UNIVERSITY OF CINCINNATI

DATE: August 06, 2003

I, Bharat Kishore Bharkhada ,
**hereby submit this as part of the requirements for the
degree of:**

Master of Science

in:

Computer Engineering

It is entitled:

Efficient FPGA Implementation of a Generic Function Approximator
and its Application to Neural Net Computation

Approved by:

Dr. Carla Purdy

Dr. Karen Tomko

Dr. James Hauser

Dr. Robert Ewing

Dr. Hoda Abdel-Aty-Zohdy

Efficient FPGA Implementation of a Generic Function Approximator and its Application to Neural Net Computation

A thesis submitted to the

Division of Research and Advanced Studies
of the
University of Cincinnati

in partial fulfillment of the
requirements for the degree of

Master of Science

in the Department of
Electrical and Computer Engineering and
Computer Science
of the
College of Engineering

August 06, 2003

by

Bharat Kishore Bharkhada

B.E. (Electronics), Shah and Anchor Kutchhi Engineering College
University of Mumbai, India, 1998

Thesis Advisor and Committee Chair: Dr. Carla Purdy

Abstract

Every neuron or processing element of an Artificial Neural Network (ANN) implementation requires an activation function. In the digital implementation of an ANN, the activation function is most widely implemented as a lookup table, and the accuracy of the output of the processing element depends partially on the number of words stored in the lookup table. The disadvantage with this approach is that, to achieve better results, the lookup table goes on increasing in size and often becomes large and unwieldy. In this thesis we explore the possibility of replacing the lookup table with direct computation of the activation function. As a case study, we compute the widely used activation function, the sigmoid function, by using a generic third order polynomial evaluator which is based on prior work addressing the problem of efficient function approximation for a system on a chip. The sigmoid function is approximated with a set of polynomials with integer coefficients. This approach results in a huge reduction in the memory required by the processing element. Booth encoding and Wallace tree techniques are used to further optimize the computation. The project is implemented in VHDL. The synthesis of the implementation and architectural simulations including place and route and timing analysis are carried using the Altera FLEX 10K family of devices. Synthesis and architectural simulations are also done in Synopsys. We show that our implementation achieves a large saving in memory and equivalent accuracy when compared with the lookup table approach, at the cost of some extra logic and execution time. We also show that our implementation achieves higher accuracy than other sigmoid approximations proposed in the literature. The current implementation can also generate a programmable sigmoid function and would allow, with minor changes and no increase in memory requirements, two to four different choices of slope for the sigmoid itself.

Dedicated to my Mother

Acknowledgements

I convey my gratitude and sincere thanks to Dr. Carla Purdy, without whose wise help and counsel, this thesis would not have been completed. Deep appreciation goes to Dr. James Hauser for his assistance with the various aspects of the thesis and for participating in my thesis committee. I am also grateful to Dr. Karen Tomko, Dr. Robert Ewing and Dr. Hoda Abdel-Aty-Zohdy for taking time to participate in my thesis committee.

I am immeasurably indebted to Mr. Eugene Rutz, Vandana Iyer, Yash Patil, Shailesh Parekh and Amar Yashlaha for supporting me in every possible way to achieve my goals. I am profoundly thankful to Thomas Curtis, Teresa Hamad, Catherine Rafter, Teri Ambrosius and Janet Luegering for their precious time and support and being a family to me away from home.

I sincerely thank Mahesh Suvarna, Jaspreet Singh Khambay, Rajsekhar Aikat, Pawan Thampi and Shalini Batra for their help, encouragement and support. Thanks are also due to Sandeep Bhatt, Asmeen Jahan, Gagan Jain, Srinivas Rayaprolu and many other friends who have made this a memorable phase in my life.

Finally and most importantly I wholeheartedly thank my family for their love. They were always present to support, encourage and motivate me and help me reach this stage. I am blessed to have them and no words are enough to express my gratitude towards them.

Contents

1. Introduction

1.1 Motivation for the Research	1
1.2 Goals of the Research	2
1.3 Research Methodology	3
1.4 Outline of the Thesis	3

2. Overview of the Technologies Used

2.1 Feed Forward Neural Networks	4
2.2 Genetic Algorithms	8
2.3 Reconfigurable Computing	10

3. Target Implementations of Feed-Forward Neural Networks

3.1 Reconfigurable Computing for Neural Networks	12
3.2 Altera SRAM-based FPGA	13
3.3 FPGA Advantages for Feed-Forward Neural Networks	17
3.4 Synopsys	19

4. Options for Implementing Sigmoid Activation Function

4.1 Dataflow Implementation	21
4.2 Lookup Table Method	23
4.3 Taylor Series Expansion	24
4.4 Least Square Technique	25
4.4.1 Least Square Approximation Technique 1	25
4.4.2 Least Square Approximation Technique 2	27

4.5 Piecewise Approximation	29
4.5.1 Piecewise Approximation Technique 1	29
4.5.2 Piecewise Approximation Technique 2	32
4.5.3 Piecewise Approximation Technique 3	35
4.6 Analog Implementation	36

5. Design, Architecture and Implementation for the Genetic

Algorithm Approximation

5.1 Genetic Algorithm Approximation	40
5.2 Booth Algorithm	43
5.2.1 Introduction to Booth Encoding	43
5.2.2 Modified Booth Algorithm	44
5.3 The Multiplier	47
5.3.1 Compressor	51
5.3.2 Wallace Tree	51
5.3.3 Carry Propagate Adder	54
5.4 Architecture Modeling	55
5.5 Sigmoid Function Implementation	60
5.5.1 Programmable Sigmoid Function Implementation	61

6. Results and Comparisons

6.1 Error Comparison for Different Methods	62
6.2 Hardware Usage and Timing Estimate	72
6.2.1 Dataflow Implementation	74
6.2.2 Lookup Table Method	74

6.2.3	Taylor Series Approximation	74
6.2.4	Least Square Approximation Method 1	75
6.2.5	Least Square Approximation Method 2	75
6.2.6	Piecewise Approximation 1	76
6.2.7	Piecewise Approximation 2	76
6.2.8	Piecewise Approximation 3	77
6.2.9	Analog Implementation	77
6.2.10	Genetic Algorithm Method	77
6.3	Practical Considerations	80
7. Conclusions and Future Work		
7.1	Conclusions	82
7.2	Future Work	83
Bibliography		84

List of Tables

3.1	FLEX10K device summary	13
4.1	Piecewise linear activation function truth table for positive values of input	30
5.1	Breakpoint and coefficients generated by the genetic algorithm	41
5.2	Bit pair recoding scheme used in modified Booth algorithm	45
5.3	Output generated by row partial product generator	50
5.4	Instruction set architecture	58
6.1	Maximum error and percentage error of different implementations	62
6.2	Number of cycles intervals required by different methods	78
6.3	Resources required to implement the genetic algorithm approximation method	78
6.4	Summary of resources required by processing element implemented in [26]	79
6.5	Summary of resources required by the processing element in [26] with the lookup ... table replaced with genetic algorithm sigmoid function approximation	79

List of Figures

2.1 General schematic of a feed-forward neural network	4
2.2 System architecture of LFFNN	5
3.1 EAB memory configurations	16
3.2 Examples of combining EABs	16
4.1 Translation of the sigmoid function into a C-like code	22
4.2 Sigmoid function and dataflow approximation	22
4.3 The sigmoid activation function circuit	23
4.4 Sigmoid function and Taylor series approximation	24
4.5 Sigmoid function and approximation using least square approximation technique 1 ...	26
4.6 Block diagram of least square approximation technique 2	28
4.7 Sigmoid function and approximation using least square approximation technique 2...	28
4.8 Plot of piecewise approximation technique 1 and sigmoid curve	30
4.9 Implementation of table 4.1	31
4.10 Implementation of the complete curve	32
4.11 The single approximating segments	32
4.12 Sigmoid function and approximation using piecewise approximation technique 2	34
4.13 Architecture for the piecewise linear approximation	35
4.14 Sigmoid function and approximation using piecewise approximation technique 3	36
4.15 Sigmoid function implementation with transistors	37
4.16 Simulated neuron activation function and its derivative compared with fitted	38
sigmoid and derivative of simulated neuron activation function	
4.17 Relative errors	39

5.1 Sigmoid function and genetic algorithm approximation in first quadrant	42
5.2 Sigmoid function and genetic algorithm approximation in third quadrant	43
5.3 Example of bit pair recoding	44
5.4 Multiplication of 3 x 29 using the conventional shift and add method	46
5.5 Multiplication of 3 x 29 using Booth encoding method	46
5.6 Multiplication of 3 x 29 using modified Booth encoding technique requiring	47
only $n/2$ partial products.	
5.7 Block diagram of the multiplier	48
5.8 Booth encoder	49
5.9 Partial product generator	49
5.10 Sign extender	50
5.11 Compressor	51
5.12 Wallace tree	52
5.13 4-2 compressor	53
5.14 9-2 compressor	54
5.15 System architecture of LFFNN	55
6.1 Errors of all methods	63
6.2 Error between genetic algorithm approximation and Taylor series	64
approximation [14] in the range $0 \leq x \leq 4$	
6.3 Error between genetic algorithm approximation and Taylor series	64
approximation [14] in the range $4 \leq x \leq 8$	
6.4 Error between sigmoid function and dataflow approximation	65
6.5 Percentage error between sigmoid function and dataflow approximation	65
6.6 Error between sigmoid function and Taylor series approximation	66

6.7	Percentage error between sigmoid function and Taylor series approximation	66
6.8	Error between sigmoid function and least square approximation method 1.....	67
6.9	Percentage error between sigmoid function and least square approximation	67
	method 1	
6.10	Error between sigmoid function and least square approximation method 2	68
6.11	Percentage error between sigmoid function and least square approximation	68
	method 2	
6.12	Error between sigmoid function and piecewise method 1 approximation	69
6.13	Percentage error between sigmoid function and piecewise method 1	69
	approximation	
6.14	Error between sigmoid function and piecewise method 2 approximation	70
6.15	Percentage error between sigmoid function and piecewise method 2	70
	approximation	
6.16	Error between sigmoid function and piecewise method 3 approximation	71
6.17	Percentage error between sigmoid function and piecewise method 3	71
	approximation	
6.18	Error between sigmoid function and genetic algorithm method approximation	72
6.19	Percentage error between sigmoid function and genetic algorithm method	72
	approximation	
6.20	Relative errors of analog implementation	73

Chapter 1

Introduction

1.1 Motivation For the Research

Neural networks have the ability to learn many examples of inputs and desired outputs, make generalizations about how outputs relate to inputs, apply these generalizations to new inputs, and make predictions. People have basically the same ability. However, neural networks have the advantage that they can make better generalizations than humans in some cases, e.g., classification and approximation [11]. For this reason, neural networks have been able to replace people in these types of applications. Both, analog and digital implementations of a neural network are possible. Analog implementation has the advantage of high accuracy and flexibility to store both fixed and adaptive weights. Generally it is more efficient than a digital implementation. This efficiency can be expressed in terms of computational density, computation time or energy efficiency. But it also has some disadvantages such as thermal instability and limited precision and it is difficult to mass-produce chips with predetermined weight matrices [13]. Exotic analog designs have a much higher complexity, and thus cost, than digital ones. Digital neural networks allow great flexibility in trading area for time. Many tools are available for digital neural network design. But digital neural networks have an obvious drawback - too slow training procedure for a complex application. Their software implementation on a sequential computing device is far from satisfying real-time requirements. This shortcoming limits their application range. Because of rapid advances in the VLSI industry, it becomes possible to implement dedicated parallel computing systems (neural processors) to speed up the intensive computation by taking advantage of a neural network's inherent

parallelism with reasonable cost.

System design and implementation, especially for parallel systems, is not an easy thing. Emerging technologies like co-design and reconfigurable computing would shorten the design period. Some of these helpful technologies have been put to use in the design of hardware-based neural networks. Our work can be used in conjunction with co-design and reconfigurable computing to improve the design of hardware-based digital neural networks.

1.2 Goals of the Research

Before emergence of reconfigurable computing, due to low ratio of performance to cost, the dedicated hardware for some specific algorithm was not an optimal choice, although it might achieve the highest performance. In the past, many researchers devoted their efforts to looking for generic parallel architectures for neural computing to balance performance and cost. With dynamic reusability of reconfigurable computing, programmable capability is brought into the hardware domain also. Therefore the research on dedicated parallel architectures on reconfigurable devices has received more attention in the recent past. Our goal is to do some research to complement the research on parallel architectures and neural computing. The activation function for the neural networks has traditionally been implemented as a lookup table. The implementation of the activation function as a lookup table requires a very large amount of memory. Our aim is to implement a generic function approximator in hardware and reduce the hardware memory requirement. We analyze the increase in the computational logic required to implement this activation function completely in hardware. The effects of this on the speed and accuracy of the computation are also observed.

1.3 Research Methodology

We use as reference a previous work on implementing neural nets and their processing elements in hardware [26]. We implement a hardware polynomial evaluator [9], which replaces the sigmoid activation function lookup table. We optimize the polynomial evaluator by using Booth encoding [6] and Wallace tree multiplication techniques [20]. The comparative results on memory savings, hardware increase, accuracy and speed are presented.

1.4 Outline of the Thesis

An overview of the different technologies used is provided in chapter 2. In chapter 3, a general description of reconfigurable computing and its advantages for feed-forward neural networks are discussed and an outline of the features of the Altera and Synopsys design analyzers is given. Chapter 4 describes the different methods being currently used for the hardware implementation of the sigmoid function. Chapter 5 presents our design, architecture and implementation of the sigmoid function evaluator and the neural network. Chapter 6 presents our evaluation results and provides a comparison between the different methods used for sigmoid function implementation. Conclusions and future work are given in chapter 7.

Chapter 2

Overview of the Technologies Used

2.1 Feed Forward Neural Networks

Figure 2.1. General schematic of a feed-forward neural network [11].

A neural network is a computational model that is based on the neuron cell structure of the biological nervous system. Given a training set of data, the neural network can learn the data with a learning algorithm. Neural architectures can be classified as feed forward, feed back, and self organizing, but the most well known and widely used class is the feed forward neural networks. The popularity of feed-forward networks derives from the fact that they have been applied successfully to a wide range of information processing tasks in such diverse fields as speech recognition, financial prediction, image compression, medical diagnosis and protein structure prediction [15]. Figure 2.1 shows a general schematic of a feed-forward neural network. It is a parallel-distributed information processing structure consisting of processing elements (neurons or layers) interconnected via unidirectional signal channels called connections or weights. Each processing element (PE) has a single output connection that branches into as many

collateral connections as desired; each carries the same signal - the processing element output signal. The processing element output signal can be of any mathematical type desired. The information processing that goes on within each processing element can be defined arbitrarily with the restriction that it must be completely local; that is, it must depend only on the current values of the input signals arriving at the processing element via incoming connections and on values stored in the processing element's local memory. A neural network can have any number of layers, units per layer, network inputs, and network outputs [11].

Figure 2.2 presents the system architecture of a lateral feed forward neural network (LFFNN) implemented in hardware [26] and which we base our work on. This network architecture, with lateral connections between neighboring neurons in the hidden layer to speed up network convergence during training, is discussed in [12]. The neural network in Figure 2.2 contains

- A global control unit, which consists of control processor, instruction memory, and data memory.

Figure 2.2. System architecture of LFFNN [26].

- A hidden layer PE array with dual ring topology connected to each two adjacent PEs, in which each hidden layer neuron is mapped into a single PE.
- An output layer PE array, where each output layer neuron is mapped into one PE. Internal broadcast buses make connections between the control unit and PE arrays.
- External host interface which makes connections between the host and the application specific hardware accelerator.

Feed forward neural networks (FFNNs) trained by a back propagation algorithm are the most popular FFNNs. They are able to solve problems, which are hard to solve using traditional approaches, very effectively for some applications [15]. FFNNs using back propagation algorithm have been employed in a wide variety of useful applications. However, there are two reasons that FFNNs require a lot of computing power.

- a) The learning algorithms themselves require powerful computation due to a large number of existing neurons and slow learning rate. FFNNs usually work adequately on small problems but can run into trouble due to mass computation when they are scaled up to problems involving large amounts of input data such as visual pattern matching and speech recognition. FFNN machines have to be idled for long hours for training these examples [11].
- b) While it can be proved theoretically that a net whose output is a linear combination of n sigmoids of linear combinations of its inputs can uniformly approximate any continuous function [11], it is difficult to characterize the class of problems which are solvable in practice. The theoretical result also makes no claim about what procedure to follow for finding good solutions. So the designer's approach is always partly empirical. It is

necessary to modify the number of neurons or change the algorithm. Each time the designer has to restart everything that has been done before. Such a procedure can be unacceptably slow.

There are three ways to enhance performance of FFNNs:

a) Make the single computing device run faster

Currently the Von Neumann model (a sequential programming model) dominates the computing area. This model is made up of three basic elements – uniprocessor, memory, and input/output device connected with buses. The product of this model is the sequential computer. Computer performance has improved dramatically over time due to improvements in these basic elements (primarily driven by IC technology) and architecture refinements (cache, pipeline, virtual memory, superscalar, etc.). The current sequential computer performance still does not meet the fast processing requirements of FFNNs. Moreover, this sequential computing mechanism wastes the intrinsic parallelism of FFNNs or exploits it only up to a very moderate extent (e.g. instruction level parallelism in pipelined RISC processors).

b) Use better algorithms

The lateral feed forward neural networks (lateral FFNNs or LFFNNS, [12]) are one of the better alternatives to speed up the tedious training. This is the type of network used in this project.

c) Use Concurrent Computing

Obviously parallel processing can achieve significant additional performance gains due to intrinsic parallelism of FFNNs. Architectures suitable for the characteristic features of the FFNN need to be developed. The support from hardware and software is indispensable.

For many applications, the execution speed provided by a software algorithm simulation running on a sequential, general-purpose computer is sufficient if we can endure the tedious training procedure. But this is not true for all applications. Moreover, there is only limited speedup based on fast uniprocessor and optimal algorithms. The huge parallel potential still hasn't been exploited for FFNNs. For real time applications of FFNNs, the effort for parallel speedup becomes a must. As VLSI technologies advance, hardware cost has reduced dramatically. Developing an extensively parallel neural computing system is now a real possibility.

2.2 Genetic Algorithms

Genetic Algorithms (GA) are heuristic methods that may be used to solve search and optimization problems. They are based on Darwin's Theory of Evolution, which emphasizes that populations evolve according to the principles of natural selection and "survival of the fittest". In particular, "GAs combine survival of the fittest among string structures with a structured yet randomized information exchange to form a search algorithm with some of the innovative flair of human search" [9]. By starting with a set of potential solutions and changing them during several iterations the GA hopes to converge to the most 'fit' solution [7].

The process begins with a set of potential solutions or chromosomes (usually in the form of bit strings) that are randomly generated or selected. The entire set of these chromosomes comprises a population. The chromosomes evolve during several iterations or generations. New generations (offspring) are generated using the crossover and mutation techniques. The crossover technique involves splitting two chromosomes and then combining one section of each chromosome with the complementary section of the other, while the mutation technique involves flipping a single bit of a chromosome. The chromosomes are then evaluated using a certain fitness criterion and

the better ones are kept while the others are discarded. This process is repeated until some chromosomes have the best fitness and this is taken as the best solution of the problem. The process can also be ended if either a predetermined number of allowed iterations are reached or there has been no improvement over a certain predetermined succession of iterations [7].

The GAs work well for global optimization, especially where the objective function is discontinuous or with several local minima. This advantage also has its potential disadvantage: since it does not use extra information such as gradients, the GA can have a slow convergence rate on well-behaved objective functions. GAs can be used in both unconstrained and constrained optimization problems. They can be applied to nonlinear programming, stochastic programming (the case where mathematical programming problems have random variables, thus introducing a degree of uncertainty), and combinatorial optimization problems such as the Traveling Salesman Problem, Knapsack Problem, Minimum Spanning Tree Problem, Scheduling Problem, and many others [9].

In reference to our work, the genetic algorithm, a type of heuristic, can be implemented to determine the integer coefficients for an interpolating polynomial of requested degree for the evaluation of the sigmoid activation function. For example, the coefficients of the third degree polynomial: $Ax^3 + Bx^2 + Cx + D$, can be determined such that the difference between the polynomial and the sigmoid function, computed in terms of the error at the training points, is minimized. In other words, the genetic algorithm can be constructed to search the restricted solution space to find the optimal coefficient set.

2.3 Reconfigurable Computing

In recent years, reconfigurable architecture has increasingly replaced the use of ASICs (Application Specific Integrated Circuits) to obtain computational speed-ups. This approach has blurred the boundaries between software and hardware, where hardware has traditionally been ‘fixed’ and software has been the ‘flexible’ component. Reconfigurable computing explores hardware / software solutions where the underlying hardware can be modified at runtime (under software control) to meet the needs of the application [26].

Field Programmable Gate Arrays (FPGAs) make reconfigurable computing possible. They are VLSI chips whose hardware functionality is user-programmable. Putting FPGAs on a workstation/PC card or motherboard allows FPGAs to serve as compute intensive co-processors. Unlike a ‘fixed’ floating-point co-processor, these FPGAs can be reconfigured over and over again, to perform any type of operation. This enables application-specific, dynamically ‘programmable’ hardware accelerators. Fast prototyping and reusability are general FPGA advantages that any application can exploit.

With the commercial availability of large reconfigurable logic devices (Altera[2], Xilinx[25], etc.) and the evolution of EDA (Electronic Design Automation) tools (Synopsys[24], etc.), it has become feasible to build fairly large and powerful systems based on reconfigurable logic. At the heart of these computing systems is the SRAM-based FPGA, an integrated circuit that consists of a large uncommitted array of programmable logic and programmable interconnect that can be easily configured and reconfigured by the end user to implement a wide range of digital circuits.

Usually, a re-programmable system is not viewed as an isolated entity, but instead, as an embedded system – an extension of a computer system. Specifically re-programmable logic resources can be added to a workstation/PC as attached processing units, greatly increasing the

processing power for some applications. These processing units are built up with multiple FPGAs and perhaps memories and other devices.

The programming methodologies of reconfigurable systems differ from the method in which general-purpose computers are programmed. In a general-purpose system, the microprocessor executes the whole program on the general-purpose computer. In a reconfigurable system, parts of the program may be modeled as hardware and mapped on to the reconfigurable hardware. The rest of the program executes as software [26].

Reconfigurable computing devices, hardware equivalents of software programs, need to be generated so that they can be downloaded into the FPGAs for execution. Usually this calls for knowledge of EDA tools and hardware design, which the general programming community does not possess. The availability of increasingly complex and higher capacity FPGAs is allowing designers to implement larger and more complex systems to meet time-to-market. Digital design can be described in HDL (Hardware Description Language – VHDL [3] or Verilog [19]), and high – level synthesis tools can then convert the technology-independent behavioral specification of the digital system into a hardware implementation.

Chapter 3

Target Implementations of Feed-Forward Neural Networks

3.1 Reconfigurable Computing for Neural Networks

Neural networks are adaptive systems that are required to adapt both on the architectural level and on the algorithmic level in response to environmental stimuli. The computational nature of neural networks is also inherently parallel. Traditionally, neural networks are trained and simulated in software, an approach that can be very slow. To apply neural networks for problem solving in real-time environments, the inherent parallelism of neural networks must be fully exploited while retaining sufficient flexibility in the implementation to facilitate the required adaptation. One approach is to implement neural networks in FPGA-based reconfigurable hardware. As the criterion for success, the implementation must be flexible enough to handle on-chip adaptations, not only for the weights, but also for the number of neurons of the neural network being trained.

Feed-forward neural networks can exploit the fact that SRAM based FPGAs may be reconfigured many times. This allows different topologies and learning algorithms to be implemented on the same FPGAs. Also, neural network algorithms (including back-propagation) may let themselves do run-time reconfiguration, where the FPGA is reconfigured during execution of the algorithm. Thus FPGAs are excellent technology manifestations for implementing hardware based feed-forward neural networks.

3.2 Altera SRAM-based FPGA

We used Altera SRAM-based FLEX 10K FPGAs in this project; hence the introduction focuses only on Altera's FLEX 10K family [2]. This description only covers basic, very limited content about FLEX 10K family, which applies to our project (See also Table 3.1). For more detailed information, Altera's website [2] provides its comprehensive datasheet.

Feature	Available Gates	LABs	EABs	RAM bits
FLEX10K10	10,000	72	3	6K
FLEX10K20	20,000	144	6	12K
FLEX10K30	30,000	216	6	12K
FLEX10K40	40,000	288	8	16K
FLEX10K50	50,000	360	10	20K
FLEX10K70	70,000	468	9	18K
FLEX10K100	100,000	624	12	24K
FLEX10K130	130,000	832	16	32K
FLEX10K250	250,000	1,520	20	40K

Table 3.1. FLEX10K device summary.

FLEX 10K devices are based on a reconfigurable CMOS SRAM FPGA. Each FLEX 10K device contains embedded arrays and logic arrays.

- The embedded array is used to implement memory. The embedded array consists of a series of EABs (Embedded Array Blocks). Each EAB provides 2,048 bits, which can be used to create RAM, ROM, dual-port RAM, or FIFO functions. EABs can be used independently, or multiple EABs can be combined to implement larger memory.

- The logic array performs complex logic functions. The logic array consists of a series of LABs (Logic Array Blocks). Each LAB contains eight LEs (Logic Elements) each of which consists of a 4 input LUT (Look up Table), a programmable flip-flop, and dedicated signal paths for carry and cascade functions. The eight LEs can be used to create medium-sized blocks of logic, 8 bit counters, address decoders, or state machines – or combined across LABs to create larger logic blocks. Each LAB represents about 96 usable gates of logic.

The combination of embedded and logic arrays provides the high performance and high density of embedded gate arrays, enabling designers to implement an entire system on a single device.

Here are some features in the FLEX 10K family [2]:

- System on a programmable chip integration
 - Embedded array for implementing mega-functions, such as memory and specialized functions.
 - Logic array for general logic functions
- High density
 - 10,000 to 250,000 gates
 - Up to 40 K RAM bits; 2K bits per EAB
- Support of PCI local bus specification
- Built-in JTAG (Joint Test Action Group) boundary-scan test circuitry available without consuming any device logic
- Less than 320 ms reconfiguration execution time.

The following covers more detailed information about the embedded array. We need to know this information when we implement distributed local memories for storing neural weights in the embedded array.

The EAB provides a few advantages over FPGAs which implement on-board RAM as arrays of small, distributed RAM blocks. These FPGA RAM blocks contain delays that are less predictable as the size of the RAM increases. In addition, FPGA RAM blocks are prone to routing problems because small blocks of RAM must be connected together to make larger blocks. Thus implementing memory is very inefficient in a logic array. In contrast, EABs can be used to implement large, dedicated blocks of RAM that eliminate these timing and routing concerns. EABs can be used to implement synchronous RAM, which is easier to use than asynchronous RAM. A circuit using asynchronous RAM must generate the RAM write enable (WE) signal, while ensuring that its data and address signals meet setup and hold time specifications relative to the WE signal. In contrast, the EABs' synchronous RAM generates its own WE signal and is self-timed with respect to the global clock. Circuits using the EABs' self-timed RAM need only meet the setup and hold time specifications of the global clock. When used as RAM, each EAB can be configured in any of the following sizes: 256 bits x 8 bits, 512 bits x 4 bits, 1,024 bits x 2 bits, and 2,048 bits x 1 bit, as shown in Figure 3.1. If necessary, all EABs in a device can be cascaded to form a single RAM block. EABs can be cascaded to form RAM blocks of up to 2,048 bits without impacting timing, as shown in Figure 3.2 [2].

Flex 10K devices are supported by the MAX+PLUS II development system, a single, integrated package that offers schematic, text and waveform design entry, compilation and logic synthesis, full simulation and worst-case timing analysis, and device configuration. Flex 10K devices contain an optimized interface that permits microprocessors to configure FLEX 10K devices

serially or in parallel and synchronously or asynchronously. The interface also enables microprocessors to treat a FLEX 10K device as memory and configure the device by writing to a virtual memory location, making it easy for the designer to reconfigure the device.

Figure 3.1. EAB memory configurations.

Figure 3.2. Examples of combining EABs.

Altera Flex 10K devices fall into the mid density FPGA family. The different categories of devices offered by Altera are high density FPGAs (Stratix, Stratix GX, APEX II, APEX 20K, Mercury), high volume FPGAs (Cyclone, ACEX), FPGA to ASIC (Hardcopy, Hardcopy Stratix, Hardcopy APEX 20KE, Hardcopy APEX 20KC), mid density FPGAs (FLEX 10K), embedded processor solutions (Excalibur, Nios), high volume CPLDs (MAX 3000A), high performance CPLDs (MAX 7000) and mature devices (FLEX 6000, FLEX 8000, MAX 9000, Classic). The details of these categories and their devices are available on the Altera website [2].

Xilinx also offers devices similar to Altera. The categories of devices offered by Xilinx are Virtex II Pro FPGAs, Virtex II FPGAs, Virtex II series EasyPath solutions, Spartan 3 FPGAs, Spartan II E FPGAs, CoolRunner II CPLDs, RocketPHY 10 Gbps Transreceivers and Configuration Solutions. The details of these categories and the devices under these categories are available on the Xilinx website [25].

The design of our project is in VHDL and hence it can be implemented in any of the above device families.

3.3 FPGA Advantages for Feed-Forward Neural Networks

The advantages that FPGAs have in implementing feed-forward neural networks can be split into two aspects. The first aspect consists of advantages that are inherent to FPGAs and that apply to anything designed for FPGAs. The second aspect consists of advantages that feed forward neural networks can exploit – these advantages cannot be exploited by all designs targeted for FPGAs.

These advantages are as follows:

- General FPGA Advantages

- Fast prototyping

Because of the easy programmability of FPGAs, a design that is thought to be correct can be implemented rapidly in hardware.

- Reusability

If design errors are found after the FPGA has been programmed, the same part can be quickly reused after the error has been corrected. Because the configuration of an FPGA is stored in internal RAM, there is no limit on the number of times a single part can be reprogrammed.

- FPGA Advantages Exploited by Feed-Forward Neural Networks

- Training versus Operation

Feed-Forward neural networks have two modes of functionality, training and operation. Training occurs when the feed-forward neural network is presented with input and output pattern pairs, and it follows some training algorithm to adapt itself and make generalization about these pairs. Operational mode occurs after training has been completed, and the feed-forward neural network applies the generalizations made during training to new inputs. Normally the hardware used during training includes the hardware used during operation plus extra hardware needed to implement the training algorithm. If a hardware-based neural network is to support training mode, it has to include the hardware for the training algorithm. After training, while the feed-forward neural network is in operational mode, the special hardware part used for training would not be needed. FPGAs have the advantage that as the feed-forward neural network is being trained, all the hardware needed for the training algorithm can be implemented. While in operational mode the FPGAs can be reconfigured leaving only the needed hardware. This would free FPGA hardware resources for other uses such as training other networks.

- Multiple Training Algorithms

Some types of feed-forward neural networks have different types of training algorithms, and each algorithm requires its own special hardware [11]. Custom implementations of feed-forward neural networks have to include all of this special hardware in order to satisfy a demand for a particular training algorithm.

While the hardware for one of these training algorithms is running, the hardware allocated to the other training algorithms would be wasted. Because the FPGA hardware is allocated at run-time, only the desired training algorithm needs to be implemented in hardware at one time. Thus the hardware resources are used more efficiently.

3.4 Synopsys

The complete system implemented in [26] consists of 12 processing elements in the hidden layer and one input and one output. The whole system does not fit into the Altera FLEX 10K family board hence Synopsys [24] was used to verify the complete design. Synopsys' Behavioral Compiler(TM) is a high-level synthesis tool that allows designers to evaluate alternative architectures and then quickly create an optimal gate-level implementation. The Behavioral Compiler creates designs that consist of a datapath, memory I/O and a control finite state machine (FSM). The Behavioral Compiler has been used in a wide variety of applications including digital communication, wireless digital signal processing (DSP), graphics, multimedia and networking ICs. For dataflow intensive designs, the Behavioral Compiler helps achieve the highest performance through automatic pipeline insertion [24], pipelined component utilization and number-of-cycles constraints, while for memory intensive designs, the Behavioral Compiler's memory I/O inferencing enables straightforward specification of memory I/O for technology-specific RAMs. The Behavioral Compiler automatically schedules the memory I/O and generates the necessary FSM. The Synopsys Behavioral Compiler synthesizes datapath, memory and control logic from a behavioral specification. It schedules operations, such as additions, multiplications, memory reads and memory writes into a number of clock cycles. This

optimization is based on user constraints to control latency, throughput, resource types and I/O activity. During optimization, the Behavioral Compiler performs hardware allocation for operators, registers and multiplexers. After the datapath and memory I/O scheduling are complete, the Behavioral Compiler creates the necessary FSM. All the features of the behavioral compiler are listed on the Synopsys website [24]. The Following is a step-by-step process for using the Synopsys Behavioral Compiler

1. Specify the functional behavior of the design using Verilog or VHDL.
2. Analyze the HDL file for syntax and simulate using the existing simulation environment.
3. Synopsys Behavioral Compiler then elaborates the design in preparation for scheduling and builds an abstract circuit representation.
4. Next, Behavioral Compiler uses the target technology library and reusable DesignWare components to time the datapath operators, FSM, registers, control chaining and multiplexer logic.
5. Behavioral Compiler then performs scheduling, allocates the hardware resources and generates the control FSM.
6. At this stage, we can use BCView to graphically analyze the scheduling and hardware allocation of the design. Based on the analysis, we can choose to modify the constraints and quickly explore design alternatives before gate-level logic optimization.
7. Once an alternative is chosen, a simulatable RTL description can be written out to verify the scheduled design.
8. Finally, perform gate-level simulation and verify the resulting implementation.

Chapter 4

Options for Implementing the Sigmoid Activation Function

Neural networks require the use of a nonlinear activation function in each neuron. Common examples of activation functions include hard-limiter, pseudolinear functions, hyperbolic tangent functions and sigmoid functions [17]. The sigmoid function is the most frequently used activation function and we have also used that here. The sigmoid function equation is

$$y = \frac{1}{1 + e^{-x}}$$

The sigmoid function is not suitable for direct digital implementation but there are different ways in which the sigmoid function or its close approximation can be implemented. Some of them are discussed below

4.1 Dataflow Implementation

The dataflow implementation method [8] uses a dataflow graph for C-like code for the sigmoid function. As the sigmoid function equation above contains the transcendental function e^x , an approximation is used:

$$f(x) = \frac{1}{2} \left(\frac{x}{1 + |x|} + 1 \right)$$

This function is a simple polynomial that uses no transcendentals. The graph in Figure 4.2 shows the actual sigmoid function and the approximation, as calculated in Matlab. Note that the curves

provide a similar limiting function. In this method, it is the general characteristics of the sigmoid that are considered and not the precise equation. A straightforward translation of the function $f(x)$ is shown in Figure 4.1.

```
/* sigmoid activation function */
sigmoid
f (sigmoid x) {
return ((1.0 / 2.0) * (x / (1 + abs(x)) + 1));
} /* end f () */
```

Figure 4.1. Translation of the sigmoid function into a C-like code [8].

Figure 4.2. Sigmoid function and dataflow approximation (Matlab) [8].

From the code fragment in Figure 4.1, a circuit may be extracted which will implement the sigmoid function. This circuit is extracted by creating the dataflow graph for the code. This

graph is then used to configure the hardware. Figure 4.3 shows the dataflow circuit for the function $f(x)$. This circuit takes as its input a value x and returns the output $f(x)$. The functional units used by the circuit are two adders, a divider, an absolute value and a divide by two unit. Some simple optimizations are performed on this circuit. A divide by two unit, for instance, has been used rather than a full divider. Once the circuit for the function has been extracted, it may be used as a macrocell. Note that the divider used to implement $\frac{x}{1+|x|}$ would require a lot of chip resources and time.

Figure 4.3. The sigmoid activation function circuit [8].

4.2 Lookup Table Method

A sigmoid function is expensive to realize on a platform that lacks a floating-point unit. To reduce the realization costs, the sigmoid transfer function is replaced by a computationally less

expensive fixed-step lookup table and a linear interpolation is performed to replace the sigmoid function. In this implementation a y value associated with each x value is stored in either a random access memory or a read only memory [21]. The value x is used to address the memory. The accuracy of the output of the processing element depends partially on the number of words stored in the lookup table. The disadvantage with this approach is that for achieving better results the size of the lookup table goes on increasing and often becomes large and unwieldy.

4.3 Taylor Series Expansion

Taylor series expansion has also been used to implement the sigmoid function in hardware [14]. It uses 5th degree polynomials to implement the sigmoid function. Figure 4.4 shows the plot of the actual sigmoid function and the Taylor series approximation, as computed in Matlab.

Figure 4.4. Sigmoid function and Taylor series approximation [14].

This method uses 3 intervals to implement the sigmoid function. The equations used for the implementation are

$$\begin{aligned}
 f(x) &= 0.571859 + (0.392773)x + (0.108706)x^2 + (0.014222)x^3 + (0.000734)x^4 & \text{for } -\infty < x \leq -1.5 \\
 &= 0.5 + (0.25)x - (0.020833)x^3 + (0.002083)x^5 & \text{for } -1.5 < x < 1.5 \\
 &= 0.428141 + (0.392773)x - (0.108706)x^2 + (0.014222)x^3 - (0.000734)x^4 & \text{for } 1.5 \leq x < \infty
 \end{aligned}$$

4.4 Least Square Technique

Two different methods using least square techniques for implementing the sigmoid function are discussed here.

4.4.1 Least Square Approximation Technique 1

This method requires only a magnitude comparator and three parameter registers together with the already present arithmetic units to generate the nonlinearity. The first principle of this approach is to determine using the comparator whether the input is positive or negative and whether or not it is saturated (highlimit and lowlimit of the input beyond which the output is considered to be either +1 or 0 respectively). The upper and lower limits or saturation points are stored in two of the parameter registers and used for comparison with the input value. The gain value that defines the slope of the nonlinearity is stored in the third parameter register. The sigmoid function curve is broken up into four segments, positive saturated, positive unsaturated, negative unsaturated and negative saturated. The nonlinear part between the saturation limits is approximated using a least-squares polynomial of degree two [22]. The input x is compared using a magnitude comparator $\text{CMP}()$ against the threshold stored. The thresholding $\text{CMP}(i,j) = 1$ if $i \geq j$ else 0, and

$$T_H = \text{CMP}(\text{highlimit}, x)$$

$$T_L = \text{CMP}(x, \text{lowlimit})$$

$$T_{\pm} = \text{CMP}(x, 0)$$

The comparator output $Y = f(x)$ is given by

$$\begin{aligned} f(x) &= 1 && \text{if } x > \text{highlimit} \\ f(x) &= 1 - \text{gain}(\text{highlimit} - x)^2 && \text{if } 0 \leq x \leq \text{highlimit} \\ f(x) &= 0 + \text{gain}(\text{highlimit} + x)^2 && \text{if } \text{lowlimit} \leq x \leq 0 \\ f(x) &= 0 && \text{if } x < \text{lowlimit} \end{aligned}$$

Thus the general formula for all possible inputs is

$$f(x) = T_{\pm} - (2T_{\pm} - 1) \times \{ T_H T_L \text{gain}[(2T_{\pm} - 1)\text{highlimit} - x]^2 \}$$

where the lowlimit and the highlimit are the saturation limits of the sigmoid curve.

Figure 4.5. Sigmoid function and approximation using least square approximation technique 1 (Matlab) [22].

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

The three parameters that form the curve may be computed using the least squares polynomial technique to give the best fit. This technique makes full use of the arithmetic hardware that might already be implemented for use by the other neural arithmetic operations. Figure 4.5 shows the sigmoid function and approximation, as computed in Matlab. This method used two intervals to implement the sigmoid function and the different values used for the computation are

$$\text{Lowlimit} = -8$$

$$\text{Highlimit} = 8$$

$$\text{Gain} = 0.0078125$$

4.4.2 Least Square Approximation Technique 2

This technique uses a second order equation of the form $y = ax^2 + bx + c$ to approximate the sigmoid function for $-4 < x < 0$ and $0 < x < 4$ [27] .

Using least square approximation for $x \in [0, 4]$ gives

$$y = -0.0363 x^2 + 0.2610x + 0.5028$$

This function can be further simplified to

$$y = 0.972 - 0.57(0.25x - 0.898)^2$$

The sigmoid function is antisymmetric around $y = 0.5$ and it has two asymptotes at $y = 0$ and $y = 1$. Hence, the function evaluation for $x < 0$ is the complement of it at $x > 0$. The approximation can be expressed as

$$y = 0.972 - 0.57(0.25x - 0.898)^2 \quad 0 < x < 4$$

$$= 0.028 + 0.57(0.25|x| - 0.898)^2 \quad -4 < x < 0$$

As a result, y can be approximated by

$$y = 1 - 2^{-1}(1 - |2^{-2}x|)^2 \quad 0 < x < 4$$

$$= 2^{-1}(1 - |2^{-2}x|)^2 \quad -4 < x < 0$$

Figure 4.6. Block diagram of least square approximation technique 2

Figure 4.7. Sigmoid function and approximation using least square approximation technique 2 (Matlab) [27].

In order to implement the approximation, two shift circuits, a squaring circuit and an additional combinational circuit are required as shown in the block diagram of Figure 4.6. The first shift block transforms the input signal into a fractional representation. The input signal is chosen to be one sign bit, two binary digits for the integer part and three binary fractional places (six binary digits). The error is kept within 0.02; hence setting five fractional places is sufficient for the output. This method used 2 intervals to implement the sigmoid function and Figure 4.7 shows the Matlab simulation.

4.5 Piecewise Approximation

Three different techniques of implementing the sigmoid function in hardware using piecewise approximation are presented here.

4.5.1 Piecewise Approximation Technique 1

The first technique [18] is inspired by the piecewise linear approximation used to implement A-law companding (combination of compression and expansion) [23] for pulse code modulation (PCM) systems. A 7 segment curve has been developed on the principles of the A-law curve so that the gradient of each section can be expressed as a power of 2. Note that at $x = 0$ the curve and the sigmoid have the same gradient ($= 0.25$). Simulation of the multiplayer perceptron (MLP) with backpropagation using the piecewise linear approximation indicates that it gives comparable performance to simulation based on the true sigmoid value. This method uses 7 intervals to compute the sigmoid function and the Matlab simulation is shown in Figure 4.8. Table 4.1 shows a possible truth table for the piecewise linear function corresponding to positive values of x , aimed at mapping a 16 bit 2's complement input value (I_0 - I_{15}) in the range -8 to $+8$

to an 8 bit 2's complement output value (R_0 - R_7) in the range 0 to 1. This could be used in a digital neural net system with 8 bit data word representation, which allows an additional 8 bits internal to each neuron for bit growth.

Figure 4.8. Plot of piecewise approximation technique 1 and sigmoid curve (Matlab) [18].

I_{15}	I_{14}	I_{13}	I_{12}	I_{11}	I_{10}	I_9	I_8	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	R_7	R_6	R_5	R_4	R_3	R_2	R_1	R_0
0	0	0	0	0	a	b	c	d	x	x	x	x	x	x	x	0	1	0	0	a	b	c	d
0	0	0	0	1	a	b	c	d	x	x	x	x	x	x	x	0	1	0	1	a	b	c	d
0	0	0	1	a	b	c	d	x	x	x	x	x	x	x	x	0	1	1	0	a	b	c	d
0	0	1	a	b	c	x	x	x	x	x	x	x	x	x	x	0	1	1	1	0	a	b	c
0	1	a	b	c	x	x	x	x	x	x	x	x	x	x	x	0	1	1	1	1	a	b	c

Table 4.1. Piecewise linear activation function truth table for positive values of input [18].

A possible implementation of the truth table of Table 4.1 given in [18] is shown in Figure 4.9. This consists of a most significant 1 selector circuit which takes I_1 - I_{14} as input and outputs a 4 bit word Y_{11} - Y_{14} containing a 1 at the position of the most significant 1 in the input or else, if all four bits are 1, 0. This output is decoded to produce output bits R_4 , R_5 . Bit $R_6 = 1$ because the output is always > 0.5 and bit $R_7 = 0$ because the output is always positive. R_7 is therefore ignored at this stage. Signals Y_{12} - Y_{14} are also used to control a bank of 2:1 multiplexer circuits which shift bits of the input word to provide R_0 - R_3 . These multiplexers connect the upper right input to the output if the control signal is 0, and the lower right input to the output otherwise.

Figure 4.9. Implementation of table 4.1 [18].

$y = 0.5$, and results in the correct value being output for negative inputs. For positive inputs, both 2's complement circuits are bypassed. The circuit for the complete curve, shown in Figure 4.10, can be compactly and elegantly implemented in VLSI.

Fig 4.10. Implementation of the complete curve [18].

4.5.2 Piecewise Approximation Technique 2

The second technique is based on selecting an integer set of breakpoints for the approximating function and correspondingly the y values are set as power of two numbers [1]. This method uses 2 intervals to implement the sigmoid function.

Figure 4.11. The single approximating segments [1].

Consider the negative semi-axis and two consecutive points P_n and P_{n-1} (as shown in Figure 4.11) where

$$P_n = (-n, \frac{1}{2^{n+1}}); P_{n-1} = (-(n-1), \frac{1}{2^n})$$

Then the equation of the line defined by the two points is:

$$\frac{y - \frac{1}{2^{n+1}}}{\frac{1}{2^n} - \frac{1}{2^{n+1}}} = \frac{x - (-n)}{-(n-1) - (-n)}$$

and the equation simplifies to

$$y(2^{n+1}) - 1 = x + n$$

which gives

$$y = \frac{x + (n+1)}{2^{n+1}} \quad (1)$$

this holds for $n = 0, 1, \dots$ and $x < 0$.

Likewise, considering the positive semi-axis and the related points $Q_{n-1}(n-1, 1 - \frac{1}{2^n})$ and

$Q_n(n, 1 - \frac{1}{2^{n+1}})$ we obtain $y = \frac{2^{n+1} + x - (n+1)}{2^{n+1}}$.

It can be proved that $y_{x>0} = 1 - y_{x<0}$; The approximation function has a symmetry point in $(0, 1/2)$, like the sigmoid function. Let's consider the negative axis (the extension to $x > 0$ is immediate). The general form of the approximating function has been given in equation 1 where $n = n(x)$. The dependence of the approximation on n can be removed by noting that, by construction and given the particular choice of breakpoints:

$$-1 \leq x < 0: n = 1$$

$$-2 \leq x < -1: n = 2$$

and so on. If we define (x) as the integer part of x , n can be written as

$$n = |(x)| + 1$$

from which

$$y = \frac{x + |(x)| + 2}{2^{(x)|+2}}$$

let $\ddot{x}$ be the decimal part of x with its own sign.

$$\ddot{x} = x + |(x)|$$

Substituting in the general expression for the approximation we have

$$y = \frac{2 + \ddot{x}}{2^{(x)|+2}} = \frac{1}{2} + \frac{\ddot{x}}{4} \quad \text{for negative values of } x.$$

$$\text{Similarly} \quad y = \frac{2^{(x)|+2} + \ddot{x} - 2}{2^{(x)|+2}} = 1 - \frac{1}{2} - \frac{\ddot{x}}{4} \quad \text{for positive values of } x.$$

Figure 4.12. Sigmoid function and approximation using piecewise approximation technique 2 (Matlab) [1].

This last formula involves only simple shifts and sums. Figure 4.12 shows the sigmoid and approximation using this technique, as computed in Matlab. Hardware implementation of the piecewise linear approximation requires a shift register and a counter to control it. The final architecture is shown in Figure 4.13. Note how the shifter is controlled by a counter and this counter is controlled by the value of $|(x)|$. This does not imply the use of a twelve-bit counter, in fact, only three or four bits (according to the chosen precision) are significant. Any one bit more significant than the first three or four will force the function to its asymptotical values and will be equivalent to a set or reset of the shift register.

Figure 4.13. Architecture for the piecewise linear approximation 2 [1].

4.5.3 Piecewise Approximation Technique 3

This technique is first presented in [4]. Its modification is implemented in [10]. This method also uses piecewise approximation to generate the sigmoid function. This method implements the sigmoid function in two intervals and the equations used for the implementation are

$$f(x) = \frac{1}{2} * \frac{1}{2^{\text{int}(x)}} * \frac{1}{1 + \text{frac}(x)} \quad x \leq 0$$

$$= 1 - \frac{1}{2} * \frac{1}{2^{\text{int}(x)}} * \frac{1}{1 + \text{frac}(x)} \quad x > 0$$

where $\text{int}(x)$ is the integer part of x and $\text{frac}(x)$ is the fractional part.

These equations generate a sigmoid function approximation as shown in Figure 4.14, as computed in Matlab.

Figure 4.14. Sigmoid function and approximation using piecewise approximation technique 3 [10].

4.6 Analog Implementation

Figure 4.15 shows the analog circuit schematic that realizes both a sigmoid activation function and its derivative. The neuron is built with strong inversion biased transistors and has a current input and voltage out. V_{out1} outputs the sigmoidal activation function while $(V_{\text{out1}} - V_{\text{out2}})$ realizes its approximate derivative [16]. Within the dashed line of the figure, the fixed voltage V_{ref1} is carefully chosen so that both transistors M1 and M2 work over their linear range. The

formed linear resistor R_{AB} can be controlled by adjusting the gate voltages of both transistors V_N and V_P . Within either dash-dotted line in the figure, a simple differential pair composed of identical transistors, together with the active loads, realizes the actual sigmoidal shaped nonlinearity. One port of the differential pair is connected to point B and the other is connected to a fixed voltage V_{ref2} or $V_{ref2} - \delta V$, where δV is a fixed small voltage. I_{ref1} and I_{ref2} are fixed current sources. The referenced current directions are shown by the arrows in Figure 4.15

Fig. 4.15. Sigmoid function implementation with transistors [16].

Assuming that M3, M4 are operating in saturation and follow an ideal square law, the drain current of transistor M3 can be expressed as

$$I_{d3}(V_d) = \frac{\beta}{2} (V_B - V_C)^2 \sim \frac{I_{ref2}}{2} + \frac{\beta}{4} V_d \sqrt{\frac{4I_{ref2}}{\beta} - V_d^2}$$

with the input differential voltage V_d ($V_d = V_b - V_{ref2}$) in a finite region where

$$|V_d| \leq \sqrt{\frac{2I_{ref2}}{\beta}}$$

Here β is the transconductance parameter for transistors M3 and M4. Current I_{AB} is given by

$$I_{AB} = I_{in} + I_{ref1}$$

When I_{in} is small, $V_d > v(2I_{ref2} / \beta)$. V_{out1} remains as the low saturation voltage. As I_{in} increases, V_B reduces gradually and V_{out1} increases slowly. When $V_d < v(2I_{ref2} / \beta)$, V_{out1} reaches and remains at the high saturation level.

Figure 4.16. Simulated neuron activation function and its derivative compared with fitted sigmoid and derivative of simulated neuron activation function [16].

Assuming that $V_{out} = V_{out}(I_{in})$ is the generated neuron activation function, using the forward difference method, the approximate derivative voltage V_{deriv} can be determined by subtracting V_{out2} from V_{out1} :

$$V'_{out}(I_{in}) = V'_{out}(V_d) \cdot V'_d(I_{in})$$

$$\cong -\frac{V_{out}(V_B - V_{ref2} + \delta V) - V_{out}(V_B - V_{ref2})}{\delta V} \cdot R_{AB}$$

$$V_{deriv}(I_{in}) = \frac{\delta V}{R_{AB}} \cdot V'_{out}(I_{in})$$

$$\cong -(V_{out}(V_B - (V_{ref2} - \delta V)) - V_{out}(V_B - V_{ref2}))$$

$$= V_{out1} - V_{out2}$$

The plots of the output voltage V_{out} and its derivative V_{deriv} are shown in Figure 4.16 and the relative errors are shown in Figure 4.17.

- - - - - Relative Error between simulated neuron activation function and fitted sigmoid
- Relative Error between simulated derivative and derivative of simulated neuron activation function

Figure 4.17. Relative errors [16].

Chapter 5

Design, Architecture and Implementation

5.1 Genetic Algorithm Approximation

We use the Genetic Algorithm (GA) method to implement the sigmoid function in hardware [9]. This is also a piecewise approximation technique, which requires a multiplier and adders for the implementation of the sigmoid function. Hence this approach is even more effective when used with architectures that already implement fast arithmetic units (i.e., fast multiplier and adders). Because of its high accuracy, it is beneficial to use this technique even where the arithmetic units need to be installed for the sole purpose of implementing the sigmoid function. In this method, piecewise polynomials with integer coefficients are used to approximate the sigmoid function. A GA is used to search for a feasible set of piecewise polynomials to fit the sigmoid function. The principle advantage of a GA is that it performs a robust search in a complex space. It is computationally simple yet powerful.

A genetic algorithm can be implemented to determine the integer coefficients for an interpolating polynomial of requested degree based on the evaluation of an objective function. The idea is to build up a set of third degree polynomials, with integer coefficients, in a piecewise fashion, so that the set of polynomials meet a prescribed error constraint at each training point for the domain they span. For example, the coefficients of the third degree polynomial $Ax^3 + Bx^2 + Cx + D$ can be determined such that the objective function, computed in terms of the error at the training points, is minimized. In other words, the genetic algorithm can be constructed to search the restricted solution space to find the optimal coefficient set. Once the coefficients and the

breakpoints are decided, the sigmoid function is implemented in hardware as a piecewise approximation of the 3rd degree polynomials.

Because it is a probabilistic search strategy, the GA will generate different sets of breakpoints and coefficients for every run. We generate eight breakpoints and correspondingly eight sets of A, B, C, and D coefficients to compute the sigmoid function approximated by the following polynomial.

$$f(x) = Ax^3 + Bx^2 + Cx + D$$

These coefficients and breakpoints are generated in [9] and are given in Table 5.1 below.

Breakpoint	A	B	C	D
0	-22688	280	16366	8192
3973	-9142	-8316	15492	10140
9584	3030	-13020	11858	12503
16302	6470	-10305	6916	14413
28672	2406	-3193	1849	15903
40960	649	-822	442	16274
51200	13304	-2751	232	16352
55296	296	-225	86	16364

Table 5.1. Breakpoint and coefficients generated by the genetic algorithm.

The number of intervals needed depends on the accuracy desired. Here we use 8 intervals to achieve a maximum error of 0.0001 as in [9]. This error constraint translates to 0.75 count as the breakpoints and coefficients are scaled by 2¹³. The result is scaled by 2¹⁴. The maximum error determines the number of intervals required. In terms of implementation, the number of

intervals affects only the cycles required to initialize the breakpoints at the beginning. It does not require additional time or resources for the computation of the sigmoid function. The above set generates the sigmoid function only in the first quadrant, i.e., only for the positive values of the input. We use the symmetry of the sigmoid function, along with the above set, in order to generate the sigmoid function in the third quadrant, i.e., for the negative values of the input. The general equation of the sigmoid function is

$$y = \frac{1}{1 + e^{-x}}$$

The important property of the above equation is that

$$1 - \frac{1}{1 + e^{-x}} = \frac{1}{1 + e^x}$$

Hence for negative values of input x , modulus of x is taken and the result is subtracted from 1.

This gives the value of the sigmoid function in the third quadrant.

Figure 5.1 Sigmoid function and genetic algorithm approximation in first quadrant [9].

The sigmoid function generated is shown in Figure 5.1 and Figure 5.2.

Figure 5.2. Sigmoid function and genetic algorithm approximation in third quadrant [9].

5.2 Booth Encoding

We use Booth encoding techniques and tree multipliers in the implementation of the necessary multipliers. These multipliers are used to implement the third degree polynomials, which in turn implement the sigmoid function in piecewise form.

5.2.1 Introduction to Booth Encoding

The original Booth algorithm [5] did not deal with parallel multiplication, but was aimed at improving the speed of the add-and-shift algorithm. The Booth algorithm belongs to the class of recoding algorithms, i.e., algorithms that “recode” one of the two operands in such a way that the number of partial products to be added together decreases. For instance, a simple recoding scheme would consist of considering all the bits of one operand sequentially and skipping all its

zeroes, because they do not contribute to the final result. However this leads to a variable execution time of the multiplication. This can be interesting in self-timed systems, where a “done” signal can announce the completion of the operation, but it is useless in other timing strategies where the user has to trigger the system on the worst case, i.e., when the operand has all ones. Actually, the original Booth algorithm itself was not constant in time, although it was much more effective than simple “skipping over 0’s” because it dealt with strings of 0’s and 1’s properly recoded.

5.2.2 Modified Booth Algorithm

The modified Booth algorithm strictly considers groups of bits in one operand, rather than being able to skip over arbitrarily long strings [6]. This leads to a longer, but constant execution time. It guarantees that an n -bit multiplier will generate at most $n/2$ partial products. This modified Booth algorithm decreases the number of rows that have to be added, therefore speeding up the computation. It can multiply 2’s complement numbers directly without regard for the signs of the two numbers. There is no need for pre-complementing the multiplier or post-complementing the product. The modified Booth algorithm uses the bit pair-recoding scheme. Table 5.2 shows the bit pair-recoding scheme and Figure 5.3 shows an example of the bit pair-recoding scheme [6].

bit position	5	4	3	2	1	0	(-1)← implied
bit value	0	1	1	1	0	1	(0)← implied
recoded string	+2		-1		+1		

Figure 5.3. Example of bit pair recoding.

The algorithm operates on one of the two operands, analyzes pairs of bits and converts them into a set of signed digits. The conversion is explained in Table 5.2.

Multiplier bit-pair		Multiplier bit on the right	Multiplicand multiples to be added
2^1	2^0		
$i + 1$	i	$i - 1$	
0	0	0	0 x multiplicand
0	0	1	+1 x multiplicand
0	1	0	+1 x multiplicand
0	1	1	+2 x multiplicand
1	0	0	-2 x multiplicand
1	0	1	-1 x multiplicand
1	1	0	-1 x multiplicand
1	1	1	0 x multiplicand

Table 5.2. Bit pair recoding scheme used in modified booth algorithm [6].

An example of multiplying 3×29 is presented in Figure 5.4, Figure 5.5 and Figure 5.6. Figure 5.4 shows the multiplication using the conventional shift and add method. Figure 5.5 shows the multiplication using Booth encoding technique and Figure 5.6 shows the multiplication using the modified Booth encoding (bit pair recoding) technique.

M = Multiplicand	x	<div style="display: flex; justify-content: space-around;"> 0 0 0 0 1 1 </div> <div style="display: flex; justify-content: space-around;"> 0 1 1 1 0 1 0 </div> <div style="display: flex; justify-content: space-around; border-top: 1px dashed black; margin-top: 5px;"> +2 x M -1 x M +1 x M </div>
		<div style="display: flex; justify-content: space-around;"> 0 0 0 0 0 0 0 0 0 0 1 1 </div> <div style="display: flex; justify-content: space-around;"> 1 1 1 1 1 1 1 1 0 1 </div> <div style="display: flex; justify-content: space-around;"> 0 0 0 0 0 1 1 0 </div>
		<div style="display: flex; justify-content: space-between; align-items: center;"> 1← 0 0 0 0 0 1 0 1 0 1 1 1 ← + 87 </div>

Figure 5.6. Multiplication of 3 x 29 using modified Booth encoding technique requiring only n/2 partial products.

5.3 The Multiplier

The sigmoid calculation requires a 16 x 16 bit multiplier. The modified Booth encoder reduces partial products by half, so we are required to sum only 8 partial products. A Wallace tree [20] using carry save adder (CSA) reduces the stages of partial product sums, hence we are using a Wallace tree with 9-2 compressor and 4-2 compressor where required. Since there are eight partial products, we need 128 (8*16) single bit partial product generators, 26 9-2 compressors and two 4-2 compressors. One carry propagate adder is needed. The block diagram of the multiplier is shown in Figure 5.7.

Let the multiplicand be

And the multiplier be

$X = X_{15} X_{14} X_{13} \dots X_2 X_1 X_0$

$Y = Y_{15} Y_{14} Y_{13} \dots Y_2 Y_1 Y_0$

Figure 5.7. Block diagram of the multiplier.

The modified Booth encoding converts the multiplier from radix-2 to radix-4 using redundant digit set $\{-2, -1, 0, 1, 2\}$. So in the new multiplier with radix-4 there are only $N/2$ digits. The new multiplier's (Z) digits are defined by the following formula:

$$Z_j = Y_{2j} + Y_{2j-1} - 2Y_{2j+1} \quad \text{with } Y_{-1} = 0$$

We are using eight basic encoder elements and each takes three inputs. Let these elements be Z7 Z6 Z5 Z4 Z3 Z2 Z1 Z0. Each corresponds to one digit of the radix-4 multiplier. Input to each element is as given below

Z0 $\Rightarrow$ Y1, Y0, Y-1, here Y-1 = 0.

Z1 $\Rightarrow$ Y3, Y2, Y1

Z2 $\Rightarrow$ Y5, Y4, Y3

Z3 $\Rightarrow$ Y7, Y6, Y5

Z4 $\Rightarrow$ Y9, Y8, Y7

Z5 $\Rightarrow$ Y11, Y10, Y9

Z6 $\Rightarrow$ Y13, Y12, Y11

Z7 $\Rightarrow$ Y15, Y14, Y13

These Z's may be $2X$, X , 0 , $-X$, $2X$. But at this point we get three signals (X , $2X$, NEG) from each Booth encoding element. The term $2X$ is generated by a left shift at the partial products generation stage. Negation is done by taking 1's complement, then adding one in appropriate compressors. The different combinations of these signals represent five distinct possibilities, i.e., ($2X$, X , 0 , $-X$, $-2X$).

Figure 5.8. Booth encoder.

Figure 5.8 above shows the Booth encoder and Figure 5.9 shows the partial product generator.

Figure 5.9. Partial product generator.

As the partial product may be any of $-2X$, $-X$, 0 , X , $2X$, each partial product point has two inputs (consecutive bits) from the multiplicand and, according to the requirement, it generates output and if necessary complements it also. In effect it generates 8 rows of partial products. Output generated by any row partial product generator is as shown in the Table 5.3.

X	2X	NEG	Partial Product
1	0	0	X
0	1	0	2X
1	0	1	1'comp(X)
0	1	1	1'comp(2X)
0	0	1	all 1's

Table 5.3. Output generated by row partial product generator.

It does not generate $-X$ or $-2X$ but the 1's complements of X or $2X$. The required 1 is added in summing units. Keeping work pending eliminates the need to generate 2's complements separately. It again does not generate a row with all 0's but generates all 1's. In this case also we add 1 in the next stage. This takes care of the special case of the maximum negative number whose 2's complement generates the number itself. We also have to extend our partial products to 32 bits. The most significant bits of the partial product will be the same as the 16th bit for the first partial product, same as the 18th bit for the second partial product and so on. The most significant bits of the eighth partial product will be the same as the 30th bit of that partial product. The logic used to achieve this is as shown in Figure 5.10.

Figure 5.10. Sign extender.

5.3.1 Compressor

A compressor C_i is a combinational device that compresses N input lines (columns of bits) in the position i , to two output lines. In addition there are L input lines coming to the compressor at different levels j , and the same number of lines L generated by the compressor in the same levels j , where $L = N - 3$, as is shown in Figure 5.11.

Figure 5.11. Compressor.

The aim of a compressor is reduction of the number of bits to two. The lines L , both coming and leaving, are carry (in and out) signals. It is extremely important that the lines L at output C_i may be connected, in the same order, with input lines L of C_{i+1} in such a way that the signals originated by some level j of C_i may enter in the same level j of C_{i+1} . That preserves the order of the signals and imposes a particular structure of interconnections on the partial product summation tree.

5.3.2 Wallace Tree

In the iterative carry save array multiplier, one column of partial products appears as a column in the architecture and the result of one CSA propagates through all other CSA's of that column. In the Wallace tree these columns are processed in parallel. Then these partial sums are added in the next stage. Figure 5.12 shows an example of a Wallace tree.

Figure 5.12. Wallace tree.

If in the level j of the tree (the levels are considered growing from the top), the number of bits is

n , then $k = \frac{n}{3}$ full-adders should be used for the summation; the k bits generated are sent to the

level $j + 1$ of the tree for product bit $i + 1$, whereas the k bits of the sum of the level j constitute

inputs of the $\frac{2k}{3}$ adders of the level $j + 1$, together with the k carry bits originating from the level

j of the tree for the product bit $i - 1$. Since the number of bits to sum is reduced three times at each level, the depth of the Wallace tree is $O(\log N)$, where N is the initial number of bits.

We have eight partial product rows (A to H). The first row contains 32 bits to be added. Partial Product (PP) A uses 32 bits, (31..0), the second row PP B uses 30 bits, (31..2), and so on till the eighth row which uses 18 bits, PP H (31..14). The sign extender handles properties of sign of the original numbers and the work of taking the 2's complement of negative numbers. We have one more input for each row to obtain 2's complements for $-2X$, $-X$, and 0.

Figure 5.13. 4-2 compressor.

In our architecture we are using 9-2 and 4-2 compressors with the Wallace tree scheme. A 9-2 compressor adds nine elements while a 4-2 compressor adds four elements of a column at a time and generates the same result as two column CSA's in the iterative array multiplier. A

compressor has one carry in and one carry out also. In this architecture we need only one stage of 9-2 and 4-2 compressors because the Booth encoder has already reduced the partial sums to eight. These partial sums are finally summed by a carry propagate adder. Figure 5.13 shows the 4-2 compressor while Figure 5.14 shows the 9-2 compressor.

Figure 5.14. 9-2 compressor.

5.3.3 Carry Propagate Adder

The Wallace tree generates two bits to be added for each bit position for the final product. We add these bits with one carry in from the previous addition using full adders. We have used a carry propagate adder to obtain the final 32-bit product $M = X * Y$.

We need to compute KY where Y can be x , x^2 or x^3 . We use one Booth encoded multiplier, which is sufficient to compute multiplication KY or x^2 or x^3 . This is more efficient for our application than using special purpose squaring and cubing circuits [28], [29] as used in [27].

5.4 Architecture Modeling

Our sigmoid calculation function is designed to replace a lookup table method in the architecture developed in [26]. This system architecture was introduced in chapter 2. The system architecture and our changes to it are explained in more detail below. It is shown in Figure 5.15. The neural network contains

- A global control unit, which consists of control processor, instruction memory, data memory. The controller broadcasts the instructions to all the Processing Elements (PEs) and controls their execution.

Figure 5.15. System architecture of LFFNN [26].

- A hidden layer PE array with dual ring topology connected to each two adjacent PEs, in which each hidden layer neuron is mapped into a single PE. Unlike a shared bus, a ring is capable of many simultaneous, different data transfers in neighbors. For example, the first PE can send to the second at the same time as the third one can send to the fourth.

- An output layer PR array, where each output layer neuron is mapped into one PE. Internal broadcast buses make connections between the control unit and PE arrays.
- An external host interface which makes connections between the host and the application specific hardware accelerator.

Each PE consists of

- An instruction decoder
This decoder receives the commands from the control processor and is responsible for the PE's operations, such as arithmetic computation, memory access, or communication.
- One arithmetic logic unit
There are only addition, subtraction and multiplication operations. Also the sigmoid function is implemented by sequentially using this arithmetic logic unit. All computational representation uses fixed point numbers instead of floating-point ones. This approach can save computing time, simplify the design, and reduce the cost, and can be implemented efficiently in an FPGA.
- Breakpoint and coefficient initializing unit
This unit reads from the breakpoints of the sigmoid function into registers. This takes 8 clock cycles (for 8 breakpoints). This supports one clock cycle comparison of breakpoint and reading of the coefficients for that breakpoint from the coefficient ROM whenever the sigmoid function needs to be computed. This unit, along with the breakpoint and the coefficient ROM, replaces the local sigmoid LUT (Lookup Table) for the sigmoid activation function.

- Breakpoint and coefficient ROM to store the breakpoints and coefficients respectively to compute the sigmoid function.
- One local RAM block for weight storage

The architecture is the same as in [26] with the few changes which need to be incorporated to enable it to compute the sigmoid function. The breakpoint and coefficient initializing unit and the breakpoint and coefficient ROM replace the lookup table in the original architecture. Also the general purpose multiplier is replaced with a Wallace tree Booth encoded multiplier explained in section 5.3. This multiplier and the breakpoint and coefficient ROM together implement the sigmoid function that was earlier read from the lookup table.

The instruction set is also the same as that of the original architecture with the addition of the sigmoid instructions. The description of the different types of instructions is given below.

- Arithmetic and input / output instructions

These instructions consist of three parts – opcode, local memory address, and register mark. The opcode is divided into three parts: the first part labels instruction attribute – sequential or parallel; the second part shows its operation, which includes load, store, add, subtract, multiply, input and output. A one bit field is used to label the internal PE's registers. The detailed format and functional description are provided in Table 5.4

- Mask and flow control instructions

These instructions consist of two parts – opcode and PE index or offset added to PC (Program Counter). The detailed information is presented in table 5.4

- Sigmoid instructions

These instructions are used to compute the sigmoid function and initialization of the breakpoint. These instructions have opcodes which specify the task to be performed. The

operands are implied. The new instructions are op2_bkpt_init, op2_sigmoid, op2_sigmoid1, op2_sigmoid2, op2_sigmoid3 and op2_sigmoid4 Table 5.2 presents information on all the instructions.

Instruction Format		Code	Description
Opcode (part I) (3 bits)	op1_noop	000	No operation
	op1_parallel_h	001	All hidden layer PEs run
	op1_parallel_o	010	All output layer PEs run
	op1_sequential	011	An appointed PE runs
	op1_mask	100	Mask an appointed PE; only an unmask instruction can reactivate an masked PE
	op1_unmask	101	Unmask an appointed PE
	op1_jump	110	This instruction is unrelated with the operation of PE array
Opcode (Part II) (4 bits)	op2_store	0001	Write data from reg3 to local memory
	op2_load	0010	Read data from local memory to reg1 or reg2 based on register marker bit
	op2_add	0011	Reg3 <= reg1 + reg2
	op2_sub	0100	Reg3 <= reg1 - reg2
	op2_mult	0101	Reg3 <= reg1 x reg2
	op2_input	1000	Read data from the left or right port to reg3 based on the address item
	op2_loadq	1001	Read data from local memory to reg3

Table 5.4. Instruction set architecture (part 1 of 2).

Instruction Format		Code	Description
Opcode (Part II) (4 bits)	op2_output	1010	Write data from reg3 to the left or right port based on the address item
	op2_data	1111	Read data from data bus to reg3
	op2_sigmoid	0110	This instruction decides which breakpoint is to be used for the computation of the sigmoid function.
	op2_sigmoid1	0111	This instruction performs the Ax part of sigmoid function computation.
	op2_sigmoid2	1011	This instruction performs the $(Ax + B)x$ part of sigmoid function computation
	op2_sigmoid3	1100	This instruction performs the $(Ax + B)x + C)x$ part of sigmoid function computation
	op2_sigmoid4	1101	This instruction performs the $(Ax + B)x + C)x + D$ part of sigmoid function computation
	op2_bkpt_init	0000	This instruction loads 8 breakpoints from breakpoint ROM into breakpoint registers so that the values are not required to be read every time the sigmoid function is computed
Local memory address (8 bits)			An ability to access 256 memory locations
Register marker (1 bit)			Marker = 0 for reg1; Marker = 1 for reg2
PE index or PC offset (13 bits)			For mask instruction, this field gives the PE serial number; for jump instruction, it stands for an offset to current PC

Table 5.4. Instruction set architecture (part 2 of 2).

Note:

1. There are three registers (reg1, reg2, reg3) in each PE.
2. There are eight breakpoint registers, break1 to break8, which hold the breakpoints after the op2_bkpt_init instruction is executed.
3. A, B, C, and D are the coefficients used to evaluate the polynomial for the corresponding breakpoint. A is the coefficient of X^3 .
4. The left port connects the left lateral bus of the PE; the right port connects the right lateral bus of the PE.

5.5 Sigmoid Function Implementation

We mentioned earlier that the multiplier along with the breakpoint and coefficient ROM together implement the sigmoid function. We also use the eight internal breakpoint registers for this purpose. The registers are initialized with the breakpoint values at the beginning so that they are readily available for comparison. This is a one time overhead requiring the number of clock cycles equal to the number of breakpoints. These breakpoints are compared against the value of 'x' used to calculate the sigmoid function. Depending upon results of the comparison, corresponding coefficients are selected from the coefficient ROM to compute the sigmoid function. This computation takes five clock cycles. The sigmoid function is approximated by the following polynomial.

$$f(x) = Ax^3 + Bx^2 + Cx + D$$

This polynomial is implemented as nested multiplication.

$$f(x) = (((Ax + B)x + C)x + D)$$

5.5.1 Programmable Sigmoid Function Implementation

Reconfigurable Neural Networks often require a programmable sigmoid function [27]. The programmable sigmoid function is given by the equation

$$y = \frac{1}{1 + e^{-kx}}$$

Where k determines the steepness of the sigmoid function.

The genetic algorithm method [9] can be trained using the data for a sigmoid function with different values of k and made to generate different sets of breakpoints and coefficients for different values of k . The hardware implementation does not change in any way to generate a sigmoid function with any fixed value of k . In order to generate a programmable sigmoid function with different values of k , the current implementation, with minor modifications, would allow up to 4 values of k assuming 8 intervals, 3 values of k assuming 10 intervals and 2 values of k assuming 16 intervals. The current implementation can accommodate up to 32 breakpoint and coefficient sets. The changes that need to be implemented to make the current implementation programmable are an input to feed k and the capability to use this value of k as an index to select appropriate breakpoint and coefficient sets.

The next chapter presents the statistics on the area usage for the implementation of the above function in an FPGA. It also presents the comparisons of resources and accuracy for various methods used for hardware implementation of the sigmoid activation function.

Chapter 6

Results and Comparisons

The figures below show the error and percentage error of different methods (digital) to implement the sigmoid function as presented in chapter 4.

6.1 Error Comparison for Different Methods

Table 6.1 gives the maximum error and the percentage error for all the different methods for hardware implementation (digital) of the sigmoid function.

Method (for the range -8 ≤ x ≤ 8)	Maximum Error	Maximum Percentage Error	
		Initial (at x = - 8)	Steady State (at x = 0)
Dataflow approximation	0.0823	1.6466×10^{-4}	11.8386
Taylor Series approximation	0.1113	3.3196×10^{-4}	11.1361
Least square approximation method 1	0.1631	1163.84	18.4714
Least square approximation method 2 (for the range $-4 \leq x \leq 4$)	0.0216	100 (at x = - 4)	2.2217
Piecewise approximation method 1	0.049	1615.0518	7.043
Piecewise approximation method 2	0.0189	482.4137	7.043
Piecewise approximation method 3	0.0445	482.4137	11.8386
Genetic Algorithm approximation	2.4736×10^{-3}	737.015	0.2472

Table 6.1. Maximum error and percentage error of different implementations.

For the analog implementation the relative error between the simulated neuron activation function and the fitted sigmoid is not more than 3 % [16] and the relative error between the simulated derivative and derivative of simulated neuron activation function is less than 5 % [16]. Table 6.1 also shows the genetic algorithm method to be the best among all the methods described. The maximum error and maximum percentage error for the genetic algorithm method is 0.00011197 and 0.0112 % respectively. Figure 6.1 captures the maximum error of all the methods. In Figure 6.1 the GA approximation and the Taylor series approximation seems to have similar error which is not the case. The GA approximation has much better error as compared to Taylor series approximation and this is shown in Figure 6.2 and Figure 6.3. Taylor series method gives maximum error at the end of the input range i.e. $x = 8$. Two graphs are shown in order to give a clear comparison between GA method and Taylor series method. The error and percentage error for each method is shown separately in Figure 6.4 – Figure 6.20.

Figure 6.1. Errors of all the methods.

Figure 6.2. Error between genetic algorithm approximating and Taylor series approximation [14] in the range $0 \leq x \leq 4$.

Figure 6.3. Error between genetic algorithm approximating and Taylor series approximation [14] in the range $4 \leq x \leq 8$.

Figure 6.4. Error between sigmoid function and dataflow approximation (Matlab) [8].

Figure 6.5. Percentage error between sigmoid function and dataflow approximation (Matlab) [8].

Figure 6.6. Error between sigmoid function and Taylor series approximation (Matlab) [14].

Figure 6.7. Percentage error between sigmoid function and Taylor series approximation (Matlab) [14].

Figure 6.8. Error between sigmoid function and least square method approximation 1 (Matlab) [22].

Figure 6.9. Percentage error between sigmoid function and least square method 1 approximation (Matlab) [22].

Figure 6.10. Error between sigmoid function and least square method approximation 2 (Matlab) [22].

Figure 6.11. Percentage error between sigmoid function and least square method 2 approximation (Matlab) [22].

Figure 6.12. Error between Sigmoid function and Piecewise approximation method 1 (Matlab) [18].

Figure 6.13. Percentage error between sigmoid function and piecewise approximation method 1 (Matlab) [18].

Figure 6.14. Error between sigmoid function and piecewise approximation method 2 (Matlab) [1].

Figure 6.15. Percentage error between sigmoid function and piecewise approximation method 2 (Matlab) [1].

Figure 6.16. Error between sigmoid function and piecewise approximation method 3 (Matlab) [10].

Figure 6.17. Percentage error between sigmoid function and piecewise method approximation (Matlab) [10].

Figure 6.18. Error between sigmoid function and GA approximation (Matlab) [9].

Figure 6.19. Percentage error between sigmoid function and GA approximation (Matlab) [9].

- - - - - Relative error between simulated neuron activation function and fitted sigmoid
 ——— Relative error between simulated derivative and derivative of simulated neuron
 activation function

Figure 6.20. Relative errors of analog implementation [16].

6.2 Hardware Usage and Timing Estimate

The hardware usage for most of the methods above is not available; hence we present a very rough estimate for the hardware requirements for the different methods.

This estimate takes into consideration only the components which consume maximum resources and significantly affect the hardware requirement and the time required to compute the sigmoid function. We assume the multipliers to be shift and add and dividers to be non-restoring. Non restoring dividers are considered for their compatibility for 2's complement notation for dividend and divisor. The quotient has to be converted to 2's complement. It performs a series of shift and add or subtract operations to achieve division. In the estimates below, n is the number of shifts

and adds or shifts and subtracts. In our work, $n = 8$, since we are using Booth encoding on 16-bit inputs.

6.2.1 Dataflow Implementation

This method uses the equation below to implement the sigmoid function.

$$f(x) = \frac{1}{2} \left(\frac{x}{1 + |x|} + 1 \right)$$

This requires a divider which can be estimated with n subtractions, hence it requires a subtractor (equivalent to an adder) and takes n cycles for every computation of the sigmoid function.

6.2.2 Lookup Table Method

The look up table method is fast. In this method a linear interpolation is performed to look up the value of the sigmoid function. The error introduced by this method depends on two factors.

- The platform's memory storage precision for storing the lookup table. It is referred to as the quantization precision.
- The value of lookup table's step t . When the step t is much smaller, the transfer function approximate is much accurate.

The memory size increases with the increase in precision and the decrease in step size [21].

6.2.3 Taylor Series Expansion

The equations implemented in this method are as given below

$$\begin{aligned} f(x) &= 0.571859 + (0.392773)x + (0.108706)x^2 + (0.014222)x^3 + (0.000734)x^4 \quad \text{for } -\infty < x \leq -1.5 \\ &= 0.5 + (0.25)x - (0.020833)x^3 + (0.002083)x^5 \quad \text{for } -1.5 < x < 1.5 \\ &= 0.428141 + (0.392773)x - (0.108706)x^2 + (0.014222)x^3 - (0.000734)x^4 \quad \text{for } 1.5 \leq x < \infty \end{aligned}$$

This method uses polynomial of the fifth order to implement the sigmoid function. We assume this to require a multiplier, which is estimated to use n additions. It would be used 5 times. Hence it requires an adder and takes $5n$ cycles for every computation of the sigmoid function.

6.2.4 Least Square Approximation Method 1

The equations required to be implemented in this method are as given below

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

$$f(x) = 1 \quad \text{if } x > \text{highlimit}$$

$$f(x) = 1 - \text{gain}(\text{highlimit} - x)^2 \quad \text{if } 0 \leq x \leq \text{highlimit}$$

$$f(x) = 0 + \text{gain}(\text{highlimit} + x)^2 \quad \text{if } \text{lowlimit} \leq x \leq 0$$

$$f(x) = 0 \quad \text{if } x < \text{lowlimit}$$

where highlimit and lowlimit are the input beyond which the output is considered to be either +1 or 0 respectively. The highlimit value is known hence the value of gain is also known. It requires computing a square value and multiplication of that value with another value. Hence we estimate that it requires an adder and takes $2n$ cycles for every computation of the sigmoid function.

6.2.5 Least Square Approximation Method 2

The equations used by this method to implement the sigmoid are as given below

$$y = 1 - 2^{-1}(1 - |2^{-2}x|)^2 \quad 0 < x < 4$$

$$= 2^{-1}(1 - |2^{-2}x|)^2 \quad -4 < x < 0$$

This method implements the sigmoid function in the input value $-4 < x < 4$. Most computations in the equation are implemented by shifting. This method requires computation of a square for every computation of the sigmoid value; hence we estimate this method to require n cycles.

6.2.6 Piecewise Approximation 1

This method involves computing the slope of a line and using that slope in the equation of a line to compute the value of the function. As the coordinates are decided early on, the slope is fixed and need not be computed every time. Hence this method needs to compute

$$f(x) = mx + c$$

where m = slope of the line

c = constant value

We estimate this method to require an adder and n cycles for every computation of the sigmoid function.

6.2.7 Piecewise Approximation 2

The equations implemented in this method are

$$y = \frac{2 + \ddot{x}}{2^{(x)+2}} = \frac{\frac{1}{2} + \frac{\ddot{x}}{4}}{2^{(x)}} \quad \text{for negative values of } x$$

$$y = \frac{2^{(x)+2} + \ddot{x} - 2}{2^{(x)+2}} = 1 - \frac{\frac{1}{2} - \frac{\ddot{x}}{4}}{2^{(x)}} \quad \text{for positive values of } x$$

Where $\ddot{x}$ = decimal part of x

(x) = integer part of x

All the divisions in the above computations are powers of 2, hence we estimate this computation to require less than n cycles. It would require the cycles necessary to perform the necessary bit shifting.

6.2.8 Piecewise Approximation 3

The equations implemented in this method are

$$\begin{aligned} f(x) &= \frac{1}{2} * \frac{1}{2^{\text{int}(x)}} * \frac{1}{1+\text{frac}(x)} & x \leq 0 \\ &= 1 - \frac{1}{2} * \frac{1}{2^{\text{int}(x)}} * \frac{1}{1+\text{frac}(x)} & x > 0 \end{aligned}$$

The division with powers of 2 in the divisors can be achieved by shifting. We estimate this method to require n cycles for division with $1 + \text{frac}(x)$.

6.2.9 Analog Implementation

This method requires 15 transistors for implementing the approximation of sigmoid function and its derivative.

6.2.10 Genetic Algorithm Approximation

In this method the sigmoid function is approximated by the equation

$$f(x) = Ax^3 + Bx^2 + Cx + D$$

This equation is implemented using nested multiplication as

$$f(x) = (((Ax + B)x + C)x + D)$$

In line with the above comparisons this would require $3n$ cycles for every computation of the sigmoid function.

Table 6.2 gives a comparison between the different methods in terms of the number of cycles and the number of intervals required to implement the sigmoid function and Table 6.3 gives the summary of the resources required to implement the genetic algorithm approximation method in

the Altera Flex10K FPGA. The Altera timing analyzer shows that the design has a maximum delay of 14.6 ns, thus allowing a clock speed of up to 68 Mhz to be used.

Method	Number of intervals	Number of cycles
Dataflow approximation	1	n
Taylor Series approximation	3	5n
Least square approximation 1	2	2n
Least square approximation 2	2	n
Piecewise approximation method 1	7	n
Piecewise approximation method 2	2	< n
Piecewise approximation method 3	2	n
Genetic Algorithm approximation	8	3n

Table 6.2. Number of cycles and intervals required by different methods.

Device	Memory Bytes (EAB)	LCs	LCs % utilized	Delay (sec)
EPF10K70RC240-2	320	1679	44 %	14.6×10^{-9}

Table 6.3. Resources required to implement the genetic algorithm approximation method.

The function evaluator is designed to accommodate 32 breakpoints and the memory requirement shown in Table 6.3 is for 32 breakpoints and coefficient sets. We implement the sigmoid function with 8 intervals; hence require only a quarter of the memory bits shown above. We

incorporate the function evaluator in the neural network processing element implemented in [26] and compare the hardware requirements of our implementation with that of [26]. Table 6.4 gives the summary of the resources required by [26].

Device	Memory Bytes for NN Weights (EAB)	Memory Bytes for sigmoid lookup (off chip)	LCs	LCs % utilized
EPF10K70RC240-2	64	32 K	807	21 %

Table 6.4. Summary of resources required by a processing element implemented in [26].

[26] uses lookup table method to implement the sigmoid activation function in the processing element. It uses 64 bytes of FPGAs on board memory (EAB) to implement the local RAM block to store the Neural Network (NN) weights. The memory required by the instruction RAM is not present in the above table. The lookup table has 16K values of 2 bytes each to implement the sigmoid function. Hence the lookup table requires 32 KB of memory. This method uses separate 32 kilobytes (KB) memory to implement the lookup table as the whole lookup table does not fit in the memory available on the FPGA board.

Device	Memory Bytes for NN Weights (EAB)	Memory Bytes (breakpoint, coefficients)	LCs	LCs % utilized
EPF10K70RC240-2	64	320	2321	61 %

Table 6.5. Summary of resources required by a processing element in [26] with the lookup table replaced with genetic algorithm sigmoid function approximation.

Table 6.5 gives the summary of the resources required by the processing element in [26] when the sigmoid function lookup table is replaced by the genetic algorithm approximation. The breakpoints and the coefficients are 2 bytes each and we implement the sigmoid function using 8 breakpoints and coefficient sets (A, B, C, D). Hence the memory requirement for the sigmoid implementation is 80 bytes. We have implemented the breakpoint and coefficient ROMs to accommodate 32 breakpoints and coefficient sets respectively, hence it uses 320 bytes for our implementation. The lookup table is eliminated and it leads to the huge memory savings of 32 KB. This is at a one time expense of 8 clock cycles to initialize the breakpoint and coefficient ROM's and 5 clock cycles each time the sigmoid function is computed instead of 2 clock cycle required by the lookup table method.

6.3 Practical Considerations

The above discussion considers implementation of the sigmoid function in the range $-8 \leq x \leq 8$. The corresponding output $f(x)$ varies in the range of $0.0003 \leq f(x) \leq 0.9996$. Consider the range of input range $-4 \leq x \leq 4$ in which the corresponding sigmoid output varies in the range $0.0179 \leq f(x) \leq 0.9820$. The sigmoid function changes by a very small value of 0.0176 when the input x varies from -8 to -4 or 4 to 8 . The active range of input x which causes the sigmoid function to traverse the interval from 0.0179 to 0.9820 is from -4 to 4 and this is the range that is of practical importance and often used [27].

If the sigmoid function is implemented in the range of input $-4 \leq x \leq 4$ using the current implementation, it would require fewer breakpoints and would present many advantages.

- The method yields better accuracy (max error = 1.5728×10^{-4}) for input range $-4 \leq x \leq 4$

- More breakpoints (intervals) can be accommodated, allowing even better accuracy
- For a programmable sigmoid function, more values of k can be accommodated
- It would require fewer initial clock cycles to initialize the breakpoints

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Our goal was replacing the lookup table method of implementing the sigmoid activation function with a hardware implementation. We have successfully used a genetic algorithm approximation method to compute the sigmoid activation function in an FPGA. The method is highly accurate and has a maximum error of 2.4736×10^{-3} , which is about eight times better than the next closest maximum error of 0.0189, of piecewise approximation method 2. We use 8 intervals to achieve this error. Still smaller maximum error would require more intervals but this does not affect the computation time other than the time (cycles) required for one time initialization of breakpoints, which is equal to the number of breakpoints used. Altera's timing analyzer derives the maximum circuit delay as 14.6×10^{-9} which allows the circuit, if implemented in an Altera 10K part, to operate at a maximum frequency of 68 MHz. The fast multipliers used for the implementation can be used for any other computations required by the system in which this method is incorporated. This would indirectly save resources. In this thesis, the sigmoid function is implemented as a case study, but the genetic algorithm approximation method can be used for general purpose function approximation where high accuracy is desired. The hardware implementation except for the number of intervals required, is independent of the change in function.

7.2 Future work

The necessary modifications can easily be done to implement a programmable sigmoid function. As mentioned above, the GA function approximation method can be employed as a general purpose function evaluator and can be used to implement any function. The implementation is capable of accommodating up to 32 breakpoints and coefficient sets. Very high accuracy is achieved by only 8 intervals. Multiple activation functions can be implemented with the proper selection method. The GA [9] can be used to generate the breakpoints and coefficient sets. This method has very good accuracy and consumes reasonable resources. It allows implementation of accurate neural networks in embedded digital processors. Thus it should be possible to apply this work to problems requiring intelligent computation for systems on a chip.

Bibliography

- [1] Alippi, C., Gajani, G. S., Simple Approximation of Sigmoidal Function: Realistic Design of Digital Neural Networks Capable of Learning, *International Symposium on Circuits and Systems IEEE-ISCAS*, Singapore 11-14 June 1991, pp. 1505-1508.
- [2] Altera, <http://www.altera.com>, August 01, 2003
- [3] Ashenden, Peter, *The Designer's Guide to VHDL*, Morgan Kaufmann, San Francisco, CA, 1995.
- [4] Beiu, V., Peperstraete, J. A., Vandewalle, J., Lauwereins, R., VLSI Complexity Reduction by Piecewise Approximation of the Sigmoid Function, *European Symposium on Artificial Neural Networks*, Brussels (Belgium), pp. 181-186, 20-21-22 April 1994.
- [5] Booth, A.D., A Signed Binary Multiplication Technique, *Quarterly Journal, Mechanics and Applied Mathematics*, Vol 4, Part 2, pp. 236-240, 1951.
- [6] Cavanagh, J. J. F., *Digital Computer Arithmetic*, McGraw-Hill Company, 1984.
- [7] Cheng, R., Gen, M., *Genetic algorithms and engineering design*, John Wiley and Sons Inc., New York, 1997
- [8] Guccione, Steven A., Gonzalez, Mario J., A Neural Network Implementation Using Reconfigurable Architectures, *More FPGAs*, Will Moore and Wayne Luk, Abingdon EE&CS Books, Abingdon, England, pp. 443-45, 1993.
- [9] Hauser, James, "Approximation of Nonlinear Functions for Fixed-Point and ASIC Applications Using a Genetic Algorithm", Doctoral Thesis, ECECS Department, University of Cincinnati, 2001.
- [10] Hernandez, M.L.G., Ascencio, R.R.L., Galicia, C.A., An Artificial Neural Network on a Complex Programmable Device as a Virtual Sensor, *Proceedings of the International*

- Symposium on Robotics and Automation - ISRA'98*, Saltillo, Coahuila, Mexico, December 12-14, 1998.
- [11] Karayiannis, N. B., Venetsanopoulos, A. N., *Artificial Neural Networks: Learning Algorithms, Performance Evaluation, and Applications*, Kluwer Academic Publishers, Boston, 1993
 - [12] Kothari, R., Agyepong, K., On Lateral Connections in Feed-Forward Neural Networks, *Proc. IEEE International Conference on Neural Networks (ICNN)*, pp. 13-18, 1996.
 - [13] Kramer, A. H, Array Based Computation: Principles, Advantages and Limitations, *IEEE Proceedings of MicroNeuro*, pp. 68-79, 1996,
 - [14] Leon, Arroyo M. A., Castro, Ruiz A., Ascencio, Leal R.R, An Artificial Neural Network on a Field Programmable Gate Array as a Virtual Sensor, *The Third International Workshop on Design of Mixed-Mode Integrated Circuits and Applications*, Puerto Vallarta, Mexico, July 26-28, pp. 114-117, 1999.
 - [15] Lisboa, P. G. J., *Neural Networks: Current Applications*, Chapman and Hall, New York, 1992
 - [16] Lu, S., Shi, B., Circuit Design of an Adjustable Neuron Activation Function and Its Derivative, *Electronics Letters*, Vol. 36, No. 6, 16th March 2000.
 - [17] Murtagh, P., Tsoi, A., C., Implementation Issues of Sigmoid Function and its Derivative for VLSI Digital Neural Networks, *IEEE Proceedings-E*, Vol. 139, No. 3, May 1992
 - [18] Myers, D. J., Hutchinson, R. A., Efficient Implementation of Piecewise Linear Activation Function for Digital VLSI Neural Networks, *Electronics Letters*, Vol. 25, No. 24, 23rd November 1989.

- [19] Palnitkar, S., *Verilog HDL: A Guide to Digital Design and Synthesis*, SunSoft Press, Mountain View, California, 1996.
- [20] Parhami, B., *Computer Arithmetic: Algorithms and Hardware Designs*, Oxford University Press, New York, 2000.
- [21] Peng, W., Brugge, M., Nijhuis, J., Spannenburg, L., Mapping a Trained Neural Network on Hardware, *Soft Computing Symposium*, 1999
- [22] Sammut, K. M., Jones, S. R., Implementing Non Linear Activation Functions In Neural Network Emulators, *Electronics Letters*, Vol 27, No. 12, 6th June 1991.
- [23] Smith, D. R., *Digital Transmission Systems*, Van Nostrand Reinhold, 1985, pp. 75-88.
- [24] Synopsys, <http://www.synopsys.com>, April 23, 2003
- [25] Xilinx, <http://www.xilinx.com>, July 16, 2003
- [26] Zhao, Minyi, FPGA-based Simulation for Lateral Feed-Forward Neural Networks, Masters Thesis, ECECS Department, University of Cincinnati, 2000.
- [27] Zohdy, H. S., Al-Nsour, M., Implementation of Programmable Digital Sigmoid Function Circuit for Neuro-Computing, *Proceedings of IEEE Midwest Symposium on Circuits and Systems*, pp. 571-574, August 1998.
- [28] Zohdy, H. S., Hiasat, A. A., VLSI Design and Implementation of An Improved Squaring Circuit by Combinational Logic, *Proceedings of 31st Annual Asilomar Conference on Signals, Systems, and Computers*, pp. 426-429, Pacific Grove, CA, November 1997.
- [29] Zohdy, H. S., Hiasat, A. A., Combinational Logic Approach for Implementing an Improved Approximate Squaring Function, *IEEE Journal of Solid State Circuits*, Vol. 34, No. 2, pp. 236-240, February 1999.