To the Graduate Council:

I am submitting herewith a thesis written by Sampath Kumar Kothandaraman entitled "Implementation of Block-based Neural Networks on Reconfigurable Computing Platforms." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

Dr. Gregory D. Peterson
Major Professor

We have read this thesis and
recommend its acceptance:

Dr. Seong G. Kong

Dr. Syed K. Islam

Accepted for the Council:
Anne Mayhew
Vice Chancellor and
Dean of Graduate Studies

(Original signatures are on file with official student records.)

IMPLEMENTATION OF BLOCK-BASED NEURAL NETWORKS ON RECONFIGURABLE COMPUTING PLATFORMS

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

Sampath Kumar Kothandaraman

August 2004

Dedicated to my family, teachers and friends

ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Dr. Peterson for all the guidance and help he has given me throughout my graduate studies. I also thank him and Dr. Kong for giving me the opportunity to work on the "Block-based Neural Networks" project. This work would never have been complete without the help of Dr. Chandra Tan, from whose extensive working knowledge with the different EDA tools, I have benefited a lot. I also thank Dr. Islam for serving on my thesis committee.

I am very grateful to Analog Devices Inc. (ADI) for supporting me with a fellowship for the year 2003. The design techniques and tricks that I learnt at ADI, Greensboro have helped me greatly in successfully completing this thesis. I also thank the Department of Electrical and Computer Engineering for the financial support they provided me during my graduate study. I enjoyed the work assigned to me. I thank the National Science Foundation for partially supporting this work under grants 0311500 and 0319002.

Special thanks Mahesh for sharing his extensive working knowledge with the Pilchard Reconfigurable Computing platform and help me fix bugs in my design at various stages of this project. I also thank Fuat, Ashwin, Sidd, Saumil, Venky, and Narayan for all their help - I have enjoyed working with each one of them and have gained in knowledge with their tips. Last, and surely not the least, I thank my family, teachers, relatives and friends for all their support. Nothing would be possible without their good wishes.

ABSTRACT

Block-based Neural Networks (BbNNs) provide a flexible and modular architecture to support adaptive applications in dynamic environments. Reconfigurable computing (RC) platforms provide computational efficiency combined with flexibility. Hence, RC provides an ideal match to evolvable BbNN applications. BbNNs are very convenient to build once a library of neural network blocks is built. This library-based approach for the design of BbNNs is extremely useful to automate implementations of BbNNs and evaluate their performance on RC platforms. This is important because, for a given application there may be hundreds to thousands of candidate BbNN implementations possible and evaluating each of them for accuracy and performance, using software simulations will take a very long time, which would not be acceptable for adaptive environments. This thesis focuses on the development and characterization of a library of parameterized VHDL models of neural network blocks, which may be used to build any BbNN. The use of these models is demonstrated in the XOR pattern classification problem and mobile robot navigation problem. For a given application, one may be interested in fabricating an ASIC, once the weights and architecture of the BbNN is decided. Pointers to ASIC implementation of BbNNs with initial results are also included in this thesis.

TABLE OF CONTENTS

CHAPTER 1 1

INTRODUCTION 1

1.1 Neural Networks 1

1.2 Block-based Neural Networks 3

1.3 Optimization of Block-based Neural Networks 7

1.4 Reconfigurable Computing (RC) 9

1.5 BbNNs on RC Platforms 10

1.6 Neural Networks on-chip - Related Work 11

1.6.1 Analog IC Implementations of NNs 12

1.6.2 Digital Implementations of NNs on ASICs and FPGAs 12

1.7 Scope and Outline of Thesis 15

CHAPTER 2 16

APPROACHES FOR IMPLEMENTING BBNNs ON HARDWARE 16

2.1 VHDL and Design Flow 16

2.2 Design-for-reuse 19

2.3 Pilchard – A Reconfigurable Computing Platform 21

2.4 Approaches for Implementing BbNNs on Hardware 24

2.4.1 Library of NN blocks and activation functions 24

2.4.2 Other possible approaches 27

2.4.3 ASIC Implementations of BbNNs 28

CHAPTER 3 29

IMPLEMENTATION DETAILS 29
3.1 Detailed design flow used to target Pilchard RC platform 29
3.2 Library of NN blocks 32
3.2.1 The Sum-of-Products function 32
3.2.2 The Activation function 36
3.2.3 The three types of NN blocks 38
3.3 Facilitating Designs with feedback - Blocks with registered outputs 39
3.4 Accuracy and Resolution Considerations 41
3.5 Validation of NN blocks 43
3.5.1 Methodology 43
3.5.2 Software Implementation 43
3.5.3 VHDL Design and Simulation 43
3.5.4 Hardware Implementation 44
3.6 Characterization of the library of NN blocks 44
3.6.1 Area 44
3.6.2 Delay 44
3.6.3 Power 46
3.7 ASIC Implementation of one data path of a 2-input-2-output NN block 46
CHAPTER 4 50
CASE STUDIES AND RESULTS 50
4.1 Building BbNNs using the library of NN blocks 50
4.2 The XOR Pattern Classification Problem 51
4.2.1 Background 51

4.2.2 BbNN for XOR Pattern Classification 51
4.2.3 Implementation on Pilchard RC platform 53
4.2.3.1 Data format 53
4.2.3.2 Data organization in Xilinx® Block RAM 54
4.2.3.3 Read and Write operation timing diagrams for Xilinx® Block RAM 56
4.2.3.4 State Machine implementation 56
4.2.3.5 Host C Program 58
4.2.4 Results 61
4.2.4.1 XOR Pattern Classification Results 61
4.2.4.2 Area 61
4.2.4.3 Speed 61
4.2.4.4 Speed- up achieved with hardware 61
4.3 The Mobile Robot Navigation Control Problem 63
4.3.1 Background 63
4.3.2 BbNN for Navigation Control 64
4.3.3 Implementation on Pilchard 66
4.3.3.1 Data format 66
4.3.3.2 Data organization in Xilinx® Block RAM 66
4.3.3.3 State Machine implementation 66
4.3.4 Results 70
4.3.4.1 Mobile Robot Controller Results 70
4.3.4.2 Area 70

4.3.4.3 Speed 70
4.3.4.4 Speed-up achieved with hardware 70
CHAPTER 5 72
CONCLUSIONS AND FUTURE WORK 72
5.1 Conclusions 72
5.2 Future Work 72
REFERENCES 74
APPENDICES 79
VITA 119

LIST OF TABLES

Table 2.1: Xilinx® Virtex™ FPGA Device XCV1000E Product Features [26] 22

Table 2.2: Features of the Pilchard RC platform [4] 23

Table 3.1: Pin description of DesignWare™ generalized sum-of-products [33] 35

Table 3.2: Parameters in DesignWare™ generalized sum-of-products [33] 35

Table 3.3: Synthesis implementations of DesignWare™ generalized sum-of-products [33] 36

Table 3.4: Area (in number of slices of Virtex™ 1000E) for the three types of NN blocks with bipolar linear saturating activation function 45

Table 3.5: Performance comparison of the three types of NN blocks with bipolar linear saturating activation function 45

Table 3.6: Power estimates of the three types of NN blocks with bipolar linear saturating activation function 46

Table 4.1: XOR truth-table 51

Table 4.2: Details of state machine for XOR BbNN 59

Table 4.3: Speed-up for XOR BbNN 63

Table 4.4: Details of state machine for BbNN robot controller 69

Table 4.5: Speed-up for BbNN robot controller 71

LIST OF FIGURES

Figure 1.1: Representations of a single p-input neuron [1] 2

Figure 1.2: Representations of a p-input m-neuron single-layer neural network [1]..... 3

Figure 1.3: Examples of activation functions used in neural networks [1] 4

Figure 1.4: Four types of basic blocks used to build BbNNs [2]..... 5

Figure 1.5: Structure of a BbNN [2] 6

Figure 1.6: Signal flow representation of BbNN structure. (a) A 3x4 BbNN example (b) 2-D binary signal-flow representation of the BbNN..... 7

Figure 1.7: Optimizing BbNNs using genetic algorithms 8

Figure 1.8: FPGA structure..... 9

Figure 2.1: Basic design flow used for FPGA or ASIC implementations 17

Figure 2.2: Using generics in VHDL..... 20

Figure 2.3: Pilchard reconfigurable computing board 22

Figure 2.4: Block diagram of Pilchard board [4]..... 23

Figure 3.1: Detailed design-flow for implementation on Pilchard RC platform 30

Figure 3.2: Computation of the outputs of a NN block 34

Figure 3.3: DesignWare™ generalized sum-of-products IP block [33] 34

Figure 3.4: Arranging the inputs in vectors for sum-of-products computation [33] 34

Figure 3.5: Unipolar and bipolar saturating activation functions 37

Figure 3.6: Unipolar and bipolar linear saturating activation functions 37

Figure 3.7: Three types of NN blocks (a) Block22 (b) Block13 (c) Block31 38

Figure 3.8: NN blocks with registered outputs 39

Figure 3.9: Inferring registers in the design using VHDL 40

Figure 3.10: Architecture of ASIC implementation in AMI 0.6u process 48

Figure 3.11: ASIC implementation of $f(w_{13}x_1 + w_{23}x_2 + b)$ in AMI 0.6u process 49

Figure 4.1: Decision boundaries for the XOR pattern classification problem 52

Figure 4.2: BbNN used for XOR pattern classification [2] 52

Figure 4.3: Activation function used in XOR pattern classification 53

Figure 4.4: Data format of inputs, weights and biases in block13 55

Figure 4.5: Data organization in 256 x 64 dual port RAM for XOR BbNN 56

Figure 4.6: Read and write operations on Xilinx® Dual port RAM [34] 57

Figure 4.7: State diagram for XOR BbNN state machine 60

Figure 4.8: XOR pattern classification by BbNN 62

Figure 4.9: Layout of XOR BbNN 62

Figure 4.10: Robot with BbNN navigation controller 64

Figure 4.11: Saturation activation function for BbNN robot controller 65

Figure 4.12: Final structure and weights of BbNN controller for robot navigation 65

Figure 4.13: Data organization in 256 x 64 Xilinx® dual port RAM for BbNN robot controller 67

Figure 4.14: State diagram for BbNN robot controller state machine 68

Figure 4.15: Layout of BbNN robot controller 71

LIST OF ABBREVIATIONS

ANN – Artificial Neural Network

API – Application Programming Interface

ASIC – Application-Specific Integrated Circuit

BbNN - Block-based Neural Network

CAD – Computer Aided Design

CLB - Configurable Logic Block

CORDIC – Coordinate Rotation Digital Computer

DIMM – Dual In-line Memory Module

DLL – Delay Locked Loops

DRC – Design Rule Check

EDA – Electronic Design Automation

EDIF – Electronic Design Interchange Format

EHW – Evolvable Hardware

FPGA – Field-Programmable Gate Array

GA – Genetic Algorithm

HDL – Hardware Description Language

IC – Integrated Circuit

I/O – Input/Output

IP – Intellectual Property

ISE® – (Xilinx®) Integrated Synthesis Environment

JTAG – Joint Test Action Group

LUT – Look-Up Table

MLP – Multi-Layer Perceptron

NCD – Native Circuit Description

NGD – Native Generic Database

NN - Neural Network

OS – Operating System

PAL – Programmable Array Logic

PAR – Place and Route

PCF – Pin Constraints File

PCI – Peripheral Component Interconnect

PLA – Programmable Logic Array

PLD – Programmable Logic Device

RAM – Random Access Memory

RC – Reconfigurable Computing

ROM – Read-Only Memory

RTL – Register Transfer Level

SDF – Standard Delay Format

SDRAM – Synchronous Dynamic Random Access Memory

VHDL – VHSIC Hardware Description Language

VHSIC – Very High-Speed Integrated Circuit

VLSI – Very Large-Scale Integration

VME – Virtual Memory Extension

XOR – Exclusive OR

CHAPTER 1

INTRODUCTION

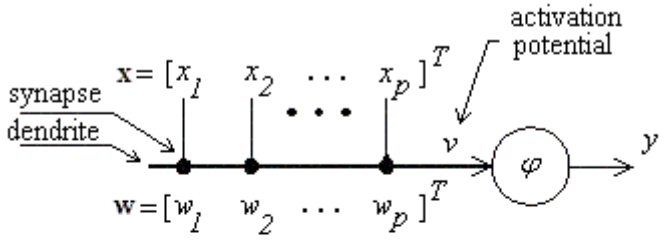
1.1 Neural Networks

A neural network (NN), or artificial neural network (ANN), is a set of processing elements or nodes, which are connected by communication links that have numerical weights associated with them. Each node performs operations on the data available at its inputs and collectively, these nodes (the NN) solve a more complex problem. ANNs are also called “connectionist classifiers” [1].

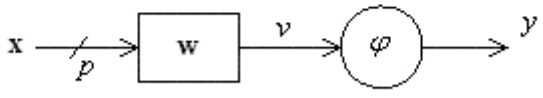
Historically, neural networks were inspired by Man’s desire to produce systems capable of performing complex functions, which the human brain performs [1]. Typically, NNs are “trained” using examples so that they “learn” from experience and adjust their weights and structure and can be applied to data beyond the training data. It can be safely said that NNs today are far from mimicking the functions of the human brain, yet are very useful for a variety of applications that have a lot of training data available. NNs are especially useful in applications which require complex problem solving and for those that have no algorithmic solution. The advantage of NNs lies in their resilience against distortions in the input data and their capability of “learning” [1]. Typical applications of NNs include pattern classification problems, pattern recognition, information (signal) processing, automatic control, cognitive sciences, statistical mechanics and so on.

Neural networks are built by interconnecting processing nodes called "neurons". A neuron (Figure 1.1) is usually an *n*-input, single-output processing element, which can be imagined to be a simple model of biological neuron [1]. Figure 1.2 shows the structure of a typical NN. The input signal is a vector, **x**. Each element of the vector is excited or inhibited by a *synapse*. Each synapse is characterized by an associated weight. Let the vector representing the weights be **w**. The synapses, which are essentially multipliers, are arranged along a *dendrite,* which aggregate the post-synaptic signals. An *activation function* φ is usually applied on the aggregated signal, resulting in the output signal. It is sometimes convenient to add a "bias" or "threshold", *b*, to the sum-of-products of weights and inputs. Thus, mathematically, a neuron computes the output according to:

$$y = \varphi(\mathbf{w}.\mathbf{x}^{\mathbf{T}} + b) = \varphi(w_1x_1 + w_2x_2 + \ldots + w_nx_n + b)$$



(a) Dendritic Representation



(b) Block diagram Representation

Figure 1.1: Representations of a single p-input neuron [1]

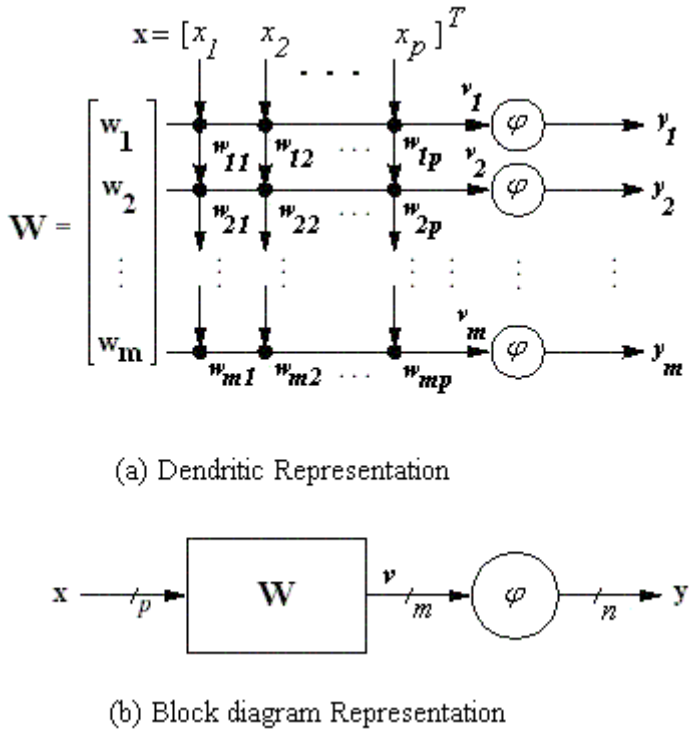


Figure 1.2: Representations of a p-input m-neuron single-layer neural network [1]

Figure 1.3 shows typical activation functions, which include linear or piecewise linear functions, step functions, sigmoid functions and so on. A NN is formed by arranging many neurons to form "layers". The most popular NNs are those consisting of an input layer, one or more hidden layers and an output layer [1].

1.2 Block-based Neural Networks

A Block-based Neural Network (BbNN) consists of a two-dimensional array of basic neural network blocks [2]. Each of these blocks has four input/output nodes, and is a simple feedforward NN. Each node can be either an input node or an output node, depending on the direction of signal flow. Thus, there are four basic configurations for

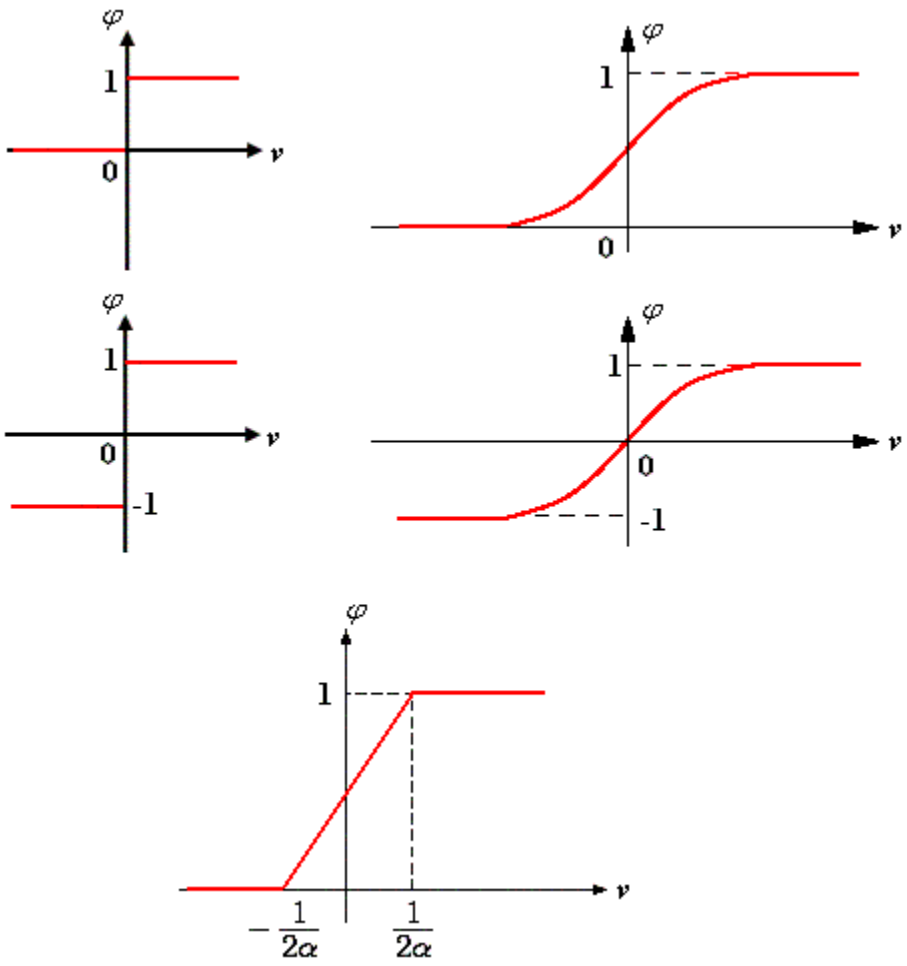


Figure 1.3: Examples of activation functions used in neural networks [1]

the blocks as shown in Figure 1.4. For easier implementation on hardware, BbNN models are restricted to have only integer or fixed-point weights.

Figure 1.5 shows the structure of an m x n BbNN model [2]. The first stage is the input stage (i = 1), and the last stage is the output stage ($i = m$). There can be as many as n inputs and n outputs. Each block is directly connected to its neighbors. The leftmost and rightmost blocks are also connected to each other. Signal flow between the blocks determines the overall structure of the BbNN as well as the internal structure of each of the blocks.

The modular structure of BbNNs makes them easy to implement on digital hardware such as field-programmable gate arrays (FPGAs). The highly modular structure

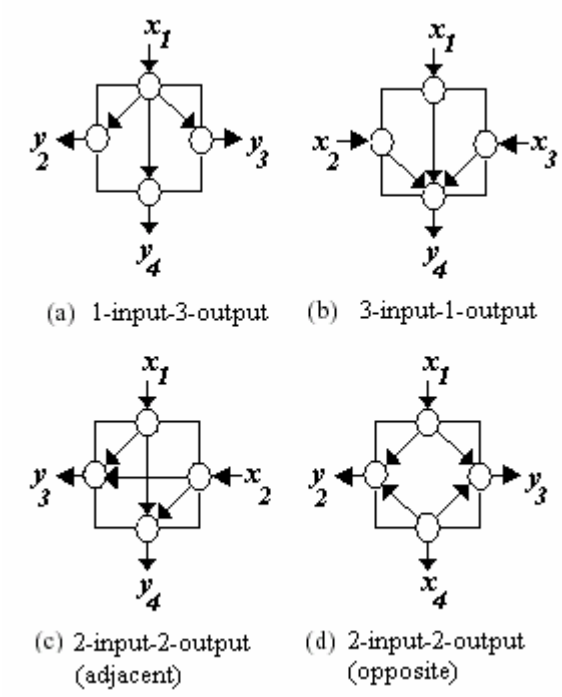


Figure 1.4: Four types of basic blocks used to build BbNNs [2]

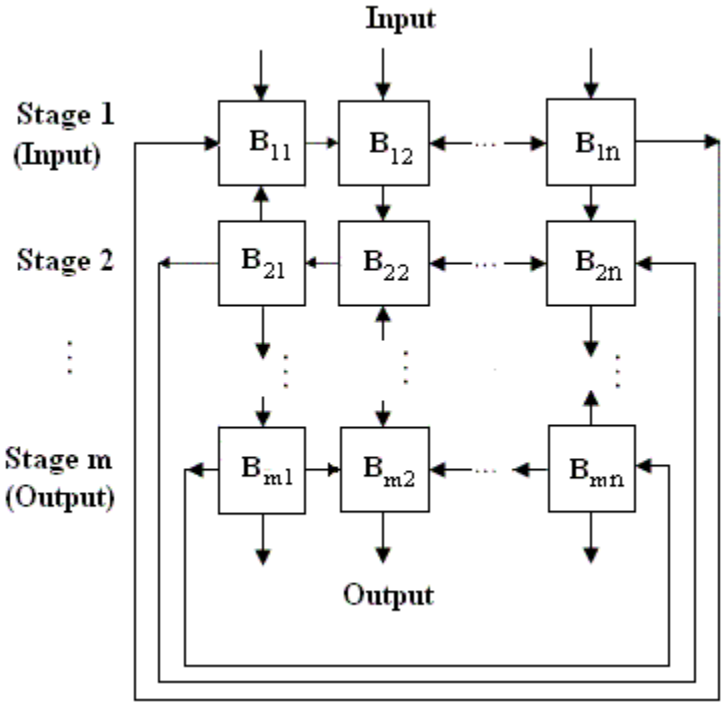


Figure 1.5: Structure of a BbNN [2]

of the BbNN enables an effective binary representation of BbNNs. An arrow between the blocks represents a signal flow. The signal flow bits between the blocks can represent network structure and internal configuration of the BbNN. A node with incoming signal flow is the input, and an output node has outgoing signal flow. A node output becomes an input to the neighboring blocks.

Figure 1.6 shows a signal flow representation of network structure that preserves 2D topological information of the BbNN. In Figure 1.6(a), all the connections between the basic blocks are represented with either 0 or 1. Bit 0 represents both downward (↓) and leftward (←) connections, while bit 1 represents upward (↑) and rightward (→) signal flows. Signal flow of all input and all output stage blocks are all 0. The number of bits to represent the signal flow of an $m \times n$ block-based neural network is $(2m\text{-}1)n$ [2].

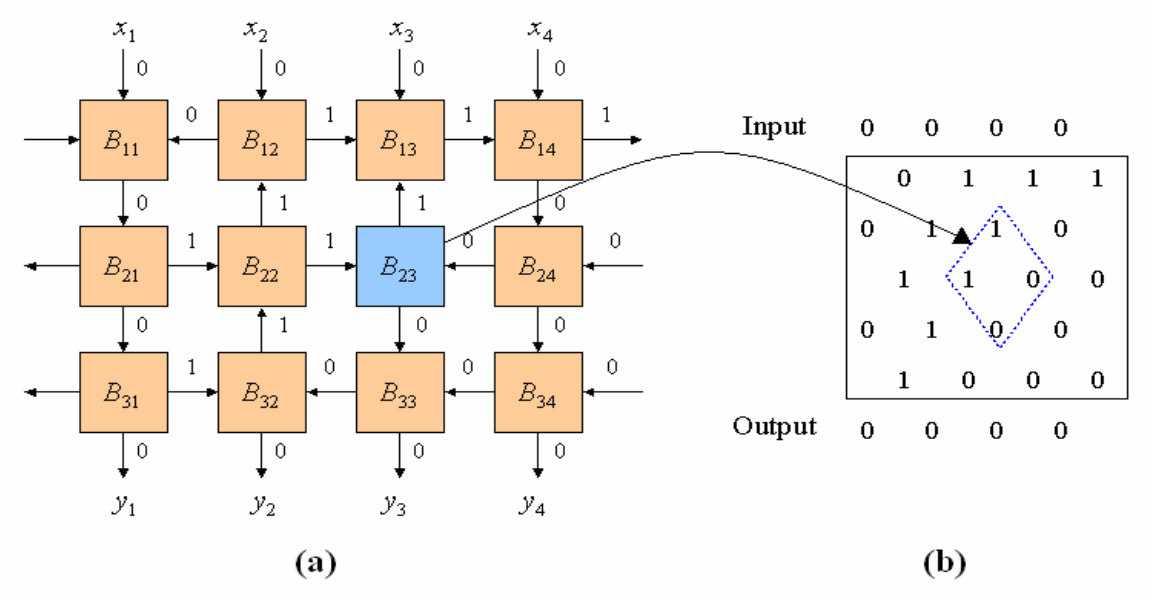


Figure 1.6: Signal flow representation of BbNN structure. (a) A 3x4 BbNN example (b) 2-D binary signal-flow representation of the BbNN

Figure 1.6(b) shows a 2D binary string that represents the signal flow and therefore the structure of the BbNN of size 3×4 given in Figure 1.6(a). Four bits in a diamond region represents the block B_{23}. Two neighboring blocks share signal-flow bits to ensure valid structures are available. Input and output signal flows are all 0s and not included in the binary representation.

1.3 Optimization of Block-based Neural Networks

Neural networks are optimized for structure and weights using a "learning" method. Sets of input data and corresponding output data are used to train the network. Once a near optimal structure and set of weights is established the NN is ready to operate on data outside the training data, to solve problems.

The structure and weights of a BbNN are encoded in bit-strings and are optimized using genetic algorithms. Different encoding schemes may be used to represent BbNNs. An encoding scheme is just a way of describing the BbNN in bit-strings or arrays, so that it can be used in the GA optimization process. The details of encoding schemes are beyond the scope of this thesis.

Figure 1.7 shows how genetic algorithms may be used to determine near-optimal structure and weights for a specific application. A set of candidate BbNNs, called "initial population", is randomly generated. Each candidate, represented in an encoded format for optimization using the genetic algorithm, is referred to as a *chromosome*. The fitness of these candidate architectures and weights is measured using a fitness function. A genetic algorithm evolves this population and the process of optimization continues until a population with acceptable fitness is obtained.

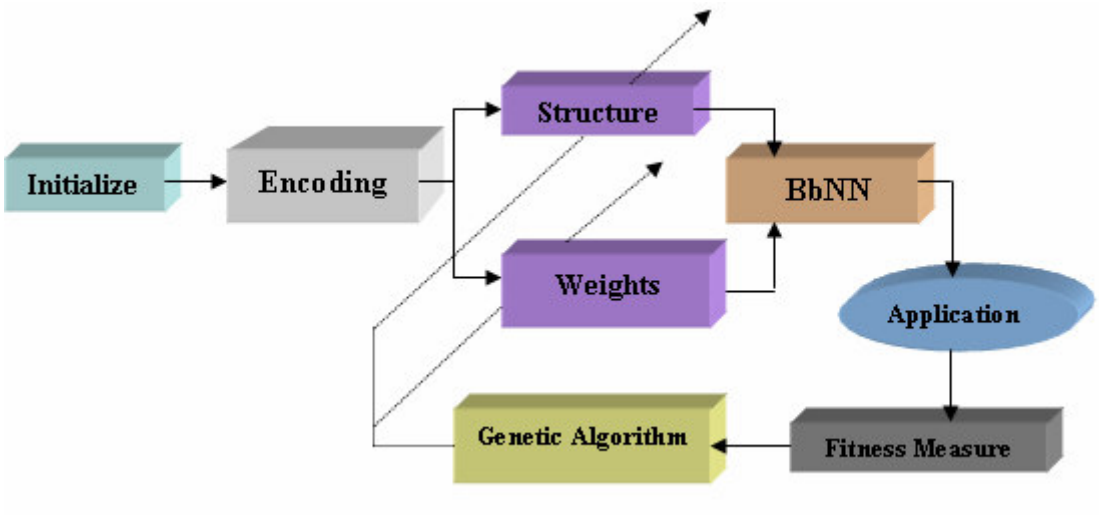


Figure 1.7: Optimizing BbNNs using genetic algorithms

1.4 Reconfigurable Computing (RC)

Reconfigurable computing is a computing paradigm that has matured greatly over the past decade. RC platforms are typically circuit boards housing one or more programmable integrated circuits, usually Field-Programmable Gate Arrays (FPGAs).

FPGAs have a regular structure consisting of a number of functional units and an interconnection fabric between them. The functional units are often implemented as look-up tables (LUTs), which can support general boolean formulas. By loading the appropriate bit patterns into the LUTs, each is configured to compute a specific boolean function. These LUTs are interconnected using a configurable communications fabric. Designers route the primary inputs to certain LUTs, the resulting intermediate values among other LUTs, and the final results to the output pins. With this flexibility in specifying the function of each LUT and their interconnection, one can implement any digital logic circuit subject to resource constraints. Figure 1.8 illustrates a typical FPGA structure. The RC boards are connected to a host microprocessor, allowing one to dynamically migrate functionality between the processor and the reconfigurable logic.

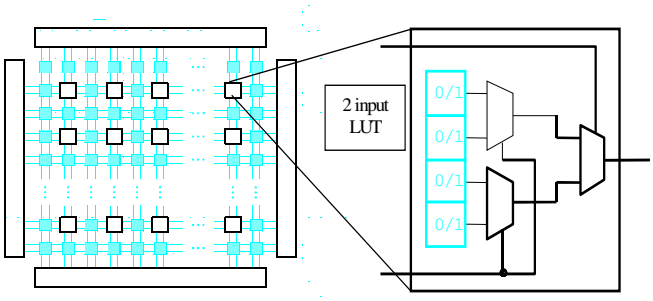


Figure 1.8: FPGA structure

This helps in implementing portions of the functionality in hardware (implemented on the FPGA) and other portions in software (running on the host processor). This makes it possible for the user to exploit inherent parallelism in algorithms and implement fast versions of it on hardware, while inherently serial tasks can be performed on the processor.

Various companies have built a wide variety of RC boards. These boards could differ in the number of FPGAs, capacity (transistor-count or gate-count) of the FPGAs, and the communication protocol used for processor-to-board communications. Latest FPGAs can accommodate millions of gates. The RC board is typically connected to the host processor through the PCI, VME or memory bus. Examples of RC boards include the SRC-6E [3] and Pilchard [4] systems that are attached to the memory bus, and platforms such as the Firebird™ and Wildcard™ from Annapolis Micro Systems Inc. [5], which can be connected to the PCI bus or other busses. A host program, running on the processor, usually controls the configuration and initialization of the FPGA(s) and communication between the processor and the RC board.

1.5 BbNNs on RC Platforms

Neural networks are inherently a collection of massively parallel processing nodes. This makes them an ideal fit for hardware implementation rather than traditional software implementations on microprocessors, which are essentially sequential computing engines. Also, many applications will require quick response times, which may not be achievable with software implementations. Hardware implementations on FPGAs are also economical. The flexible and modular architecture of BbNNs, which

makes them suitable for adaptive applications in dynamic environments, combined with flexibility (due to reconfigurability) and computational efficiency, in terms of speed and power-consumption (due to implementation on customized digital logic) of FPGAs, makes the implementation of BbNNs on RC platforms a natural choice. Various BbNN architectures, differing in structure and weights, may be downloaded onto the FPGA and evaluated for fitness, according to the flow described in Section 1.3. RC platforms seem to be the best environment to prototype candidate BbNNs for a given application, because the number of possible network structures increases exponentially with the network size (i.e., number of basic blocks used to build the BbNN). For example, 2 x 2 BbNNs can have 64 possible structures, while a 2 x 4 BbNN will be one of 4096 possible structures. The number of generations of chromosomes that have to be evaluated before a satisfactory solution is obtained could run into hundreds even for BbNNs with less than ten blocks. Exploiting the reconfigurability and computational efficiency of FPGAs seem the best way to zero-in on the final BbNN structure and weights. Of course, this would require using modified genetic algorithms and/or genetic operators to account for the encoding scheme used to represent the BbNNs.

1.6 Neural Networks on-chip - Related Work

A significant amount of work has been done in the field of NN simulation environments on von-Neumann (sequential) machines. These are summarized in [25]. However, hardware implementations are better suited for dynamic environments and adaptive NN applications like speech-recognition and robot navigation control, because of their quicker response time when compared to traditional software implementations.

There have been successful attempts for implementing NNs on hardware [6–18, 27, 28]. Digital, analog and hybrid ASICs have been used for NN implementation.

1.6.1 Analog IC Implementations of NNs

Analog electronic devices possess characteristics which may directly aid the implementation of NNs. For example, operations like multiplication and integration may be easily implemented using operational-amplifier circuits or even single transistors [24]. Also, the fact that analog VLSI chips interface more naturally to real-world signals (analog) makes them the fastest responding hardware implementation for a given problem. Analog circuits can consume less power when compared to their software and digital hardware implementations. However, the limitation with analog circuits is that they are not as suitable for dynamic environments and adaptive applications as they are generally not reconfigurable on the fly, and are difficult to design. Moreover, design of analog circuits becomes increasingly difficult as the noise immunity constraints get tighter. The circuit is also highly dependent on process-parameter variations, making prediction with a great degree of accuracy, difficult. [7] and [8] detail some of the issues and results achieved in the attempt to implement analog neural chips.

1.6.2 Digital Implementations of NNs on ASICs and FPGAs

Digital ASIC implementations for NNs are probably the most powerful hardware implementations of NNs today [24]. This is despite the fact that digital circuits for NN applications will most likely have less computational density (amount of computation per unit area on silicon), occupy more area and consume more power. The reason for this is that they are easier to design than analog chips and are more reliable and programmable. Examples of digital NN chips include CNAPS [27] and SYNAPSE-1 [28]. Custom

ASICs have the drawback that they are not reconfigurable and so provide very little scope for implementing NN for adaptive applications in dynamic environments. However they prove useful in applications where the NN has fixed structure and weights.

The first successful FPGA implementation of ANNs was published more than a decade ago [21]. In the GANGLION project [6], FPGAs were used for rapid prototyping of different ANN implementation strategies and for initial simulation or proof of concept. In the work done by Eldredge et al. [13], the "density enhancement" method was used, which essentially aims at increasing the functionality per unit area on the FPGA through reconfiguration. In this work, a back-propagation learning algorithm is divided into a series of feed-forward and back-propagation stages, and each is executed on the same FPGA successively. James-Roxby et al. [14] implemented a technique known as "dynamic constant folding" where an FPGA is shared over time (time-multiplexing) for different NN circuits. A 4-8-8-4 multi-layer perceptron (MLP) was implemented by them on a Xilinx® Virtex™ FPGA. Zhu et al. [15] explored similar techniques, where they exploited training-level parallelism with batch-updates of weights at the end of each training epoch [21]. Iterative construction of NNs [16] can be realized through "topology adaptation". During training, the topology and desired computational precision can be adjusted according to some criteria. GA-based evolution of NNs was used by de Garis et al. [29]. Zhu and Sutton [17] implemented Kak's fast classification NNs on FPGAs.

It is not that there are no difficulties involved in implementing NNs on digital hardware. Nonlinear activation functions and real-valued weights make it difficult to realize NNs on digital hardware. Usual methods of getting around this problem are to use piece-wise linear approximation of the function or use look-up tables. In either case, the

cost of implementation usually grows very rapidly as more resolution is demanded. Another problem with digital hardware implementations of NNs is the difficulty in accommodating floating-point precision weights. A few attempts have been made to implement NNs with floating-point weights, but none have reported any success [21]. Recent work by Nichols et al. [19] show that even with advances in FPGA capacities, it is impractical to implement NNs with floating-point precision weights. FPGA realizations for large NNs have been a problem in the past, because it is expensive to implement many multipliers on fine-grained FPGAs. Marchesi et al. [18] explored the possibility of implementing NNs without multipliers. They achieved this by restricting weights to be powers of two, or sums of powers of two, hence performing multiplication by a series of shift operations. This greatly reduces the area occupied by the design on the FPGA, but restricts the domain of weight-values, which is already reduced to being just integers or fixed-point numbers. The limited area resources on an FPGA may not be as significant an issue with today's and future-generation FPGAs, which will be capable of supporting circuitry containing millions of gates.

The capabilities of BbNNs have been proven using software simulations for applications such as pattern classification, pattern recognition and mobile robot navigation control [30] [31] [32]. This thesis describes the development of the first implementations of BbNNs on FPGAs and lays out a methodology to implement BbNNs for various applications using a library-based approach.

1.7 Scope and Outline of Thesis

The work presented in this thesis is part of a bigger research project, which aims at implementing BbNNs on digital hardware, analyzing encoding schemes, coming up with modified genetic operators and implementing evolution schemes for optimizing the BbNN using genetic algorithms, optimizing bit-widths, and addressing issues like design automation and runtime reconfiguration of the FPGAs. This thesis focuses on the development of VHDL models for the basic neural network blocks of a BbNN and putting them together to solve problems like the XOR pattern classification problem. The implementations target the Xilinx® Virtex™ 1000E FPGA [26], which is housed on the Pilchard RC board.

The rest of this document is organized as follows. Chapter 2 describes the approach adopted for the development of VHDL models and the basic design-flow used in digital design. The target RC platform, namely, the Pilchard RC board, is also briefly described in chapter 2. Chapter 3 discusses in detail, the implementation, verification and characterization of the library of basic NN blocks needed for building BbNNs on digital hardware. Preliminary investigations and results of ASIC implementations are also presented. In chapter 4, applications of BbNNs and their implementation using the library of NN blocks, on the Pilchard RC platform are described. The applications chosen for the demonstration are the XOR pattern classification problem and mobile robot navigation control problem. Results of these case-studies are analyzed. Conclusions from the research are drawn, and a discussion on future work is presented in chapter 5.

CHAPTER 2

APPROACHES FOR IMPLEMENTING BBNNs ON HARDWARE

2.1 VHDL and Design Flow

VHDL is a programming language used to model and test hardware designs. VHDL stands for VHSIC (Very High-Speed Integrated Circuit) Hardware Description Language. Programs written in VHDL can be used to simulate the functional behavior of hardware implementations of the design as well as its timing characteristics. VHDL programs may be "synthesizable", which means that they can be translated into physically realizable circuits using CAD tools. The target hardware for synthesis may be a programmable logic device (PLD) (like PALs, PLAs and FPGAs) or an ASIC. In either case, the job of the synthesis tool is to map the design described in VHDL to a physically realizable circuit, using the components available in a library. PLDs are often used for testing algorithms implemented in VHDL before using them to fabricate ASICs, because the functionality of ASICs cannot be altered once fabricated. However, with FPGAs today capable of housing circuits with millions of transistors, they are gaining popularity for applications like hardware acceleration and reconfigurable computing, where the design in the chip can be configured on the fly.

Figure 2.1 shows the basic design flow used to design test and implement hardware models using a hardware description language like VHDL. The target hardware may be an FPGA or an ASIC, the major difference in them being the library used by the synthesis tool for mapping the design into hardware. While in the case of targeting

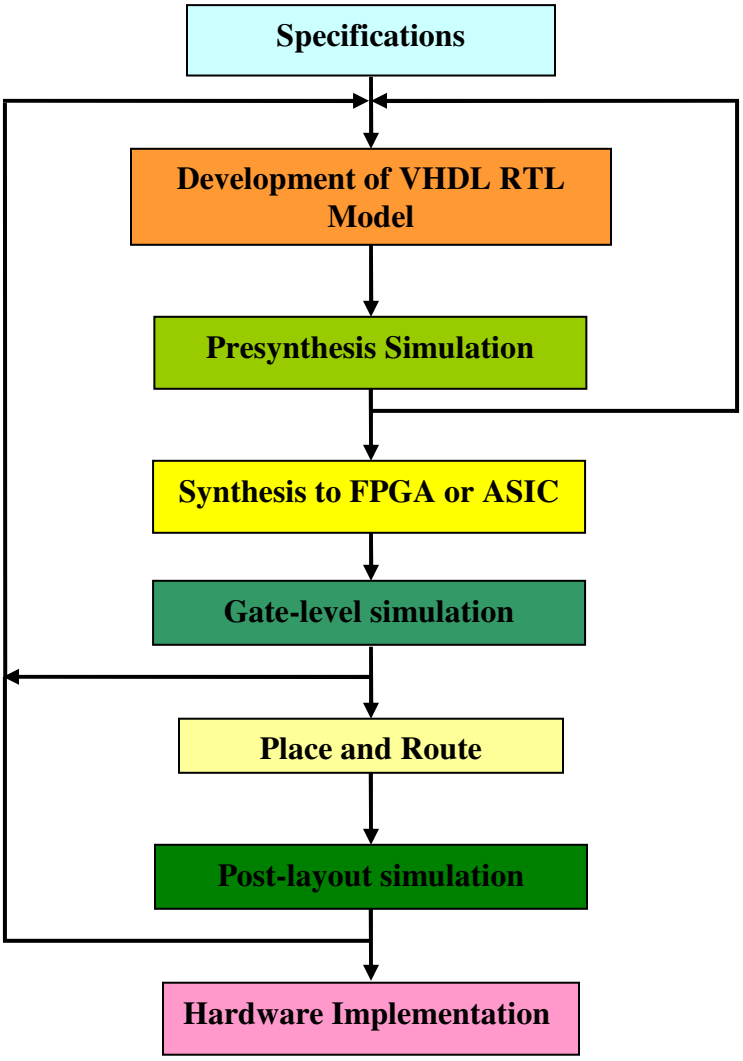


Figure 2.1: Basic design flow used for FPGA or ASIC implementations

FPGAs, the tool is provided with libraries describing the blocks available inside the FPGA fabric, for ASICs, a library of "standard height cells" of a particular technology, typically containing logic gates, flip-flops and latches, buffers and so on, are provided to the tool to use.

The first step in the design process is to understand the requirements, generate specifications and translate it into VHDL code, usually written at the Register-Transfer Logic (RTL) level. This has to be done by a human programmer and requires specialized skills. A "test-bench", also written in VHDL, is used to test the design by providing test-vectors and collecting and comparing responses with known "correct" results. This is known as pre-synthesis simulation of the design. There are many commercially available VHDL simulators which help the programmer in this phase.

Once the VHDL models have been verified for functional correctness using simulation, it is ready to be synthesized. Not all VHDL statements are synthesizable. This means that some statements used may not be physically realizable on hardware. For example, the **`wait`** statement in VHDL can be used only for simulation purposes, because it is not possible to introduce an element in hardware which will introduce an exact delay in the signal path. Synthesis is the process of converting a design from one level of abstraction to a lower level of abstraction. Thus, synthesis could mean, converting a design described in RTL level to gate-level, or from gate-level to switch-level and so on. Synthesis is done using CAD tools which usually accept design constraints like area, delay and power-consumption from the user and try to implement and optimize the design for one or more of these variables. Once synthesis is performed, the resulting implementation from the CAD tool is verified for functional correctness

using simulation. However, it is now common to use formal verification techniques to check that the implemented design (written out as a "netlist" from the synthesis tool) is equivalent to the design input to the tool.

Once synthesis and post-synthesis verification are completed successfully, Place and Route (PAR) CAD tools are used for the physical implementation of the design. As the name suggests, there are two basic tasks performed by the tools – placement of the different design blocks and routing the interconnections inside and between them. These tools work with the timing and area constraints provided by the user. A post-layout simulation is done to check whether the final placed-and-routed design meets the timing goals desired. It is not unusual to simply do a static timing analysis instead of regression testing (using test vectors) of the routed design. Equivalence checking as described before may be done again at this stage – this time, between the post-synthesis and post-PAR netlists. Timing analysis at this stage is the most accurate as the design includes both cell delays and the delays due to parasitic capacitance and resistance of the interconnecting wires. Any failure to meet timing constraints means that a more aggressive PAR has to be done in the next iteration or may be the synthesis or even the RTL coding may have to be modified.

2.2 Design-for-reuse

Design-for-reuse is a method of developing hardware designs so that they may be reused in other designs. This helps in greatly reducing design-time of future projects, though the development-time for the initial reusable code may take longer to develop than a regular application-specific code. Designs which can be reused are generally

"parameterized". This means that the code allows a developer to change some parameters (e.g., bit-widths of signals and variables) depending on specific needs for the application.

In VHDL, design-for-reuse may be done using the **generic** declaration and/or the **generate** statement. A **generic** declaration allows the programmer to describe constants whose values may be changed in different instances of the block of code. For example, the number of bits used to represent the weights and inputs to a neural network block may be 8 for one particular application and 16 for another which requires more resolution for these inputs. Declaring the bit-width of inputs and weights as a parameter, using the **generic** declaration, allows one to easily reuse the block of code for the neural network block, by simply changing the value assigned to the parameter in different implementations. The section of VHDL code shown in Figure 2.2 illustrates the use of **generic** declaration.

A **generate** statement is handled during synthesis of the VHDL code. It can be

```
entity block22 is
      generic ( data_width: integer = 8;
                wt_width: integer = 8
              );
      port ( clock, resetn: in std_logic;
             x1, x2: in std_logic_vector(data_width-1 downto 0);
             w13, w14: in std_logic_vector(wt_width-1 downto 0);
             w23, w24: in std_logic_vector(wt_width-1 downto 0);
             y3, y4: out std_logic_vector(data_width-1 downto 0)
           );
end block22;
```

Figure 2.2: Using generics in VHDL

used as a "conditional" **generate** statement or, an "unconditional" **generate** statement. A conditional **generate** statement creates logic depending on the truth-value of a condition. An unconditional **generate** statement uses a **loop** to create designs which involve repeating operations. Examples include adder trees, combinational multipliers and so on.

2.3 Pilchard – A Reconfigurable Computing Platform

The FPGA prototyping board used in this research is the Pilchard reconfigurable computing board (shown in Figure 2.3), which was developed at the Chinese University of Hong Kong [4]. It houses a million-gate FPGA, the Xilinx® Virtex™1000E (XCV1000E). Table 2.1 lists the important features of the XCV1000E part, obtained from the manufacturer's website [26]. The Pilchard board has a memory slot interface with the host processor, unlike most commercially available reconfigurable boards like the Firebird™ or Wildcard™ from Annapolis Micro Systems, Inc [5], which have a PCI bus interface to the host processor. The advantage of a DIMM interface over the standard PCI interface is higher bandwidth for communication between the host processor and the RC board. Figure 2.4 shows the block diagram of the Pilchard board [4]. Table 2.2 summarizes the features of the Pilchard board.

Communications between the host processor and the RC board is achieved through a software interface program running on the host Pentium® III processor. The software interface has four API functions to help this communication possible:

(i) void read64(int64, char *) - To read 64 bits from Pilchard

(ii) void write64(int64, char *) - To write 64 bits to Pilchard

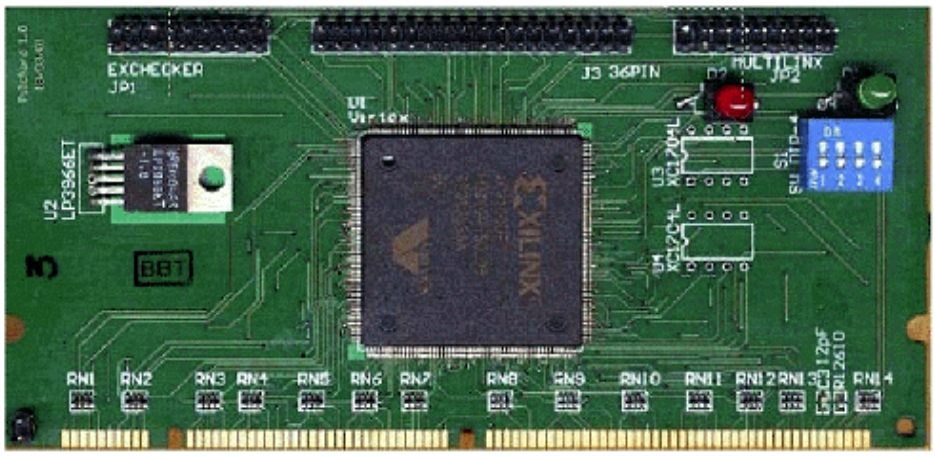


Figure 2.3: Pilchard reconfigurable computing board

Table 2.1: Xilinx® Virtex™ FPGA Device XCV1000E Product Features [26]

Feature	Specification
Package Used in Pilchard	HQ240 (32mm x 32mm)
CLB Array (Row x Col.)	64x96
Logic Cells	27,648
System Gates	1,569,178
Max. Block RAM Bits	393,216
Max. Distributed RAM Bits	393,216
Delay Locked Loops (DLLs)	8
I/O Standards Supported	20
Speed Grades	6,7,8
Available User I/O	158 pins (for package PQ240) max. 660

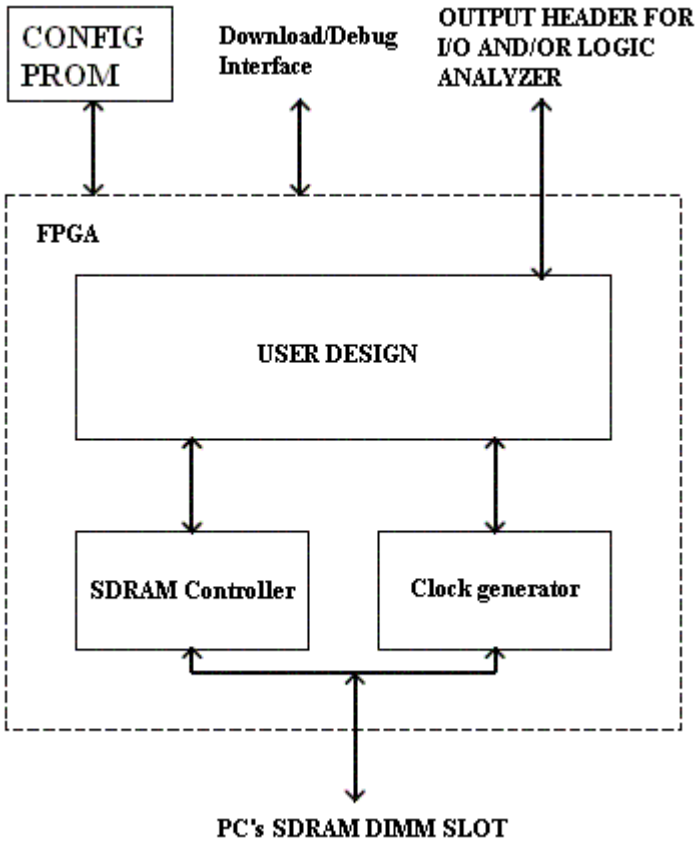


Figure 2.4: Block diagram of Pilchard board [4]

Table 2.2: Features of the Pilchard RC platform [4]

Feature	Specification
Host Interface	• DIMM Interface • 64-bit Data I/O • 12-bit Address Bus
External (Debug) Interface	27-Bits I/O
Configuration Interface	X-Checker, MultiLink and JTAG
Maximum System Clock Rate	133 MHz
Maximum External Clock Rate	240 MHz
FPGA Device	XCVE1000E-HQ240-6
Dimension	133mm x 65mm x 1mm
OS Supported	GNU/LINUX
Configuration Time	16s Using Linux download program

(iii) void read32(int, char *) - To read 32 bits from Pilchard

(iv) void write32(int, char *) - To write 32 bits from Pilchard

"int64" is a special data type that is defined in "iflib.h", as a two-element integer array. "download.c" is a utility that can be used to configure the FPGA with the design bit-stream.

2.4 Approaches for Implementing BbNNs on Hardware

BbNNs may be implemented on hardware (ASICs or FPGAs) using the library-based approach or by using a generic processing-engine for computing the output of each neural network block. In this thesis, the library-based approach has been investigated and implemented. The library-based approach seems to be well-suited for automation and ease-of-use. Parameterized structural VHDL models provide great flexibility as far as resolution of the inputs, outputs and weights are concerned, and also make it possible to easily implement and automate the process of building any BbNN from the basic NN blocks and activation functions available in the library. Chapter 3 discusses the implementation details of the models in the library of NN blocks. This section provides an overview of the approach taken to implement the models, keeping in mind the need for flexibility, ease-of-use and automation of the process of developing any *mxn* BbNN from components available the library.

2.4.1 Library of NN blocks and activation functions

A BbNN consists of an array of basic NN blocks, which may be one of three types, depending on the number of input and output ports in the block. Every NN block computes the sum-of-products of weights and inputs. The result is passed through an

activation function, which could be one of many possible forms, depending on the requirement of the specific application.

Typical activation functions include unipolar and bipolar linear, ramp, saturation and ramp-with-saturation functions (as an approximation of the sigmoid function that is normally used with NNs). These functions are tailor-made to suit easy hardware implementation. As a general rule, it is easier to implement functions with minimal non-linearity, on hardware, than those that involve more non-linearity. Also, multiplication is easier to implement than division. One way to implement functions which involve non-linearities is to use LUTs. In this method, values of the function for different values of the independent variable(s) involved are stored in a ROM and read when required, instead of actually computing the numerical value of the function each time. Another possible method of implementing non-linear activation functions on hardware is to use algorithmic methods to approximate the function. An example of this is method is the commonly used algorithm to generate sinusoids - the CORDIC algorithm. The strategy adopted in the hardware implementation of BbNNs is to use activation functions which do not involve non-linearities, but still provide fairly accurate results in applications such as pattern classification and mobile robot navigation control. This keeps the design simple and easy to implement, and at the same time, provide better performance when compared to implementation with more complex activation functions. Thus, this research has focused on building activation functions which do not involve non-linearities of any kind.

The library of basic NN blocks consists of the three types of basic NN blocks, namely, 3-input 1-output block, 2-input 2-output block and 1-input 3-output block. Each

of these consists of the sum-of-products function and an activation function, which may be chosen from the library of activation functions.

Many variations are possible in each of the components of the library. These are:

(i) **Variation in bit-widths** of inputs, weights and outputs help in adjusting the resolution of the BbNN as required for the application. This is achieved by developing parameterized, synthesizable VHDL models. The VHDL models of the different activation functions are also parameterized.

(ii) **Variation in architectures** for the multiplier and adder units helps in exploring the design-space for different speed, area and power goals. For example, fast adders (e.g., carry-look-ahead, carry-save and so on) could be used to improve the performance of the circuit. Gated-clock designs may be developed to make low-power designs.

(iii) **Synchronous and asynchronous** NN blocks are designed. Synchronous blocks are invariably needed for implementations in which the RC unit (FPGA) has to interact with the host processor. However, asynchronous blocks will be useful for the ASIC implementation of a BbNN for a given application. Asynchronous designs have better performance typically, but are more difficult to design. For BbNN designs with feedback, synchronous designs have to be used. However, in applications where feedback is not required, it would be a good idea to make use of the performance gains provided by asynchronous designs.

(iv) Different **forms of activation functions** (with different possible resolutions) have been designed and implemented in the library of activation functions.

The approach for building the library can thus be summarized in the following steps:

(i) Development of VHDL models for each of the three types of NN blocks and activation functions

(ii) Validation of the above NN models through pre-synthesis simulations

(iii) Synthesis to Xilinx® Virtex™ 1000E FPGA

(iv) Place-and-route (PAR)

(v) Implementation and verification on the FPGA

(vi) Characterization of the components of the library

(vii) Building test BbNNs using components from the library

(viii) Simulation, synthesis and PAR of models developed in (vii)

(ix) Implementation and verification of test BbNNs on the FPGA

2.4.2 Other possible approaches

The library-based approach for implementing BbNNs was chosen for this research keeping in mind the ease of automation and the ease of use. Parameterized and synthesizable structural VHDL models support the flexibility and modularity that BbNNs require. However, this is not the only possible method for implementing BbNNs. One alternate method would be to build a generic processing engine, which could be used to compute the output of each neuron. This would save a lot of real-estate on the FPGA because replication of the NN blocks is avoided. The NN blocks are multiplier-rich and multipliers are very area-greedy. The area-saving however, comes at a cost – the latency (time taken to obtain an output after the inputs have been sensitized) will increase. Latency may be reduced by pipe-lining the design, but this would increase the area of the design. One other factor to be weighed while implementation BbNNs using a generic

processing engine would be the maintenance-of-state and state-machine implementation for the controller. Maintenance-of-state means that memory elements (register/register-files or RAM) are required to store intermediate processing results. A state-machine would be required to orchestrate various processing actions and this would become more complex with increase in the number of blocks and depth of pipelining, because there would be more states to keep track of. Automatic generation of VHDL models in this case would be difficult. Thus, choosing the appropriate implementation is a trade-off among various factors which the user has to decide depending on the need for the application.

2.4.3 ASIC Implementations of BbNNs

FPGA implementations of BbNNs help one zero-in on a specific structure and set of weights for a given application. They fit very well in the GA-scheme of "design-implement-evaluate-modify" cycle illustrated in Figure 1.7, because of the ability to reconfigure FPGAs. ASIC implementations have proved to be superior in performance than FPGA implementations for a given circuit. However, the circuit cannot be modified, once the ASIC has been fabricated. Thus, it would be very practical to use FPGAs for fitness evaluation and prototyping of a given BbNN, and then fabricate an ASIC for actual deployment, once the structure and weights of the BbNN have been optimized to the required level. The ASIC can be custom-designed and implemented as an analog or digital IC, to reduce power consumption, and at the same time operate at higher speeds than FPGA.

CHAPTER 3

IMPLEMENTATION DETAILS

3.1 Detailed design flow used to target Pilchard RC platform

The target platform used for this research is the Pilchard RC platform, described in chapter 2. It houses a Xilinx® Virtex™ 1000E FPGA. The detailed design-flow used for the implementation of BbNNs on this FPGA with the role of the various EDA tools is described in this section.

Figure 3.1 shows the detailed design-flow for implementing BbNNs on the Pilchard RC platform. The BbNN is designed using the library of NN blocks and modeled in VHDL. Before steps to translate this design into a physically realizable circuit are taken, it is simulated using test-vectors provided from a test-bench written in VHDL. A VHDL simulator (in this case, ModelSim® version 5.7d, from Mentor Graphics®) is used to simulate the design. The in-built waveform viewer helps the designer analyze the design for correct functionality. Once the design is verified to be functionally correct, it is synthesized using a design synthesis tool to target the Xilinx® Virtex™ 1000E FPGA. In this research Synopsys® FPGA Compiler-II was used. The tool is supplied with all the design files and delay constraints and it generates a structural netlist in EDIF format. This is a gate-level circuit-description. The synthesis tool also reports timing violations in this implementation of the design. These have to be fixed before going to the next step, namely, PAR. Xilinx® ISE tools are used for PAR. The tool is provided with the EDIF

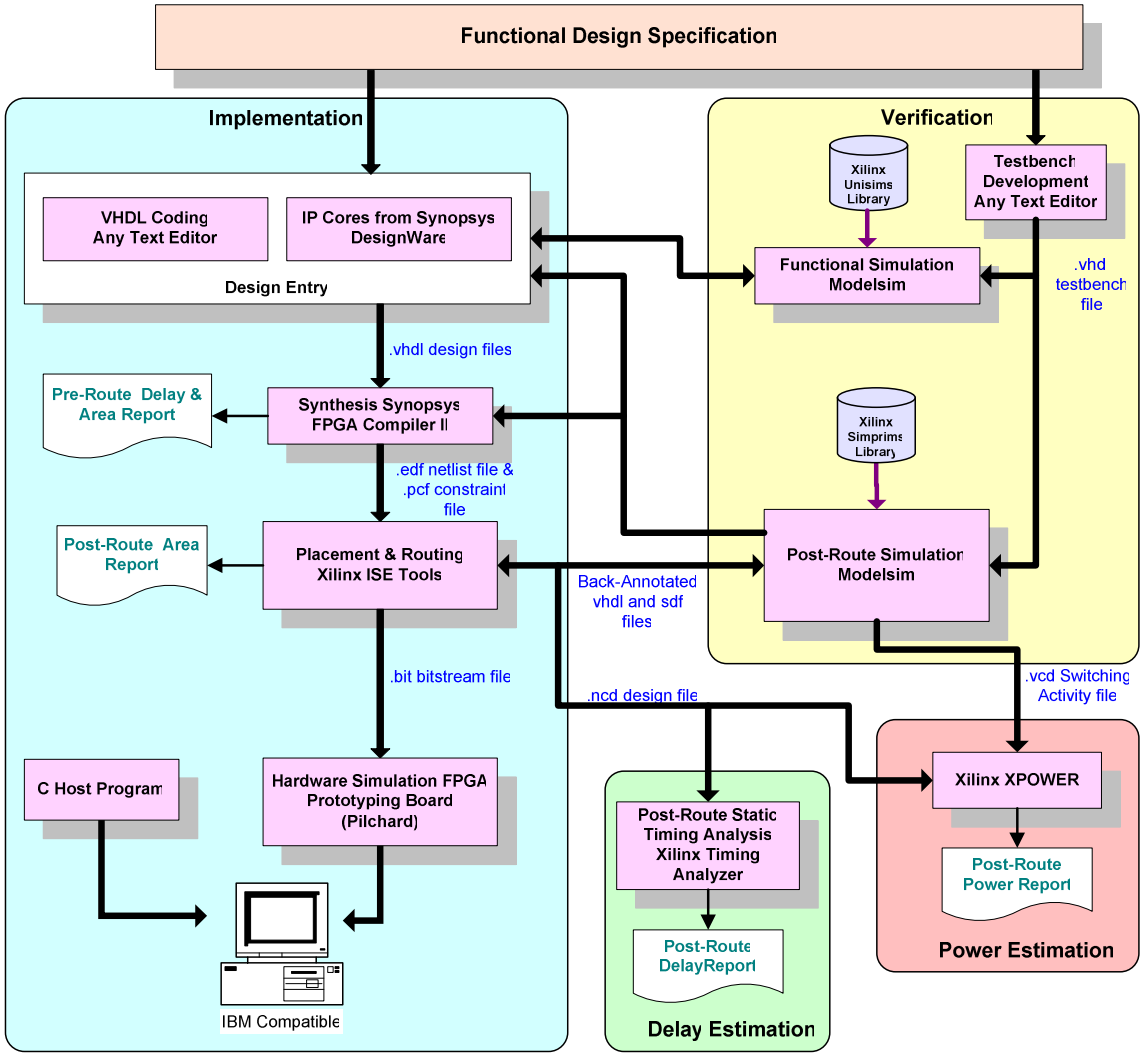


Figure 3.1: Detailed design-flow for implementation on Pilchard RC platform

(Courtesy: Dr. Chandra Tan)

netlist generated from synthesis and a pin constraints file (PCF). The design is first translated into Xilinx's Native Generic Database (NGD) format, and then mapped to primitives inside the particular FPGA being targeted (in this case Virtex™ 1000E). The output from the "map" program is a NCD file which serves as an input to the "par" program. The next step is to place and route (PAR) the design. A logical DRC (Design Rule Check) (device-independent) is performed after building the NGD files to verify the logical design. The output of the PAR program is a native-circuit description (NCD) file which can be used with the "bitgen" program to generate the FPGA configuration file. Post-route static timing analysis is done using the "trce" (trace) program and this would report any timing violations in the design. The designer may have to run a more aggressive PAR in the next iteration, to overcome these violations, or may have to step back and re-synthesize the design, or may even have to change the VHDL design files to meet constraints. On successful bit-file generation using the "bitgen" program, the design is ready to be "downloaded" onto the FPGA residing on the Pilchard RC board. A post-route simulation netlist (VHDL or Verilog) and back-annotation timing file (in the Standard Delay Format or SDF) may be written out from the toolset if the user wishes to perform post-layout simulation. Power estimation of the design may be done using the Xilinx® Xpower™ tool.

In the case of the Pilchard RC board, a host program is written to communicate to the board. Typically, the parameters and data required for the design are written to the Xilinx® block RAM from the host program, and these are used by the design residing inside the FPGA.

3.2 Library of NN blocks

A library of NN blocks consisting of different parameterized, synthesizable VHDL models is first created. VHDL models for different types of activation functions are also built. This library can be used to build any *mxn* BbNN easily.

Every NN block consists of 4 ports to interface to the primary inputs or outputs of the system, or the neighboring blocks. There are 3 types of NN blocks that are considered "legal" – 3-input-1-output block ("block31"), 2-input-2-output block ("block22", and 1-input-3-output block ("block13"). Blocks with all four ports being of the same type are not considered valid NN blocks. The computation performed by each NN block is a weighted sum of the inputs and biases that is passed through an "activation function", before being passed to the output port. The weight for the bias-inputs is always 1. Mathematically, the output of each output port from a NN block can be represented by the equation:

$$y = \varphi(\mathbf{w}.\mathbf{x}^{\mathbf{T}} + b) = \varphi(w_1x_1 + w_2x_2 + \ldots + w_nx_n + b)$$

where y is the output, **w** is the weight vector, **x** is the input vector, b is the bias and φ is the activation function. The form of the equation shown is identical to that of computations performed by neurons in ordinary NNs, but in BbNNs, the inputs, weights, biases and outputs are restricted to be integers or fixed-point numbers to facilitate easy implementation on-chip. Also, the activation functions do not involve non-linearities.

3.2.1 The Sum-of-Products function

The Synopsys® DesignWare™ IP library contains parameterized IP cores (VHDL and Verilog models) which can be incorporated in designs through component instantiation or inference. This saves a lot of development time for the designer who can

spend more time on verification and system-level architectural improvements. As depicted in Figure 3.2, every NN block consists of a sum-of-products of inputs and weights, to which a bias is added. This result is followed by an activation function block devoid of non-linearities, and its output is passed on to the output port.

The DesignWare™ generalized sum-of-products IP was used to implement the sum-of-products and adder function combined. The bias input can be considered to be a regular x-input with its associated weight being 1. Thus, the bias can be augmented with the x-input vector and a "1" can be augmented to the weights-vector and the computation can be implemented with just one DesignWare™ core. The activation function has been implemented in RTL VHDL code from scratch, without using the DesignWare™ library.

Figure 3.3 shows the pin-diagram of the DesignWare™ sum-of-products IP core. The primary inputs to the block are the vectors whose sum-of-products is to be found and a bit ("TC") to indicate the format of the numbers in the vectors, namely two's-complement mode (to accommodate representation of negative numbers) or unsigned numbers. The number of inputs, word-length of input data on the two vectors and the width of the resulting sum are parameters. Figure 3.4 shows how the data has to be organized into the input vectors for the block to do the sum-of-products computation. Table 3.1 shows the pin description of the IP block and table 3.2 explains the parameters involved in it.

During synthesis of a DesignWare™ IP block, the implementation architecture can be chosen from one of the existing varieties, by using compiler directives in the HDL description itself, or in the Synopsys® synthesis script. If a Synopsys® synthesis tool is

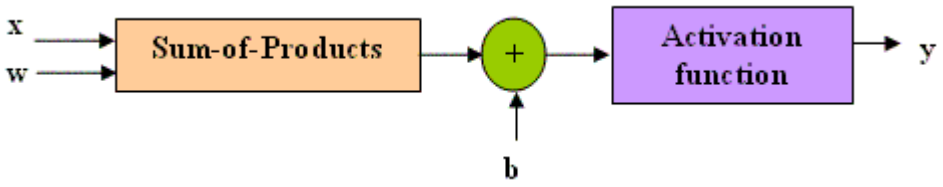


Figure 3.2: Computation of the outputs of a NN block

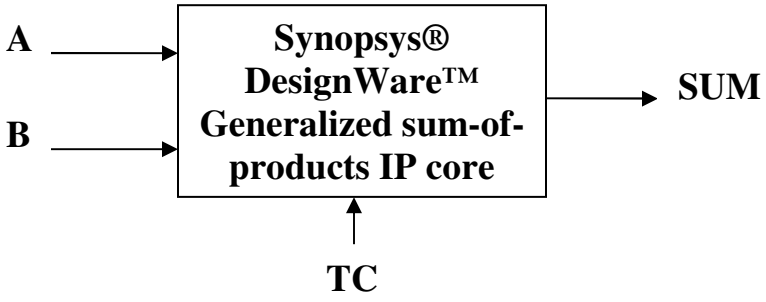


Figure 3.3: DesignWare™ generalized sum-of-products IP block [33]

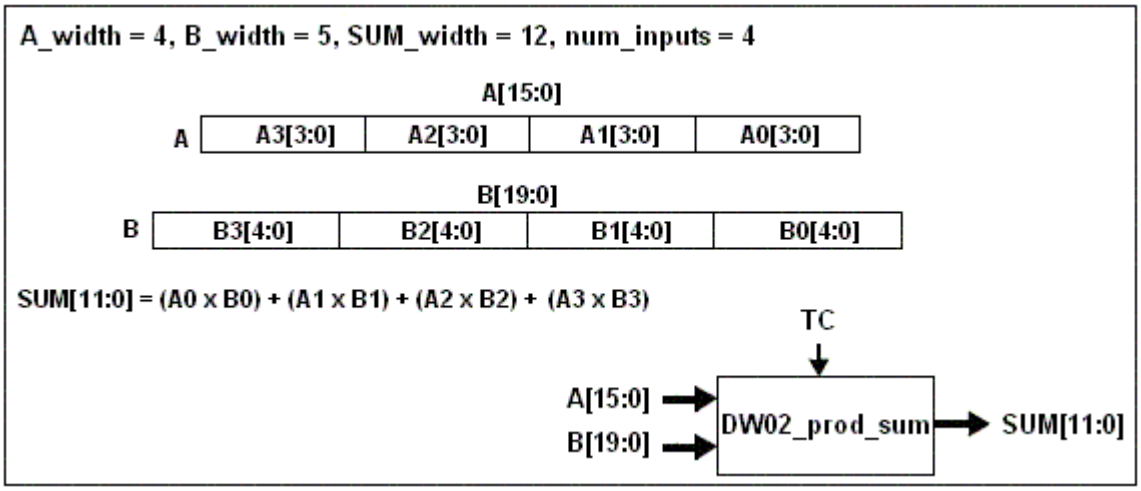


Figure 3.4: Arranging the inputs in vectors for sum-of-products computation [33]

Table 3.1: Pin description of DesignWare™ generalized sum-of-products [33]

Pin Name	Width	Direction	Function
A	*A_width x num_inputs* bit(s)	Input	Concatenated input data
B	*B_width x num_inputs* bit(s)	Input	Concatenated input data
TC	1 bit	Input	Two's complement 0 = unsigned; 1 =signed
SUM	*SUM_width* bit(s)	Output	Sum of products

Table 3.2: Parameters in DesignWare™ generalized sum-of-products [33]

Parameter	Values	Description
A_width	≥ 1	Word length of A
B_width	≥ 1	Word length of B
num_inputs	≥ 1	Number of inputs
SUM_width	≥ 1	Word length of SUM

used, the tool can choose the synthesis implementation by itself, depending on the constraints given to the tool. However, the user may choose to force the synthesis tool to implement a particular architecture. In the case of the sum-of-products IP core, the user can choose from one of the options specified in Table 3.3. In the design of NN blocks and BbNNs, the unpipelined synthesis models have been chosen for their ease of implementation and performance gain. Also, in BbNN circuits with feedback, pipelining the NN blocks will complicate the state-machine used to control the BbNN.

3.2.2 The Activation function

The output of the sum-of-products function is passed on to an "activation function" before it is sent as output of the NN block. Typical activation functions in traditional NNs consist of the sigmoid function or other functions involving non-linearities. However, the activation functions in BbNNs are simple and involve no non-linearities. The activation functions implemented in the library of NN blocks that have been developed in this research, include unipolar and bipolar saturation functions (Figure 3.5) and unipolar and bipolar saturating linear (or ramp) functions (Figure 3.6). The VHDL models for these functions are parameterized and synthesizable. The bit-widths of

Table 3.3: Synthesis implementations of DesignWare™ generalized sum-of-products [33]

Implementation	Function
csa	Carry-save array synthesis model
wall	Booth-recoded Wallace-tree synthesis model
nbw	Either a non-booth (A_width + B_width ≤ 41) or a Booth Wallace-tree (A_width + B_width > 41) synthesis model
mcarch	MC-inside-DW Wallace-tree
csmult	MC-inside-DW flexible Booth Wallace

Figure 3.5: Unipolar and bipolar saturating activation functions

Figure 3.6: Unipolar and bipolar linear saturating activation functions

input and output are parameters and so is the slope of the linear portion of the linear saturation function. The activation function to be used on the output nodes depends on the application. For example, the bipolar saturating linear function is used in the XOR pattern classification problem, and the bipolar saturation function is used in the mobile robot navigation control problem described in chapter 4.

3.2.3 The three types of NN blocks

The three types of NN blocks are block31, block22 and block13. These are formed by choosing the sum-of-product IP core (with a desired synthesis implementation) and an activation function. Each of these is depicted in Figure 3.7. The NN blocks are designed such that all the inputs and outputs are of the same bit-widths. This ensures that a block can get its input directly from another block or its output can be directly fed into the input of a neighboring block.

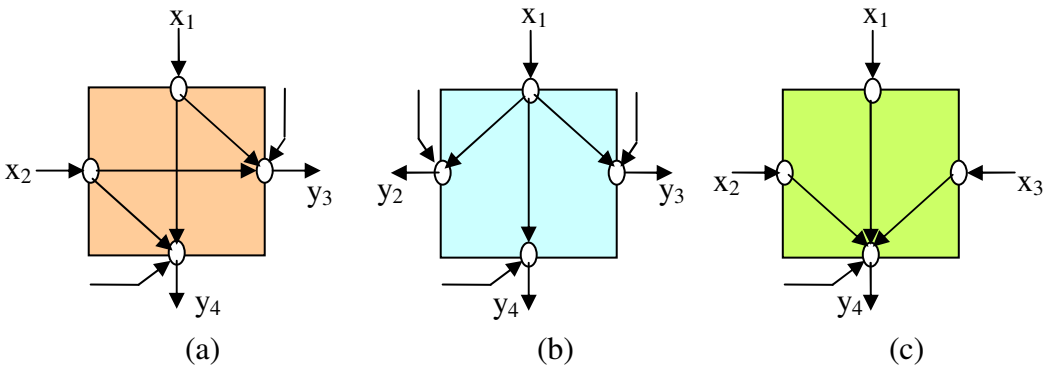


Figure 3.7: Three types of NN blocks (a) Block22 (b) Block13 (c) Block31

3.3 Facilitating Designs with feedback - Blocks with registered outputs

Most BbNNs are designed with lateral feedback from the last layer of NN blocks to the first. An example is shown in Figure 3.8. a, b and c are inputs to the BbNN and Y_1, Y_2 and Y_3 are the outputs. Feedback connections in synchronous digital circuits have to be made through memory elements. Flip-flops or registers are the most commonly inferred memory elements by automatic synthesis tools. A clock signal is required control the functioning of the memory element. This class of designs, where operations are performed on a clock-edge is known as "synchronous design".

In VHDL, memory elements are inferred using clock-sensitive processes. To register output signals, the output port is assigned a value only on a clock-edge (e.g., rising edge). The section of VHDL code in Figure 3.9 illustrates how registers are inferred.

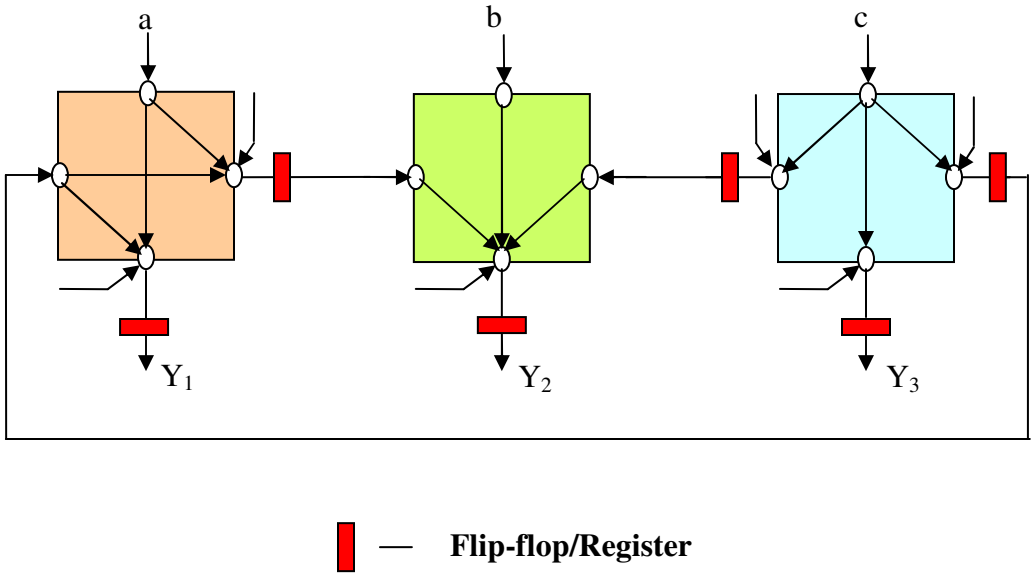


Figure 3.8: NN blocks with registered outputs

```
...

entity regout_block13 is

    generic ( x_width : NATURAL := 8;
              w_width : NATURAL := 8;
              b_width : NATURAL := 8;
              y_width : NATURAL := 64
            );
    port (  clk, rst      : in std_logic;
            x1            : in std_logic_vector(x_width-1 downto 0);
            w12, w13, w14 : in std_logic_vector(w_width-1 downto 0);
            b2, b3, b4    : in std_logic_vector(b_width-1 downto 0);
            tc            : in std_logic;
            y2_y3_y4      : out std_logic_vector(y_width-1 downto 0)
         );

end regout_block13;

architecture regout_structure13 of regout_block13 is

      constant bias_wt : std_logic_vector(w_width-1 downto 0) :=
      "00000001";

      signal y2, y3, y4 : std_logic_vector(x_width-1 downto 0);

      ...

begin

-- calculation of output y2

      w12_bias_wt <= w12 & bias_wt;
      x1_b2 <= x1 & b2;

      U1_sop: DW02_prod_sum_inst
                port map ( inst_A => x1_b2,
                           inst_B => w12_bias_wt,
                           inst_TC => tc,
                           SUM_inst => sop2
                         );
      U1_y2:  rampsat_bi_13
                  port map (  act_in => sop2,
                              act_out => y2
                           );

      ...

-- calculation of outputs y3 and y4
      ...
```

Figure 3.9: Inferring registers in the design using VHDL

```
-- Inferring registers at outputs using clock-sensitive process

      process (clk, rst, y2, y3, y4)
      begin
            if (rst = '1') then
                  y2_y3_y4 <= (others => '0');
            elsif (clk'event and clk = '1') then
                  y2_y3_y4(63 downto 24) <= (others => '0');
                  y2_y3_y4(23 downto 16) <= y2;
                  y2_y3_y4(15 downto 8) <= y3;
                  y2_y3_y4(7 downto 0) <= y4;
            end if;
      end process;
end regout_structure13;
```

Figure 3.9: (Continued)

In this example, `y2`, `y3` and `y4` are signals which are passed on to the output port `y2_y3_y4` only on every rising clock-edge.

3.4 Accuracy and Resolution Considerations

BbNNs are designed to have integer or fixed-point weights and inputs to facilitate easy implementation on hardware. Moreover the number of bits used to represent the inputs and outputs are the same (i.e., same word-length or bit-width for inputs and outputs) to ensure that the blocks can be used to interface directly to other blocks or primary inputs and outputs. The output from the sum-of-products function present in every NN block will have more bits than the inputs and weights. Thus, the activation function has to scale the input it receives from the sum-of-products computation to fall within a certain range of numbers which can be represented with the same number of bits as the inputs. A mathematical analysis to show how the word-length increases is shown below.

Let the number of bits used to represent each input and bias = *x_width*

Let the number of bits used to represent each weight = *w_width*

Let the number of inputs to the block = *n*

Let the number of outputs from the block = *m*

Clearly,

$m + n = 4$ where $m, n \in \{1, 2, 3\}$

Number of bits in each partial-product of the sum-of-products

= *x_width* + *w_width*

Number of partial-product terms (including the bias term)

= *n* + 1

Number of bits on the output from the sum-of-products function

= *n* + 1 + *x_width* + *w_width* - 1

= *n* + *x_width* + *w_width*

There are *m* such outputs from the NN block.

Thus, the activation function has to scale an input of (*n* + *x_width* + *w_width*) bits to *x_width* bits so that it can serve as input to the neighboring block. The designer of the BbNN has to make sure that the bits lost due to truncation of bits at the activation function stage do not adversely affect the performance of the BbNN for that specific application. Typically, one would try to drop bits in the fractional part of the fixed point notation number. A detailed analysis example of these issues is presented in the XOR pattern classification problem in chapter 4.

3.5 Validation of NN blocks

3.5.1 Methodology

The method adopted to validate the functional correctness of the NN blocks is described here. First, a software implementation of each type of NN block is done in the C programming language. The results of computations from this implementation are taken as the "correct results" and then VHDL models are developed. These are simulated using ModelSim® VHDL simulator. Test vectors are supplied to the design from a test-bench and responses collected and compared to the software implementation. Once the two results match, the rest of the hardware design steps are performed, namely synthesis, PAR and implementation on the FPGA.

3.5.2 Software Implementation

The software is written in the C programming language and executed on the Pilchard RC board's host-processor, which is a Pentium® III processor running at 933 MHz. The host computer has 256 MB of RAM. The OS on this is Mandrake Linux 9.1. In the XOR pattern classification problem and mobile robot navigation control problems, the time for computation in software and hardware is compared.

3.5.3 VHDL Design and Simulation

Parameterized, structural VHDL models are developed and simulated using the VHDL simulator, ModelSim®. These are verified for correct functioning for integer and fixed-point, signed and unsigned data. Extensive test-benches have been written to maximize the coverage of possible test-cases. The VHDL models are provided in the appendices at the end of this document.

3.5.4 Hardware Implementation

The NN blocks are implemented on the Pilchard RC platform and tested. A host C program is capable of communicating with the board. Weights, biases and data are loaded into the Xilinx® Block RAM inside the Virtex™ 1000E FPGA and then a "start" signal is issued to the design inside the FPGA. The design performs the computations and writes the results back to the RAM, which can then be read by the host code. The working details of this process of hardware implementation are presented in the case studies in chapter 4.

3.6 Characterization of the library of NN blocks

3.6.1 Area

The area occupied by the NN block is obtained from the Xilinx® PAR tool. It reports the device utilization summary for the designer to review. Table 3.4 shows the number of slices occupied by each of the three types of blocks with registered outputs, for different bit-widths.

3.6.2 Delay

The maximum clock frequency that can be used with the design is also reported by the Xilinx® ISE tools. On the Pilchard RC platform, the clock used for the design is derived from the SDRAM interface clock. The Pilchard RC board allows this clock (133 MHz) to be divided by 2, 2.5, 3, 4, 5, 8 or 16. Thus the design can be implemented only at one of these speeds. However, the design itself may be run at the maximum speed reported by the PAR tool. Table 3.5 summarizes the performance of the different blocks.

Table 3.4: Area (in number of slices of Virtex™ 1000E) for the three types of NN blocks with bipolar linear saturating activation function

	Block13	**Block22**	**Block31**
4 – bit	104	39	69
8 – bit	185	103	176
16 – bit	582	327	532

Table 3.5: Performance comparison of the three types of NN blocks with bipolar linear saturating activation function

	Block13	**Block22**	**Block31**
4 – bit	64.259 MHz	81.606 MHz	43.985 MHz
8 – bit	46.856 MHz	52.751 MHz	34.108 MHz
16 – bit	29.641 MHz	37.063 MHz	30.975 MHz

3.6.3 Power

Power estimation is done using Xilinx® XPower™. These estimates, shown in table 3.6, are done by setting a 0.25 switching activity on each node. This is useful for comparison of the various designs, but to make better estimates of power consumption, realistic test-vectors have to be used to generate a switching activity (VCD) file from the VHDL simulator, to serve as an input to the power-estimation tool.

3.7 ASIC Implementation of one data path of a 2-input-2-output NN block

ASIC implementations yield better performance than FPGA implementations as the circuit designed is specific to the application. Thus, there is a performance gain at the cost of reconfigurability. A single data path of a 2-input-2-output ASIC was designed, simulated and physically laid out for fabrication in the AMI-C5F/N (0.6u technology). The design is a fully combinational circuit that performs the operation:

$$f(w_{13}\,x_1 + w_{23}\,x_2 + b)$$

Table 3.6: Power estimates of the three types of NN blocks with bipolar linear saturating activation function

	Block13	**Block22**	**Block31**
4 – bit	976 mW	973 mW	976 mW
8 – bit	979 mW	975 mW	979 mW
16 – bit	994 mW	981 mW	984 mW

where x_1 and x_2 are 4-bit inputs, w_{13} and w_{23} are 4-bit weights, b is the 4-bit bias input and *f* is the activation function, in this case a constant multiplier with the constant being a parameter input for testing purposes. The multipliers were generated using Cadence® Silicon Ensemble® automatic layout-generation tool and the adders were custom-designed data paths. The area occupied by the design is about 198000 square microns. Simulation was done using Cadence® Spectre® transistor-level simulator.

Figure 3.10 shows the architecture implemented in for this ASIC. Figure 3.11 shows the ASIC layout including the pads for the chip.

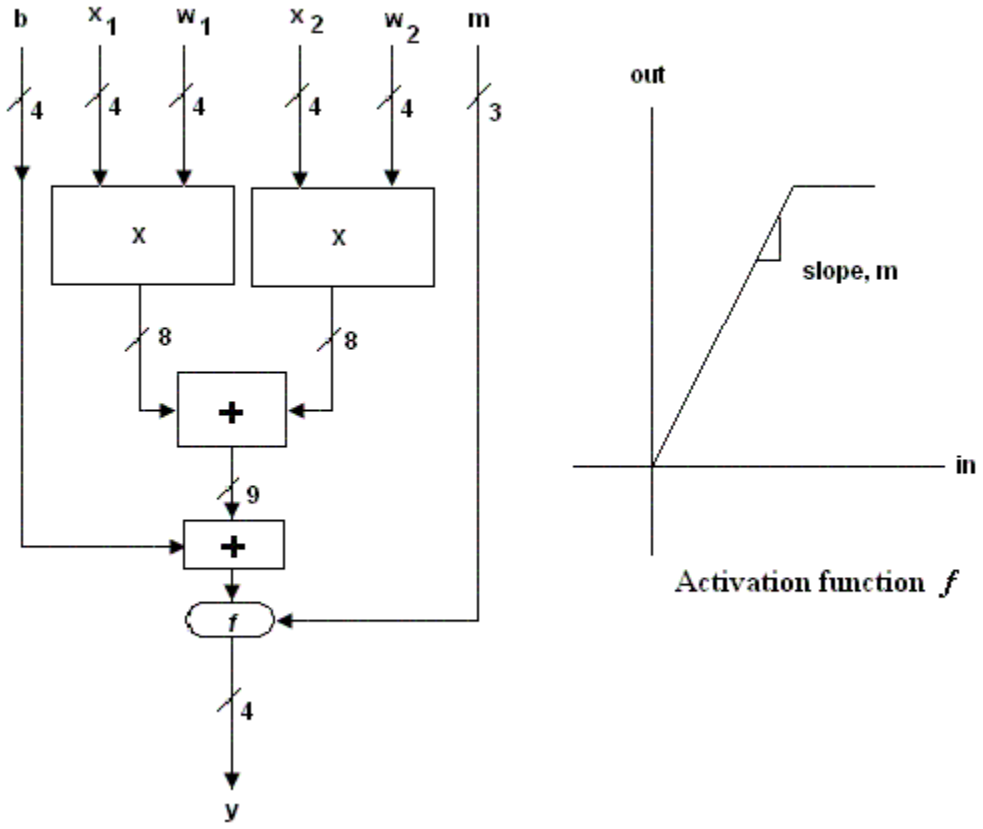


Figure 3.10: Architecture of ASIC implementation in AMI 0.6u process

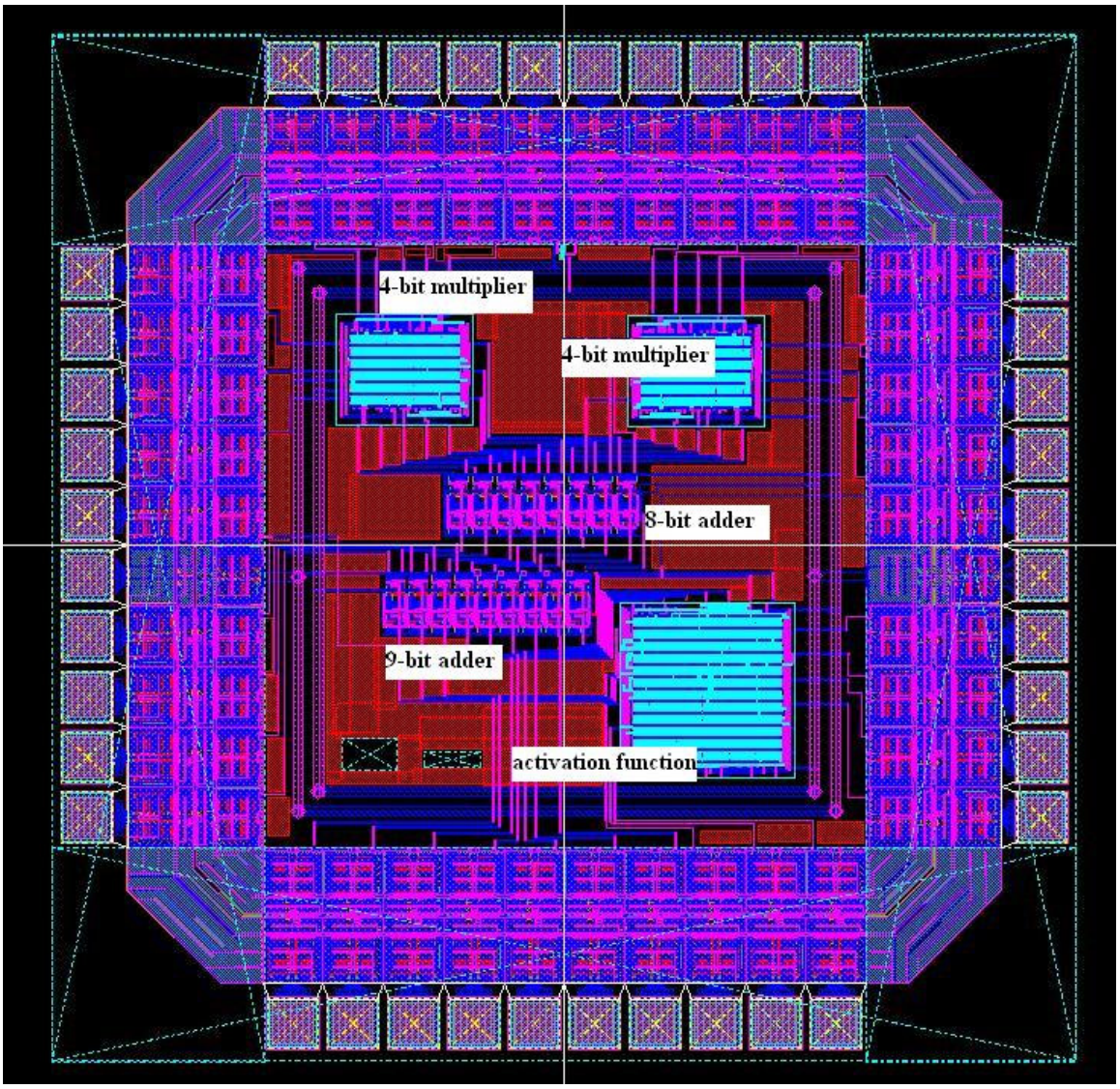


Figure 3.11: ASIC implementation of $f(w_{13}x_1 + w_{23}x_2 + b)$ in AMI 0.6u process

CHAPTER 4

CASE STUDIES AND RESULTS

4.1 Building BbNNs using the library of NN blocks

The library of NN blocks that has been developed and characterized can be used to build BbNNs for various applications. Software simulations have shown that BbNNs are suitable for pattern classification and robotic control problems. In this chapter, two specific applications of BbNNs, namely, the XOR pattern classification problem and the mobile robot navigation control problem, and their implementations on the Pilchard RC platform are described in detail. The issues related to the FPGA implementations and the results obtained from the hardware and software implementations are analyzed. Both the BbNNs for the above-mentioned applications involve feedback and hence NN blocks with registered outputs are used.

Building a BbNN from the library of blocks involves describing a structural VHDL model to interconnect the various blocks through port-mapping and integrating the design with a RAM which can hold the weights, biases and input data. Also, a state machine will have to be designed to orchestrate data-flow between various blocks and maintaining the states during each clock cycle so that computations are performed correctly. Additional design files required for implementation of the design on the Pilchard RC platform include a VHDL wrapper with descriptions for the pads of the Virtex™ 1000E FPGA ("pilchard.vhd"). A host C program ("iftest.c") is also required to interface to the Pilchard RC platform.

4.2 The XOR Pattern Classification Problem

4.2.1 Background

The XOR function takes in two inputs, x_1 and x_2. The output of this boolean function is 0, when both inputs are identical and 1, when they are not the same. The truth-table of the XOR function is shown in Table 4.1 below. The XOR pattern classification problem aims at classifying input data other than (0,0), (0,1), (1,0) and (1,1) into one of the output classes – output 0 and output 1. The input data are in the range from 0 to 1. For example, the data (0.95, 0.89) may be classified to an output 0 class while (0.05, 0.98) may be classified as an output 1 class. The XOR pattern classification problem is a linearly inseparable problem. This means that a single line (or "decision boundary") in the x_1-x_2 plane cannot separate the regions representing class 0 and class 1. Figure 4.1 shows the linearly inseparable output classes 0 and 1.

4.2.2 BbNN for XOR Pattern Classification

The BbNN used for XOR pattern classification is shown in Figure 4.2 [2]. It consists of four 2-input-2-output blocks (labeled A, B, C and D). The structure and weights shown in the figure are obtained after optimization using genetic algorithms. The inputs x_1 and x_2 are fractional numbers (represented as fixed-point numbers) between 0

Table 4.1: XOR truth-table

X_1	X_2	Output
0	0	0
0	1	1
1	0	1
1	1	0

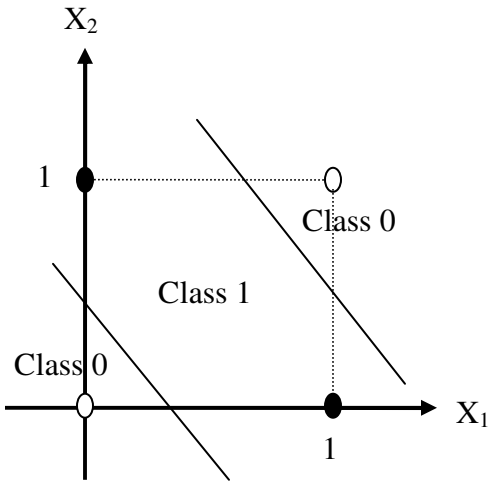


Figure 4.1: Decision boundaries for the XOR pattern classification problem

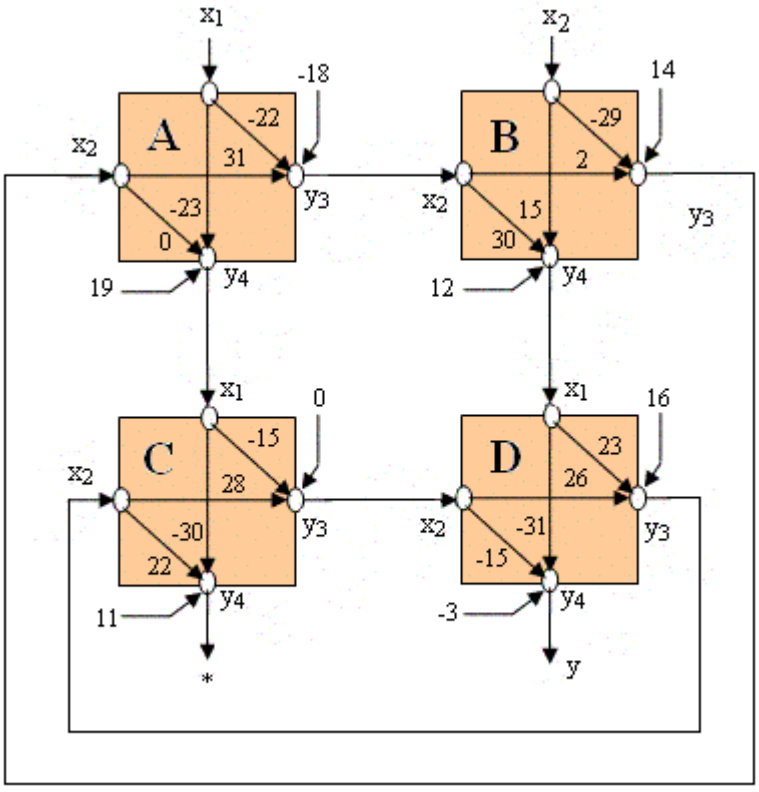


Figure 4.2: BbNN used for XOR pattern classification [2]

and 1, and the output y, is positive for class 0 (output = 0) and negative for class 1 (output = 1) data. The activation function used, is a bipolar linear saturating function with a slope of 1/20 and saturation occurring if the output goes above 1 (saturated to 1) or below -1 (saturated to -1) (Figure 4.3).

4.2.3 Implementation on Pilchard RC platform

4.2.3.1 Data format

The weights and biases in this problem are integers and fall within the range -32 to 31. In hardware implementation at least 6 bits are required to represent these numbers in the two's complement form. The input data x_i's lie in the range 0 to 1. These are fractional numbers and have to be represented using some number of bits depending on the resolution desired for the input data. Using n bits to represent the magnitude of a fractional number gives a resolution of 2^{-n}. In this problem, 8 bits are used for representing numbers between 0 and 1. Thus, the interval between 0 and 1 is divided uniformly to represent 256 numbers with a resolution of 0.00390625.

The DesignWare™ sum-of-products IP core requires that all inputs be in the same

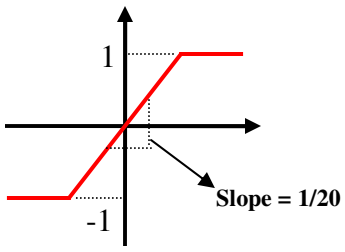


Figure 4.3: Activation function used in XOR pattern classification

format – signed or unsigned. Also, there is a need to represent all inputs, weights, biases and outputs in the same data format – in terms of signed representation and number of bits in the fractional part. The total number of bits used to represent the input need not necessarily be the same as those of the weights and biases. Thus, the integers have to be shifted left by *m* bits, if *m* bits are used to represent the magnitude of the fractional inputs. All the numbers are represented in two's-complement form.

In this implementation of BbNN for XOR pattern classification 9 bits are used to represent the fractional input data x_1 and x_2. The leading bit is a signed bit, and is always '0' because x_1 and x_2 are always positive. Thus, only the 8 bits representing the magnitude needs to be stored in the RAM, and the leading sign-bit can be appended before sum-of-products computation, because it is known to be '0' always. Similarly, the integer weights and biases should be represented in the two's-complement *p.q* format where *p* is the number of bits used to represent the integer part (8 in this case) and *q* is the number of bits for the fractional part (should be 8 in this case). The extension to include the fractional part of the integer number (always "00000000") is done just before it is presented as input to the sum-of-products function. The weights and biases stored in the RAM need to be only 8 bits wide and this saves considerable space in the RAM. Figure 4.4 shows the format of data inside the RAM and that when presented as inputs to the block.

4.2.3.2 Data organization in Xilinx® Block RAM

The weights, biases and inputs are stored in the Xilinx® Block RAM. A 256 (rows) x 64 (bits per row) dual-port RAM, generated using Xilinx® Core Generator™,

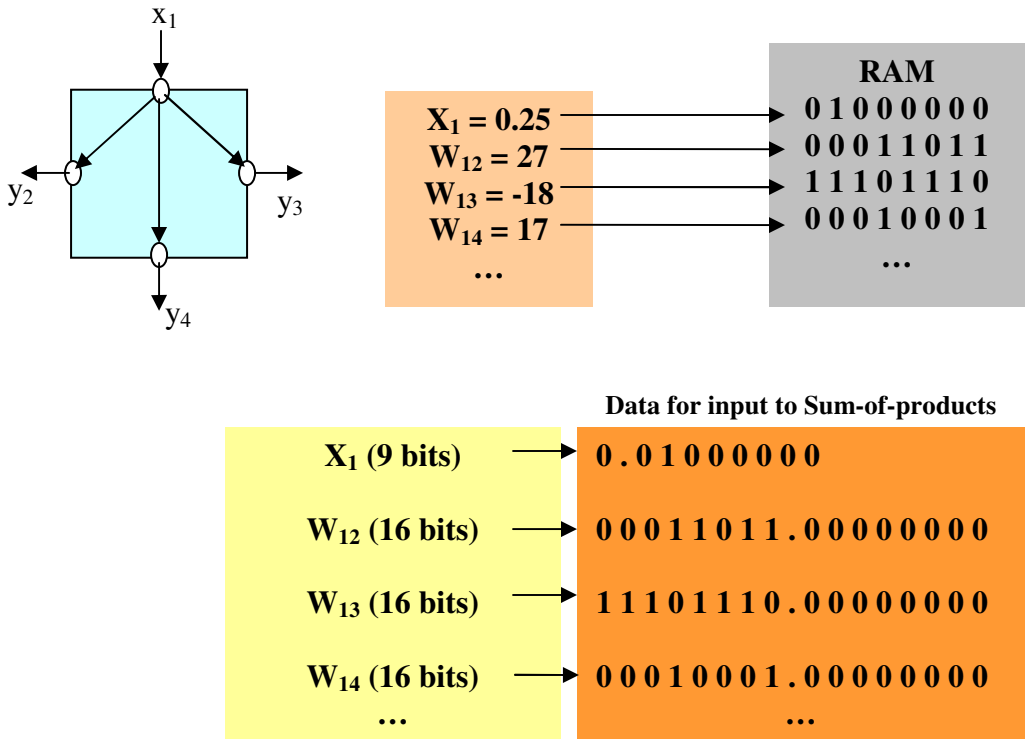


Figure 4.4: Data format of inputs, weights and biases in block13

is used in this design. Port A is used by the host processor to access the RAM on the FPGA. Port B is used by the design residing in the FPGA to access the RAM. For the XOR BbNN, 16 weights and 8 biases are required. Each needs 8 bits to be represented in the RAM. Thus 192 bits or 3 rows of the RAM are needed to store these inputs. Each pair of data is stored in one row, for convenience. The 8 biases are concatenated and stored in the first row (address 0) of the RAM. The second row (address location 1) contains the weights for blocks A and B. The third row (address location 2) stores the weights for blocks C and D. This is followed by 64 rows of data input pairs (x_1, x_2). This is from address location 3 to 66. Address locations 67 to 130 will store the outputs. The arrangement of weights, biases and inputs in the RAM, is shown in Figure 4.5.

ADDR	CONTENTS
0	Biases for A, B, C and D blocks
1	Weights for blocks A and B
2	Weights for blocks C and D
3	Input dataset 1 (x1, x2)
4	Input dataset 2 (x1, x2)
…	…
66	Input dataset 64 (x1, x2)
67	Output data 1
68	Output data 2
…	…
130	Output data 64

Figure 4.5: Data organization in 256 x 64 dual port RAM for XOR BbNN

4.2.3.3 Read and Write operation timing diagrams for Xilinx® Block RAM

To use the dual port RAM, it is important to understand its read and write operations clearly so that the state machine to control data flow to and from the RAM can be described correctly. The Xilinx® dual port RAM is generated using the Xilinx® Core Generator™. The RAM generated is implemented on-chip by mapping the RAM design to physically separate RAM blocks and it does not consume the CLBs inside the FPGA. Figure 4.6 shows the write and read operation timing diagrams.

4.2.3.4 State Machine implementation

The BbNN used for XOR pattern classification (Figure 4.2) involves a circuit which does not have a complete ordering when described as an acyclic graph. This is

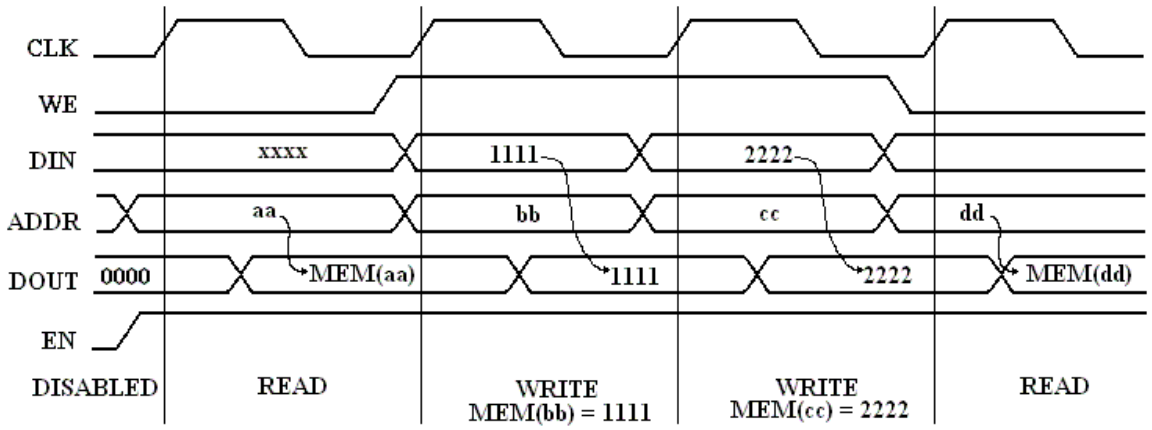


Figure 4.6: Read and write operations on Xilinx® Dual port RAM [34]

because the output of block A depends on the output of block B, and vice-versa. This is the same with blocks C and D. Thus, unless we assume some initial state for the outputs (i.e., on reset), a "valid" solution can never be obtained. In this design, to begin with (on reset), the outputs of all blocks are assumed to be zero. This gives block A, a starting point to compute its outputs and in subsequent cycles, the outputs of other blocks can be considered "valid". In this particular BbNN, output 'y' from block D is the final answer (which represents the class where the input belongs) and is taken to be "valid" after three clock cycles. Each block takes one clock cycle to compute the outputs and clearly, there are 3 cycles of delay from input x_1 to output y. It can be shown that in a BbNN with *m* NN blocks, the final output can be obtained in a maximum of *m*+1 clock cycles, if each NN block is implemented as a purely combinational circuit with the outputs alone being registered on every clock cycle.

A state machine is implemented to initially load the weights and biases for the BbNN and then successively read input data sets, i.e., (x_1, x_2), one at a time, from the

RAM, compute the result and write the output back to the RAM. The important points to be remembered when writing the state machine are:

(i) The outputs of the blocks have to be initialized to zero and kept at that value till the end of the first clock cycle. The outputs of the blocks at the end of the first clock cycle are then used as valid inputs to the neighboring blocks in subsequent cycles.

(ii) The next input data set is not read until the computation for the previous data set is complete.

(iii) The outputs are reset to zero at the end of the computation for every data set.

Table 4.2 describes the states in detail. The state diagram of the state machine implemented for the XOR BbNN is shown in Figure 4.7. It involves 12 states.

The state machine waits in state 0 till it receives a "start" signal issued by the host program. This indicates that the data required for computation are ready in the RAM and that the BbNN can start its computations. Once this signal is sensed by the state machine, it starts loading the biases and weights of the BbNN in successive cycles. *ip_addr_ctr* and *op_addr_ctr* are counters used to keep track of the address location to read the input data set and write the output. The states 6 – 12 form the loop for repeating the process of reading input data, computing output and writing the output to the RAM, for 64 data sets.

4.2.3.5 Host C Program

The host C program communicates with the Pilchard RC board. "read64" and "write64" are the two APIs which facilitate reading and writing data from and to the Xilinx® dual port RAM, through its port A. For this application, the program loads the biases, weights and inputs into the RAM and issues a "start" signal to the BbNN

Table 4.2: Details of state machine for XOR BbNN

STATE	OPERATIONS
0	If (start = 1), then initialize all outputs of the 4 blocks to 0, initialize *ip_addr_ctr* to 3 and *op_addr_ctr* to 67, STATE = 1; else, wait for start to become 1
1	Issue address for reading row 1 of RAM (bias values for all 4 blocks); STATE = 2
2	Issue address for reading row 2 of RAM (weights for A and B blocks); STATE = 3
3	Store the bias values into a temporary buffer; Issue address for reading row 3 of RAM (weights for C and D blocks); STATE = 4
4	Store the weight values of A and B blocks into a temporary buffer; Issue address for reading row 4 of RAM (input data set 1); STATE = 5
5	Store the weight values of C and D blocks into a temporary buffer; STATE = 6
6	If (*ip_addr_ctr* = 67) set STATE = 0; else if (*ip_addr_ctr* = 3) store the input data set 1 value into a buffer; STATE = 8; else STATE = 7
7	Store the input data set value into a temporary buffer; STATE = 8
8	STATE = 9; Allow feedback values to serve as "valid" inputs to the neighboring blocks (i.e., no more forcing outputs to 0)
9	STATE = 10 (this is a dummy state to wait for completion of computation)
10	STATE = 11 (this is a dummy state to wait for completion of computation)
11	Write the output data to RAM; Increment *ip_addr_ctr* by 1; Reset all block outputs to 0 for computation using next data set; STATE = 12
12	Issue address for next input data set; Increment *op_addr_ctr* by 1; STATE = 6

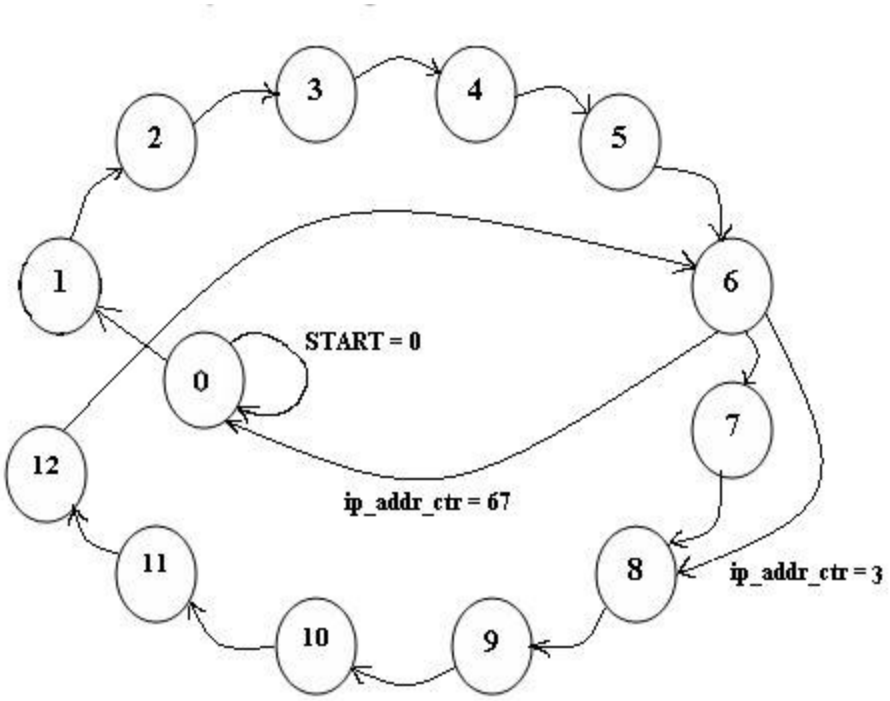


Figure 4.7: State diagram for XOR BbNN state machine

controller (state machine). The "start" signal is nothing but a write operation (data being "00000000") to the 256th row of the 256 x 64 RAM used in the design. The BbNN controller checks for this signal on every rising clock-edge. Once the required weights, biases and inputs are loaded into the RAM, the host program enters an idle state (state 0) in which it waits for computation to be completed by the BbNN core, so that it can read the results from the RAM. The C program polls for a write operation to address location 130 (where the last output is written) and then begins reading the contents of the RAM. States 6 to 12 form the loop in which the computation of outputs and storing results in the RAM takes place. The state machine ensures that the FPGA need not be configured each time classification operation on a new data set has to be performed.

4.2.4 Results

4.2.4.1 XOR Pattern Classification Results

The results obtained from hardware and software implementations of the XOR BbNN are found to be consistent. Figure 4.8 shows the decision boundaries of the XOR pattern classification.

4.2.4.2 Area

The XOR BbNN occupies about 5 % of the Virtex™ 1000E FPGA. Figure 4.9 shows the layout of the design on the FPGA.

4.2.4.3 Speed

The circuit can operate at a maximum speed of 26.824 MHz. However, the design was tested on the Pilchard board at a speed of 16.63 MHz, because only clocks of certain frequencies can be easily generated on-board in the Pilchard RC system. The clock used for the design is derived from the 133 MHz SDRAM interface clock. For generating other frequencies, this clock has to be divided inside the FPGA. The factors that can be used for this division should be one of 2, 2.5, 3, 4, 5, 8 and 16.

4.2.4.4 Speed-up achieved with hardware

The time taken by the hardware and software implementations are measured using the C function, "`gettimeofday()`". For the software implementation, the data is presented in a file, and the time is measured only for the computation of the outputs for the 64 input data sets. For hardware execution-time measurement, the time is measured from the

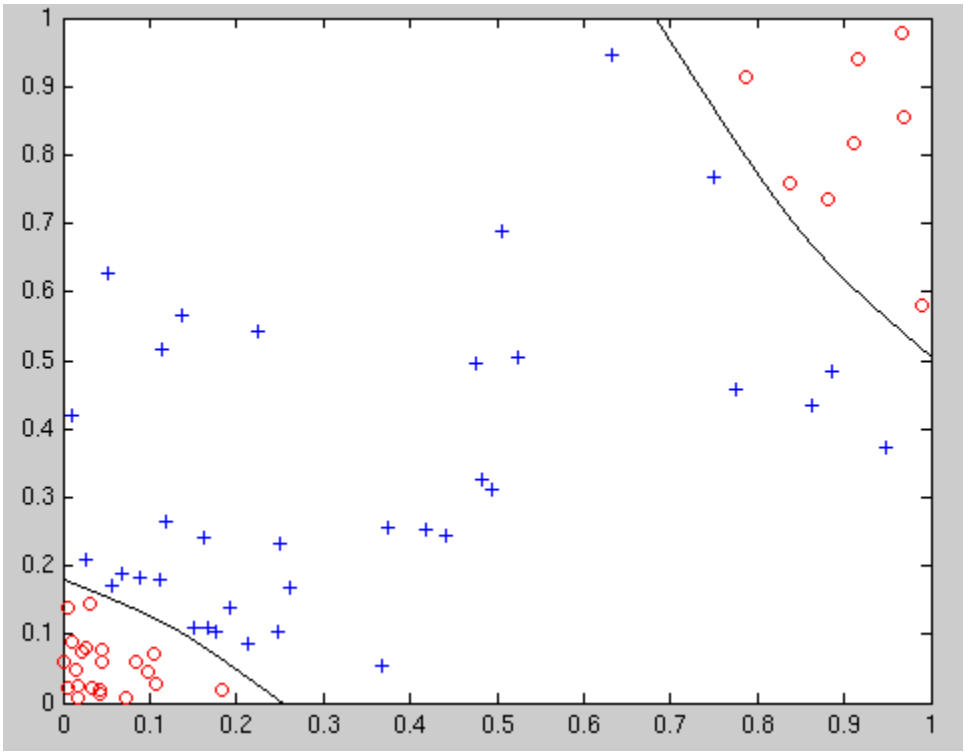


Figure 4.8: XOR pattern classification by BbNN

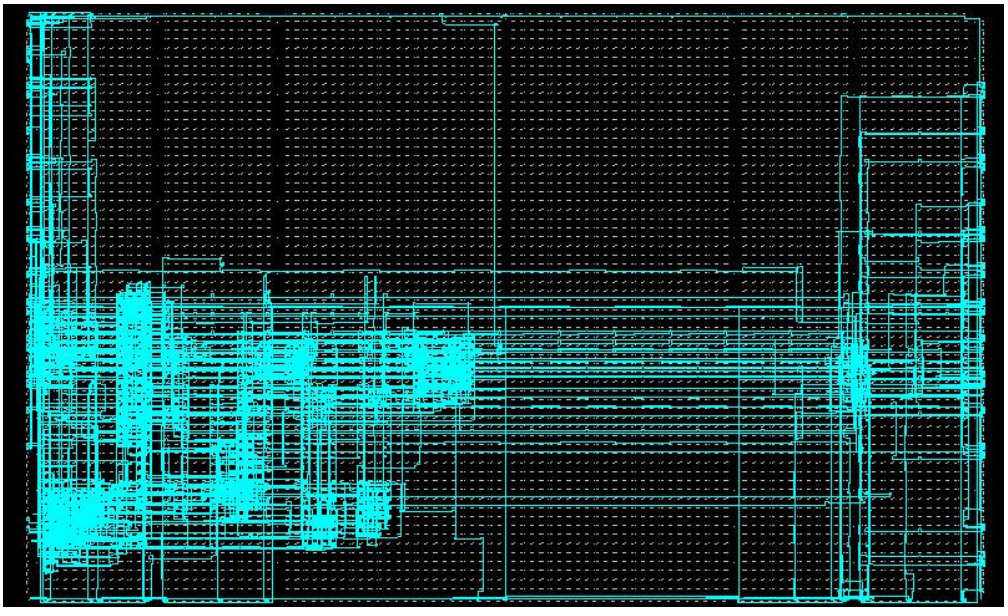


Figure 4.9: Layout of XOR BbNN

Table 4.3: Speed-up for XOR BbNN

Run-time in hardware(μs) t_{hw}	Run-time in software(μs) t_{sw}	Speed-up = t_{hw}/t_{sw}
23	52362	2277

instant when it issues the "start" signal to the BbNN controller, to when the output data is started to be read from the block RAM. Table 4.3 shows the comparison between hardware and software execution times.

4.3 The Mobile Robot Navigation Control Problem

4.3.1 Background

Robots as arm manipulators are mostly programmed in a very explicit way to execute well-defined tasks. Mobile robots, however, may work under unknown and dynamic environments in nature, avoiding obstacles. The robot has no prior knowledge of the environment, navigation paths, shapes and positions of obstacles. The robot also has to acquire knowledge about the configuration of its sensors, which can be blinded or disabled from time to time. The robot behaves based on its current sensory input and its previous interactions with the environment. Its controller can be considered as a kind of adaptive reactive system, which can be described by a dynamic Boolean function easily implemented in hardware using EHW. The robot learns how to navigate in the environment by on-line evolution and builds an explicit model of the environment as a collection of events. The mobile robot considered in this case study, consists of 5 sensors to detect obstacles, and 2 wheels which are controlled by a navigation controller. The

controller decides the rotation angle of the wheels depending on the inputs from the sensor.

4.3.2 BbNN for Navigation Control

Figure 4.10 shows the mobile robot with BbNN controller. $S_1 – S_5$ are five sensors placed on the periphery of the base and outputs from the sensor provide inputs to the 1x5 BbNN. Each sensor outputs a '1' or '0' to indicate the presence or absence of an obstacle. The bipolar saturation activation function (Figure 4.11) is used in this BbNN. Thus, the output of each block is either '1' or '-1'. The outputs of blocks with inputs from S2 – S5 are provided to a decoder which translates the encoded BbNN output to inputs for the motor used to control the angle of rotation of the 2 wheels. Figure 4.12 shows the optimized weights, biases and structure used to implement the BbNN controller. It consists of all the three types of NN blocks – three blocks of 2-input-2-output type, and one each of 1-input-3output and 3-input-1-output types.

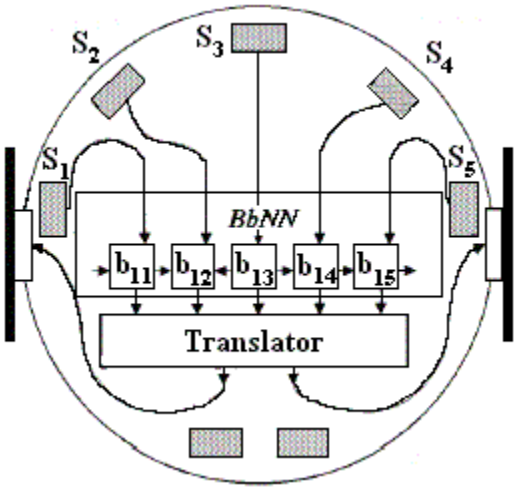


Figure 4.10: Robot with BbNN navigation controller

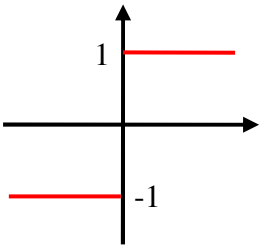


Figure 4.11: Saturation activation function for BbNN robot controller

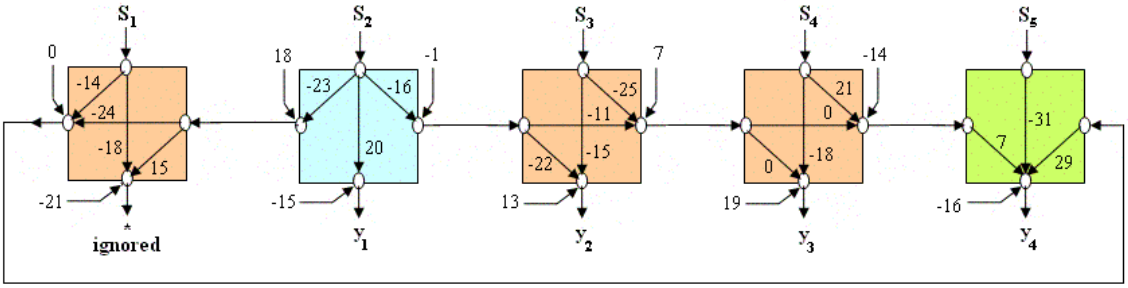


Figure 4.12: Final structure and weights of BbNN controller for robot navigation

4.3.3 Implementation on Pilchard

4.3.3.1 Data format

The weights and biases in this problem are integers and fall within the range -32 to 31. At least 6 bits are required to represent these numbers in the two's complement form. The input data S_i's are either 0 or 1. One bit is enough to represent these.

Just as in the XOR pattern classification problem, there is a need to represent all inputs, weights, biases and outputs in the same data format – in terms of signed representation. There is no fractional number involved in this problem. All the numbers are represented in two's-complement form. Thus, leading zeros have to be appended to the input data to ensure that they are interpreted as positive numbers. 8 bits are used to represent weights, biases and inputs in this implementation.

4.3.3.2 Data organization in Xilinx® Block RAM

Data organization in the RAM, for a set of 64 input data sets ($S_1 – S_5$) is shown in Figure 4.13.

4.3.3.3 State Machine implementation

The state machine implementation for the robotic BbNN controller can be represented as an acyclic graph with complete ordering. The block B does not depend on inputs from any block and its outputs are all valid after one clock cycle. All four outputs are valid after 3 clock cycles. Figure 4.14 and Table 4.4 explain the states in the BbNN controller.

ADDR	CONTENTS
0	Biases for A, B and C blocks
1	Biases for D and E blocks
2	Weights for blocks A and B
3	Weights for blocks C and D
4	Weights for block E
5	Input dataset 1 (S_1, S_2, S_3, S_4, S_5)
…	…
68	Input dataset 64 (S_1, S_2, S_3, S_4, S_5)
69	Output data 1 (y_1, y_2, y_3, y_4)
…	…
132	Output data 64 (y_1, y_2, y_3, y_4)

Figure 4.13: Data organization in 256 x 64 Xilinx® dual port RAM for BbNN robot controller

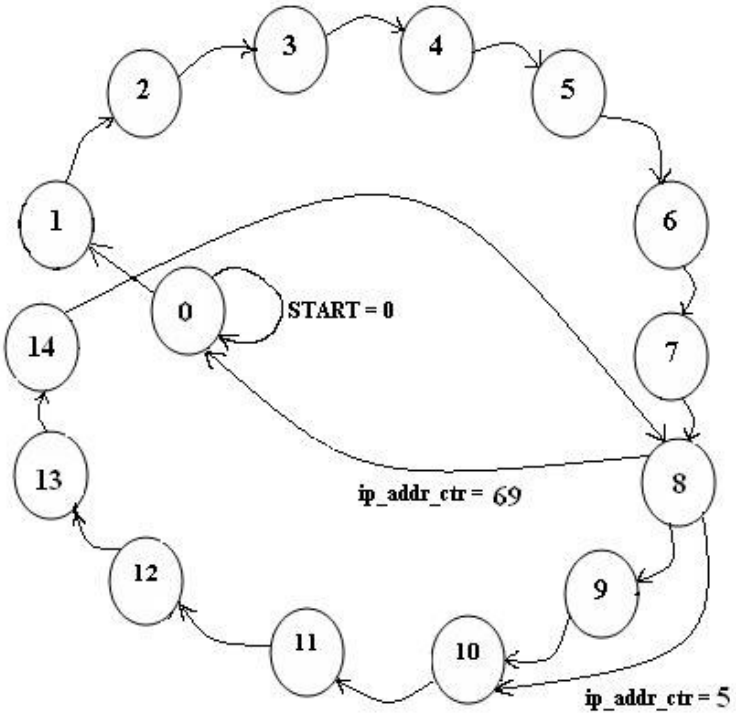


Figure 4.14: State diagram for BbNN robot controller state machine

Table 4.4: Details of state machine for BbNN robot controller

STATE	OPERATIONS
0	If (start = 1), then initialize all outputs of the 4 blocks to 0, initialize *ip_addr_ctr* to 5 and *op_addr_ctr* to 69, STATE = 1; else, wait for start to become 1
1	Issue address for reading row 1 of RAM (bias values for blocks A, B and C); STATE = 2
2	Issue address for reading row 2 of RAM (bias values for blocks D and E); STATE = 3
3	Store the bias values of A, B and C blocks into a temporary buffer; Issue address for reading row 3 of RAM (weights for A and B blocks); STATE = 4
4	Store the bias values of D and E blocks into a temporary buffer; Issue address for reading row 4 of RAM (weights for C and D blocks); STATE = 5
5	Store the weight values of A and B blocks into a temporary buffer; Issue address for reading row 5 of RAM (weights for E block); STATE = 6
6	Store the weight values of C and D blocks into a temporary buffer; Issue address for reading row 6 of RAM (input data set 1); STATE = 7
7	Store the weights of block E into a temporary buffer; STATE = 8
8	If (*ip_addr_ctr* = 69) set STATE = 0; else if (*ip_addr_ctr* = 5) store the input data set 1 value into a buffer; STATE = 10; else STATE = 9
9	Store the input data set value into a temporary buffer; STATE = 10
10	STATE = 11 (this is a dummy state)
11	Allow feedback values to serve as “valid” inputs to the neighboring blocks (i.e., no more forcing outputs to 0); STATE = 12
12	STATE = 13 (this is a dummy state to wait for completion of computation)
13	Set Write the output data to RAM; Increment *ip_addr_ctr* by 1; Reset all block outputs to 0 for computation using next data set; STATE = 14
14	Issue address for next input data set; Increment *op_addr_ctr* by 1; STATE = 8

4.3.4 Results

4.3.4.1 Mobile Robot Controller Results

The results obtained from hardware and software implementations of the BbNN controller are found to be consistent. The 4-bit output is translated into motor rotation angles by the translator.

4.3.4.2 Area

The BbNN robot controller occupies about 7 % of the Virtex™ 1000E FPGA. Figure 4.15 shows the layout of the design on the FPGA.

4.3.4.3 Speed

The circuit can operate at a maximum speed of 22.936 MHz. The design was tested on the Pilchard board at a speed of 16.63 MHz.

4.3.4.4 Speed-up achieved with hardware

The time taken by the hardware and software implementations are measured using the C function, "`gettimeofday()`". For the software implementation, the data is presented in a file, and the time is measured only for the computation of the outputs for the 64 input data sets. For hardware execution-time measurement, the time is measured from the instant when it issues the "start" signal to the BbNN controller, to when the output data is started to be read from the block RAM. Table 4.5 shows the comparison between hardware and software execution times.

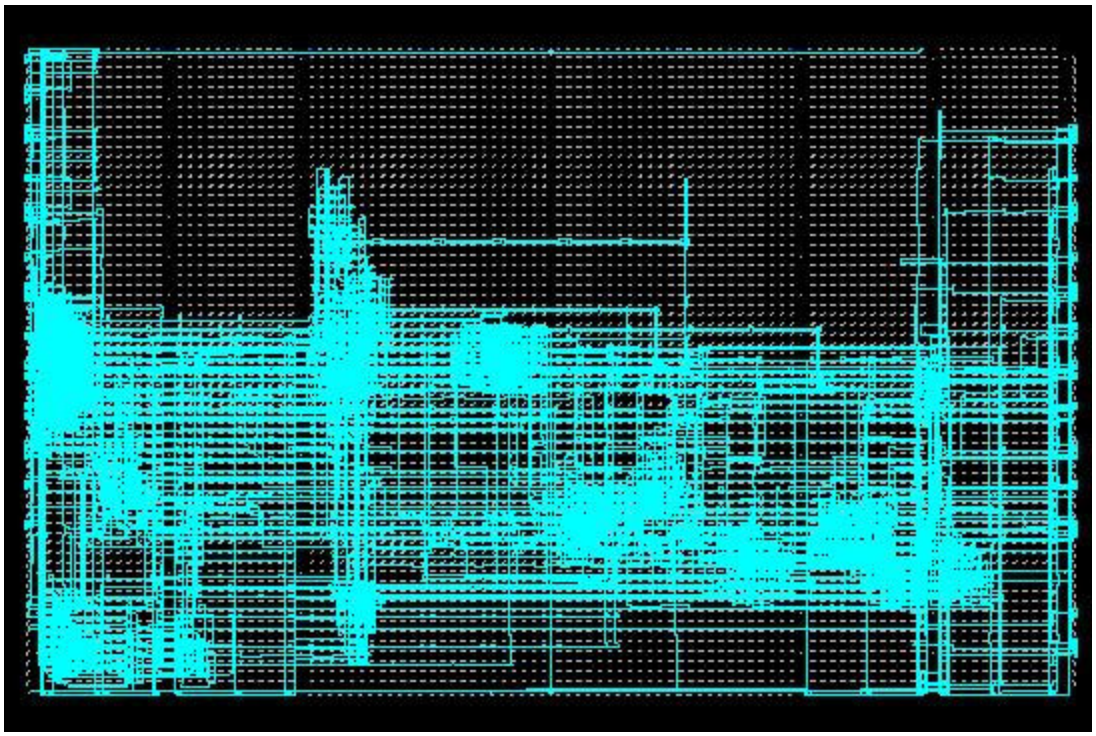


Figure 4.15: Layout of BbNN robot controller

Table 4.5: Speed-up for BbNN robot controller

Run-time in hardware(μs) t_{hw}	Run-time in software(μs) t_{sw}	Speed-up = t_{hw}/t_{sw}
50	8678	174

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

A library of parameterized synthesizable VHDL models for the three different types of neural network blocks has been developed. The use of the library has been demonstrated in two BbNN applications, namely, the XOR pattern classification problem and the mobile robot navigation control problem. Significant speed-ups over software implementations of BbNNs have been achieved in hardware. This lays the foundation for using hardware implementations of BbNNs in adaptive and dynamic environments.

5.2 Future Work

This work is part of a bigger project that aims at implementing evolvable BbNNs for dynamically changing environments. The structure and weights of the BbNN will be generated using an iterative genetic algorithm after fitness evaluation of each candidate solution (BbNN). The immediate extension to this work would be implement a generic parameterized state machine to act as BbNN controller for any *m x n* BbNN. The other important issue to be addressed before implementing BbNNs for dynamic environments is to reduce the time taken to go from the generation of the BbNN structure and weights using the genetic algorithms, to physical implementation on FPGAs. This process involves generation of a HDL model for the BbNN, synthesis, PAR and downloading the FPGA configuration file. The time taken for this process could vary anywhere between a

few minutes to a few hours, depending on the size of the BbNN. Ways to reduce this time have to be explored.

REFERENCES

1. Monash University lecture notes on Neural Networks (Neuro-Fuzzy Computing), http://www.csse.monash.edu.au/~app/CSC437nn/
2. S. W. Moon and S. G. Kong, "Block-based Neural Networks", *IEEE Transactions on Neural Networks*, Vol. 12, No. 2, pp.307-317, March 2001
3. SRC Computers, Inc., http://www.srccomp.com/
4. P. H. W. Leong, M. P. Leong, O. Y. H. Cheung, T. Tung, C. M. Kwok, M. Y. Wong, and K. H. Lee, "Pilchard - A Reconfigurable Computing Platform With Memory Slot Interface", *Proc. of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2001.
5. Annapolis Microsystems, http://www.annapmicro.com
6. Cox, C.E. and Blanz, W.E., "GANGLION-a fast field-programmable gate array implementation of a connectionist classifier", *IEEE Journal of Solid-State Circuits*, Vol. 27, Issue 3, p. 288-299, Mar 1992
7. Montalvo, A.J., R.S. Gyurcsik, and J.J. Paulos, "Towards a general-purpose analog VLSI neural network with on-chip learning", *IEEE Transactions on Neural Networks*, Vol. 8, No. 2, pp.413-423
8. L. Raffo, S.P. Sabatini, G.M. Bo, G.M. Bisio, "Analog VLSI Circuits as Physical Structures for Perception in Early Visual Tasks", *IEEE Transaction on Neural Networks*, 9(3):525-531,1998
9. J.G. Elredge and B.L. Hutchings, "RRANN: A Hardware Implementation of the Backpropagation Algorithm Using Reconfigurable FPGAs", *Proceedings of the IEEE International Conference on Neural Networks*, June 1994.

10. N. Izeboudjen, A. Farah, S. Titri, H. Boumeridja, “Digital Implementation of Artificial Neural Networks: From VHDL Description to FPGA Implementation”, *International Work-conference on Artificial Neural Networks*, (2) 1999: 139-148
11. Rafael Gadea Gironés, Joaquín Cerdá, Francisco J. Ballester, Antonio Mocholí Salcedo, “Artificial Neural Network Implementation on a Single FPGA of a Pipelined On-Line Backpropagation”, *IEEE International Symposium on System Synthesis*, 2000: 225-230
12. M. Alderighi, E. L. Gummati, Vincenzo Piuri, Giacomo R. Sechi, “A FPGA-Based Implementation of a Fault-Tolerant Neural Architecture for Photon Identification”, *FPGA 1997*: 166-172
13. J. G. Eldredge and B. L. Hutchings, “Density enhancement of a neural network using FPGAs and run-time reconfiguration”, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 180--188, Napa, CA, April 1994
14. Philip James-Roxby, Brandon Blodget, “Adapting Constant Multipliers in a Neural Network Implementation”, *FCCM 2000*: 335-336
15. J. Zhu and B.K. Gunther, “Towards an FPGA Based Reconfigurable Computing Environment for Neural Network Implementations”, *Ninth International Conference on Artificial Neural Networks*. 1999, pp.661-667.
16. A. Pérez-Uribe, E. Sanchez, “FPGA Implementation of an Adaptable-Size Neural Network”, *VI International Conference on Artificial Neural Networks ICANN'96*
17. Zhu, Jihan and Sutton, Peter (2003), “An FPGA Implementation of Kak's Instantaneously-Trained, Fast-Classification Neural Networks”, *Proceedings*

2003 IEEE International Conference on Field-Programmable Technology (FPT 2003), December, 2003, pages 126-133, Tokyo, Japan.

18. Marchesi, M.; Orlandi, G.; Piazza, F.; Uncini, A., "Fast neural networks without multipliers", *IEEE Transactions on Neural Networks*, 1993. 4(1): p53-62
19. K. Nichols, M. Moussa, S. Areibi, "Feasibility of Floating-Point Arithmetic in FPGA based Artificial Neural Networks", *CAINE*, San Diego California, pp:8-13, Nov 2002
20. Draghici S, "On the capabilities of neural networks using limited precision weights", *NEURAL NETWORKS* 15 (3): 395-414 APR 2002
21. Jihan Zhu and Peter Sutton, "FPGA Implementations of Neural Networks – a Survey of a Decade of Progress", *Proceedings of 13th International Conference on Field Programmable Logic and Applications* (FPL 2003), Lisbon, Sep 2003.
22. X. Yao, "Promises and Challenges of Evolvable Hardware", *IEEE Transactions on Systems, Man and Cybernetics*, 1999
23. X. Yao, "A New Evolutionary System for Evolving Artificial Neural Networks", *IEEE Transactions on Neural Networks*, 1997
24. Yihua Liao, "Neural Networks in Hardware: A Survey", Department of Computer Science, University of California, Davis
25. Murre J. M. J., "Handbook of Brain Research and Neural Networks", MIT Press, 1995
26. Xilinx Inc., http://www.xilinx.com

27. McCartor, H., “A Highly Parallel Digital Architecture for Neural Network Emulation”, *VLSI for Artificial Intelligence and Neural Networks,* 357-366, Plenum Press, NY, 1991

28. Ramacher, U., Raab, W., Anlauf, J., Hachmann, U., Beichter, J., Bruls, N., Webeling, M. and Sicheneder, E., “Multiprocessor and Memory Architecture of the Neurocomputers SYNAPSE-1”, *Proceedings of the 3rd International Conference on Microelectronics for Neural Networks (MicroNeuro)*, 227-231, 1993

29. de Garis, H., et al., “Initial evolvability experiments on the CAM-brain machines (CBMs)”, *Proceedings of the 2001 Congress on Evolutionary Computation,* 2001. pp 635-641, vol. 1.

30. S. W. Moon and S. G. Kong, "Pattern Classification using Block-based Neural Networks," *Journal of Fuzzy Logic and Intelligent Systems*, Vol. 9, No. 4, pp.393-403, September 1999.

31. S. W. Moon and S. G. Kong, "Pattern Recognition with Block-based Neural Networks," *Proceedings of the International Joint Conference on Neural Networks (IJCNN-2002)*, Honolulu, HI, May 2002.

32. S. W. Moon and S. G. Kong, "Behavior Control of Autonomous Mobile Robots using the Block-based Neural Networks," *Proceedings of the 5th International Symposium on Artificial Life and Robotics (AROB'2000)*, Vol. 1, pp.355-358, Oita, Japan, Jan. 2000.

33. Synopsys® DesignWare™ Generalized sum-of-products data sheet

34. Xilinx® Dual Port block RAM data sheet

APPENDICES

PILCHARD.VHD

```
library ieee;
use ieee.std_logic_1164.all;

entity pilchard is
port (
      PADS_exchecker_reset: in std_logic;
      PADS_dimm_ck: in std_logic;
      PADS_dimm_cke: in std_logic_vector(1 downto 0);
      PADS_dimm_ras: in std_logic;
      PADS_dimm_cas: in std_logic;
      PADS_dimm_we: in std_logic;
      PADS_dimm_s: std_logic_vector(3 downto 0);
      PADS_dimm_a: in std_logic_vector(13 downto 0);
      PADS_dimm_ba: in std_logic_vector(1 downto 0);
      PADS_dimm_rege: in std_logic;
      PADS_dimm_d: inout std_logic_vector(63 downto 0);
      PADS_dimm_cb: inout std_logic_vector(7 downto 0);
      PADS_dimm_dqmb: in std_logic_vector(7 downto 0);
      PADS_dimm_scl: in std_logic;
      PADS_dimm_sda: inout std_logic;
      PADS_dimm_sa: in std_logic_vector(2 downto 0);
      PADS_dimm_wp: in std_logic;
      PADS_io_conn: inout std_logic_vector(27 downto 0) );
end pilchard;

architecture syn of pilchard is

      component INV
      port (
            O: out std_logic;
            I: in std_logic );
      end component;

      component BUF
      port (
            I: in std_logic;
            O: out std_logic );
      end component;

      component BUFG
      port (
            I: in std_logic;
            O: out std_logic );
      end component;

      component CLKDLLHF is
      port (
            CLKIN: in std_logic;
            CLKFB: in std_logic;
            RST: in std_logic;
            CLK0: out std_logic;
            CLK180: out std_logic;
```

```
      CLKDV: out std_logic;
      LOCKED: out std_logic );
end component;

component FDC is
port (
      C: in std_logic;
      CLR: in std_logic;
      D: in std_logic;
      Q: out std_logic );
end component;

component IBUF
port (
      I: in std_logic;
      O: out std_logic );
end component;

component IBUFG
port (
      I: in std_logic;
      O: out std_logic );
end component;

component IOB_FDC is
port (
      C: in std_logic;
      CLR: in std_logic;
      D: in std_logic;
      Q: out std_logic );
end component;

component IOBUF
port (
      I: in std_logic;
      O: out std_logic;
      T: in std_logic;
      IO: inout std_logic );
end component;

component OBUF
port (
      I: in std_logic;
      O: out std_logic );
end component;

component STARTUP_VIRTEX
port (
      GSR: in std_logic;
      GTS: in std_logic;
      CLK: in std_logic );
end component;

component pcore
port (
```

```
      clk: in std_logic;
      clkdiv: in std_logic;
      rst: in std_logic;
      read: in std_logic;
      write: in std_logic;
      addr: in std_logic_vector(13 downto 0);
      din: in std_logic_vector(63 downto 0);
      dout: out std_logic_vector(63 downto 0);
      dmask: in std_logic_vector(63 downto 0);
      extin: in std_logic_vector(25 downto 0);
      extout: out std_logic_vector(25 downto 0);
      extctrl: out std_logic_vector(25 downto 0) );
end component;

signal clkdllhf_clk0: std_logic;
signal clkdllhf_clkdiv: std_logic;
signal dimm_ck_bufg: std_logic;
signal dimm_s_ibuf: std_logic;
signal dimm_ras_ibuf: std_logic;
signal dimm_cas_ibuf: std_logic;
signal dimm_we_ibuf: std_logic;
signal dimm_s_ibuf_d: std_logic;
signal dimm_ras_ibuf_d: std_logic;
signal dimm_cas_ibuf_d: std_logic;
signal dimm_we_ibuf_d: std_logic;
signal dimm_d_iobuf_i: std_logic_vector(63 downto 0);
signal dimm_d_iobuf_o: std_logic_vector(63 downto 0);
signal dimm_d_iobuf_t: std_logic_vector(63 downto 0);
signal dimm_a_ibuf: std_logic_vector(14 downto 0);
signal dimm_dqmb_ibuf: std_logic_vector(7 downto 0);
signal io_conn_iobuf_i: std_logic_vector(27 downto 0);
signal io_conn_iobuf_o: std_logic_vector(27 downto 0);
signal io_conn_iobuf_t: std_logic_vector(27 downto 0);

signal s,ras,cas,we : std_logic;

signal VDD: std_logic;
signal GND: std_logic;

signal CLK: std_logic;
signal CLKDIV: std_logic;
signal RESET: std_logic;
signal READ: std_logic;
signal WRITE: std_logic;
signal READ_p: std_logic;
signal WRITE_p: std_logic;
signal READ_n: std_logic;
signal READ_buf: std_logic;
signal WRITE_buf: std_logic;
signal READ_d: std_logic;
signal WRITE_d: std_logic;
signal READ_d_n: std_logic;
signal READ_d_n_buf: std_logic;

signal pcore_addr_raw: std_logic_vector(13 downto 0);
```

```
      signal pcore_addr: std_logic_vector(13 downto 0);
      signal pcore_din: std_logic_vector(63 downto 0);
      signal pcore_dout: std_logic_vector(63 downto 0);
      signal pcore_dmask: std_logic_vector(63 downto 0);
      signal pcore_extin: std_logic_vector(25 downto 0);
      signal pcore_extout: std_logic_vector(25 downto 0);
      signal pcore_extctrl: std_logic_vector(25 downto 0);
      signal pcore_dqmb: std_logic_vector(7 downto 0);

--    CLKDIV frequency control, default is 2
--  uncomment the following lines so as to redefined the clock rate
--  given by clkdiv
      attribute CLKDV_DIVIDE: string;
      attribute CLKDV_DIVIDE of U_clkdllhf: label is "8";


begin

      VDD <= '1';
      GND <= '0';

      U_ck_bufg: IBUFG port map (
            I => PADS_dimm_ck,
            O => dimm_ck_bufg );

      U_reset_ibuf: IBUF port map (
            I => PADS_exchecker_reset,
            O => RESET );

      U_clkdllhf: CLKDLLHF port map (
            CLKIN => dimm_ck_bufg,
            CLKFB => CLK,
            RST => RESET,
            CLK0 => clkdllhf_clk0,
            CLK180 => open,
            CLKDV => clkdllhf_clkdiv,
            LOCKED => open );

      U_clkdllhf_clk0_bufg: BUFG port map (
            I => clkdllhf_clk0,
            O => CLK );

      U_clkdllhf_clkdiv_bufg: BUFG port map (
            I => clkdllhf_clkdiv,
            O => CLKDIV );

      U_startup: STARTUP_VIRTEX port map (
            GSR => RESET,
            GTS => GND,
            CLK => CLK );

      U_dimm_s_ibuf: IBUF port map (
            I => PADS_dimm_s(0),
            O => dimm_s_ibuf );
```

```
U_dimm_ras_ibuf: IBUF port map (
      I => PADS_dimm_ras,
      O => dimm_ras_ibuf );

U_dimm_cas_ibuf: IBUF port map (
      I => PADS_dimm_cas,
      O => dimm_cas_ibuf );

U_dimm_we_ibuf: IBUF port map (
      I => PADS_dimm_we,
      O => dimm_we_ibuf );

G_dimm_d: for i in integer range 0 to 63 generate

      U_dimm_d_iobuf: IOBUF port map (
            I => dimm_d_iobuf_i(i),
            O => dimm_d_iobuf_o(i),
            T => dimm_d_iobuf_t(i),
            IO => PADS_dimm_d(i) );

      U_dimm_d_iobuf_o: IOB_FDC port map (
            C => CLK,
            CLR => RESET,
            D => dimm_d_iobuf_o(i),
            Q => pcore_din(i) );

      U_dimm_d_iobuf_i: IOB_FDC port map (
            C => CLK,
            CLR => RESET,
            D => pcore_dout(i),
            Q => dimm_d_iobuf_i(i) );

      U_dimm_d_iobuf_t: IOB_FDC port map (
            C => CLK,
            CLR => RESET,
            D => READ_d_n_buf,
            Q => dimm_d_iobuf_t(i) );

end generate;

G_dimm_a: for i in integer range 0 to 13 generate

      U_dimm_a_ibuf: IBUF port map (
            I => PADS_dimm_a(i),
            O => dimm_a_ibuf(i) );

      U_dimm_a_ibuf_o: IOB_FDC port map (
            C => CLK,
            CLR => RESET,
            D => dimm_a_ibuf(i),
            Q => pcore_addr_raw(i) );

end generate;

pcore_addr(3 downto 0) <= pcore_addr_raw(3 downto 0);
```

```
addr_correct: for i in integer range 4 to 7 generate
      ADDR_INV: INV port map (
            O => pcore_addr(i),
            I => pcore_addr_raw(i) );
end generate;
pcore_addr(13 downto 8) <= pcore_addr_raw(13 downto 8);

G_dimm_dqmb: for i in integer range 0 to 7 generate

      U_dimm_dqmb_ibuf: IBUF port map (
            I => PADS_dimm_dqmb(i),
            O => dimm_dqmb_ibuf(i) );

      U_dimm_dqmb_ibuf_o: IOB_FDC port map (
            C => CLK,
            CLR => RESET,
            D => dimm_dqmb_ibuf(i),
            Q => pcore_dqmb(i) );

end generate;

pcore_dmask(7 downto 0) <= (others => (not pcore_dqmb(0)));
pcore_dmask(15 downto 8) <= (others => (not pcore_dqmb(1)));
pcore_dmask(23 downto 16) <= (others => (not pcore_dqmb(2)));
pcore_dmask(31 downto 24) <= (others => (not pcore_dqmb(3)));
pcore_dmask(39 downto 32) <= (others => (not pcore_dqmb(4)));
pcore_dmask(47 downto 40) <= (others => (not pcore_dqmb(5)));
pcore_dmask(55 downto 48) <= (others => (not pcore_dqmb(6)));
pcore_dmask(63 downto 56) <= (others => (not pcore_dqmb(7)));

G_io_conn: for i in integer range 2 to 27 generate

      U_io_conn_iobuf: IOBUF port map (
            I => io_conn_iobuf_i(i),
            O => io_conn_iobuf_o(i),
            T => io_conn_iobuf_t(i),
            IO => PADS_io_conn(i) );

      U_io_conn_iobuf_o: IOB_FDC port map (
            C => CLK,
            CLR => RESET,
            D => io_conn_iobuf_o(i),
            Q => pcore_extin(i - 2) );

      U_io_conn_iobuf_i: IOB_FDC port map (
            C => CLK,
            CLR => RESET,
            D => pcore_extout(i - 2),
            Q => io_conn_iobuf_i(i) );

      U_io_conn_iobuf_t: IOB_FDC port map (
            C => CLK,
            CLR => RESET,
            D => pcore_extctrl(i - 2),
            Q => io_conn_iobuf_t(i) );
```

```
end generate;

U_io_conn_0_iobuf: IOBUF port map (
      I => dimm_ck_bufg,
      O => open,
      T => GND,
      IO => PADS_io_conn(0) );

U_io_conn_1_iobuf: IOBUF port map (
      I => GND,
      O => open,
      T => VDD,
      IO => PADS_io_conn(1) );

READ_p <=
      (not dimm_s_ibuf) and
      (dimm_ras_ibuf) and
      (not dimm_cas_ibuf) and
      (dimm_we_ibuf);

U_read: FDC port map (
      C => CLK,
      CLR => RESET,
      D => READ_p,
      Q => READ );

U_buf_read: BUF port map (
      I => READ,
      O => READ_buf );

U_read_d: FDC port map (
      C => CLK,
      CLR => RESET,
      D => READ,
      Q => READ_d );

WRITE_p <=
      (not dimm_s_ibuf) and
      (dimm_ras_ibuf) and
      (not dimm_cas_ibuf) and
      (not dimm_we_ibuf);

U_write: FDC port map (
      C => CLK,
      CLR => RESET,
      D => WRITE_p,
      Q => WRITE );

U_buf_write: BUF port map (
      I => WRITE,
      O => WRITE_buf );

U_write_d: FDC port map (
      C => CLK,
```

```
            CLR => RESET,
            D => WRITE,
            Q => WRITE_d );

      READ_n <= not READ;

      U_read_d_n: FDC port map (
            C => CLK,
            CLR => RESET,
            D => READ_n,
            Q => READ_d_n );

      U_buf_read_d_n: BUF port map (
            I => READ_d_n,
            O => READ_d_n_buf );

      -- User logic should be placed inside pcore
      U_pcore: pcore port map (
            clk => CLK,
            clkdiv => CLKDIV,
            rst => RESET,
            read => READ,
            write => WRITE,
            addr => pcore_addr,
            din => pcore_din,
            dout => pcore_dout,
            dmask => pcore_dmask,
            extin => pcore_extin,
            extout => pcore_extout,
            extctrl => pcore_extctrl );

end syn;
```

PCORE.VHD

```
-- pcore wrapper for parith/block31 and DPRAM
-- author: Sampath Kothandaraman

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity pcore is

    port ( clk: in std_logic;
         clkdiv: in std_logic;
         rst: in std_logic;
         read: in std_logic;
         write: in std_logic;
         addr: in std_logic_vector(13 downto 0);
         din: in std_logic_vector(63 downto 0);
         dout: out std_logic_vector(63 downto 0);
         dmask: in std_logic_vector(63 downto 0);
         extin: in std_logic_vector(25 downto 0);
         extout: out std_logic_vector(25 downto 0);
         extctrl: out std_logic_vector(25 downto 0)
       );

end pcore;


architecture syn of pcore is

    component dpram256_64
      port ( addra: IN std_logic_VECTOR(7 downto 0);
               clka: IN std_logic;
               dina: IN std_logic_VECTOR(63 downto 0);
               douta: OUT std_logic_VECTOR(63 downto 0);
               wea: IN std_logic;
               addrb: IN std_logic_VECTOR(7 downto 0);
               clkb: IN std_logic;
               dinb: IN std_logic_VECTOR(63 downto 0);
               doutb: OUT std_logic_VECTOR(63 downto 0);
               web: IN std_logic
           );
    end component;

    component parith
        port ( clk: in std_logic;
               rst: in std_logic;
               data_fromram: in std_logic_vector(63 downto 0);
               start: in std_logic;
             addr: out std_logic_vector(7 downto 0);
               data2ram: out std_logic_vector(63 downto 0);
               we: out std_logic;
               finish: out std_logic
         );
```

```
    end component;

    signal addrb:std_logic_VECTOR(7 downto 0);
    signal clkb: std_logic;
    signal dinb: std_logic_VECTOR(63 downto 0);
    signal doutb: std_logic_VECTOR(63 downto 0);
    signal web: std_logic;
    signal start: std_logic;
    signal finish: std_logic;
    signal bram_dout : std_logic_VECTOR(63 downto 0);
    --debug signal
    signal  start_debug:std_logic;
    signal start_ctr : std_logic_vector(4 downto 0);
    signal tmp_start_ctr : std_logic_vector(3 downto 0);
    --register interface
    --signal reg0: std_logic_VECTOR(31 downto 0);

begin

    ram0:dpram256_64 port map  ( addra    => addr(7 downto 0),
                               clka => clk,
                               dina => din,
                               douta      => bram_dout,
                                 wea      => write,
                                 addrb    => addrb,
                               clkb => clkb,
                               dinb => dinb,
                               doutb      => doutb,
                               web  => web
                               );

    parith0: parith port map ( clk  => clkb,
                             rst    => rst,
                             data_fromram => doutb,
                             start  => start_debug,
                               addr => addrb,
                             data2ram => dinb,
                             we     => web,
                             finish => finish
                             );


    process(clk,rst)
    begin
        if (rst = '1') then
          start_debug <= '0';
        elsif (clk'event and clk ='1') then
            start_debug <= start_debug or start;
      end if;
    end process;


    dout <= bram_dout;
```

```
    start <= '1' when (write = '1' and addr(7 downto 0) = "11111111")
else '0';

-- define the core clock

    clkb <= clkdiv;

end syn;
```

PARITH.VHD FOR XOR PATTERN CLASSIFICATION

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity parith is

    port ( clk: in std_logic;
           rst: in std_logic;
           data_fromram: in std_logic_vector(63 downto 0);
-- data read from block ram
           start: in std_logic;
         addr: out std_logic_vector(7 downto 0);
           data2ram: out std_logic_vector(63 downto 0);
-- data to write to block ram
           we: out std_logic;                          -- write enable
           finish: out std_logic
         );

end parith;


architecture rtl of parith is
    --signal state_flag : std_logic_vector( 1 downto 0);
    signal state : std_logic_vector(3 downto 0);
    signal idx: std_logic_vector(7 downto 0);
    signal buf_inp, buf_bias, buf_wAB, buf_wCD: std_logic_vector(63
downto 0);
    signal out2: std_logic_vector(63 downto 0);
    signal flag_parith: std_logic;        -- tc,
    signal ip_addr_ctr, op_addr_ctr: std_logic_vector(7 downto 0);

    constant zeros : std_logic_vector(63 downto 8) := (others => '0');

component xor_bbnn
      generic ( x_width: NATURAL := 8;
              b_width : NATURAL := 8;
              w_width : NATURAL := 8;
              y_width: NATURAL := 64
            );
      port (  clk : in std_logic;
            rst : in std_logic;
            inp : in std_logic_vector(2*x_width-1 downto 0);
            bias_ABCD : in std_logic_vector(8*x_width-1 downto 0);
            w_AB: in std_logic_vector(8*x_width-1 downto 0);
            w_CD: in std_logic_vector(8*x_width-1 downto 0);
            flag: in std_logic;
            out2: out std_logic_vector(y_width-1 downto 0)
           );
end component;

begin
```

```
    -- retrieve data from block RAM and do the block31 opereation

    U0: xor_bbnn port map ( clk => clk,
                      rst => rst,
                      inp => buf_inp(15 downto 0),
                      bias_ABCD => buf_bias,
                      w_AB => buf_wAB,
                      w_CD => buf_wCD,
                      flag => flag_parith,
                      out2 => out2
           );

 --   tc <= '0';

    process (clk, rst)
    begin
        if (rst = '1') then
          flag_parith <= '1';
          state <= "0000";
          finish <= '0';
          ip_addr_ctr <= "00000011";
-- initialize ip_addr_ctr = 3, the location of first input dataset
--(in1, in2)
          op_addr_ctr <= "01000011";
                                    -- initialize ip_addr_ctr = 67, the
location of first output data out2
        --  state_flag <= (others => '0');
          idx <= (others => '0');
          we <= '0';
        elsif (clk = '1' and clk'event)  then
          if (state = "0000") then
            if (start = '1') then         -- state machine
                  flag_parith <= '1';
                  ip_addr_ctr <= "00000011";
                        -- 3 - start loc of inp data
                  op_addr_ctr <= "01000011";
                        -- 67 - start loc of output data
                  state <= "0001";
                  finish <= '0';
                  --state_flag <= (others => '0');
            end if;
          elsif (state = "0001") then
            idx <= "00000000";      -- issue addr for bias (0)
            state <= "0010";
          elsif (state = "0010") then -- issue addr for wts_AB (1)
            idx <= "00000001";
            state <= "0011";
          elsif (state = "0011") then -- get bias values, issue addr
for wts_CD (2)
            idx <= "00000010";
            buf_bias <= data_fromram;
            state <= "0100";
          elsif (state = "0100") then  -- get wts_AB values, issue addr
for inp dataset in row 1 (3) (in1, in2)
            idx <= ip_addr_ctr;
```

```
            buf_wAB <= data_fromram;
            state <= "0101";
          elsif (state = "0101") then  -- get wts_CD values
            buf_wCD <= data_fromram;
            state <= "0110";

-- ^^ loading biases and weights for a given BbNN and for a set of 64 -
-- data ^^
-- vv reading input data, computing output and writing to RAM vv

          elsif (state = "0110") then -- get inp row 1 values
            if (ip_addr_ctr = "01000011") then
                  finish <= '1';
                  state <= "0000";
            elsif (ip_addr_ctr = "00000011") then
                  buf_inp <= data_fromram;
                  state <= "1000";
            else
                  state <= "0111";
            end if;
          elsif (state = "0111") then
            buf_inp <= data_fromram;
            state <= "1000";
          elsif (state = "1000") then
            flag_parith <= '0';
            state <= "1001";
          elsif (state = "1001") then
            state <= "1010";
          elsif (state = "1010") then
            state <= "1011";
          elsif (state = "1011") then -- BRK
            idx <= op_addr_ctr;
            ip_addr_ctr <= ip_addr_ctr + 1;
            we <= '1';
            state <= "1100";
          elsif (state = "1100") then
            we <= '0';
            idx <= ip_addr_ctr;
            op_addr_ctr <= op_addr_ctr + 1;
            state <= "0110";
       --     state <= "1101";
        --    elsif (state = "1101") then
        --          state <= state;
            else    -- dummy, NDR
              finish <= '0';
            end if;
        end if;
    end process;

    addr <= idx;
    data2ram <= out2;
end rtl;
```

XOR_BBNN.VHD

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

entity xor_bbnn is
      generic ( x_width: NATURAL := 8;
              b_width : NATURAL := 8;
              w_width : NATURAL := 8;
              y_width: NATURAL := 64
            );
      port (  clk : in std_logic;
            rst : in std_logic;
            inp : in std_logic_vector(2*x_width-1 downto 0);
            bias_ABCD : in std_logic_vector(8*b_width-1 downto 0);
            w_AB: in std_logic_vector(8*w_width-1 downto 0);
            w_CD: in std_logic_vector(8*w_width-1 downto 0);
            flag: in std_logic;
--          out2: out std_logic_vector(x_width-1 downto 0)
            out2: out std_logic_vector(y_width-1 downto 0)
           );
end xor_bbnn;

architecture structure_xor of xor_bbnn is

      signal tc: std_logic;
      signal tmp_a_out, tmp_b_out, tmp_c_out, tmp_d_out:
std_logic_vector(y_width-1 downto 0);
      signal sig_BA, sig_DC : std_logic_vector(x_width-1 downto 0);
      signal sig_b_inx2 : std_logic_vector(x_width-1 downto 0);
      signal sig_c_inx1 : std_logic_vector(x_width-1 downto 0);
      signal sig_d_inx1 : std_logic_vector(x_width-1 downto 0);
      signal sig_d_inx2 : std_logic_vector(x_width-1 downto 0);

      component regout_block22
            generic ( x_width : NATURAL := 8;
                        w_width : NATURAL := 8;
                  b_width : NATURAL := 8;
                        num     : NATURAL := 2;
                        y_width : NATURAL := 64
                      );
            port (      clk, rst, flag : in std_logic;
                  x1, x2 : in std_logic_vector(x_width-1 downto 0);
                  w13, w23, w14, w24 : in std_logic_vector(w_width-1
downto 0);
                  b3, b4 : in std_logic_vector(b_width-1 downto 0);
                  tc : in std_logic;
                  y3_y4 : out std_logic_vector(y_width-1 downto 0)
                 );
      end component;

begin
```

```
-- concurrent statements

      tc <= '0';

      -- output

      out2 <= tmp_d_out;

-- instantiating the basic blocks

      A_22: regout_block22
            port map ( clk => clk,
                       rst => rst,
                       flag => flag,
                       x1 => inp(2*x_width-1 downto x_width),
                       x2 => sig_BA,
                       w13 => w_AB(w_width-1 downto 0),
                       w23 => w_AB(2*w_width-1 downto w_width),
                       w14 => w_AB(3*w_width-1 downto 2*w_width),
                       w24 => w_AB(4*w_width-1 downto 3*w_width),
                       b3 => bias_ABCD(b_width-1 downto 0),
                       b4 => bias_ABCD(2*b_width-1 downto b_width),
                       tc => tc,
                       y3_y4 => tmp_a_out
                     );

      B_22: regout_block22
            port map ( clk => clk,
                       rst => rst,
                       flag => flag,
                       x1 => inp(x_width-1 downto 0),
                       x2 => sig_b_inx2(x_width-1 downto 0),
                       w13 => w_AB(5*w_width-1 downto 4*w_width),
                       w23 => w_AB(6*w_width-1 downto 5*w_width),
                       w14 => w_AB(7*w_width-1 downto 6*w_width),
                       w24 => w_AB(8*w_width-1 downto 7*w_width),
                       b3 => bias_ABCD(3*b_width-1 downto 2*b_width),
                       b4 => bias_ABCD(4*b_width-1 downto 3*b_width),
                       tc => tc,
                       y3_y4 => tmp_b_out
                     );

      C_22: regout_block22
            port map ( clk => clk,
                       rst => rst,
                       flag => flag,
                       x1 => sig_c_inx1(x_width-1 downto 0),
                       x2 => sig_DC,
                       w13 => w_CD(w_width-1 downto 0),
                       w23 => w_CD(2*w_width-1 downto w_width),
                       w14 => w_CD(3*w_width-1 downto 2*w_width),
                       w24 => w_CD(4*w_width-1 downto 3*w_width),
                       b3 => bias_ABCD(5*b_width-1 downto 4*b_width),
                       b4 => bias_ABCD(6*b_width-1 downto 5*b_width),
                       tc => tc,
```

```
                  y3_y4 => tmp_c_out
                );

      D_22: regout_block22
            port map ( clk => clk,
                  rst => rst,
                  flag => flag,
                  x1 => sig_d_inx1(x_width-1 downto 0),
                  x2 => sig_d_inx2(x_width-1 downto 0),
                  w13 => w_CD(5*w_width-1 downto 4*w_width),
                  w23 => w_CD(6*w_width-1 downto 5*w_width),
                  w14 => w_CD(7*w_width-1 downto 6*w_width),
                  w24 => w_CD(8*w_width-1 downto 7*w_width),
                  b3 => bias_ABCD(7*b_width-1 downto 6*b_width),
                  b4 => bias_ABCD(8*b_width-1 downto 7*b_width),
                  tc => tc,
                  y3_y4 => tmp_d_out
                );

-- processes

-- flag = 1 means, reset all block outputs to 0; flag = 0 means, work
-- with feedbck data

      process (rst, flag, tmp_a_out, tmp_b_out, tmp_c_out, tmp_d_out)
-- clk removed from sensitivity list : MUX on flag

      begin
            if (rst = '1' or flag = '1') then
                  sig_DC <= (others => '0');
                  sig_BA <= (others => '0');
                  sig_b_inx2 <= (others => '0');
                  sig_c_inx1 <= (others => '0');
                  sig_d_inx1 <= (others => '0');
                  sig_d_inx2 <= (others => '0');
            elsif (flag = '0') then
                  sig_b_inx2 <= tmp_a_out(x_width-1 downto 0);
                  sig_c_inx1 <= tmp_a_out(2*x_width-1 downto x_width);
      -- y4 output of A
                  sig_d_inx1 <= tmp_b_out(2*x_width-1 downto x_width);
      -- y4 output of B
                  sig_d_inx2 <= tmp_c_out(x_width-1 downto 0);
      -- y3 output of C
                  sig_BA <= tmp_b_out(x_width-1 downto 0);
-- y3 output of B
                  sig_DC <= tmp_d_out(x_width-1 downto 0);
-- y3 output of D
            end if;
      end process;

end structure_xor;
```

REGOUT_BLOCK22.VHD

```
-- Author : Sampath Kothandaraman

-- Synthesizable 2-input 2-output block
-- Uses IP from Synopsys(C) DesignWare(TM): generalized sum-of-products
-- 8-bit version
-- activation function: bipolar-ramp-with-saturation

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

entity regout_block22 is

    generic ( x_width : NATURAL := 8;
              w_width : NATURAL := 8;
            b_width : NATURAL := 8;
              num     : NATURAL := 2;
              y_width : NATURAL := 64
             );
    port ( clk, rst, flag : in std_logic;
         x1, x2 : in std_logic_vector(x_width-1 downto 0);
           w13, w23, w14, w24 : in std_logic_vector(w_width-1 downto
0);
           b3, b4 : in std_logic_vector(b_width-1 downto 0);
         tc : in std_logic;
           y3_y4 : out std_logic_vector(y_width-1 downto 0)
         );

end regout_block22;


architecture regout_structure22 of regout_block22 is

      constant bias_wt : std_logic_vector(w_width-1 downto 0) :=
"00000001";
      signal sop3, sop4: std_logic_vector(num+x_width+w_width-1 downto
0);
      signal x1_x2_b3, x1_x2_b4 : std_logic_vector(2*x_width+b_width-1
downto 0);
      signal w13_w23_bias_wt, w14_w24_bias_wt :
std_logic_vector(2*w_width+b_width-1 downto 0);
      signal y3, y4 : std_logic_vector(x_width-1 downto 0);

        component DW02_prod_sum_inst
                generic ( inst_A_width : NATURAL := 8;
                          inst_B_width : NATURAL := 8;
                          inst_num_inputs : POSITIVE := 3;
                          inst_SUM_width : NATURAL := 18
                        );
                port ( inst_A : in
std_logic_vector(inst_num_inputs*inst_A_width-1 downto 0);
```

```
                        inst_B : in
std_logic_vector(inst_num_inputs*inst_B_width-1 downto 0);
                        inst_TC : in std_logic;
                        SUM_inst : out std_logic_vector(inst_SUM_width-1
downto 0)
                      );
        end component;

      component rampsat_bi
            generic ( in_width: POSITIVE := 18;
                    out_width: POSITIVE := 8
                  );
            port ( rst, flag : in std_logic;
                   act_in  : in std_logic_vector(in_width-1 downto 0);
                   act_out: out std_logic_vector(out_width-1 downto 0)
                 );
      end component;

begin

-- leading bits of output vector padded with 0's

            y3_y4(63 downto 16) <= (others => '0');   -- new

-- calculation of output y3

      x1_x2_b3 <= x1 & x2 & b3;
      w13_w23_bias_wt <= w13 & w23 & bias_wt;

        U1_sop: DW02_prod_sum_inst
                port map ( inst_A => x1_x2_b3,
                           inst_B => w13_w23_bias_wt,
                           inst_TC => tc,
                           SUM_inst => sop3
                         );
      U1_y3: rampsat_bi
            port map( rst => rst,
                    flag => flag,
                    act_in => sop3,
                    act_out => y3
                  );

-- calculation of output y4

      x1_x2_b4 <= x1 & x2 & b4;
      w14_w24_bias_wt <= w14 & w24 & bias_wt;

        U2_sop: DW02_prod_sum_inst
                port map ( inst_A => x1_x2_b4,
                           inst_B => w14_w24_bias_wt,
                           inst_TC => tc,
                           SUM_inst => sop4
                         );
      U2_y4: rampsat_bi
            port map ( rst => rst,
```

```
                    flag => flag,
                      act_in => sop4,
                    act_out => y4
                  );

      process (clk, rst, y3, y4)
      begin
            if (clk'event and clk = '1') then
                  y3_y4(15 downto 8) <= y3;
                  y3_y4(7 downto 0) <= y4;
            end if;
      end process;

end regout_structure22;
```

DW02_PROD_SUM_INST.VHD

```
library IEEE,DWARE,DW02;
use IEEE.std_logic_1164.all;
use DWARE.DWpackages.all;
use DW02.DW02_components.all;

entity DW02_prod_sum_inst is
      generic (
          inst_A_width : NATURAL;
          inst_B_width : NATURAL;
          inst_num_inputs : POSITIVE;
          inst_SUM_width : NATURAL
          );
      port (
          inst_A : in std_logic_vector(inst_num_inputs*inst_A_width-1
downto 0);
          inst_B : in std_logic_vector(inst_num_inputs*inst_B_width-1
downto 0);
          inst_TC : in std_logic;
          SUM_inst : out std_logic_vector(inst_SUM_width-1 downto 0)
          );
    end DW02_prod_sum_inst;

architecture inst of DW02_prod_sum_inst is

begin

    -- Instance of DW02_prod_sum
    U1 : DW02_prod_sum
      generic map ( A_width => inst_A_width, B_width => inst_B_width,
num_inputs => inst_num_inputs, SUM_width => inst_SUM_width )
      port map ( A => inst_A, B => inst_B, TC => inst_TC, SUM =>
SUM_inst );


end inst;


-- pragma translate_off
library DW02;
configuration DW02_prod_sum_inst_cfg_inst of DW02_prod_sum_inst is
for inst
    for U1 : DW02_prod_sum use configuration
DW02.DW02_prod_sum_cfg_sim; end for;
end for; -- inst
end DW02_prod_sum_inst_cfg_inst;
-- pragma translate_on
```

RAMPSAT_BI.VHD

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

entity rampsat_bi is
      generic ( in_width: POSITIVE := 18;
              out_width: POSITIVE := 8
            );
      port(       rst, flag : in std_logic;
            act_in  : in std_logic_vector(in_width-1 downto 0);
            act_out: out std_logic_vector(out_width-1 downto 0)
          );
end rampsat_bi;

architecture working of rampsat_bi is

      constant sat_high: std_logic_vector(in_width+out_width-1 downto
0) := "01111111111111111111111111";
        constant sat_low: std_logic_vector(in_width+out_width-1 downto
0) :=  "10000000000000000000000000";
        constant sat_xlow: std_logic_vector(out_width-1 downto 0) :=
"10000011";
        constant sat_xhigh: std_logic_vector(out_width-1 downto 0) :=
"01111111";
      constant sat_slope: std_logic_vector(out_width-1 downto 0) :=
"00000001";

      signal act_out_sig : std_logic_vector(in_width+out_width-1 downto
0);

begin

      process(rst, flag, act_in)
      begin
            if (rst = '1') then
                  act_out_sig <= (others => '0');

            else
                  if (flag = '1') then
                        act_out_sig <= (others => '0');
                  else
                              act_out_sig <= sat_slope * act_in;
                  end if;
            end if;
      end process;

      act_out <= act_out_sig(out_width-1 downto 0);

end working;
```

TB.VHD FOR XOR PATTERN CLASSIFICATION

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity tb is
end tb;

architecture syn of tb is

component pcore
port (
      clk: in std_logic;
      clkdiv: in std_logic;
      rst: in std_logic;
      read: in std_logic;
      write: in std_logic;
      addr: in std_logic_vector(13 downto 0);
      din: in std_logic_vector(63 downto 0);
      dout: out std_logic_vector(63 downto 0);
      dmask: in std_logic_vector(63 downto 0);
      extin: in std_logic_vector(25 downto 0);
      extout: out std_logic_vector(25 downto 0);
      extctrl: out std_logic_vector(25 downto 0)
);
end component;

signal clk: std_logic;
signal clkdiv: std_logic;
signal reset: std_logic;
signal read: std_logic;
signal write: std_logic;
signal addr: std_logic_vector(13 downto 0);
signal din: std_logic_vector(63 downto 0);
signal dout: std_logic_vector(63 downto 0);
signal dmask: std_logic_vector(63 downto 0);
signal extin: std_logic_vector(25 downto 0);
signal extout: std_logic_vector(25 downto 0);
signal extctrl: std_logic_vector(25 downto 0);

begin


pcore0: pcore port map(
clk, clkdiv, reset, read, write, addr,
din, dout, dmask, extin, extout, extctrl
);

process
begin

reset <= '1';
clk <= '0';
```

```
wait for 100 ns;

loop
      reset <= '0';
      clk <= '1';
      wait for 50 ns;
      clk <= '0';
      wait for 50 ns;
end loop;
end process;

process (clk, reset)
begin
if reset = '1' then
      clkdiv <= '0';
elsif clk'event and clk = '1' then
      clkdiv <= not clkdiv;
end if;
end process;

process
begin
read <= '0';
write <= '0';
addr <= "00000000000000";

wait for 200 ns;
read <= '0';

----------------------------
-- insert the 64 data here

write <= '1'; addr(7 downto 0) <= "00000000" ;
din <=
"0000000000000001000000100000001100000000000000010000001000000011";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00000001" ;
din <=
"0000000000000001000000100000001100000000000000010000001000000011";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00000010" ;
din <=
"0000000000000001000000100000001100000000000000010000001000000011";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00000011" ;
din <=
"0110110100100001110011001010010100110100110000000000101010111001";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00000100" ;
din <=
"1000000000111110111101011101111001100101101100110011100010110110";
```

```
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00000101" ;
din <=
"100000011000001010011000011001101110100011000010000101010011110 0";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00000110" ;
din <=
"001000010010100001101001100100001101110111101010100111111101110 1";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00000111" ;
din <=
"100101101110111001100011100101011100101001001110111000000111100 0";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00001000" ;
din <=
"010001010100000000110010011010011011011001011000000000100000001 1";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00001001" ;
din <=
"011100011011000100111001111001110011010101010100100010111101111 1";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00001010" ;
din <=
"100010111010001000011100101011010010101000000111101110001010010 00";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00001011" ;
din <=
"000010000100111111001111010110111001000101110100100100111101000 10";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00001100" ;
din <=
"011010110001001101000111000001101110100001100110110000010110001 0";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00001101" ;
din <=
"001010010011001101010011101111011101100111101111111000001110101 1";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00001110" ;
din <=
"001011010001110001001011011000110101010110101010011010001010010 1";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00001111" ;
din <=
"100100000101110011001000011110011111111100011010000110001110110 1";
```

```
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00010000" ;
din <=
"10101100000100010010111011110011111001000110111011101011110100011";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00010001" ;
din <=
"10001011000111110000110011001000101101101110010100001111101011100";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00010010" ;
din <=
"01101011111000001001000110011010100000101100010100101011001000010";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00010011" ;
din <=
"00111100111001000010111011001001001001001001100000011000111110101";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00010100" ;
din <=
"10100110011100111101001100000000010100011011110110001111010111110";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00010101" ;
din <=
"11001000000110001101001011100011001000100001110101111000100110011001";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00010110" ;
din <=
"01000001100111101001110000111111001111000000000110000000111011100";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00010111" ;
din <=
"01111011011111100010100000111010100010010101010000011110011111011";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00011000" ;
din <=
"10001011010001001111010001001001000110011010010110011010111110001";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00011001" ;
din <=
"00001100111100111101010010101011000011110011000000111011000111111";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00011010" ;
din <=
"11000100110001111000011101011000001011010111110001001110100101010";
```

```
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00011011" ;
din <=
"0100101100110101101000110010001000100001000111000111111110001111";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00011100" ;
din <=
"0110100010110101011100111110000011011011010100110001011010100 1";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00011101" ;
din <=
"1110110101001100000000100000001011100111111100111011101001011010";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00011110" ;
din <=
"0110110001111010101001110101010110000110111101101011111001001011";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00011111" ;
din <=
"1101000111010111100101101100001000111001101110110101100100001001";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00100000" ;
din <=
"0101011110011101111000010000111000100110001010110110111001010001";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00100001" ;
din <=
"0001111100101111100010000000110000010111001101101101111110001001";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00100010" ;
din <=
"1011110011001001010101100011010110001011010001111100000111000111";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00100011" ;
din <=
"1110001011110111010010101000100101000110001001100011110100111001";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00100100" ;
din <=
"0111011100001101100000111111010010010011001010110000010101100 1100";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00100101" ;
din <=
"1110101011100011010100100110011011011000000101100001011110011101";
```

```
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00100110" ;
din <=
"000100011000000010110010001001110111110011110010110011101110000110";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00100111" ;
din <=
"000010100101011100001001111111010110001000001010001000000011111001";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00101000" ;
din <=
"110011100000011011111000100001110000001000101011001010100011101 1";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00101001" ;
din <=
"111001111101110011100001001011100010101000010011100001111011001 1";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00101010" ;
din <=
"001010100001111001001000111001111101001010001100101100111000000 0";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00101011" ;
din <=
"010110001101000110011101011111110100110001000100101101011011101 1";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00101100" ;
din <=
"001000111010011110011100001100011000000110100011101011111110000 1";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00101101" ;
din <=
"001000010111110011101101101110100111111000110001100101000011011 1";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00101110" ;
din <=
"010111110010111100000010000001111111110111001001110010010100001 1";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00101111" ;
din <=
"000010000010010011101110100001011010001011001110000110101100100 0";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00110000" ;
din <=
"010000110101110010001000100011100110001001111110000000101001101 1";
```

```
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00110001" ;
din <=
"00000001001011101000010000111001111000000000001000110000000011010";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00110010" ;
din <=
"01011001001001100010010001010110100100011011100001000011011101110";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00110011" ;
din <=
"00111110001100011001010111010111010101100011010101000001000100011";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00110100" ;
din <=
"11001010010000110111110111110111111011010100010110000000101000011";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00110101" ;
din <=
"00110110000000000110010010011101011010110010111000000000001110100";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00110110" ;
din <=
"11000010110100001110100111111011011001100111010011010000010110001";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00110111" ;
din <=
"00110110011110001110101001011001011101100111110100011100011100100";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00111000" ;
din <=
"11011010010101111101011010111000011011100111100110000011101110101";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00111001" ;
din <=
"00000011011011010001101110000110101010110000101100101101101111100";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00111010" ;
din <=
"10011101000111111111000110001000000100011101110000100111111011111";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00111011" ;
din <=
"10111111010000000001111111100100100101011101010011111011011100111";
```

```
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00111100" ;
din <=
"10100111011001010100111111011111010111011001000111000000011001101";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00111101" ;
din <=
"00101110010000110101101001110000001011100000010000100101111111000";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00111110" ;
din <=
"00111000110111001010100101010110001000101101011011111000001100001";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "00111111" ;
din <=
"11000010110001111001010111111011111101011101111010011010101110101";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "01000000" ;
din <=
"11000010110001111001010111111011111101011101111010011010101110101";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "01000001" ;
din <=
"11000010110001111001010111111011111101011101111010011010101110101";
wait for 100 ns;

write <= '1'; addr(7 downto 0) <= "01000010" ;
din <=
"11000010110001111001010111111011111101011101111010011010101110101";
wait for 100 ns;

---------------------------
write <= '1'; addr(7 downto 0) <= (others => '1');
-- start
din <=
"00000000000000000000000000000000000000000000000000000000000000000";
wait for 100 ns;

write <= '0'; addr(7 downto 0) <= (others => '1');
din <=
"00000000000000000000000000000000000000000000000000000000000000000";
wait for 200 ns;

wait;

end process;
```

```
-- dummy signal
extin <= (others => '0');
dmask <= (others => '0');
end syn;
```

PARITH.VHD FOR MOBILE ROBOT NAVIGATION CONTROL

```
-- Modified by Sampath Kothandaraman

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity parith is

    port ( clk: in std_logic;
           rst: in std_logic;
           data_fromram: in std_logic_vector(63 downto 0);
-- data read from block ram
           start: in std_logic;
           addr: out std_logic_vector(7 downto 0);
           data2ram: out std_logic_vector(63 downto 0);
-- data to write to block ram
           we: out std_logic;                          -- write enable
           finish: out std_logic
         );

end parith;


architecture rtl of parith is
    --signal state_flag : std_logic_vector( 1 downto 0);
    signal state : std_logic_vector(3 downto 0);
    signal idx: std_logic_vector(7 downto 0);
    signal buf_inp, buf_biasABC, buf_biasDE, buf_wAB, buf_wCD, buf_wE:
std_logic_vector(63 downto 0);
    signal out1234: std_logic_vector(63 downto 0);
    signal flag_parith: std_logic;        -- tc,
    signal ip_addr_ctr, op_addr_ctr: std_logic_vector(7 downto 0);

    constant zeros : std_logic_vector(63 downto 8) := (others => '0');

      component robot
            generic ( x_width: NATURAL := 8;
                    b_width: NATURAL := 8;
                    w_width: NATURAL := 8;
                    y_width: NATURAL := 64
                  );
            port (  clk: in std_logic;
                        rst: in std_logic;
                        inp: in std_logic_vector(8*x_width-1 downto 0);
      -- 64; using 40
                        bias_ABC: in std_logic_vector(8*b_width-1
downto 0);  -- 64; using 56
                        bias_DE: in std_logic_vector(8*b_width-1 downto
0);   -- 64; using 24
                        w_AB: in std_logic_vector(8*w_width-1 downto
0);   -- 64; using 56
```

```
                        w_CD: in std_logic_vector(8*w_width-1 downto
0);   -- 64;
                        w_E: in std_logic_vector(8*w_width-1 downto 0);
      -- 64;
                        flag: in std_logic;
                        out1234: out std_logic_vector(y_width-1 downto
0)    -- 64;
                );
      end component;

begin

    -- retrieve data from block RAM and do the block31 opereation

    U0: robot port map ( clk => clk,
                  rst => rst,
                  inp => buf_inp,
                  bias_ABC => buf_biasABC,
                  bias_DE => buf_biasDE,
                  w_AB => buf_wAB,
                  w_CD => buf_wCD,
                  w_E => buf_wE,
                  flag => flag_parith,
                  out1234 => out1234
                  );

    process (clk, rst)
    begin
        if (rst = '1') then
          flag_parith <= '1';
          state <= "0000";
          finish <= '0';
          ip_addr_ctr <= "00000101";
-- initialize ip_addr_ctr = 5, the location of first input dataset
-- (in1, in2)
          op_addr_ctr <= "01000101";
-- initialize ip_addr_ctr = 69, the location of first output data out2
          idx <= (others => '0');
          we <= '0';
        elsif (clk = '1' and clk'event)  then
          if (state = "0000") then
            if (start = '1') then         -- state machine
                  flag_parith <= '1';
                  ip_addr_ctr <= "00000101";
-- 5 - start loc of inp data
                  op_addr_ctr <= "01000101";
-- 69 - start loc of output data
                  state <= "0001";
                  finish <= '0';
            end if;
          elsif (state = "0001") then -- issue addr for bias_ABC (0)
            idx <= "00000000";
            state <= "0010";
          elsif (state = "0010") then -- issue addr for bias_DE (1)
            idx <= "00000001";
```

```
            state <= "0011";
          elsif (state = "0011") then
-- get bias_ABC, issue addr for wts_AB (2)
            idx <= "00000010";
            buf_biasABC <= data_fromram;
            state <= "0100";
          elsif (state = "0100") then
-- get bias_DE value, issue addr for wts_CD (3)
            idx <= "00000011";
            buf_biasDE <= data_fromram;
            state <= "0101";
          elsif (state = "0101") then
-- get wts_AB values, issue addr for wts_E (4)
            idx <= "00000100";
            buf_wAB <= data_fromram;
            state <= "0110";
          elsif (state = "0110") then
-- get wts_CD values, issue addr for input dataset 1
            buf_wCD <= data_fromram;
            state <= "0111";
          elsif (state = "0111") then  -- get wts_E values,
            buf_wE <= data_fromram;
            state <= "1000";

-- ^^ loading biases and weights for a given BbNN and for a set of 64 -
-- data ^^

-- vv reading input data, computing output and writing to RAM vv

          elsif (state = "1000") then           -- get input dataset 1
or stop or pass this state
            if (ip_addr_ctr = "01000101") then
-- stop when ip_addr_ctr is > last ip_addr
                  finish <= '1';
                  state <= "0000";
            elsif (ip_addr_ctr = "00000101") then
                  buf_inp <= data_fromram; -- getting input dataset 1
                  state <= "1010";
            else
                  state <= "1001";
            end if;
          elsif (state = "1001") then
-- getting input dataset other than 1st set
            buf_inp <= data_fromram;
            state <= "1010";
          elsif (state = "1010") then
            state <= "1011";
          elsif (state = "1011") then
            flag_parith <= '0';
            state <= "1100";
          elsif (state = "1100") then
            state <= "1101";
          elsif (state = "1101") then
-- writing output of that particular dataset to RAM
            idx <= op_addr_ctr;
```

```
            ip_addr_ctr <= ip_addr_ctr + 1;
            we <= '1';
            state <= "1110";
            flag_parith <= '1';
          elsif (state = "1110") then
-- issue address for next dataset
            we <= '0';
            idx <= ip_addr_ctr;
            op_addr_ctr <= op_addr_ctr + 1;
            state <= "1000";
          else                          -- dummy
            finish <= '0';
          end if;
      end if;
    end process;

    addr <= idx;
    data2ram <= out1234;

end rtl;
```

ROBOT.VHD

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

entity robot is
      generic ( x_width: NATURAL := 8;
              b_width: NATURAL := 8;
              w_width: NATURAL := 8;
              y_width: NATURAL := 64
            );
      port ( clk: in std_logic;
             rst: in std_logic;
             inp: in std_logic_vector(8*x_width-1 downto 0);
      -- 64; using 40
             bias_ABC: in std_logic_vector(8*b_width-1 downto 0);
-- 64; using 56
             bias_DE: in std_logic_vector(8*b_width-1 downto 0);
-- 64; using 24
             w_AB: in std_logic_vector(8*w_width-1 downto 0);
-- 64; using 56
             w_CD: in std_logic_vector(8*w_width-1 downto 0);
-- 64; using 64
             w_E: in std_logic_vector(8*w_width-1 downto 0);
-- 64; using 24
             flag: in std_logic;
             out1234: out std_logic_vector(y_width-1 downto 0)
-- 64;
           );
end robot;

architecture structure_robot of robot is

      signal tmp_a_out, tmp_b_out, tmp_c_out, tmp_d_out, tmp_e_out:
std_logic_vector(y_width-1 downto 0);
      signal tc: std_logic;
      signal sig_a_inx2, sig_AE, sig_c_inx2, sig_d_inx2, sig_e_inx2:
std_logic_vector(x_width-1 downto 0);
      signal out_robot: std_logic_vector(4*x_width-1 downto 0);
      signal zeros: std_logic_vector(8*x_width-1 downto 4*x_width);

      component regout_block13
            generic ( x_width : NATURAL := 8;
                        w_width : NATURAL := 8;
                        b_width : NATURAL := 8;
                        y_width : NATURAL := 64
                      );
            port ( clk, rst, flag : in std_logic;
                  x1       : in std_logic_vector(x_width-1 downto 0);
                  w12, w13, w14 : in std_logic_vector(w_width-1 downto
0);
                  b2, b3, b4    : in std_logic_vector(b_width-1 downto
0);
```

```
                tc            : in std_logic;
                y2_y3_y4      : out std_logic_vector(y_width-1 downto
0)
               );
      end component;

      component regout_block22
            generic ( x_width : NATURAL := 8;
                        w_width : NATURAL := 8;
                        b_width : NATURAL := 8;
                        num     : NATURAL := 2;
                        y_width : NATURAL := 64
                      );
            port (      clk, rst, flag : in std_logic;
                  x1, x2 : in std_logic_vector(x_width-1 downto 0);
                  w13, w23, w14, w24 : in std_logic_vector(w_width-1
downto 0);
                  b3, b4 : in std_logic_vector(b_width-1 downto 0);
                  tc : in std_logic;
                  y3_y4 : out std_logic_vector(y_width-1 downto 0)
                 );
      end component;

      component regout_block31
            generic ( x_width : NATURAL := 8;
                        w_width : NATURAL := 8;
                        b_width : NATURAL := 8;
                        num     : NATURAL := 3;
                        y_width : NATURAL := 64
                      );
            port (      clk, rst, flag : in std_logic;
                  x1, x2, x3 : in std_logic_vector(x_width-1 downto 0);
                  w14, w24, w34 : in std_logic_vector(w_width-1 downto
0);
                  tc : in std_logic;
                  b4 : in std_logic_vector(x_width-1 downto 0);
                  y4 : out std_logic_vector(y_width-1 downto 0)
                 );
      end component;

begin


-- instantiating the basic blocks

      A_22: regout_block22
            port map ( clk => clk,
                       rst => rst,
                       flag => flag,
                       x1 => inp(x_width-1 downto 0),
                       x2 => sig_a_inx2,
                       w13 => w_AB(w_width-1 downto 0),
                       w23 => w_AB(2*w_width-1 downto w_width),
                       w14 => w_AB(3*w_width-1 downto 2*w_width),
                       w24 => w_AB(4*w_width-1 downto 3*w_width),
```

```
              b3 => bias_ABC(b_width-1 downto 0),
              b4 => bias_ABC(2*b_width-1 downto b_width),
              tc => tc,
              y3_y4 => tmp_a_out
            );

B_13: regout_block13
      port map ( clk => clk,
              rst => rst,
              flag => flag,
              x1 => inp(2*x_width-1 downto x_width),
              w12 => w_AB(5*w_width-1 downto 4*w_width),
              w13 => w_AB(6*w_width-1 downto 5*w_width),
              w14 => w_AB(7*w_width-1 downto 6*w_width),
              b2 => bias_ABC(3*b_width-1 downto 2*b_width),
              b3 => bias_ABC(4*b_width-1 downto 3*b_width),
              b4 => bias_ABC(5*b_width-1 downto 4*b_width),
              tc => tc,
              y2_y3_y4 => tmp_b_out
              );

C_22: regout_block22
      port map ( clk => clk,
              rst => rst,
              flag => flag,
              x1 => inp(3*x_width-1 downto 2*x_width),
              x2 => sig_c_inx2,
              w13 => w_CD(w_width-1 downto 0),
              w23 => w_CD(2*w_width-1 downto w_width),
              w14 => w_CD(3*w_width-1 downto 2*w_width),
              w24 => w_CD(4*w_width-1 downto 3*w_width),
              b3 => bias_ABC(6*b_width-1 downto 5*w_width),
              b4 => bias_ABC(7*b_width-1 downto 6*b_width),
              tc => tc,
              y3_y4 => tmp_c_out
            );

D_22: regout_block22
      port map ( clk => clk,
              rst => rst,
              flag => flag,
              x1 => inp(4*x_width-1 downto 3*x_width),
              x2 => sig_d_inx2,
              w13 => w_CD(5*w_width-1 downto 4*w_width),
              w23 => w_CD(6*w_width-1 downto 5*w_width),
              w14 => w_CD(7*w_width-1 downto 6*w_width),
              w24 => w_CD(8*w_width-1 downto 7*w_width),
              b3 => bias_DE(b_width-1 downto 0),
              b4 => bias_DE(2*b_width-1 downto b_width),
              tc => tc,
              y3_y4 => tmp_d_out
            );

E_31: regout_block31
      port map ( clk => clk,
```

```
                    rst => rst,
                    flag => flag,
                    x1 => inp(5*x_width-1 downto 4*x_width),
                    x2 => sig_e_inx2,
                    x3 => sig_AE,
                    w14 => w_E(w_width-1 downto 0),
                    w24 => w_E(2*w_width-1 downto w_width),
                    w34 => w_E(3*w_width-1 downto 2*w_width),
                    b4 => bias_DE(b_width-1 downto 0),
                          tc => tc,
                    y4 => tmp_e_out
                  );

      tc <= '0';
      zeros <= "00000000000000000000000000000000";  -- 32
      out1234(8*x_width-1 downto 4*x_width) <= zeros;
      out1234(4*x_width-1 downto 0) <= out_robot;

-- processes

      -- flag = 1 means, reset all block outputs to 0; flag = 0 means,
work with feedbck data


      process (rst, flag, tmp_a_out, tmp_b_out, tmp_c_out, tmp_d_out)
-- clk removed from sensitivity list: MUX on flag, no FF
      begin
            if (rst = '1' or flag = '1') then
                  sig_a_inx2 <= (others => '0');
                  sig_AE <= (others => '0');
                  sig_c_inx2 <= (others => '0');
                  sig_d_inx2 <= (others => '0');
                  sig_e_inx2 <= (others => '0');
            elsif (flag = '0') then
                  sig_a_inx2 <= tmp_b_out(x_width-1 downto 0);
-- y4 output from block B_13
                  sig_AE <= tmp_a_out(2*x_width-1 downto x_width);
-- y3 output from block A_22
                  sig_c_inx2 <= tmp_b_out(3*x_width-1 downto
2*x_width); -- y2 output from block B_13
                  sig_d_inx2 <= tmp_c_out(2*x_width-1 downto x_width);
-- y3 output from block C_22
                  sig_e_inx2 <= tmp_d_out(2*x_width-1 downto x_width);
-- y3 output from block D_22
            end if;
      end process;

      process(tmp_b_out, tmp_c_out, tmp_d_out, tmp_e_out)
      begin
            out_robot <= tmp_b_out(x_width-1 downto 0) &
tmp_c_out(x_width-1 downto 0) & tmp_d_out(x_width-1 downto 0) &
tmp_e_out(x_width-1 downto 0);
      end process;

end structure_robot;
```

VITA

Sampath Kothandaraman was born in Madurai, India. He grew up in Trivandrum, Kerala, and then moved to Madras, for the last two years of his high-school education. He then went to the College of Engineering, Trivandrum, and obtained his Bachelor of Technology degree in Electrical and Electronics Engineering from the University of Kerala, in 2001. Since August 2001, he has attended graduate school at the University of Tennessee, Knoxville and plans to graduate with a Master's degree in Electrical Engineering in August 2004. He has been a teaching assistant with the Department of Electrical and Computer Engineering and a recipient of the Analog Devices Inc. Fellowship for 2003.