The Impact of Arithmetic Representation on Implementing MLP-BP on FPGAs: A Study

Antony W. Savich, Medhat Moussa, *Member, IEEE*, and Shawki Areibi, *Member, IEEE*

***Abstract*—In this paper, arithmetic representations for implementing multilayer perceptrons trained using the error backpropagation algorithm (MLP-BP) neural networks on field-programmable gate arrays (FPGAs) are examined in detail. Both floating-point (FLP) and fixed-point (FXP) formats are studied and the effect of precision of representation and FPGA area requirements are considered. A generic very high-speed integrated circuit hardware description language (VHDL) program was developed to help experiment with a large number of formats and designs. The results show that an MLP-BP network uses less clock cycles and consumes less real estate when compiled in an FXP format, compared with a larger and slower functioning compilation in an FLP format with similar data representation width, in bits, or a similar precision and range.**

***Index Terms*—Artificial neural networks (AANs), fixed-point (FXP) arithmetic, floating-point (FLP) arithmetic.**

I. Introduction

ARTIFICIAL neural networks (ANNs) are currently used in a wide variety of applications either for function approximation and data fitting or classification and pattern recognition. One of the most common architectures consists of multilayer perceptrons trained using the error backpropagation algorithm (MLP-BP) [1]. One major problem in using an MLP-BP network is the lack of clear methodology in setting up the network initial topology and parameters, especially for large networks. The network topology has a significant impact on the network's computational ability to learn the target function and to generalize from training data to unseen data. If the network has too few free parameters (weights), training could fail to achieve the required error threshold. On the other hand, if the network has too many free parameters, then a large data set is needed. In this case, the probability of the overfit is higher, which jeopardizes generalization as well [2].

To solve this problem, heuristics are typically used to obtain a rough estimate of the learning space complexity and the number of free parameters needed. Once training starts, small variations in topology are possible with minute adjustments to the network's properties. It is typically not possible to experiment with a wide variety of network topologies to determine various effects of the changes on network performance, at least not in a timely manner, because of the long training sessions required. Furthermore, this approach is possible for offline design process, for applications where training data is static, or where conditions initially determined will stay the same for the duration of network's useful function. However, when online training is necessary or when the solution space is dynamic and new data is being added continuously, there is a critical need for testing wide range of topologies in a limited time period.

Since ANNs are inherently parallel architectures, there have been several earlier attempts to build custom application-specific integrated circuits (ASICs) that include multiple parallel processing units [3]–[5]. However, intrinsic to the ASIC design principles, the resulting networks were constrained by size and type of algorithm implemented. More recently, the focus of ANN hardware design shifted toward implementation on reconfigurable hardware. This allows for more flexibility of network size, type, topology, and other constraints while maintaining increased processing density by taking advantage of the natural parallel structure of neural networks [6]–[9]. Currently, field-programmable gate arrays (FPGAs) are the preferred reconfigurable hardware platform. Current FPGAs provide performance and logic density similar to ASIC but with the flexibility of quick design/test cycles. Thus, they are superior in research, and, often, industrial applications.

However, current FPGAs are not limitless in their resources when implementing an ANN, specifically MLP-BP, and the design poses a number of challenges [10]. One is to determine the most efficient arithmetic representation format. While most general computing microprocessors/software implementations currently implement single (32 bit) and double (64 bit) IEEE floating-point (FLP) formats (IEEE 754-1985 [11]–[13]) using these formats on FPGAs requires significant resources. On the other hand, using shorter integer, FLP or fixed-point (FXP) formats, which consume less FPGA area to process, often means loss of precision [14]. Depending on the data format chosen, efficient implementation of MLP-BP on FPGAs can result in completely different outputs than those of a similar architecture implemented in software using the IEEE FLP formats. This is called the *area versus precision* design tradeoff. The tradeoff is to choose, in a data format, the correct balance between the precision required to carry on network functionality, and the size and cost of the FPGA resources consumed.

A. Contributions and Organization

This paper investigates in detail the *area versus precision* design tradeoff for implementing MLP-BP on FPGAs. Several FXP and FLP arithmetic formats are tested using a generic very high speed integrated circuit hardware description language

Manuscript received September 12, 2005; revised March 6, 2006. This work was supported by the Natural Science and Engineering Research Council of Canada (NSERC).

The authors are with the School of Engineering, University of Guelph, Guelph, ON N1G 2W1, Canada (e-mail: mmoussa@uoguelph.ca).

Digital Object Identifier 10.1109/TNN.2006.883002

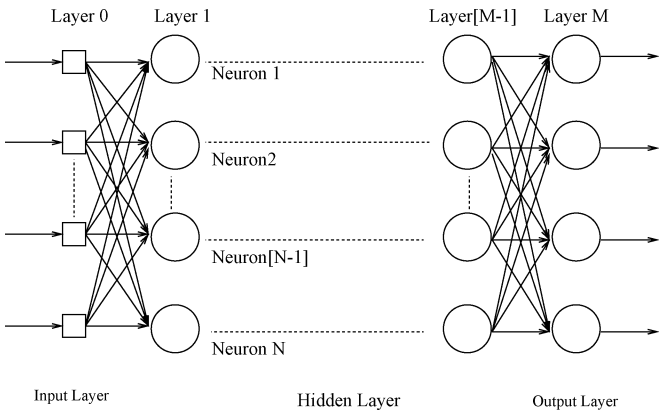


Fig. 1. MLP architecture.

(VHDL) implementation of an MLP-BP network. The subcomponents involved in implementing each of the formats, and the impact the width and format of the data representation have on the overall consumed resources by the network are discussed. This paper also discusses the effect of numeric precision on network convergence and training time performance. The remainder of this paper is organized as follows. Section II reviews the background material related to various arithmetic formats and multilayer perceptron (MLP) artificial neural networks. Section III presents previous work in approaching the precision versus area tradeoff. The requirements and methodology of the design is discussed in Section IV. This is followed by Section V, where the structure, functionality, and rationale of the design are discussed in detail. Section VI describes the experimental setup and Section VII presents the results and analysis. Finally, conclusions are presented in Section VIII.

II. Background

Before proceeding further, it is helpful to briefly review the MLP-BP algorithm as well as the FXP and FLP arithmetic formats. This will provide the background foundation for later sections.

A. Error Backpropagation Algorithm

An MLP network contains neurons structured in parallel layers, from inputs to outputs, as illustrated in Fig. 1. The layers are numbered 0 to M and the neurons are numbered 1 to N. When used in conjunction with the error backpropagation (BP) algorithm, each neuron contains two key arithmetic functions which perform forward and backward computations. The forward computation step uses internal weights associated with each neuron for calculating the neuron's output. The backward computation step compares the network's overall output to a target, computes an error gradient, and propagates the error through layers by adjusting each neuron's weights to correct for it.

In general, an MLP network can be composed of any number of layers as seen in Fig. 1. However, when reviewing the mathematical properties of the relationship of outputs from inputs, a network with M number of layers can be fit into a network with two layers, namely one hidden and one output layer, without loss of these properties. This comes about from the fact that each output of the network must be dependant on a nonlinear combination of any or all inputs. While a network with many hidden layers can learn complex nonlinear input–output relationships efficiently (with few neurons in each layer), the same relationships can be learned with a two-layer topology, though using more neurons in each layer. In this paper, the VHDL network constructed uses this property, and is designed to be a two-layer structure. The standard MLP-BP algorithm is described as follows.

1) Forward Computation: The computation performed by each neuron in layer s during this stage is as follows:

$$o_k^{(s)} = f\left(H_k^{(s)}\right) = f\left(\sum_{j=1}^{N^{(s-1)}} w_{kj}^{(s)} o_j^{(s-1)} + w_{ko}^{(s)}\right) \quad (1)$$

where network layers are $s = 1, \ldots, M$, and for each kth neuron in the sth layer the following holds:

M	total number of layers;
$N^{(s-1)}$	number of neurons in layer $(s-1)$;
$o_k^{(s)}$	*output* of the current neuron;
f	*activation function* computed on $H_k^{(s)}$;
$H_k^{(s)}$	*weighted input sum*;
$o_j^{(s-1)}$	*output* of the jth neuron in the $(s-1)$th layer;
$w_{kj}^{(s)}$	*synaptic weight* contained in the current neuron k, associated with output of neuron j of layer $(s-1)$;
$w_{ko}^{(s)}$	current neuron's *bias weight*.

2) Backward Computation: At this stage, the weights and biases are updated according to the error gradient-descent vector. After an input vector is applied during the forward computation stage, a network output vector is obtained (the set of layer M neurons' outputs). A target vector t is provided to the network, compared with the network's output, and the error is minimized to drive the network's output toward the expected target value. The following steps are performed.

1) Starting with the output layer, and moving back toward the input layer, calculate the error terms and local gradients as follows:

$$\varepsilon_k^{(s)} = \begin{cases} t_k - o_k^{(s)}, & s = M \\ \sum_{j=1}^{N^{(s+1)}} w_{jk}^{(s+1)} \delta_j^{(s+1)}, & s = 1, \ldots, M-1 \end{cases} \quad (2)$$

where $\varepsilon_k^{(s)}$ is the *error term* for the kth neuron in the sth layer (for output layer M, this is simply the difference between target and actual output); $w_{jk}^{(s+1)}$ is the *synaptic weight* of neuron j in the $(s+1)$ layer, associated with the output of current neuron k; t_k is the *target value* from the provided target vector t, associated with neuron k of the output layer; and $\delta_j^{(s+1)}$ is the *local gradient* for the jth neuron in the $(s+1)$th layer, defined as follows:

$$\delta_k^{(s+1)} = \varepsilon_k^{(s+1)} f'\left(H_k^{(s+1)}\right), \qquad s = 1, \ldots, M \quad (3)$$

where $\varepsilon_k^{(s+1)}$ is the error term of neuron k in layer $(s+1)$; and $f'(H_k^{(s+1)})$ is the derivative of the activation function,

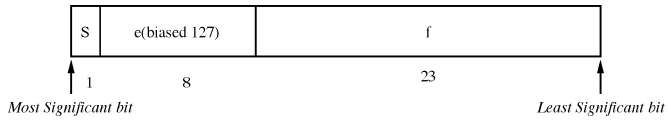


Fig. 2. IEEE standard 754-1985 format.

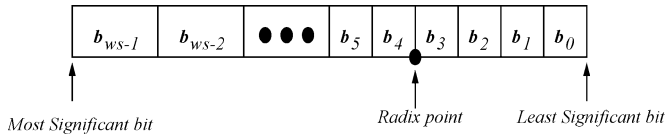


Fig. 3. Format of an FXP number.

which is calculated on the weighted sum $H_k^{(s+1)}$, described by (1).

2) Calculate the changes for all the weights as follows:

$$\Delta w_{kj}^{(s)} = \eta \delta_k^{(s)} o_j^{(s-1)}, \qquad k = 1, \ldots, N^{(s)}, \quad j = 1, \ldots, N^{(s-1)}$$
$$\Delta w_{ko}^{(s)} = \eta \delta_k^{(s)} \tag{4}$$

where η is the learning rate.

3) Update all the weights as follows:

$$w_{kj}^{(s)}(n+1) = w_{kj}^{(s)}(n) + \Delta w_{kj}^{(s)}(n) \tag{5}$$

where $k = 1, \ldots, N^{(s)}$ and $j = 0, \ldots, N^{(s-1)}$; $w_{kj}^{(s)}(n)$ is the current synaptic weight; and $w_{kj}^{(s)}(n+1)$ is the updated synaptic weight to be used in the next feedforward iteration.

B. FLP and FXP Formats

1) FLP Format: In general, an FLP number is represented as $\pm d.dd\ldots d \times \beta^e$. More precisely, $\pm d_0.d_1 d_2 \ldots d_{p-1} \times \beta^e$ represents the number

$$\pm \left(d_0 + d_1 \beta^{-1} + \ldots + d_{p-1} \beta^{-(p-1)} \right) \beta^e, \qquad (0 \le d_i < \beta) \tag{6}$$

where β represents the *base* (which is always assumed to be even), e represents the *exponent*, and p is the precision expressed as number of significant digits or bits for $\beta = 2$. The exponent is said to be *biased* when the actual value of the exponent is

$$e = k - (\beta^{m-1} - 1) \tag{7}$$

where k is the value of the exponent bits interpreted as an unsigned integer and m is the number of bits in the exponent. The number is said to be normalized when $d_0 = 1$ in (6).

One of the most common FLP formats is the single precision IEEE 754-1985 format detailed in [11]. The bit representation for this format is given in Fig. 2 and its corresponding value is given in (8). Implications of implementing FLP arithmetic on FPGAs are discussed further in [15]

$$v = -1^s (1.f) 2^e$$
$$\text{for} \quad \beta = 2 \quad p = 24 \quad m = 8 \quad e = k - 127. \tag{8}$$

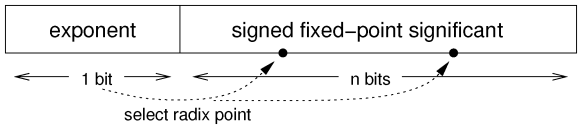


Fig. 4. Format of a dual FXP number.

2) FXP Format: FXP format is illustrated in Fig. 3. There are two parts in an FXP number. The first is the integer part which is b_{ws-1} to b_4, and the second is the fractional part which is b_3 to b_0, as illustrated in Fig. 3. If the base of this FXP number is β and it is a positive number, the decimal equivalent value can be calculated by

$$b_{ws-1}\beta^{ws-5} + \ldots + b_4 + b_3\beta^{-1} + b_2\beta^{-2} + b_1\beta^{-3} + b_0\beta^{-4}. \tag{9}$$

If the base of the FXP number is 2, then the value is determined by the type of representation used (generally 2's complement is used).

A special note must be given to a recently proposed "dual FXP" representation [16], [17]. This format is encoded similarly to an FXP number with one additional "exponent" bit representing the position of the radix point. With two settings preselected during system design, this gives two possible ranges and precisions the number can represent, selected by the exponent bit, as seen in Fig. 4. We will focus on the simple FXP representation leaving analysis of MLP-BP network performance using dual FXP format for future work.

III. Analysis of the Previous Approaches

A careful analysis of the MLP-BP network operation as summarized by (1)–(5) shows the following.

- Weight values start around zero and increase or decrease as learning progresses. Large weights, however, lead to network "saturation" where only a small subset of the network's weights are changed as illustrated by (3). Large weights also lead to poor generalization after training.
- Transfer functions (also called squashing functions) must be bounded to avoid large convergence errors. For example, the log-sigmoid function (10) maps any number >3 to a number ≈ 1 and maps any number < -3 to ≈ 0.
- As learning progresses, the error between the target and output values becomes smaller (2). Once the error is very small or below the stopping criteria, learning stops. Thus, it is critical that the arithmetic format precision allows for very small error numbers to be represented.

From the previous discussion, one can conclude that the ideal numeric format for MLP-BP networks should have a high precision. Range can be limited as long as inputs and outputs are normalized since weights should be limited for learning progress. However, as shown in [18], the occupied area of an FPGA-based MLP-BP is proportional to the multiplier used. We are interested in finding the *minimum allowable precision* and *minimum allowable range* which minimize hardware area usage while not affecting ANN convergence or performance.

For MLP-BP algorithm, Holt and Baker [14] showed using simulations and theoretical analysis that *16-bit FXP* (1-bit sign, 3 bit left and 12 bit right of the radix point) was the minimum allowable range/precision assuming that both inputs and outputs were normalized between [0,1] and a log-sigmoid transfer function was used. Ligon III *et al.* [19] have also shown the density advantage of FXP over FLP for older generation Xilinx 4020E FPGAs. They show that the space/time requirements for 32-bit FXP adders and multipliers are less than those of their 32-bit FLP counterparts.

The goal of this paper is, therefore, directed to providing a comprehensive study of the impact of various precision FLP and FXP formats on hardware resources used by an entire MLP-BP network, while taking into consideration adequate convergence performance and current FPGAs architectures (e.g., Xilinx Virtex-II has embedded multipliers [20]). An MLP-BP network is implemented in VHDL for this purpose. The bit widths of fields in the number representation, and the number representation itself can be freely specified. For number representation, a 2's complement FXP and generic FLP parameterizable libraries are used.

IV. Overall VHDL Design

In designing a VHDL implementation of a MLP-BP network, there are four primary requirements:

- generic topology;
- generic precision and range;
- generic numeric[1] representation;
- easy selection of training or purely testing configuration.

The first design requirement is set to allow for a flexible network topology to ease adjustment of the number of neurons in each layer without complex changes in the VHDL code. This problem has a two-part solution: custom typed arrays for data routing between layers with varying number of neurons, and concurrent instantiation of neuron elements in a hierarchical design structure. Three constant values are defined in a header structure—the number of network inputs, hidden neurons, and outputs. These three values fully define the topology, which in turn is sufficient to instantiate the proper routing and arithmetic structures. This means that the proper number of routing buses and arithmetic units are instantiated, while the actual width of these structures is left to the custom data types created for this purpose when implementing the next goal.

The second design requirement dictates the flexible bit widths of all data being processed in the network. This is accomplished using generic data typing and parameterized mathematical operators. The arithmetic definitions in the header specify the bit width to be used by all mathematical operators which ensure data integrity and intermodule connectivity. In order to handle the variable bit widths, parameterized math units are required. These units, for both FXP and FLP arithmetic, are designed to handle a wide input width range specified by VHDL generic statements.

The third design requirement dictates that the network code is easily switched between FXP and FLP representations. This problem is solved by utilizing wrapper files. A wrapper is created for addition, subtraction, multiplication, and the log-sigmoid function. Each wrapper contains an instantiation of both the FXP and FLP version of its operations. Each operator gets synthesized in the design depending on the VHDL constant set in the header file. Other hierarchically higher units which use a "+," "−," "∗," or "/" arithmetic operations simply reference the wrapper with disregard to its configuration. The wrappers in turn provide a consistent use of arithmetic processing modules for either of the two formats implemented throughout the design. Additionally, the use of wrappers allows for alternate implementations of the same operation. For example, the log-sigmoid wrapper allows for the lookup table or linear approximation implementations in FXP format. This feature greatly enhances the flexibility of the entire architecture and helps ease future development.

The final design requirement is to allow an easy switch between training and testing modes of synthesis (the testing mode does not need the additional BP circuitry required for training). This problem is solved by arranging the design into an architecture separated at top level by the network's two stages (Sections II-A1 and II-A2) as explained in Section V.

V. Detailed Design and Implementation

This section reviews the various stage modules needed for the calculation purposes of the MLP-BP network as well as the control structure which handles input and communication timing between stages.

A. General Structure

Fig. 5 shows the general layout and interconnections of data and control in the network. The layout consists of four major units: the forward stage, BP stage, weight update stage, and the controller.

1) Forward Stage: The forward stage module consists of neurons for both hidden and output layers.[2] Each neuron consists of one or many multipliers in an array, accumulator, and a swappable transfer function unit. This is taken directly from (1).

Currently, generic transfer function units for a five-piece linear approximation of (10) are implemented for both FXP and FLP formats. In addition, two lookup table units for the FXP and FLP formats with fixed internal precision, but parameterizable input–output widths, are implemented as well.[3]

2) BP Stage: The BP stage module implements (2) and (3) to calculate the error between the final outputs of the forward stage and the target value provided to the neural network. Following the error calculation, a delta value is calculated for each of the output neurons. Next, a delta value is calculated for each of the hidden neurons based on the output deltas and associated weights in the output layer.

3) Update Stage: The update stage module implements (4) and (5) and adjusts the network's weights accordingly. It also contains the registered arrays of weights which are routed to the forward stage as needed. The adjustment of hidden weights is

[1]FXP and FLP arithmetic has been implemented.

[2]In Fig. 5, the outputs of all neurons are marked OUTPUTS, and the first derivatives of those outputs are marked OUTPUTS′.

[3]These units trim or expand the stored samples appropriately to the number format (FXP or FLP), thus producing a variable width output for an arbitrary width input.

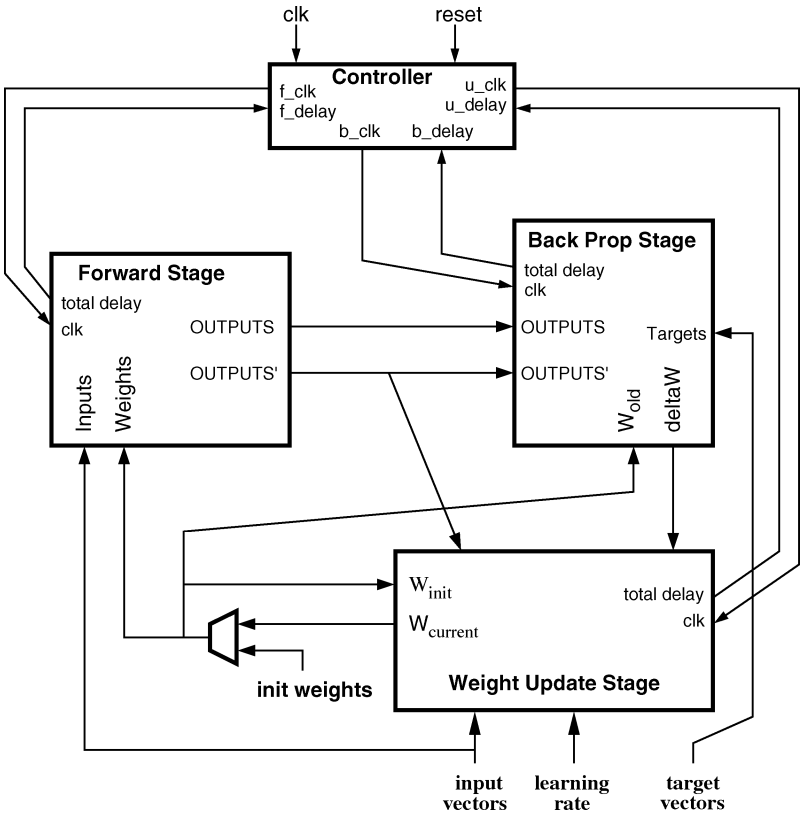


Fig. 5. Neural network data and control flow.

calculated based on network inputs, hidden delta, and learning rate. The adjustment for output weights is calculated using hidden neuron outputs, output neurons' deltas, and learning rate. The adjustment value is added to the existing weight to produce a new weight for the next cycle of the forward stage.

4) Controller Unit: The controller module is used to coordinate data routing and timing during operation of the three previous stages. The controller has a signal for each of the three stages. The rising edge of each signal corresponds with the end of processing for the corresponding stage. The timing of the dedicated signals is determined by the processing time of each stage. Every submodule of the three stages provides the number of cycles needed for data to propagate through that submodule. During the synthesis stage, these timings are combined to generate the totals necessary for the controller to dictate the proper function. Thus, the swappable modular architecture is seamlessly integrated into control timing.

B. Basic Neuron Implementation

The most basic element of the neural network, the neuron, transforms its inputs in the feedforward stage of operation according to the steps in (1) as shown in Fig. 6.

1) Multiply each input by its corresponding weight.[4]
2) Sum the multiplication results.
3) "Squash" the sum using a transfer function.

[4]Weights are generated in registers internal to the FPGA and are a part of the weight update module (Fig. 5). The inputs are provided from external sources (i.e., external memory).

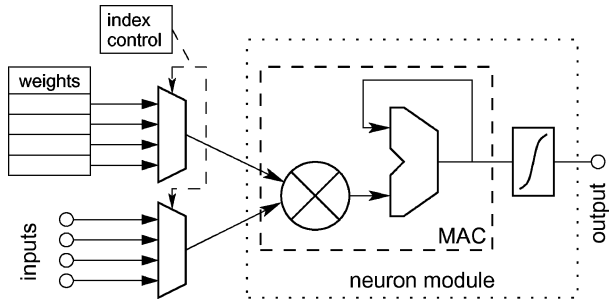


Fig. 6. Neuron structure—serial processing.

However, the feedforward processing of each neuron can be executed in several different ways. This design allows for three different processing techniques: serial, partial parallel, and full parallel. In our paper, only serial and partial parallel processing are implemented. Extension to include implementation of full parallel processing would be left for future work.

The modular design of this VHDL network allows for simple switching between the different processing techniques. The following examples will demonstrate the techniques using implementation of a four-input neuron.

1) Serial Processing: Referring to Fig. 6, serial processing uses a multiply-accumulator (MAC) unit which accepts a series of input pairs. The MAC unit consists of a multiplier and an adder. Each input pair is multiplied together and a running total is recorded. An index control module controls the multiplexing order. Once all input pairs have been processed, the final sum is passed through the transfer function to produce the neuron's output.

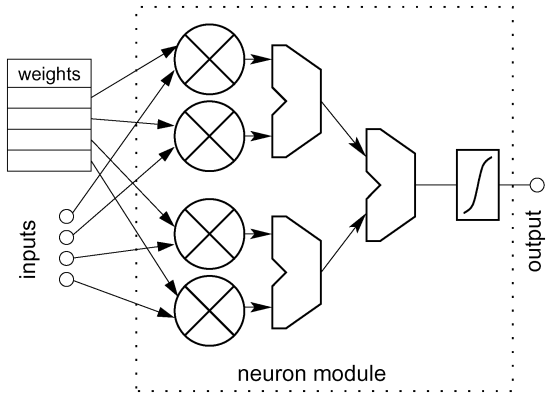


Fig. 7. Neuron structure—partial parallel processing.

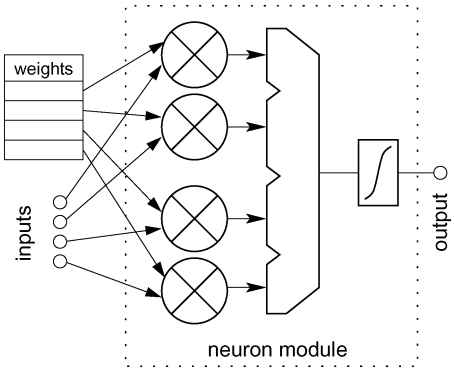


Fig. 8. Neuron structure—full parallel processing.

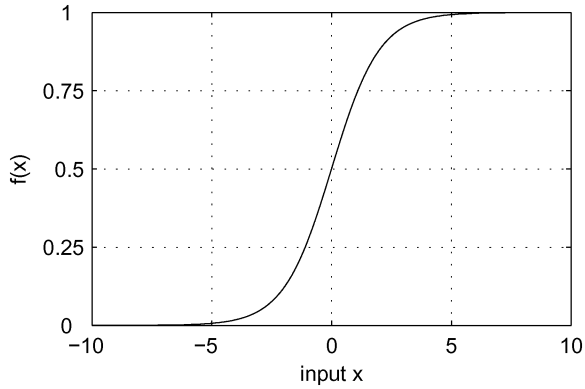


Fig. 9. Logsig function.

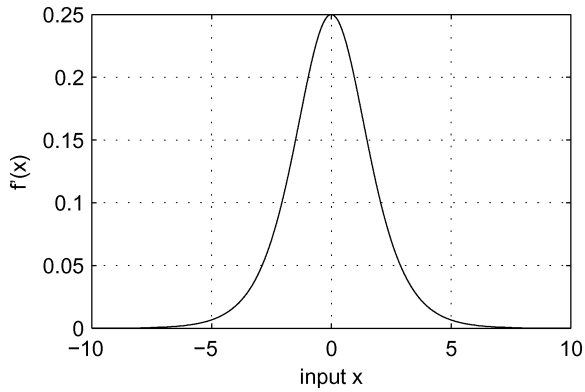


Fig. 10. First derivative of the logsig function.

The main advantage of serial processing is the small constant area required, regardless of topology, to implement one MAC and some routing for one input and one weight contained in the weight update stage. The obvious disadvantage is the processing speed. If the network involves a large number of inputs, serial processing will suffer from slow processing.

2) Partial Parallel Processing: Partial parallel processing uses a number of multipliers followed by a cascade of adders. There is an array of multipliers, one for each input/weight pair. The results of the first two multipliers are added together using an adder wrapper and then added to the result of the third multiplier, and so on. This process continues until all multiplication results are accumulated. The final sum is then passed to the transfer function unit, as demonstrated by Fig. 7.

The advantage of the partial parallel processing is faster processing time. The disadvantage is the larger area requirements compared to serial processing, since each additional neuron input requires a dedicated multiplier and an adder in the adder tree.

3) Full Parallel Processing: Full parallel processing is very similar to the partial parallel technique (Fig. 8). The difference is that in addition to the parallel array of multipliers, the adders are also fully parallelized as one multi-input parallel adder module performing the calculation in one step. This technique is the fastest, particularly for larger networks, without using more area than the partial parallel technique. The parallel adder unit implementation is left for future work.

C. Sigmoid Transfer Function Implementation

The last block in Figs. 6–8 is a transfer function, labeled as $\boxed{\int}$. Its functionality is to *squash* the neuron's output to a small range—(0, 1) for our transfer function of (10). This is a key functionality for the MLP-BP network.

One popular function is the *log-sigmoid transfer function* (further referred to as *logsig*), described by (10), with its first derivative given by (11)

$$f(x)_{\text{logsig}} = \frac{1}{1+\exp(-x)} \tag{10}$$

$$f'(x)_{\text{logsig}} = f(x)\cdot(1-f(x))\,. \tag{11}$$

This function limits the output in the (0, 1) range. The nonlinear squashing property of the logsig allows the linear segment to adequately correct errors with weight modifications as the output rises and falls; alternatively, the squashed regions *cap off* neurons which begin to dominate the network or carry no significance on the network output. Fig. 9 shows the logsig function while Fig. 10 shows its derivative. The first derivative $f'(x)_{\text{logsig}}$ is used in the BP algorithm as outlined in (3). The derivative is easily calculated at the BP stage from $f(x)_{\text{logsig}}$, (11), provided by the forward stage calculations.

Typical software implementations of the logsig function consist of direct calculation of (10). In general, purpose processors, where x^y or $\exp(x)$ functions are internally built into the hardware [21], or an operating system emulates such calculations using some predefined algorithm (e.g., using Taylor series), no

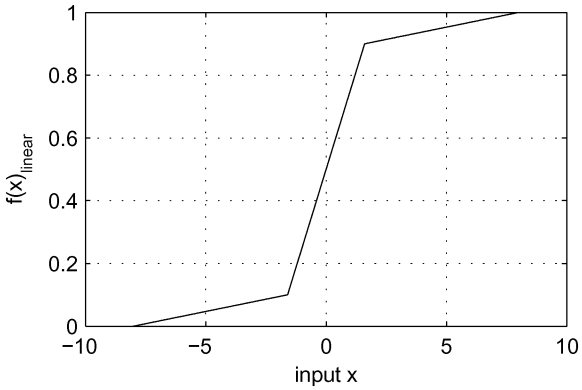


Fig. 11. Implementation of linear logsig approximation function.

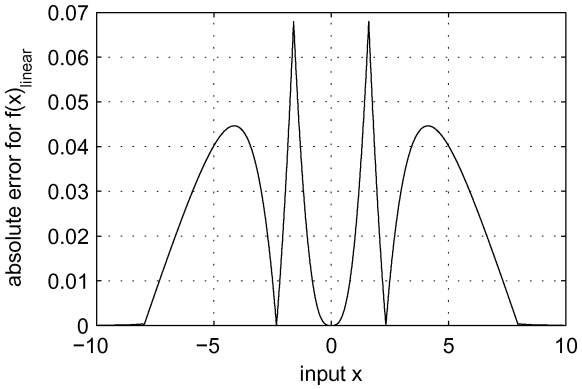


Fig. 12. Absolute error between logsig and its linear approximation.

additional area resources or external processing units need to be dedicated.

In hardware implementations of neural networks, especially those targeting FPGA architectures, the following issues must be taken into consideration.

- Arithmetic modules of x^y or $\exp(x)$ type are not built into FPGAs, and are difficult to synthesize to match the performance of parallel neural networks and the underlying simple arithmetic units. Typical efficient designs are also not suitable for the highly generic VHDL network considered in this paper.
- A built-in specialized divider circuit is not present in any current FPGA architecture. Some current FPGA architectures carry specialized adder and multiplier circuits [20], which are mapped to directly by the synthesizer using IEEE VHDL library's "+," "−," and "∗" functions. The "/" function is currently not supported by Xilinx or Altera synthesis tools. VHDL implementations of divider circuits do exist, but they are far from being area and speed efficient to be successfully incorporated into this design.

Therefore, the more common way of implementing the logsig function in current FPGA-based neural networks is to have a *lookup table module* (LTM) [10].

Usually, the memory mapping of inputs to outputs in an LTM is not 1 : 1 but many : 1. There are several reasons for this reduction. Some least significant bits of input are ignored; as well, some size is saved by simple optimization of internal contents of the LTM itself. Since the output of the function is constrained between (0, 1), neither the sign, nor the integer bits in an FXP implementation need to be saved. The logsig function is also odd, which means that it is symmetrical, and an expression can be used to transform its negative domain into the positive domain with little cost, saving half the size of the LTM. This also saves another bit of storage required to represent 0.5, since the values can be stored now for the (0, 0.5) range, with the (0.5, 1) range being calculated. The resolution can typically be further decreased by using a less precise definition of each value in the lookup table, and capping the output to one or zero when the input is outside a certain domain (when the function sufficiently approaches one or zero). The lookup table design does not lend itself well to optimizations under the FLP format, thus large table size results.

Similarly, the option of logsig function calculation on-chip via direct e^x and x/y module implementation is not attractive either. Both these approaches, when compared to the size of other components in a neuron (see [18]), bump up the size of each neuron considerably. Note that information regarding the size of e^x modules on FPGA is unavailable since most existing implementations in literature are neither synthesized by the authors nor available to be synthesized on an FPGA for comparison. These solutions strictly deal with very large scale integration (VLSI) implementations.

In this paper, a different approach for implementing the logsig function is adopted. It consists of a linear approximation of (10). The formal mathematical representation of this approximation is expressed by (12)

$$f(x) = \begin{cases} 0, & \text{(region 1) if} \quad x \leq -8 \\ \frac{8-|x|}{64}, & \text{(region 2) if} \quad -8 < x \leq -1.6 \\ \frac{x}{4} + 0.5, & \text{(region 3) if} \quad |x| < 1.6 \\ 1 - \frac{8-|x|}{64}, & \text{(region 4) if} \quad 1.6 \leq x < 8 \\ 1, & \text{(region 5) if} \quad x > 8. \end{cases} \tag{12}$$

Fig. 11 shows the graph of this linear approximation function. Fig. 12 shows that maximum absolute error (difference) between (12) and the original logsig (10) is 6.79% with a mean of 2.63% over the [−8, 8] range.

It must be noted that the actual derivative of this linear approximation from (12) is a stepwise function. It should not be mistakenly used in BP algorithm calculations from (3). Equation (11) should be used as the derivative function. The derivative should have an appropriate sloping curvature for the BP gradient-descent error calculations to produce convergence. Convergence could not be obtained in training experiments using a set of linear approximations (up to 11 pieces) of the derivative function in Fig. 10. Implementing linear approximations with higher piece resolution is unjustifiably expensive in terms of FPGA resources consumed versus implementing (11) directly using a multiplier.

D. FXP Format Implementation

The arithmetic modules which implement arithmetic functions in the FXP format are put together into a parameterized generic library. Both the FXP and FLP modules can be used in the architecture with ease via wrapper files written to implement either one by a simple argument selection. However, this FXP

TABLE I
RESOURCE COST OF VARIOUS SIGMOID IMPLEMENTATIONS

Module:	Divider	Lookup table	Logsig	Logsig
Type:	FXP	FXP	FXP	FLP
Size (LUTs):	~1200	3182	53	640

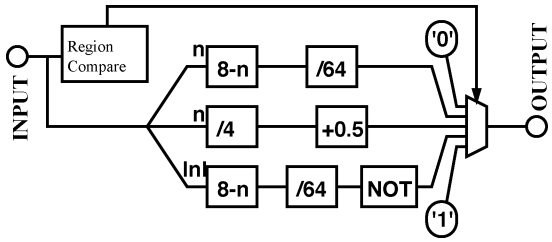


Fig. 13. Structural diagram of the logsig linear approximation module.

discussion is separated into two parts: the libraries and logsig function, since the FXP logsig function uses the highly optimized linear approximation and does not reference any of the FXP library's arithmetic modules.

1) Math Libraries: The FXP library used in this paper consists of VHDL generics to control the use of existing *ieee.std* arithmetic functions—e.g., the input widths, as well as significant bit select of the outputs for the "+," "−," and "∗" operations. These generic parameters are taken from the top level VHDL header file and through wrappers are propagated to each module to maintain compatibility of input and output formats.

2) Sigmoid: The FXP sigmoid function, (12) approximation, utilizes a variety of bit manipulation techniques. This achieves its small size (Table I) as well as extremely high performance. Fig. 13 shows its basic operating principle.

All the subblocks of this design are implemented through bit manipulation of FXP numbers. The $\boxed{/x}$ blocks are implemented as constant bit shifts, $\boxed{8-\mathrm{n}}$ is a inverter module,[5] $\boxed{|\mathrm{n}|}$ is incorporated into the subtraction, and the $\boxed{+0.5}$ is a plain adder limited to fractional bits of n. Both sign-magnitude and 2's complement format are applicable through a generic parameter. This specifies whether or not a one-least-significant-bit correction is performed during bit manipulation at appropriate locations.

As the $f'_{\mathrm{logsig}}(x)$ is needed as an output of the current neuron, its calculation is hierarchically incorporated under the sigmoid module. There is no method as yet to obtain this result using a simple bitwise manipulation. A multiplier must be used here (the additional size of the f' calculation module is not incorporated into the data of Table I).

E. Floating Point Format Implementation

The FLP format incorporates similar wrapper files and parameters as FXP. From the architectural point of view, it has the same pluggable structure.

1) Math Libraries: The FLP libraries are used from the work in [22]. They are mostly compliant to the IEEE single precision FLP standard (IEEE 754-1985 [11]). Some modification and addition to this work had to be made in order to handle special cases, described in [23], in order to fully comply to the IEEE standard. For a complete description of the specific generic VHDL FLP modules refer to [22].

2) Sigmoid: The FLP linear logsig approximation is of the same mathematical form as (12) (Fig. 11). Its structure is similar to that of the FXP sigmoid approximation function shown in Fig. 13. Note that the same calculation as in (11) has to be made in order to determine the first derivative of the transfer function.

Table I shows the differences in FPGA area requirements between the various implementations. The first column refers to the cost of implementing a divider. This is the only information available regarding size if the logsig is implemented using (10). The second column refers to the cost of implementing the logsig by a lookup table as outlined previously. The third and forth columns refer to implementation by (12). Note that the difference between the FXP and FLP implementations arises since in the FLP arithmetic bitwise manipulation of inputs is straightforward. The FLP sigmoid function in the current design consists primarily of instantiations to FLP arithmetic modules to perform the constituent operations, as shown in Fig. 13. This is the primary cause of the size of the sigmoid function geared to an FLP format being 12 times larger than its FXP counterpart. Also note that the derivative module, together with its multiplier, is included in the overall logsig structure, but is excluded from the size data found in Table I.

VI. EXPERIMENTAL SETUP AND TESTING

The VHDL network is synthesized for a Xilinx Virtex II, XC2v2000ff896-4 chip, using Xilinx ISE 7.1.04i. Scalability data is synthesized for the largest Xilinx Virtex II Pro xc2vp100ff1696-6 chip. Synthesis emphasis is not on area because we are comparing relative performance of numeric representations, and the values presented in Section VII are by no means absolute. Synthesis is performed using default Xilinx ISE optimization settings, namely "optimization goal = speed" and "optimization effort = normal."

To test the design, several VHDL test-benches are written. The test-benches provide the network with all necessary default values (learning rate, initial preselected random weights) and also provide the network inputs. The classic exclusive OR (XOR) problem, which is described in detail in [24], is chosen as the base of training for the network. The topology selected for this problem is (2,2,1), two inputs, two neurons in the hidden layer, and a single neuron in the output layer.

In the test-benches, the input/target pairs are presented to the network in sequence. An option in the test-benches is designed to allow the weights, gradients, input–output (I/O), and various other parameters to be written to a text file. This option is very useful in debugging the network. C++ and Matlab simulations are used to generate data flow from start to end of training. C++ simulation is useful in finding weight sets which guarantee convergence, and Matlab simulation is used to compare VHDL test-bench output with the theoretical data generated for every training step—this helps correct any design errors and verify correct hardware functionality to MLP-BP algorithm equations.

[5]Applied only to the first three integer bits and all fractional bits of n.

TABLE II
RANGE AND PRECISION OF FXP AND FLP FORMATS USED

Notation	Number format (bits)	Range	Precision
	sign-integer-fraction	Fixed point configurations	
fix1	1-3-12	[-8,8-2^{-12}]	2^{-12}
fix2	1-3-13	[-8,8-2^{-13}]	2^{-13}
fix3	1-3-14	[-8,8-2^{-14}]	2^{-14}
fix4	1-3-15	[-8,8-2^{-15}]	2^{-15}
fix5	1-3-16	[-8,8-2^{-16}]	2^{-16}
fix6	1-4-12	[-16,16-2^{-12}]	2^{-12}
fix7	1-4-13	[-16,16-2^{-13}]	2^{-13}
fix8	1-4-14	[-16,16-2^{-14}]	2^{-14}
fix9	1-4-15	[-16,16-2^{-15}]	2^{-15}
fix10	1-4-16	[-16,16-2^{-16}]	2^{-16}
fix11	1-5-12	[-32,32-2^{-12}]	2^{-12}
fix12	1-5-13	[-32,32-2^{-13}]	2^{-13}
fix13	1-5-14	[-32,32-2^{-14}]	2^{-14}
fix14	1-5-15	[-32,32-2^{-15}]	2^{-15}
fix15	1-5-16	[-32,32-2^{-16}]	2^{-16}
	sign-exponent-mantissa	Floating point configurations	
float1	1-2-12	[-8+2^{-10},8-2^{-10}]	2^{-12}
float2	1-2-13	[-8+2^{-11},8-2^{-11}]	2^{-13}
float3	1-2-14	[-8+2^{-12},8-2^{-12}]	2^{-14}
float4	1-2-15	[-8+2^{-13},8-2^{-13}]	2^{-15}
float5	1-2-16	[-8+2^{-14},8-2^{-14}]	2^{-16}
float6	1-3-12	[-32+2^{-8},32-2^{-8}]	2^{-14}
float7	1-3-13	[-32+2^{-9},32-2^{-9}]	2^{-15}
float8	1-3-14	[-32+2^{-10},32-2^{-10}]	2^{-16}
float9	1-3-15	[-32+2^{-11},32-2^{-11}]	2^{-17}
float10	1-3-16	[-32+2^{-12},32-2^{-12}]	2^{-18}

Simulation of the synthesized network is performed using ModelSim 6.0 SE. The network is trained during simulation for 3000 epochs.[6]

VII. RESULTS AND ANALYSIS

Twenty five arithmetic formats were tested as illustrated in Table II. Fifteen formats are fixed point formats while the other ten are floating point formats.

A. Numeric Format and Size

Tables III and IV present the raw *slice* data extracted from the "map report" in Xilinx ISE during synthesis of the design with various parameters. Table III presents details regarding FXP arithmetic formats while Table IV covers FLP arithmetic formats. All data refers to a network that implements the XOR problem. Additional data is shown for the number of MULT18x18's. These are dedicated multiplier blocks available in Virtex2, as well as other FPGAs [20].

Figs. 14 and 15 provide a visual comparison of the area consumed by the network with various numeric representations. It is obvious from these figures that FLP configurations take up a considerably larger area on the FPGA (approximately two times larger) compared with similar FXP configurations.

It should be noted that FXP and FLP formats are compared based on the similarity in range and precision. For example, by referring to Table II, one could compare fix4 (1-3-15) and float4 (1-2-15) configurations for both parallel and serial network implementations. In this case, both formats have approximately equal ranges as well as equal maximum precision. For a parallel implementation, FXP (1-3-15) takes up 5184 slices, whereas FLP (1-2-15) takes up 9606 slices.

[6]Simulations just for the XOR network are quite time consuming.

TABLE III
FXP PARALLEL AND SERIAL ARCHITECTURES

Number format (sig-int-frac)	Parallel		Serial	
	slices	MULT18x18	slices	MULT18x18
1-3-12	3,307	31	2,884	25
1-3-13	3,720	31	3,220	25
1-3-14	4,468	31	3,845	25
1-3-15	5,184	31	4,494	25
1-3-16	5,756	124	4,893	100
1-4-12	3,393	31	2,981	25
1-4-13	4,149	31	3,636	25
1-4-14	4,962	31	4,238	25
1-4-15	5,362	124	4,603	100
1-4-16	5,894	124	4,897	100
1-5-12	4,209	31	3,647	25
1-5-13	5,002	31	4,293	25
1-5-14	5,460	121	4,674	97
1-5-15	5,967	124	4,963	100
1-5-16	6,140	124	5,163	100

It is also interesting to observe from Table III that the consumption of the built-in multipliers (i.e., MULT18x18) remains constant until the FXP format exceeds the 18-bit width limit (excluding the sign bit). At this point, more than one MULT18x18 is required to implement a logical multiplier block. On the other hand, for the FLP implementation the usage of the MULT18x18 remains constant, since a logical multiplier only operates on a mantissa whose width does not exceed 18 bits for the compared formats.

B. Numeric Format and Performance

To compare the performance of various formats previously presented, the maximum operating frequency of individual

TABLE IV
FLP PARALLEL AND SERIAL ARCHITECTURES

Number format (sig-exp-man)	Parallel		Serial	
	slices	MULT18x18	slices	MULT18x18
1-2-12	7,645	31	7,210	25
1-2-13	8,315	31	7,415	25
1-2-14	9,408	31	8,401	25
1-2-15	9,606	31	8,490	25
1-2-16	10,069	31	8,965	25
1-3-12	9,720	31	8,973	25
1-3-13	9,897	31	8,714	25
1-3-14	10,712	31	9,539	25
1-3-15	10,919	31	9,979	25
1-3-16	11,882	31	10,438	25

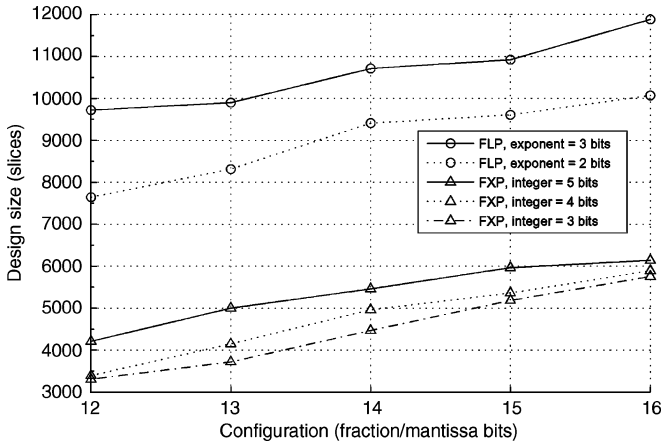


Fig. 14. Parallel architecture XOR network size.

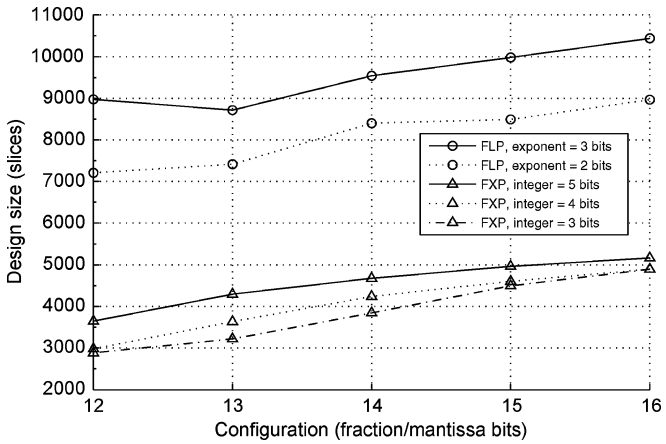


Fig. 15. Serial architecture XOR network size.

arithmetic units (adders and multipliers) of the FXP and FLP types is presented in Table V.[7] We have limited the comparison to individual units (i.e., adders, multipliers), instead of the overall network clock rate, since the network is limited in clock rate by the controller unit (Fig. 5) and not the performance of the neuron arithmetic modules.

In terms of latency, both the multiplier and adder in the FXP implementation have a latency of one clock cycle. On the other hand, the FLP multiplier has a latency of five clock cycles, and the FLP adder has a latency of six clock cycles.

[7]Note that the sigmoid function implementation for FXP formats outperforms an FXP adder of the same width while the implementation for FLP formats performs the same as an FLP adder of the same width.

TABLE V
CLOCK RATE OF VARIOUS REPRESENTATIONS

Number format	MULT (MHz)	ADD (MHz)
FXP		
1-3-12	192.957	210.526
1-3-13	193.817	208.203
1-3-14	200.985	205.549
1-3-15	79.697	203.707
1-3-16	78.737	201.531
1-4-12	193.817	208.203
1-4-13	200.985	205.549
1-4-14	79.697	203.707
1-4-15	78.737	201.531
1-4-16	77.558	199.401
1-5-12	200.985	205.549
1-5-13	79.697	203.707
1-5-14	78.737	201.531
1-5-15	77.558	199.401
1-5-16	75.623	197.316
FLP		
1-2-12	200.985	153.645
1-2-13	158.140	172.846
1-2-14	158.140	167.518
1-2-15	158.140	139.655
1-2-16	158.140	155.982
1-3-12	158.140	152.648
1-3-13	158.140	155.897
1-3-14	158.140	147.640
1-3-15	158.140	143.410
1-3-16	158.140	143.390

In terms of frequency, however, as the total FXP representation width increases, a logical multiplier slows down considerably due to the hierarchical connection of several MULT18x18's in the datapath. An FXP adder gracefully slows down due to a gradual increase in length of the carry chain. Furthermore, the FLP adder resembles in frequency performance its FXP counterpart, but the FLP multiplier does not. It appears that the FLP multiplier is limited internally by a component not directly dependant on the input data width.

Since the original MLP-BP algorithm is not pipelined, and batch training is not used, the network tends to wait for the completion of training for one pattern before presenting the next. For one pattern in the 2-2-1 (XOR) network there is a 42/53 (parallel/serial) cycle latency from start to finish using FXP libraries, and a 117/177 cycle latency using FLP libraries, respectively. It is clear that the FXP representation is superior over the FLP representation in operating frequency and pattern training latency.

C. Numeric Format and Convergence

It is important to check the impact of using the linear sigmoid function approximation on convergence properties of the network when using the formats described in Table II. Figs. 16 and 17 show convergence for the sample XOR problem using various FXP and FLP representations, respectively. The XOR problem is notorious for its symmetry. When using the classic MLP-BP algorithm as per Section II-A2, not every randomly selected weight set tends to converge—a portion of randomly selected weight sets will cause the network to oscillate in a local minimum at the center between output targets.

TABLE VI
SIZES OF SERIAL AND PARALLEL FXP 1-4-13 TOPOLOGIES

Topology	Num of Weights	Parallel		Serial	
		slices	MULT18x18	slices	MULT18x18
2-2-1	9	4,149	31	3,636	25
5-5-2	42	16,844	118	4,917	34
10-5-2	67	24,694	168	4,915	34
10-10-2	132	41,995	323	4,915	34
10-10-5	165	51,268	425	8,783	58
10-10-10	265	NA	NA	15,022	98

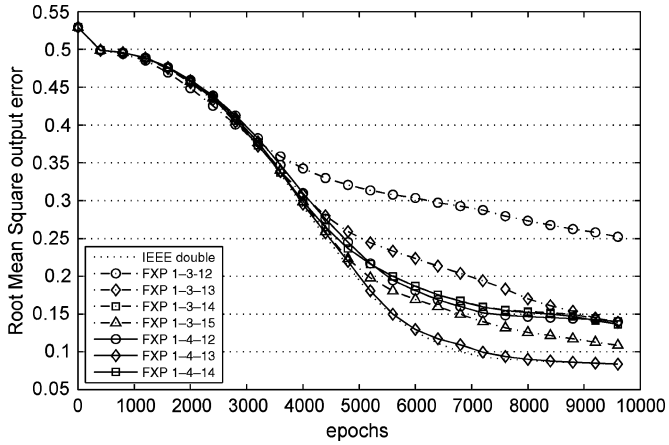


Fig. 16. Average convergence of network using FXP formats.

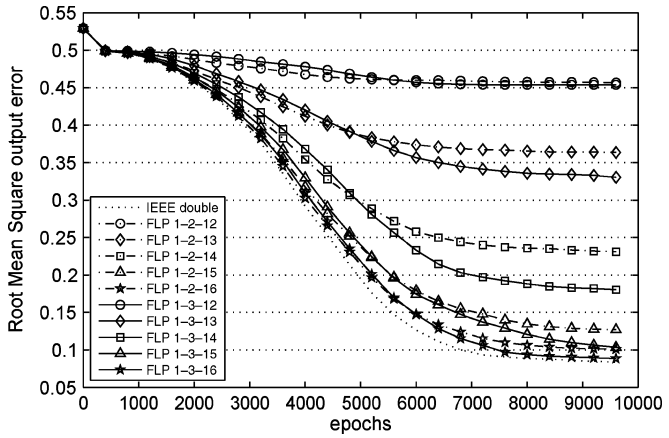


Fig. 17. Average convergence of network using FLP formats.

To obtain convergence data, weight sets are generated at random and training is performed using a learning rate of 0.1 for 10 000 epochs and using IEEE double precision format. Of these sets, a 100 sets which give adequate convergence [<20% root-mean-square (rms) error] are selected, while the remainder, out of a total 173 sets generated, are discarded. For each tested number format, the network is then trained using these 100 random weight sets for 10 000 epochs. Root mean square error after each training epoch is recorded. The error data for the 100 simulations in each numeric format is then averaged and a plot of average rms error is obtained.

Results for numeric formats which use one MULT18x18 block per implemented multiplier are plotted in Figs. 16 and 17. In the FXP formats, it is evident that using FXP 1-3-12 format does not result in adequate convergence performance achieved, nor does it result in significant area saving among formats which also efficiently use the built-in multiplier resources on Xilinx FPGAs. In addition, FXP 1-5-12 and FXP 1-5-13 formats provide wider range than FXP 1-4-12 and FXP 1-4-13 formats but result in identical average convergence performance for a greater SLICE resource cost, and thus are not considered. From evaluating the FXP convergence graph in Fig. 16, it is clear that the {FXP 1-4-13} format has the highest convergence performance out of all FXP representations which efficiently use the built-in multipliers.

Looking at convergence results for FLP formats in Fig. 17, the largest FLP 1-3-16 format is close in convergence to IEEE double precision format, but is somewhat worse than the FXP 1-4-13 format. A closer look at Figs. 14 and 15 indicate that the FLP format is inferior in FPGA resources consumed and does not provide better convergence performance than similar or smaller, by precision and range, FXP formats.

Thus, choosing FXP 1-4-13 is optimal as it provides best convergence performance of the FXP group of formats, and optimally uses the hardware resources of Xilinx FPGAs.

D. Scalability

The XOR network is a small network that allowed testing of 25 arithmetic representations in reasonable time. But for real world application, much larger networks are used. Would the results presented so far hold for larger networks? Table VI presents experimental data collected by synthesizing larger networks using the FXP 1-4-13 number representation. Fig. 18 visually demonstrates that MLP-BP networks in both serial and parallel architectures scale fairly linearly in number of slices occupied as the number of free parameters (weights) in the network grows. In addition, in Fig. 18 network sizes are compared to available resources in Xilinx Virtex II/IIPro/4 family of FPGAs. An attempt was made at synthesizing larger networks. Unfortunately, Xilinx ISE software is unable to effectively synthesize designs that are larger than their largest FPGA offerings.

With scalability also arises problems of interface bottlenecks. The larger the network, the more data it needs to be fed at every training cycle. However, new larger FPGAs are continuously developed, and larger networks, which are able to fit on a single chip, will always have proportionally wider I/O resources available to handle the required I/O load. Additionally, our network architecture is not pipelined, conforming to the unmodified MLP-BP algorithm described in Section II-A. When the network complexity is increased, the latency required to process each pattern to completion also increases. The training patterns, which are typically stored in off-chip memory, can be loaded sequentially into the FPGA, and then be latched in parallel as they

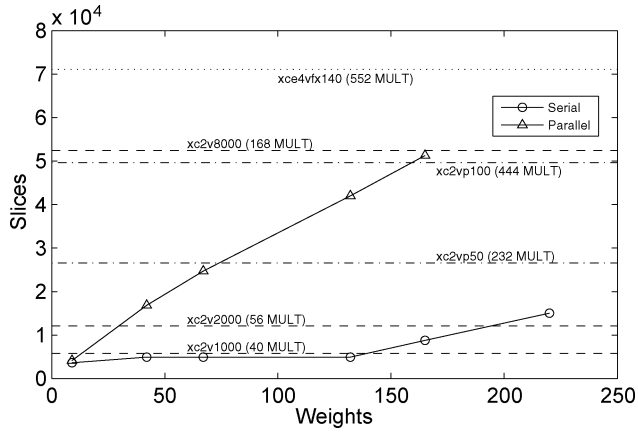


Fig. 18. Scalability of serial and parallel FXP 1-4-13 topologies.

are loaded into the network module, thus avoiding any further I/O bottlenecks.

E. Analysis

Analysis of the results presented shows the following two points.

1) **Fixed versus floating point arithmetic**
   The FXP architecture is always smaller in area, as compared to an FLP architecture with similar precision and range by approximately a factor of two. The FXP representation is also somewhat faster in clock rate and significantly better in latency than its FLP counterpart. As well, for similar precision and range representations, the FXP format produces better convergence results over FLP format. The FXP 1-4-13 format was found to be optimal for MLP-BP networks implementation on FPGAs.
2) **Serial versus parallel architecture**
   By examining Fig. 18, while scaling linearly using parallel and serial architecture for large network topologies, an MLP-BP network can achieve significant area savings when implemented serially—around 80%. Though at an increase in training time due to increased per-pattern latency, a topology which cannot fit into a single FPGA when implemented in parallel, can in fact fit into the same FPGA via a serial implementation.

Can we generalize these results to other neural networks algorithms and topologies? This is highly dependent on the type of algorithm used. As discussed in Section III, the MLP-BP has special characteristics that allow less precise arithmetic representation to be used with no impact on the functionality of the algorithm. If similar discussion can be made for other algorithms then we believe that our result will hold for these algorithms. We plan to explore this issue in future research.

VIII. Conclusion

This paper discussed techniques for implementing MLP-BP on FPGAs and the impact of arithmetic formats on the required resources. The resource requirements of networks implemented with FXP arithmetic are approximately two times less than its counterpart FLP designs, with similar precision and range of the data representation used, without compromising the effectiveness of training and operational function. Also, FXP arithmetic produces better convergence and takes less time (clock cycles) to train, and has a somewhat faster clock rate than FLP arithmetic. We concluded our findings by demonstrating a network's ability to scale to larger topologies while maintaining the simplicity of a single chip design.

Acknowledgment

The authors would like to thank J. Ewing for his work done in writing and testing the VHDL code used in our simulations.

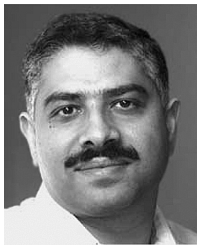
References

[1] D. Rumelhart, J. McClelland, and P. R. Group, *Parallel Distrubuted Processing: Explorations in the Microstructure of Cognition*. Cambridge, MA: MIT Press, 1986, vol. 1, Foundations.
[2] G. N. Karystinos and D. A. Pados, "On overfitting, generalization, and randomly expanded training sets," *IEEE Trans. Neural Netw.*, vol. 11, no. 5, pp. 1050–1057, Sep. 2000.
[3] H. McCartor, "A highly parallel digital architecture for neural network emulation," *VLSI Artif. Intell. Neural Netw.*, pp. 357–366, 1991.
[4] U. Ramacher, W. Raab, and J. A. *et al.*, "Multiprocessor and memory architecture of the neurocomputers SYNAPSE-1," in *Proc. 3rd Int. Conf. Microelectron. Neural Netw.*, 1993, pp. 227–231.
[5] S. Tam, B. Gupta, H. Castro, and M. Holler, "Learning on an analog VLSI neural network chip," in *Proc. IEEE Int. Conf. Syst., Man Cybern.*, Nov. 1990, pp. 701–703.
[6] J. G. Eldredge, "FPGA density enhancement of a neural network through run-time reconfiguration," M.S. thesis, Dept. Elect. Comput. Eng., Brigham Young Univ., Provo, UT, 1994.
[7] D. Anguita, A. Boni, and S. Ridella, "A digital architecture for support vector machines: Theory, algorithm, and FPGA implementation," *IEEE Trans. Neural Netw.*, vol. 14, no. 5, pp. 993–1009, Sep. 2003.
[8] H. Ng and K. Lam, "Analog and digital FPGA implementation of brin for optimization problems," *IEEE Trans. Neural Netw.*, vol. 14, no. 5, pp. 1413–1425, Sep. 2003.
[9] Y. Maeda and M. Wakamura, "Simultaneous perturbation learning rule for recurrent neural networks and its fpga implementation," *IEEE Trans. Neural Netw.*, vol. 16, no. 6, pp. 1664–1672, Nov. 2005.
[10] J. Zhu and P. Sutton, "FPGA implementations of neural networks—a survey of a decade of progress," in *Proc. Conf. Field Programm. Logic*, Lisbon, Portugal, 2003, pp. 1062–1066.
[11] *IEEE Standard for Binary Floating Point Arithmetic*, Std 754-1985 edition, New York ANSI/IEEE, 1985.
[12] G. Even and P. Wolfgang, "On the design of IEEE compliant floating point units," *IEEE Trans. Comput.*, vol. 49, no. 5, pp. 398–413, May 2000.
[13] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Comput. Surv.*, vol. 23, no. 1, pp. 5–48, Mar. 1991.
[14] J. Holt and T. Baker, "Backpropagation simulations using limited precision calculations," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN-91)*, Seattle, WA, Jul. 1991, vol. 2, pp. 121–126.
[15] G. Govindu, L. Zhuo, S. Choi, and V. Prasanna, "Analysis of high-performance floating-point arithmetic on FPGAs," in *Proc. IEEE Int. Parallel and Distributed Process. Symp.*, Santa Fe, NM, Apr. 2004, pp. 149–156.
[16] C. T. Ewe, "Dual fixed-point: an efficient alternative to floating-point computation for DSP applications," in *Proc. Field Programm. Logic Appl.*, Tampere, Finland, Aug. 2005, pp. 715–716.
[17] C. T. Ewe, P. Y. K. Cheung, and G. A. Constantinides, "Error modelling of dual fixed-point arithmetic and its application in field programmable logic," in *Proc. Field Programm. Logic Appl.*, Tampere, Finland, Aug. 2005, pp. 124–129.
[18] S. Li, M. Moussa, and S. Areibi, "Arithmetic formats for implementing artificial neural networks on FPGAs," *Can. J. Elect. Comput. Eng.*, vol. 31, no. 1, pp. 31–40, 2006.
[19] W. Ligon, III, S. McMillan, G. Monn, K. Schoonover, F. Stivers, and K. Underwood, "A re-evaluation of the practicality of floating point operations on FPGAs," in *Proc. IEEE Symp. FPGAs Custom Comput. Mach.*, K. L. Pocek and J. Arnold, Eds., Los Alamitos, CA, 1998, pp. 206–215 [Online]. Available: citeseer.nj.nec.com/95888.html

[20] Xilinx, Virtex-II Platform FPGAs: Detailed Description Mar. 2004 [Online]. Available: www.xilinx.com, DS031-2 v3.2 ed.
[21] *"Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference (243191)"* Intel Corp., 1999 [Online]. Available: www.intel.com
[22] P. Belanovic, "Library of parameterized hardware modules for floating-point arithmetic with an example application," M.A.Sc. thesis, Elect. Comp. Eng. Dept., Northeastern Univ., Boston, MA, 2002.
[23] Sun Microsystems, What Every Computer Scientist Should Know About Floating-Point Arithmetic Mountain View, CA, Tech. Rep., 1994.
[24] S. Haykin, *Neural Networks: A Comprehensive Foundation, Edition II*. Englewood Cliffs, NJ: Prentice-Hall, 1999.

Antony W. Savich received the B.A.Sc. degree in computer engineering from the University of Toronto, Toronto, ON, Canada, in 2002. He is currently working towards the M.Sc. degree in engineering systems and computing at the University of Guelph, Guelph, ON, Canada.

Before resuming his studies in 2004, he participated as an Engineering Consultant in a variety of prototyping projects. His research and professional interests include reconfigurable computing, machine learning, artificial neural networks, embedded systems, and cross-system integration in mechanical applications.



Medhat Moussa (S'96–M'03) received the B.A.Sc. degree in mechanical engineering from the American University, Cairo, Egypt, the M.A.Sc. degree in mechanical engineering from the Universite de Moncton, Moncton, NB, Canada, and the Ph.D. degree in systems design engineering from the University of Waterloo, Waterloo, ON, Canada in 1987, 1991, 1996, respectively.

He is now an Associate Professor with the School of Engineering, University of Guelph, Guelph, ON, Canada. His research interests include user-adaptive robots, machine learning, neural networks, and human–robot interaction.

Dr. Moussa is a member of Association for Computing Machinery (ACM).

Shawki Areibi (S'88–M'89) received the B.Sc. degree in computer engineering from Elfateh University, Tripoli, Libya, in 1984 and the M.A.Sc. and Ph.D. degrees in electrical/computer engineering from the University of Waterloo, Waterloo, ON, Canada, in 1991 and 1995, respectively.

From 1985 to 1987, he was at NCR Research Laboratories, Libya. From 1995 to 1997, he was a Research Mathematician with Shell International Oil Products in The Netherlands. Currently, he is an Associate Professor with the School of Engineering, University of Guelph, Guelph, ON, Canada. He has authored/coauthored over 60 papers in international journals and conferences. His research interests include VLSI physical design automation, combinatorial optimization, reconfigurable computing systems, embedded systems, and parallel processing.

Dr. Areibi served on the technical program committees for several international conferences on computer engineering and embedded systems including Genetic and Evolutionary Computation Conference (GECCO) and High Performance Computing (HPC). He is the Associate Editor of the *International Journal of Computers and Their Applications*.