilliadis, and J. Fris, "Sigmoid generators for neural computing using piece-wise approximation," *IEEE transactions on Computers*, vol. 45 No. 9, , September 1996.