

Chapter 2

ON THE ARITHMETIC PRECISION FOR IMPLEMENTING BACK-PROPAGATION NETWORKS ON FPGA: A CASE STUDY

Medhat Moussa and Shawki Areibi and Kristian Nichols

University of Guelph

School of Engineering

Guelph, Ontario, N1G 2W1, Canada

mmoussa@uoguelph.ca sareibi@uoguelph.ca knichols@uoguelph.ca

Abstract Artificial Neural Networks (ANNs) are inherently parallel architectures which represent a natural fit for custom implementation on FPGAs. One important implementation issue is to determine the numerical precision format that allows an optimum tradeoff between precision and implementation areas. Standard single or double precision floating-point representations minimize quantization errors while requiring significant hardware resources. Less precise fixed-point representation may require less hardware resources but add quantization errors that may prevent learning from taking place, especially in regression problems. This chapter examines this issue and reports on a recent experiment where we implemented a Multi-layer perceptron (MLP) on an FPGA using both fixed and floating point precision. Results show that the fixed-point MLP implementation was over 12x greater in speed, over 13x smaller in area, and achieves far greater processing density compared to the floating-point FPGA-based MLP.

Keywords: Reconfigurable Computing, Back-propagation Algorithm, FPGAs, Artificial Neural Networks

2.1 Introduction

Artificial neural networks (ANNs) have been used in many applications in science and engineering. The most common architecture consists of multi-layer perceptrons trained using the error back-propagation algorithm (MLP-BP) [37]. One of the main problems in training a BP Network is the lack of a clear methodology to determine the network topology before training starts. Experimenting with various topologies is difficult due to the long time required

for each training session especially with large networks. Network topology is an important factor in the network's ability to generalize after training is completed. A larger than needed network may over-fit the training data and result in poor generalization on testing data, while a smaller than needed network may not have the computational capacity to approximate the target function. Furthermore, in applications where online training is required, training time is often a critical parameter. Thus it is quite desirable to speed up training. This allows for reasonable experimentation with various network topologies and ability to use BP networks in online applications.

Since Neural Networks in general are inherently parallel architectures [55], there have been several earlier attempts to build custom ASIC based boards that include multiple parallel processing units such as the NI1000. However, these boards suffered from several limitations such as the ability to run only specific algorithms and limitations on the size of a network. Recently, much work has focused on implementing artificial neural networks on reconfigurable computing platforms. Reconfigurable computing is a means of increasing the processing density (i.e greater performance per unit of silicon area) above and beyond that provided by general-purpose computing platforms. Field Programmable Gate Arrays (FPGAs) are a medium that can be used for reconfigurable computing and offer flexibility in design like software but with performance speeds closer to Application Specific Integrated Circuits (ASICs).

However, there are certain design tradeoffs which must be dealt with in order to implement Neural Networks on FPGAs. One major tradeoff is *area vs. precision*. The problem is how to balance between the need for numeric precision, which is important for network accuracy and speed of convergence, and the cost of more logic areas (i.e. FPGA resources) associated with increased precision. Standard precisions floating-point would be the ideal numeric representation to use because it offers the greatest amount of precision (i.e. minimal quantization error) and matches the representation used in simulating Neural Networks on general purpose microprocessors. However, due to the limited resources available on an FPGA, standard floating-point may not be as feasible compared to more area-efficient numeric representations, such as 16 or 32 bit fixed-point.

This chapter explores this design trade-off by testing an implementation of an MLP-BP network on an FPGA using both floating-point and fixed-point representations. The network is trained to learn the XOR problem. The study's goal is to provide experimental data regarding what resources are required for both formats using current FPGA design tools and technologies. This chapter is organized as follows: In Section 2.2, background material on the area vs precision range trade-off is presented as well as an overview of the back-propagation algorithm and FPGA architectures. Section 2.3 provides details about the architecture design used to implement a BP network on FPGA. In

section 2.4 the XOR problem is presented. Finally validation of the proposed implementations, and benchmarked results of floating-point and fixed-point arithmetic functions implemented on a FPGA are discussed in Section 2.5.

2.2 Background

One way to help achieve the density advantage of reconfigurable computing over general-purpose computing is to make the most efficient use of the hardware area available. In terms of an optimal *range-precision vs area trade-off*, this can be achieved by determining the *minimum allowable precision* and *minimum allowable range*, where their criterion is to minimize hardware area usage without sacrificing quality of performance. These two concepts combined can also be referred to as the *minimum allowable range-precision*.

2.2.1 Range-Precision vs. Area Trade-off

A reduction in precision usually introduces many errors into the system. Determining the minimum allowable precision is actually a question of determining the maximum amount of uncertainty (i.e. quantization error due to limited precision) an application can withstand before performance begins to degrade. It is often dependent upon the algorithm used and the application at hand.

For MLP using the BP algorithm, Holt and Baker [41] showed using simulations and theoretical analysis that *16-bit fixed-point* (1 bit sign, 3 bit left and 12 bit right of the radix point) was the minimum allowable range-precision for the back-propagation algorithm assuming that both input and output were normalized between [0,1] and a sigmoid transfer function was used.

Ligon III *et al.* [45] have also shown the density advantage of fixed-point over floating-point for older generation Xilinx 4020E FPGAs, by showing that the space/time requirements for 32-bit fixed-point adders and multipliers were less than that of their 32-bit floating-point equivalents.

Other efforts focused on developing a complete reconfigurable architecture for implementing MLP. Eldredge [3] successfully implemented the back-propagation algorithm using a custom platform he built out of Xilinx XC3090 FPGAs, called the Run-Time Reconfiguration Artificial Neural Network (RRANN). He showed that the RRANN architecture could learn how to approximate centroids of fuzzy sets. Heavily influenced by the Eldredge's RRANN architecture, Beuchat *et al.* [13] developed a FPGA platform, called RENCO—a REconfigurable Network Computer. As its name implies, RENCO contains four Altera FLEX 10K130 FPGAs that can be reconfigured and monitored over any LAN (i.e. Internet or other) via an on-board 10Base-T interface. RENCO's intended application was hand-written character recognition. Ferrucci and Martin [14, 15] built a custom platform, called Adaptive Connec-

tionist Model Emulator (ACME) which consists of multiple Xilinx XC4010 FPGAs. They validated ACME by successfully carrying out a 3-input, 3-hidden unit, 1-output network used to learn the 2-input XOR problem. Skrbek's FPGA platform [26], called the ECX card, could also implement Radial Basis Function (RBF) neural networks, and was validated using pattern recognition applications such as parity problem, digit recognition, inside-outside test, and sonar signal recognition.

Since the size of an FPGA-based MLP-BP is proportional to the multiplier used, it is clear that given an FPGA's finite resources, a 32-bit signed (2's complement) *fixed-point* representation will allow larger [54] ANNs to be implemented than could be accommodated when using a 32-bit IEEE (a 32-bit floating point multiplier can be implemented on a Xilinx Virtex-II or Spartan-3 FPGA using four of the dedicated multiplier blocks and CLB resources) *floating-point*. However, while 32 fixed-point representation allows high processor density implementation, the quantization error of 32 floating-point representation is negligible. Validating an architecture on an FPGA using 32-bit floating point arithmetic might be easier than fixed point arithmetic since a software version of the architecture can be run on a Personal Computer with 32-bit floating point arithmetic. As such its use is justifiable if the relative loss in processing density is negligible in comparison.

FPGA architectures and related development tools have become increasingly sophisticated in more recent years, including improvements in the space/time optimization of arithmetic circuit designs. As such, the objective of this study is to determine the feasibility of floating-point arithmetic in implementing MLP-BP using today's FPGA design tools and technologies. Both floating-point and fixed-point precision are considered for implementation and are classified as *amplitude-based* digital numeric representations. Other numeric representations, such as digital *frequency-based* [42] and analog were not considered because they promote the use of low precision, which is often found to be inadequate for **minimum allowable range-precision**.

2.2.2 Overview of Back-propagation Algorithm

It is helpful before proceeding to discuss architecture design to give a brief review of MLP and the error Back-propagation algorithm. The general structure of a Multi-layer perceptron (MLP) neural network is shown in Figure 2.1, where layers are numbered 0 to M , and neurons are numbered 1 to N .

A MLP using the back-propagation algorithm has five steps of execution:

(1) Initialization

The following parameters must be initialized before training starts: (i) $w_{kj}^{(s)}(n)$ is defined as the *synaptic weight* that corresponds to the connection from neuron unit j in the $(s - 1)^{th}$ layer, to k in the s^{th} layer. This weight

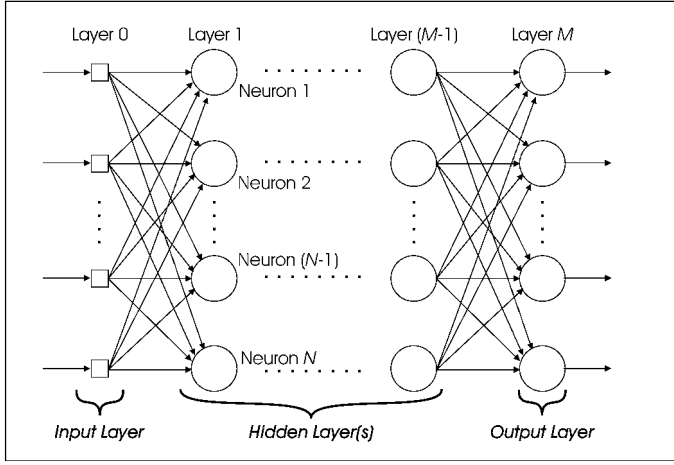


Figure 2.1. Generic structure of a feedforward ANN

is updated during the n^{th} iteration, where $n = 0$ for initialization. (ii) η is defined as the *learning rate* and is a constant scaling factor used to control the step size in error correction during each iteration of the back-propagation algorithm. (iii) $\theta_k^{(s)}$ is defined as the *bias* of a neuron, which is similar to synaptic weight in that it corresponds to a connection to neuron unit k in the s^{th} layer. Statistically, biases can be thought of as noise, which better randomizes initial conditions, and increases the chances of convergence.

(2) Presentation of Training Examples

Available training data are presented to the network either individually or as a group (a.k.a. epoch).

(3) Forward Computation

During the forward computation, data from neurons of a lower layer (i.e. $(s - 1)^{th}$ layer), are propagated forward to neurons in the upper layer (i.e. s^{th} layer) via a feed-forward connection network. The computation performed by each neuron (in the hidden layer) is as follows:

$$H_k^{(s)} = \sum_{j=1}^{N_{s-1}} w_{kj}^{(s)} o_j^{(s-1)} + \theta_k^{(s)} \quad (2.1)$$

where $j < k$ and $s = 1, \dots, M$

$H_k^{(s)}$ = weighted sum of the k^{th} neuron in the s^{th} layer

$w_{kj}^{(s)}$ = synaptic weight sd defined above

$o_j^{(s-1)}$ = neuron output of the j^{th} neuron in the $(s - 1)^{th}$ layer

$\theta_k^{(s)}$ = bias of the k^{th} neuron in the s^{th} layer.

On the other hand for the output layer neurons the computation is as follows:

$$o_k^{(s)} = f(H_k^{(s)}) \quad (2.2)$$

where $k = 1, \dots, N$ and $s = 1, \dots, M$

$o_k^{(s)}$ = neuron output of the k^{th} neuron in the s^{th} layer

$f(H_k^{(s)})$ = *activation function* computed on the weighted sum $H_k^{(s)}$

Note that a unipolar sigmoid function is often used as the nonlinear activation function, such as the following *logsig* function:

$$f(x)_{logsig} = \frac{1}{1 + \exp(-x)} \quad (2.3)$$

(4) Backward Computation

In this step, the weights and biases are updated. The learning algorithm's goal is to minimize the error between the expected (or teacher) value and the actual output value that was determined in the Forward Computation. The following steps are performed:

- 1 Starting with the output layer, and moving back towards the input layer, calculate the local gradients, as follows:

$$\varepsilon_k^{(s)} = \begin{cases} t_k - o_k^{(s)} & s = M \\ \sum_{j=1}^{N_{s+1}} w_{kj}^{s+1} \delta_j^{(s+1)} & s = 1, \dots, M - 1 \end{cases} \quad (2.4)$$

where

$\varepsilon_k^{(s)}$ = *error term* for the k^{th} neuron in the s^{th} layer; the difference between the teaching signal t_k and the neuron output $o_k^{(s)}$

$\delta_j^{(s+1)}$ = *local gradient* for the j^{th} neuron in the $(s + 1)^{th}$ layer.

$$\delta_k^{(s)} = \varepsilon_k^{(s)} f'(H_k^{(s)}) \quad s = 1, \dots, M \quad (2.5)$$

where $f'(H_k^{(s)})$ is the derivative of the activation function.

- 2 Calculate the weight (and bias) changes for all the weights as follows:

$$\Delta w_{kj}^{(s)} = \eta \delta_k^{(s)} o_j^{(s-1)} \quad k = 1, \dots, N_s \\ j = 1, \dots, N_{s-1} \quad (2.6)$$

where $\Delta w_{kj}^{(s)}$ is the change in synaptic weight (or bias) corresponding to the gradient of error for connection from neuron unit j in the $(s - 1)^{th}$ layer, to neuron k in the s^{th} layer.

3 Update all the weights (and biases) as follows:

$$w_{kj}^s(n+1) = \Delta w_{kj}^{(s)}(n) + w_{kj}^{(s)}(n) \quad (2.7)$$

where $k = 1, \dots, N_s$ and $j = 1, \dots, N_{s-1}$

$w_{kj}^s(n+1)$ = updated synaptic weight (or bias) to be used in the $(n+1)^{th}$ iteration of the Forward Computation

$\Delta w_{kj}^{(s)}(n)$ = change in synaptic weight (or bias) calculated in the n^{th} iteration of the Backward Computation, where n = the current iteration

$w_{kj}^{(s)}(n)$ = synaptic weight (or bias) to be used in the n^{th} iteration of the Forward and Backward Computations, where n = the current iteration.

(5) Iteration

Reiterate the Forward and Backward Computations for each training example in the epoch. The trainer can continue to train the MLP using one or more epochs until some stopping criteria is met. Once training is complete, the MLP only needs to carry out the Forward Computation when used in applications.

2.2.3 Field Programmable Gate Arrays

FPGAs are a form of programmable logic, which offer flexibility in design like software, but with performance speeds closer to Application Specific Integrated Circuits (ASICs). With the ability to be reconfigured an endless number of times after having been manufactured, FPGAs have traditionally been used as a prototyping tool for hardware designers. However, as growing die capacities of FPGAs have increased over the years, so has their use in reconfigurable computing applications too.

The fundamental architecture of Xilinx FPGAs consists of a two-dimensional array of programmable logic blocks, referred to as *Configurable Logic Blocks (CLBs)*. Figure 2.2 shows the architecture of a CLB from the Xilinx Virtex-E family of FPGAs, which contains four *logic cells (LCs)* and is organized in two similar *slices*. Each LC includes a 4-input look-up table (LUT), dedicated fast carry-lookahead logic for arithmetic functions, and a storage element (i.e. a flip-flop). A CLB from the Xilinx Virtex-II family of FPGAs, on the other hand, contains eight 4-input LUTs, and is over twice the amount of logic as a Virtex-E CLB. As we will see, the discrepancies in CLB architecture from one family to another is an important factor to take into consideration when comparing the spatial requirements (in terms of CLBs) for circuit designs which have been implemented on different Xilinx FPGAs.

2.3 Architecture design and implementation

There has been a rich history of attempts at implementing ASIC-based approaches for neural networks - traditionally referred to as *neuroprocessors* [29]

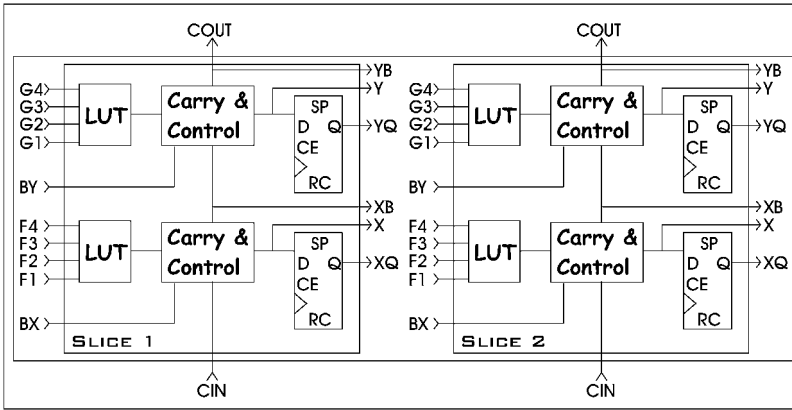


Figure 2.2. Virtex-E Configurable Logic Block

or *neurochips*. FPGA-based implementations, on the other hand, are still a fairly new approach which has only been in effect since the early 1990s. The type of neural network used in a FPGA-based implementation, and/or the algorithm used for *on-chip learning* is a classification feature which often depends on its intended application. On-chip learning [11] occurs when the learning algorithm is implemented in hardware, or in this case, on the FPGA. *Offline learning* occurs when learning occurs on a general-purpose computing platform before the learned system is implemented in hardware.

2.3.1 Non-RTR Implementation

The digital ANN architecture implemented in this chapter is an example of a non-RTR (Run-Time Reconfiguration) reconfigurable computing application, where all stages of the algorithm reside together on the FPGA at once. A finite state machine was used to ensure proper sequential execution of each step of the back-propagation algorithm as described in Section 2.2.2, which consists of the following two states:

- 1 Forward state (**F**) - used to emulate the *forward pass* associated with the back-propagation algorithm. Only the ANN's input signals, synapses, and neurons should be active in this state, in order to calculate the ANN's output. All forward pass operations (i.e. Forward Computations as described by Equations 2.1, 2.2, and 2.3) should be completed by the time the Forward State (**F**) ends.
- 2 Backward state (**B**) - used to emulate the *backward pass* associated with the back-propagation algorithm. All the circuitry associated with helping the ANN learn (i.e. essentially all the circuitry *not* active in Forward State) should be active here. All backward pass operations (i.e. Back-

ward Computations as described by Equations 2.4, 2.5, and 2.6) should be completed by the time the Backward state ends.

It should be noted that both states of the finite state machine continually alternate, and synaptic weights are updated (as described in Equation 2.7) during the transition from Backward State to Forward State.

As far as the ANN's components (eg. neurons, synapses) were concerned, the finite state machine is generally a means of synchronizing when various sets of components should be active. The duration of each state depends on the number of clock cycles required to complete calculations in each state, the length of the system's clock period, and the propagation delay associated with each state (Note that propagation delay is platform dependent, and can only be determined after the digital VLSI design has been synthesized on a targeted FPGA. The propagation delay is then determined through a timing analysis/simulation using the platform's EDA tools). The architecture of the active ANN components associated with each state dictates the propagation delay for that state.

Each of the ANN components implemented in hardware, such as the synapse and neuron, housed a *chip select* input signal in their architecture which is driven by the finite state machine. This chip select feature ensured that only those components that were associated with a particular state were enabled or active throughout that state's duration. With regards to initialization of the circuit, a *reset* input signal was used which would fulfill two important requirements when activated:

- Ensure the finite state machine initially starts in "Forward State".
- Initialize the synaptic weights of the ANN, to some default value.

Finally, the BP algorithm calculations, Equations 2.1–2.7, are realized using a series of arithmetic components, including addition, subtraction, multiplication, and division. Standardized high-description language (HDL) libraries for digital hardware implementation can be used in synthesizing all the arithmetic calculations involved with the back-propagation algorithm, in analogous fashion of how typical math general programming language (GPL) libraries are used in software implementations of ANNs. The architecture described here is generic enough to support arithmetic HDL libraries of different amplitude-based precision, whether it be floating-point or fixed-point.

2.3.2 Arithmetic Library

The architecture was developed using *VHDL*. Unfortunately, there is currently no explicit support for fixed- and floating-point arithmetic in *VHDL* (according to the IEEE Design Automation Standards Committee [43], an extension of IEEE Std 1076.3 has been proposed to include support for fixed-

and floating-point numbers in VHDL, and is to be addressed in a future review of the standard). As a result, two separate arithmetic VHDL libraries were custom designed for use with the FPGA-based ANN. One of the libraries supports the IEEE-754 standard for single-precision (i.e. 32-bit) floating-point arithmetic, and is referred to as `uog_fp_arith`, which is an abbreviation for *University of Guelph Floating-Point Arithmetic*. The other library supports 16-bit fixed-point arithmetic, and is referred to as `uog_fixed_arith`, which is an abbreviation for *University of Guelph Fixed-Point Arithmetic*. These two representations were chosen based on previous results from the literature [41] that showed that 16 bit fixed point representation is the minimum needed to allow the BP algorithm to converge and the fact that 32 bit floating point precision is the standard floating point representation. We could have used a custom floating point representation (maybe with less precision) but it is very likely that any future VHDL floating point implementation will follow this standard representation. As such we specifically wanted to test the tradeoff with this standard presentation. This is also important for applications in Hardware/Software co-design using languages like SystemC and HandleC.

Fixed-point representation is signed 2's complement binary representation, which is made rational with a *virtual* decimal point. The virtual radix point location used in `uog_fixed_arith` is *SIII.FFFFFFFF*, where

S = sign bit

I = integer bit, as implied by location of binary point

F = fraction bit, as implied by location of binary point

The range for a 16-bit fixed-point representation of this configuration is [-8.0, 8.0), with a quantization error of 2.44140625E-4. Description of the various arithmetic VHDL design alternatives considered for use in the `uog_fp_arith` and `uog_fixed_arith` libraries are summarized in Table 2.1. All HDL designs with the word *std* in their name signify that one of the IEEE standardized VHDL arithmetic libraries was used to create them. For example, `uog_std_multiplier` was easily created using the following VHDL syntax:

```
 $z \leq x * y;$ 
```

where x and y are the input signals, and z the output signal of the circuit. Such a high level of abstract design is often associated with *behavioral* VHDL designs, where ease of design comes at the sacrifice of letting the FPGA's synthesis tools dictate the fine-grain architecture of the circuit.

On the other hand, an engineer can explicitly define the fine-grain architecture of a circuit by means of *structural* VHDL and schematic-based designs, as was done for `uog_ripple_carry_adder`, `uog_c1_adder` (please refer to Figure 2.4 for detailed implementation) and `uog_sch_adder` respectively. How-

Table 2.1. Summary of alternative designs considered for use in custom arithmetic VHDL libraries

HDL Design	Description
uog_fp_add*	IEEE 32-bit single precision floating-point pipelined parallel adder
uog_ripple_carry_adder	16-bit fixed-point (bit-serial) ripple-carry adder
uog_c_l_addr	16-bit fixed-point (parallel) carry lookahead adder
uog_std_adder	16-bit fixed-point parallel adder created using standard VHDL arithmetic libraries
uog_core_adder	16-bit fixed-point parallel adder created using Xilinx LogiCORE <i>Adder Subtractor v5.0</i>
uog_sch_adder	16-bit fixed-point parallel adder created using Xilinx <i>ADD16</i> schematic-based design
uog_pipe_adder	16-bit fixed-point pipelined parallel adder created using Xilinx LogiCORE <i>Adder Subtractor v5.0</i>
uog_fp_sub*	IEEE 32-bit single precision floating-point pipelined parallel subtracter
uog_par_subtractor	16-bit fixed-point carry lookahead (parallel) subtracter, based on uog_std_adder VHDL entity
uog_std_subtractor	16-bit fixed-point parallel subtracter created with standard VHDL arithmetic libraries
uog_core_subtractor	16-bit fixed-point parallel subtracter created using Xilinx LogiCORE <i>Adder Subtractor v5.0</i>
uog_fp_mult*	IEEE 32-bit single precision floating-point pipelined parallel multiplier
uog_booth_multiplier	16-bit fixed-point shift-add multiplier based on Booth's algorithm (with carry lookahead adder)
uog_std_multiplier	16-bit fixed-point parallel multiplier created using standard VHDL arithmetic libraries
uog_core_bs_mult	16-bit fixed-point bit-serial (non-pipelined) multiplier created using Xilinx LogiCORE <i>Multiplier v4.0</i>
uog_pipe_serial_mult	16-bit fixed-point bit-serial (pipelined) multiplier created using Xilinx LogiCORE <i>Multiplier v4.0</i>
uog_core_par_multiplier	16-bit fixed-point parallel (non-pipelined) multiplier created using Xilinx LogiCORE <i>Multiplier v4.0</i>
uog_pipe_par_mult	16-bit fixed-point parallel (pipelined) multiplier created using Xilinx LogiCORE <i>Multiplier v4.0</i>
active_func_sigmoid	Logsig (i.e. sigmoid) function with IEEE 32-bit single precision floating-point
uog_logsig_rom	16-bit fixed-point parallel logsig (i.e. sigmoid) function created using Xilinx LogiCORE <i>Single Port Block Memory v4.0</i>

* Based on VHDL source code donated by Steven Derrien (sderrien@irisa.fr) from *Institut de Recherche en Informatique et systèmes aleatoires (IRISA)* in France. In turn, Steven Derrien had originally created this through the adaptation of VHDL source code found at <http://flex.ee.uec.ac.jp/yamaoka/vhdl/index.html>.

ever, having complete control over the architecture's fine-grain design comes at the cost of additional design overhead for the engineer.

Many of the candidate arithmetic HDL designs described in Table 2.1 were created by the *Xilinx CORE Generator System*. This EDA tool helps an engineer parameterize ready-made Xilinx intellectual property (ip) designs (i.e. *LogiCOREs*), which are optimized for Xilinx FPGAs. For example, `uog_core_adder` was created using the Xilinx proprietary LogiCORE for an adder design.

Approximation of the logsig function in both floating-point and fixed-point precision, were implemented in hardware using separate lookup-table architectures. In particular, `active_func_sigmoid` was a modular HDL design, which encapsulated all the floating-point arithmetic units necessary to carry out calculation of logsig function. According to Equation 2.3, this would require the use of a multiplier, adder, divider, and exponential function. As a result, `active_func_sigmoid` was realized in VHDL using `uog_fp_mult`, `uog_fp_add`, a custom floating-point divider called `uog_fp_div`, and a *table-driven* floating-point exponential function created by Bui *et al* [44]. While this is not the most efficient implementation of the logsig, it allows implementing other transfer functions with min efforts (like Tan Hyperbolic) since it shares the basic functions with the Sigmoid. It is very common in training BP networks to test different transfer functions.

The `uog_logsig_rom` HDL design utilized a Xilinx LogiCORE to implement single port block memory. A lookup-table of 8192 entries was created with this memory, which was used to approximate the logsig function in fixed-point precision.

In order to maximize the processing density of the digital VLSI ANN design proposed in Section 2.3.1, only the most area-optimized arithmetic HDL designs offered in Table 2.1 should become part of the `uog_fp_arith` and `uog_fixed_arith` VHDL libraries. However, the space-area requirements of any VHDL design will vary from one FPGA architecture to the next. Therefore, all the HDL arithmetic designs found in Table 2.1 have to be implemented on the same FPGA as was targeted for implementation of the digital VLSI ANN design, in order to determine the most area-efficient arithmetic candidates.

2.4 Experiments using logical-XOR problem

The logical-XOR problem is a *classic* toy problem used to benchmark the learning ability of an ANN. It is a simple example of a non-linearly separable problem.

The minimum ANN *topology* (a *topology* includes the number of neurons, number of layers, and the layer interconnections (i.e. synapses)) required to

Table 2.2. Truth table for logical-XOR function

Inputs		Output
x_0	x_1	y
0	0	0
0	1	1
1	0	1
1	1	0

solve a non-linearly separable problem consisting of at least one hidden layer. An overview of the ANNs topology used in this particular application, which consists of only one hidden layer, is shown in Figure 2.3.

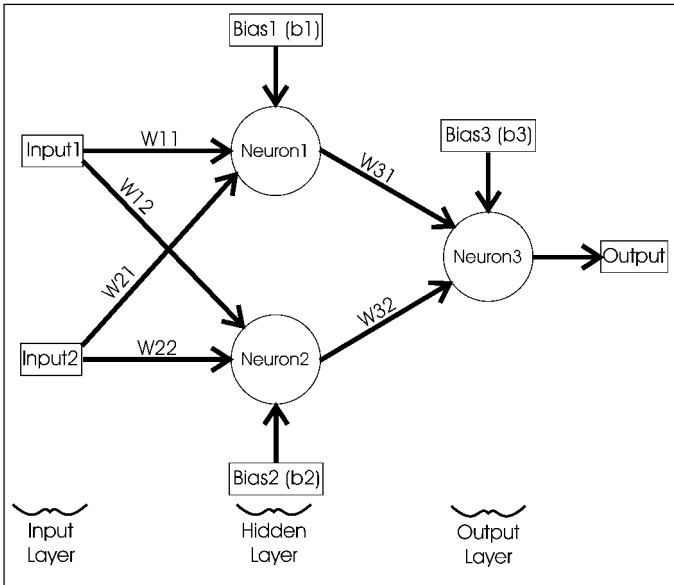


Figure 2.3. Topology of ANN used to solve logical-XOR problem

For each ANN implementation, a set of thirty training sessions were performed individually. Each training session lasted for a length of 5000 epoch, and used a learning rate of 0.3. Each of the training sessions in the set used slightly different initial conditions, in which all weights and biases were randomly generated with a mean of 0, and a standard deviation of ± 0.3 . Once generated, every BP implementation was tested using the same set of thirty training sessions. This way, the logical-XOR problem discussed acts as a common testing platform, used to benchmark the performance of all BP implementations.

Xilinx Foundation ISE 4.1i EDA tools were used to synthesize, and map (i.e. place and route) two variations of the FPGA-based ANN designs – one

using `uog_fp_arith` library, and one using `uog_fixed_arith` library. All experiments and simulations were carried out on a PC workstation running Windows NT (SP6) operating system, with 1 GB of memory and Intel PIII 733MHz CPU.

These circuit designs were tested and validated in simulation only, using ModelTech's ModelSim SE v5.5. *Functional* simulations were conducted to test the syntactical and semantical correctness of HDL designs, under ideal FPGA conditions (i.e. where no propagation delay exists). *Timing* simulations were carried out to validate the HDL design under non-ideal FPGA conditions, where propagation delays associated with the implementation as targeted on a particular FPGA are taken into consideration.

Specific to VHDL designs, timing simulations are realized using an IEEE standard called VITAL (VHDL Initiative Toward ASIC Libraries). VITAL libraries contain information used for modeling accurate timing of a particular FPGA at the gate level, as determined *a priori* by the respective FPGA manufacturer. These VITAL libraries are then used by HDL simulators, such as ModelSim SE, to validate designs during timing simulations.

A software implementation of a back-propagation algorithm was created using MS Visual C++ v6.0 IDE. The software simulator was set up to solve the logical-XOR problems using the topology shown in Figure 2.3. The purpose for creating this software simulator was to generate expected results for testing and validating FPGA-based BP. To speed up development and testing, two other software utilities were created to automate numeric format conversions—one for converting real decimal to/from IEEE-754 single precision floating-point hexadecimal format, and one for converting real decimal to/from 16-bit fixed-point binary format.

2.5 Results and discussion

We implemented the previous HDL designs on Xilinx FPGAs. The resulting space-time requirements for each arithmetic HDL design are summarized in Table 2.3. In order to maximize the neuron density of the FPGA-based MLP, the area of the various arithmetic HDL designs that a neuron is comprised of should be minimized. As a result, the focus here is to determine the most area-optimized arithmetic HDL designs for use in the implementations.

2.5.1 Comparison of Digital Arithmetic Hardware

Comparison of the different adder results, shown in Table 2.3, reveals that the three carry lookahead adders (i.e. `uog_std_adder`, `uog_core_adder`, and `uog_sch_adder`) require the least amount of area and are the fastest among all non-pipelined adders. Note that the sophistication of today's EDA tools have

allowed the VHDL-based designs for carry lookahead adders to achieve the same fine-grain efficiency of their equivalent schematic-based designs.

Since a carry lookahead adder is essentially a ripple-carry adder with additional logic as seen in Figure 2.4, it isn't immediately clear why a carry lookahead adder is shown here to use less area compared to a ripple-carry adder

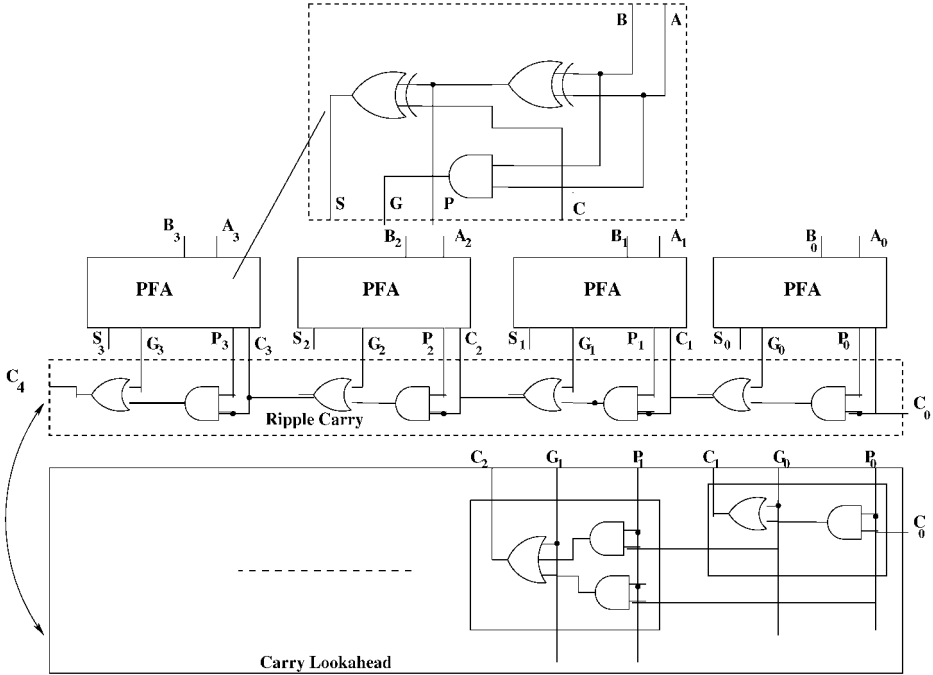


Figure 2.4. Carry Lookahead Adder

when implemented on a Xilinx Virtex-E FPGA. The carry lookahead design can be obtained [35] by a transformation of the ripple carry design in which the carry logic over fixed groups of bits of the adder is reduced to two-level logic. The carry lookahead adder would consist of (for example a 4-bit carry lookahead adder) four partial full adders (PFA) each consisting of two EXOR gates and an AND gate. The ripple carry logic (AND gate and OR gate for each bit) will be substituted with the Carry lookahead logic. Since the Virtex-II FPGA has built in (fast arithmetic functions) of look-ahead carry chains, the number of CLBS utilized by the carry lookahead adder will be equivalent or smaller than that of the ripple adder (i.e the extra logic used by the Carry Lookahead is free! since it is custom built within the FPGA). In addition, the Virtex-E CLBs dedicated fast lookahead logic enhances the performance of the

adder. As a result, it's best to use HDL adder designs which take advantage of the Virtex-E's fast carry lookahead logic.

The Virtex-E's fast carry-lookahead logic is again utilized to produce the best area-optimized subtractors (i.e. `uog_std_subtractor` and `uog_core_subtractor`), as well as, the best area-optimized multiplier (i.e. `uog_booth_multiplier`).

Only the most area-optimized arithmetic HDL designs discussed here were used in the construction of custom arithmetic HDL libraries, as listed in Table 2.4. In the case where there was more than one choice of best area-optimized arithmetic HDL design to choose from, *behavioral* VHDL designs were preferred because they promote high-level abstract designs and portability. For example, such was the case in selecting a fixed-point adder and subtractor for the `uog_fixed_arith` library.

Table 2.4 also reveals how much more area-optimized the individual fixed-point arithmetic HDL designs in `uog_fixed_arith` were compared to the floating-point arithmetic HDL designs in `uog_fp_arith`. Since a floating-point adder is essentially a fixed-point adder plus additional logic, not to mention the fact that floating-point uses more precision than fixed-point arithmetic, it's no surprise to find that the 16-bit fixed-point adder is much smaller than the 32-bit floating-point adder. Similar in nature is the case for subtractor and multiplier comparisons shown in Table 2.4.

The comparison of area-optimized logsig arithmetic HDL designs reveals that the 32-bit floating-point version is over 250 times bigger than the 16-bit fixed-point version. Aside from the difference in amount of precision used, the significant size difference between logsig implementations is due to the fact that floating-point implementation encapsulates a table-lookup architecture in addition to other *area-expensive* arithmetic units, while the fixed-point version *only* encapsulates a table-lookup via memory.

`Uog_fp_arith` and `uog_fixed_arith` have been clearly defined with only the best area-optimized components, as shown in Table 2.4. This will help to ensure that 32-bit floating-point and 16-bit fixed-point FPGA-based MLP implementations achieve a processing density advantage over the software-based MLP simulations. As was shown here, the larger area requirements of floating-point precision in FPGA-based ANNs makes it not nearly as feasible as fixed-point precision.

2.5.2 Comparison of ANN Implementations

Table 2.5 summarizes logical-XOR benchmark results for each of the following implementations with identical topology:

- 32-bit floating-point FPGA-based MLP, which utilizes `uog_fp_arith` library.

Table 2.3. Space/Time Req'ts of alternative designs considered for use in custom arithmetic VHDL libraries

HDL Design	Area (CLB)s	Max. Clock Rate (MHz)	Pipe-lining Used?	Clock cycles per calc.	Min. Time per calc. (ns)	Total
uog_fp_add	174	19.783	1-stage	2	101.096	(for first calc.)
uog_ripple_carry_adder	12	67.600	No	16	236.688	
uog_c_l_addr	12	34.134	No	1	29.296	
uog_std_adder	4.5	66.387	No	1	15.063	
uog_core_adder	4.5	65.863	No	1	15.183	
uog_sch_adder	4.5	72.119	No	1	13.866	
uog_pipe_adder	96	58.624	15-stage	16	272.928	
uog_fp_sub	174	19.783	1-stage	2	101.096	
uog_par_subtractor	8.5	54.704	No	1	18.280	
uog_std_subtractor	4.5	56.281	No	1	17.768	
uog_core_subtractor	4.5	60.983	No	1	16.398	
uog_fp_mult	183.5	18.069	1-stage	2	110.686	(for first calc.)
uog_booth_multiplier	28	50.992	No	34	668.474	
uog_std_multiplier	72	32.831	No	1	30.459	
uog_core_bs_mult	34	72.254	No	20	276.800	
uog_pipe_serial_mult	39	66.397	?-stage	21	316.281	(for first calc.)
uog_core_par_multiplier	80	33.913	No	1	29.487	
uog_pipe_par_mult	87.5	73.970	?-stage	2	27.038	(for first calc.)
active_func_sigmoid*	3013	1.980	No	56	29282.634	
uog_logsig_rom	12	31.594	No	1	31.652	

*Target platform used here was Xilinx Virtex-II FPGA (xc2v8000-5bf957)

Please note the following:

- 1 All fixed-point HDL designs use signed 2's complement arithmetic
- 2 Unless otherwise mentioned, all arithmetic functions were synthesized and implemented (i.e. place and route) under the following setup:

Target Platform: Xilinx Virtex-E FPGA (xcv2000e-6bg560)

Development Tool: Xilinx Foundation ISE 4.1i (SP2)

Synthesis Tool: FPGA Express VHDL

Optimization Goal: Area (Low Effort)

- 3 Max. Clock Rate is determined using the Xilinx Timing Analyzer on Post-Place and Route Static Timing of HDL design. $Max.ClockRate = \min\{(Min.CombinationalPathDelay)^{-1}, [(Min.InputArrivalTimeBeforeClk) + Max.OutputRequiredTimeBeforeClk]^{-1}\}$

Table 2.4. Area comparison of uog_fp_arith vs. uog_fixed_arith

Arithmetic Function	uog_fixed_arith HDL Design	uog_fp_arith HDL Design	Area Optimization (CLB/CLB)
Adder	uog_std_adder	uog_fp_add	38.66x smaller
Subtractor	uog_std_subtractor	uog_fp_sub	38.66x smaller
Multiplier	uog_booth_multiplier	uog_fp_mult	6.55x smaller
Logsig Function	uog_logsig_rom	activ_func_sigmoid	251.08x smaller

- 16-bit fixed-point FPGA-based MLP, which utilizes uog_fixed_arith library.
- software-based MLP simulations using C++.

Due to the relative difference in size of arithmetic components used, the fixed-point FPGA-based MLP is over 13 times smaller than the floating-point FPGA-based MLP. It can only be assumed that the area requirements for the software-based MLP implemented on an Intel PIII CPU (i.e. general-purpose computing platform) is infinitely big in comparison to the FPGA-based MLPs.

Of concern was the fact that timing simulations via ModelSim SE v5.5 required two weeks for floating-point and six days for fixed-point runs just to complete one training session in each. In general, any VLSI design which is not area-optimized may impede the design and test productivity.

The fact that all three MLP-BP implementations converged at all is enough to validate the successful design of each. Note that a MLP is not always guaranteed to converge since it may get trapped in local minima.

What's interesting about the convergence percentages given in Table 2.5 is that they're the same for the software-based and 32-bit FPGA-based MLPs, but not for the 16-bit FPGA-based MLPs. The software-based MLP and FPGA-based MLP that used uog_fp_arith achieved the same convergence percentages because they both use 32-bit floating-point calculations, and will follow identical paths of gradient descent when given the same initial MLP parameters. Due to the quantization errors found in 16-bit fixed-point calculations, its respective FPGA-based MLP will follow down a slightly different path of gradient descent when exposed to the same initial MLP parameters as the other two implementations.

In the context of MLP applications, reconfigurable computing looks to increase the neuron density above and beyond that of general-purpose computing. Due to the fact that three neurons exist in the MLP topology used to solve the logical-XOR problem, and based on the benchmarked speeds of back-propagation iteration for each particular MLP implementation, the processing density can be calculated for each. For MLP applications, processing density is realized as the number of weight updates per unit of space-time. As shown in

Table 2.5. Summary of logical-XOR ANN benchmarks on various platforms

XOR ANN Architecture	Precision	Total Area (CLBs, [Slices])*	% of Convergence in thirty trials**	Max. Clock Rate (MHz)
Xilinx Virtex-E xcv2000e FPGA	16-bit fixed-pt	1239 [2478]	100%	10
Xilinx Virtex-II xc2v8000 FPGA	32-bit floating-pt	8334.75 [33339]	73.3%	1.25
Intel Pentium III CPU	32-bit floating-pt	NA	73.3%	733
	Total Clock Cycles per Backprop Iteration	Backprop Iteration Period (μs)	Weight Updates per Sec (WUPS)	Processing Density (per Slice)
Xilinx Virtex-E xcv2000e FPGA	478	47.8	62762	25.33
Xilinx Virtex-II xc2v8000 FPGA	464	580	5172	0.1551
Intel Pentium III CPU	N/A	2045.15***	1466.89	NA

* Note Virtex-II CLB is over twice the size of Virtex-E CLB. Virtex-II CLB consists of 4 slices, whereas Virtex-E CLB consists of 2 slices.

** Convergence is defined here as less than 10% error in the ANN’s output, after it has been trained.

*** This is an average based on time taken to complete 200,000,000 iterations of the backpropagation algorithm for the software-based ANN. Microsoft Platform SDK multimedia timers were used, which had a resolution of 1ms.

Table 2.5, the relative processing density of the 16-bit fixed-point implementation is significantly higher than that of the 32-bit floating-point one. This reveals how a combination of minimum allowable range-precision and greater degree of area-optimization results in a direct impact on the processing density in implementation.

In addition to infinitely large area requirements, the software-based MLP was shown to be over 40x slower in comparison to the 16-bit fixed-point FPGA-based implementation. Therefore, it can only be assumed that the relative processing density of the software-based MLP is infinitely small in comparison to the other two implementations.

2.6 Conclusions

In general, we have shown that the choice of range-precision and arithmetic hardware architecture used in reconfigurable computing applications has a direct impact on the processing density achieved. A minimal allowable range-

precision of 16-bit fixed-point continues to provide the most optimal *range-precision vs. area trade-off* for MLP-BP implemented on today's FPGAs.

The *classic* logical-XOR problem was used as a common benchmark for comparing the performance of a software-based MLP, and two FPGA-based MLPs – one with 16-bit fixed-point precision, and the other with 32-bit floating-point precision. Despite the limited range-precision, the MLP with area-optimized fixed-point arithmetic managed to maintain the same quality of performance (i.e. in terms of the MLPs ability to learn) as demonstrated with floating-point arithmetic. Results showed that the fixed-point MLP-BP implementation was over 12x greater in speed, over 13x smaller in area, and achieved far greater processing density compared to the floating-point FPGA-based MLP-BP. Also, the processing density achieved by the FPGA-based MLP-BP with 16-bit fixed-point precision compared to the software-based MLP-BP best demonstrates the processing density advantage of reconfigurable computing over general-purpose computing for this particular application. As a result, floating-point precision is not as feasible as fixed-point in this type of application.

One disadvantage of using 16-bit fixed-pt, is that its limited range poses risk of saturation. Saturation adds error to a system, the extent of which is application dependent. The logical-XOR example demonstrated in this chapter still managed to achieve convergence, despite the saturation error caused by 16-bit fixed-pt with range [-8.0,8.0). Another important lesson to learn from this study is that the area savings of using 16-bit fixed-point rather than floating-point precision in a FPGA-based ANN help minimize simulation durations when validating HDL designs. The current performance rate of digital HDL simulators, like *ModelSim SE 5.5*, is an ongoing concern. Not only does the duration of timing simulations increase proportionally with the size of the circuit being simulated, but the magnitude of duration is in the order of 'days' and even 'weeks' for large VLSI HDL designs.

References

- [1] Xin Yao. Evolutionary Artificial Neural Networks, In: *Encyclopedia of Computer Science and Technology*, A. Kent and J. G. Williams, Eds., Vol. 33, Marcel Dekker Inc., New York, NY 10016, pp. 137-170, 1995
- [2] K. Balakrishnan and V. Honavar, Evolutionary Design of Neural Architectures – A Preliminary Taxonomy and Guide to Literature, Tech. Report no. CS TR95-01, Artificial Intelligence Research Group, Iowa State University, pp. January, 1995.
- [3] J. G. Eldredge, FPGA Density Enhancement of a Neural Network Through Run-Time Reconfiguration, Department of Electrical and Computer Engineering, Brigham Young University, pp. May, 1994.

- [4] J. G. Eldridge and B. L. Hutchings, Density Enhancement of a Neural Network using FPGAs and Run-Time Reconfiguration, In: *IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 180-188, 1994.
- [5] J. G. Eldridge and B. L. Hutchings, RRANN: A Hardware Implementation of the Backpropagation Algorithm Using Reconfigurable FPGAs, In: *Proceedings, IEEE International Conference on Neural Networks*, Orlando, FL, 1994.
- [6] J. D. Hadley and B. L. Hutchings. Design Methodologies for Partially Reconfigured Systems, In: *Proceedings, IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 78-84, 1995.
- [7] Hugo de Garis and Michael Korkin . The CAM-BRAIN MACHINE (CBM) An FPGA Based Hardware Tool which Evolves a 1000 Neuron Net Circuit Module in Seconds and Updates a 75 Million Neuron Artificial Brain for Real Time Robot Control, *Neurocomputing journal*, Vol. 42, Issue 1-4, 2002.
- [8] Amanda J. C. Sharkey, (Ed.). *Combining Artificial Neural Nets – Ensemble and Modular Multi-Net Systems*, Perspectives in Neural Computing, Springer-Verlag London Publishing, 1999.
- [9] Eric Ronco and Peter Gawthrop. Modular Neural Networks: a state of the art, Tech. Report, no. CSC-95026, Center for System and Control, University of Glasgow, Glasgow, UK, May 12, 1999.
- [10] Hugo de Garis and Felix Gers and Michael Korkin. CoDi-1Bit: A Simplified Cellular Automata Based Neuron Model, *Artificial Evolution Conference (AE97)*, Nimes, France, 1997
- [11] Andres Perez-Uribe. Structure-Adaptable Digital Neural Networks, Ph.D. Thesis, Logic Systems Laboratory, Computer Science Department, Swiss Federal Institute of Technology-Lausanne, 1999.
- [12] H. F. Restrepo and R. Hoffman and A. Perez-Uribe and C. Teuscher and E. Sanchez . A Networked FPGA-Based Hardware Implementation of a Neural Network Application. In: *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'00)*, pp. 337-338, 2000.
- [13] J.-L. Beuchat and J.-O. Haenni and E. Sanchez. Hardware Reconfigurable Neural Networks, 5th Reconfigurable Architectures Workshop (RAW'98), Orlando, Florida, USA, pp. March 30, 1998.
- [14] Aaron Ferrucci. ACME: A Field-Programmable Gate Array Implementation of a Self-Adapting and Scalable Connectionist Network, University of California, Santa Cruz, January, 1994.

- [15] Marcelo H. Martin. A Reconfigurable Hardware Accelerator for Back-Propagation Connectionist Classifiers, University of California, Santa Cruz, 1994.
- [16] Tomas Nordstrom. Highly Parallel Computers for Artificial Neural Networks, Ph.D. Thesis, Division of Computer Science and Engineering, Lulea University of Technology, Sweden, 1995.
- [17] T. Nordstrom and E. W. Davis and B. Svensson. Issues and Applications Driving Research in Non-Conforming Massively Parallel Processors, book. In: *Proceedings of the New Frontiers, a Workshop of Future Direction of Massively Parallel Processing*, 1992
- [18] T. Nordstrom and B. Svensson. Using and Designing Massively Parallel Computers for Artificial Neural Networks, *Journal of Parallel and Distributed Computing*, Vol. 14, 1992.
- [19] B. Svensson and T. Nordstrom and K. Nilsson and P.-A. Wiberg. Towards Modular, Massively Parallel Neural Computers, In: *Connectionism in a Broad Perspective: Selected Papers from the Swedish Conference on Connectionism - 1992*, L. F. Niklasson and M. B. Boden, Eds., Ellis Harwood, pp. 213-226, 1994.
- [20] Arne Linde and Tomas Nordstrom and Mikael Taveniku. Using FPGAs to implement a reconfigurable highly parallel computer. In: *Field-Programmable Gate Arrays: Architectures and Tools for Rapid Prototyping*, Springer-Verlag, Berlin, 1992.
- [21] T. Nordstrom. On-line Localized Learning Systems Part 1 - Model Description, Research Report, Division of Computer Science and Engineering, Lulea University of Technology, Sweden, 1995.
- [22] Tomas Nordstrom. On-line Localized Learning Systems Part II - Parallel Computer Implementation, Research Report, no. TULEA 1995:02, Division of Computer Science and Engineering, Lulea University of Technology, Sweden, 1995.
- [23] Tomas Nordstrom. Sparse distributed memory simulation on REMAP3, Research Report, no. TULEA 1991:16, Division of Computer Science and Engineering, Lulea University of Technology, Sweden, 1991.
- [24] T. Nordstrom. Designing Parallel Computers for Self Organizing Maps, In: *Proceedings of the 4th Swedish Workshop on Computer System Architecture (DSA-92)*, Linkoping, Sweden, January 13-15, 1992.
- [25] B. Svensson and T. Nordstrom. Execution of neural network algorithms on an array of bit-serial processors. In: *Proceedings, 10th International Conference on Pattern Recognition, Computer Architectures for Vision and Pattern Recognition*, Vol. II, pp. 501-505, 1990.

- [26] M. Skrbek. Fast Neural Network Implementation. In: *Neural Network World*, Elsevier, Vol. 9, no. No. 5, pp. 375-391, 1999.
- [27] Introducing the XC6200 FPGA Architecture: The First FPGA Architecture Optimized for Coprocessing in Embedded System Applications, *Xcell*, Xilinx Inc., No. 18 : Third Quarter, pp. 22-23, 1995, url = <http://www.xilinx.com/apps/6200.htm>.
- [28] Xilinx. XC6200 Field Programmable Gate Arrays, Data Sheet, Version 1.7, 1996.
- [29] Mikael Taveniku and Arne Linde. A reconfigurable SIMD computer for artificial neural networks, Licentiate Thesis, Department of Computer Engineering, Chalmers University of Technology, Goteborg, Sweden, 1995.
- [30] Gate Count Capacity Metrics for FPGAs, Application Note, no. XAPP 059 - Version 1.1, Xilinx, Inc., Feb. 1, 1997, URL = <http://www.xilinx.com/xapp/xapp059.pdf>.
- [31] FLEX 10K Embedded Programmable Logic Device Family, Data Sheet, Version 4.1, Altera, Inc., March, 2001, URL = <http://www.altera.com/literature/ds/dsf10k.pdf>
- [32] XC3000 Series Field Programmable Gate Arrays, Product Description, Version 3.1, Xilinx, Inc., November 9, 1998, URL = <http://www.xilinx.com/partinfo/3000.pdf>
- [33] XC4000XLA/XV Field Programmable Gate Arrays, Product Specification, no. DS015 - Version 1.3, Xilinx, Inc., October 18, 1999, URL = <http://www.xilinx.com/partinfo/ds015.pdf>
- [34] Andres Perez-Urbe and Eduardo Sanchez. Speeding-Up Adaptive Heuristic Critic Learning with FPGA-Based Unsupervised Clustering, In: *Proceedings of the IEEE International Conference on Evolutionary Computation ICEC'97*, pp. 685-689, 1997.
- [35] M. Morris Mano and Charles R. Kime. *Logic And Computer Design Fundamentals*, Prentice Hall Inc., New Jersey, USA, 2000.
- [36] Andre Dehon. The Density Advantage of Configurable Computing, *IEEE Computer*, vol. 33, no. 5, pp. 41-49, 2000.
- [37] David E Rumelhart and James L McClelland and PDP Research Group. *Parallel Distrubuted Processing: Explorations in the Microstructure of Cognition*, vol. Volume 1: Foundations, MIT Press, Cambridge, Massachusetts, 1986.
- [38] Simon Haykin, *Neural Networks: A Comprehensive Foundation*, Prentice-Hall, Englewood Cliffs, New Jersey, 1999.

- [39] Stephen D. Brown and Robert J. Francis and Jonathan Rose and Zvonko G. Vranesic, *Field-Programmable Gate Arrays*, Kluwer Academic Publishers, USA, 1992
- [40] Virtex-E 1.8 V Field Programmable Gate Arrays, Preliminary Product Specification, no. DS022-2 (v2.3), Xilinx, Inc., November 9, 2001, URL = <http://www.xilinx.com/partinfo/ds022-2.pdf>
- [41] Jordan L Holt and Thomas E Baker. Backpropagation simulations using limited precision calculations. In: *Proceedings, International Joint Conference on Neural Networks (IJCNN-91)*, vol. 2, Seattle, WA, USA, pp. 121 - 126, 1991.
- [42] Hikawa Hiroomi. Frequency-Based Multilayer Neural Network with On-chip Learning and Enhanced Neuron Characteristics. *IEEE Transactions on Neural Networks*, vol. 10, no. 3, pp. 545-553, May, 1999.
- [43] Peter J. Ashenden. VHDL Standards. *IEEE Design & Test of Computers*, vol. 18, no. 6, pp. 122-123, September–October, 2001.
- [44] Hung Tien Bui and Bashar Khalaf and Sofiene Tahar. Table-Driven Floating-Point Exponential Function, Technical Report, Concordia University, Department of Computer Engineering, October, 1998, URL = <http://www.ece.concordia.ca/tahar/pub/FPE-TR98.ps>
- [45] W.B. Ligon III and S. McMillan and G. Monn and K. Schoonover and F. Stivers and K.D. Underwood. A Re-evaluation of the Practicality of Floating Point Operations on FPGAs. In: *Proceedings, IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 206–215, 1998.
- [46] Pete Hardee. System C: a realistic SoC debug strategy, *EETimes*, 2001.
- [47] G.M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities, In: *AFIPS Conference Proceedings*, vol. 30, AFIPS Press, Reston, Va., pp. 483–485, 1967.
- [48] Stephen Chappell and Chris Sullivan, Celoxica Ltd. Oxford UK. Handel-C for co-processing & co-design of Field Programmable System on Chip FPSoC, 2002, url = www.celoxica.com/technical-library/
- [49] Synopsys, Inc. Describing Synthesizable RTL in SystemC v1.1, Synopsys, Inc., January, 2002.
- [50] Martyn Edwards. Software Acceleration Using Coprocessors: Is it Worth the Effort?, In: *Proceedings, 5th International Workshop on Hardware/Software Co-design Codes/CASHE'97*, Braunschweig, Germany, pp. 135–139, March 24-26, 1997.
- [51] Giovanni De Micheli and Rajesh K. Gupta. Hardware/Software Co-design. *Proceedings of the IEEE*, vol. 85, no. 3, pp. 349-365, March, 1997.

- [52] John Sanguinetti and David Pursley. High-Level Modeling and Hardware Implementation with General- Purpose Languages and High-level Synthesis, White Paper, Forte Design Systems, 2002.
- [53] Don Davis. Architectural Synthesis: Unleashing the Power of FPGA System-Level Design. *Xcell Journal*, Xilinx Inc., no. 44, vol. 2, pp. 30-34, pp. Winter, 2002.
- [54] Xilinx, XAPP467 Using Embedded Multipliers in Spartan-3 FPGAs, Xilinx Application Note, May 13, 2003, <http://www.xilinx.com>.
- [55] Nestor Inc. Neural Network Chips, <http://www.nestor.com>.



<http://www.springer.com/978-0-387-28485-9>

FPGA Implementations of Neural Networks

Omondi, A.R.; Rajapakse, J.C. (Eds.)

2006, XII, 360 p., Hardcover

ISBN: 978-0-387-28485-9