Spring 2010

FPGA implementation of PSO algorithm and neural networks

Parviz Palangpour

Follow this and additional works at: http://scholarsmine.mst.edu/masters_theses

Recommended Citation
FPGA IMPLEMENTATION OF PSO ALGORITHM AND NEURAL NETWORKS

by

PARVIZ MICHAEL PALANGPOUR

A THESIS

Presented to the Faculty of the Graduate School of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE IN COMPUTER ENGINEERING

2010

Approved by

Ganesh Kumar Venayagamoorthy, Advisor
Waleed Al-Assadi
Maciej Zawodniok
ABSTRACT

This thesis describes the Field Programmable Gate Array (FPGA) implementations of two powerful techniques of Computational Intelligence (CI), the Particle Swarm Optimization algorithm (PSO) and the Neural Network (NN).

Particle Swarm Optimization (PSO) is a popular population-based optimization algorithm. While PSO has been shown to perform well in a large variety of problems, PSO is typically implemented in software. Population-based optimization algorithms such as PSO are well suited for execution in parallel stages. This allows PSO to be implemented directly in hardware and achieve much faster execution times than possible in software. In this thesis, a pipelined architecture for hardware PSO implementation is presented. Benchmark functions solved by software and FPGA hardware PSO implementations are compared.

NNs are inherently parallel, with each layer of neurons processing incoming data independently of each other. While general purpose processors have reached impressive processing speeds, they still cannot fully exploit this inherent parallelism due to their sequential architecture. In order to achieve the high neural network throughput needed for real-time applications, a custom hardware design is needed. In this thesis, a digital implementation of an NN is developed for FPGA implementation.

The hardware PSO implementation is designed using only VHDL, while the NN hardware implementation is designed using Xilinx System Generator. Both designs are synthesized using Xilinx ISE and implemented on the Xilinx Virtex-II Pro FPGA Development Kit.
ACKNOWLEDGMENT

I am deeply grateful to my advisor Dr. Ganesh Kumar Venayagamoorthy, who has provided me with guidance, knowledge and financial support throughout the research and preparation of this thesis.

I would also like to thank Dr. Waleed Al-Assadi and Dr. Maciej Zawodniok for their assistance and serving on my committee. I would also like to thank Dr. Scott C. Smith for his advice and comments towards my work.

I would like acknowledge funding from the following grants:

1. GAANN: Advanced Computational Techniques and Real-Time Simulation Studies for the Next Generation Energy Systems

2. NSF CAREER: Scalable Learning and Adaption with Intelligent Techniques and Neural Networks for Reconfiguration and Survivability of Complex Systems (ECCS # 0348221)

3. EFRI-COPN: Neuroscience and Neural Networks for Engineering the Future Intelligent Electric Power Grid (EFRI #0836017)

Most importantly, I would like to thank my parents for their unconditional support and encouragement towards reaching my goals.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>SECTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 INTRODUCTION</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 BACKGROUND</td>
<td>1</td>
</tr>
<tr>
<td>1.2 THESIS OBJECTIVE</td>
<td>4</td>
</tr>
<tr>
<td>1.3 THESIS OVERVIEW</td>
<td>4</td>
</tr>
<tr>
<td>1.4 CONTRIBUTIONS OF THIS THESIS</td>
<td>5</td>
</tr>
<tr>
<td>1.5 RESEARCH PUBLICATIONS</td>
<td>5</td>
</tr>
<tr>
<td>1.6 SUMMARY</td>
<td>5</td>
</tr>
<tr>
<td><strong>2 IMPLEMENTATIONS OF PSO ALGORITHM</strong></td>
<td>6</td>
</tr>
<tr>
<td>2.1 INTRODUCTION</td>
<td>6</td>
</tr>
<tr>
<td>2.2 PARTICLE SWARM OPTIMIZATION</td>
<td>7</td>
</tr>
<tr>
<td>2.3 PSO HARDWARE IMPLEMENTATION</td>
<td>9</td>
</tr>
<tr>
<td>2.3.1 The Hardware Velocity Update</td>
<td>10</td>
</tr>
<tr>
<td>2.3.2 Random Number Generation</td>
<td>11</td>
</tr>
<tr>
<td>2.3.3 Control Module</td>
<td>12</td>
</tr>
<tr>
<td>2.4 RESULTS</td>
<td>13</td>
</tr>
<tr>
<td><strong>3 IMPLEMENTATION OF NEURAL NETWORK</strong></td>
<td>19</td>
</tr>
<tr>
<td>3.1 INTRODUCTION</td>
<td>19</td>
</tr>
<tr>
<td>3.2 DESIGN APPROACH</td>
<td>20</td>
</tr>
<tr>
<td>3.3 NN HARDWARE IMPLEMENTATION</td>
<td>21</td>
</tr>
</tbody>
</table>
# LIST OF ILLUSTRATIONS

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>PSO hardware implementation with 6-stage pipeline.</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>An example of a linear-feedback shift register.</td>
<td>11</td>
</tr>
<tr>
<td>2.3</td>
<td>The structure of the neighborhood-of-four PRNG.</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>The hardware PSO execution flowchart.</td>
<td>14</td>
</tr>
<tr>
<td>2.5</td>
<td>A two-dimensional view of the sphere benchmark problem fitness surface.</td>
<td>15</td>
</tr>
<tr>
<td>2.6</td>
<td>A two-dimensional view of the Rosenbrock benchmark problem fitness surface.</td>
<td>16</td>
</tr>
<tr>
<td>2.7</td>
<td>The hardware PSO compared with a software PSO for the sphere function of 10 dimensions.</td>
<td>17</td>
</tr>
<tr>
<td>2.8</td>
<td>The hardware PSO compared with a software PSO for the Rosenbrock function of 10 dimensions.</td>
<td>18</td>
</tr>
<tr>
<td>3.1</td>
<td>An NN with an input, hidden and output layer.</td>
<td>19</td>
</tr>
<tr>
<td>3.2</td>
<td>A diagram of the neuron model.</td>
<td>22</td>
</tr>
<tr>
<td>3.3</td>
<td>MAC hardware subsystem.</td>
<td>22</td>
</tr>
<tr>
<td>3.4</td>
<td>LUTAF approximation.</td>
<td>25</td>
</tr>
<tr>
<td>3.5</td>
<td>LUTAF approximation subsystem.</td>
<td>25</td>
</tr>
<tr>
<td>3.6</td>
<td>The complete hardware neuron.</td>
<td>26</td>
</tr>
<tr>
<td>3.7</td>
<td>The LCB finite state machine (2 × 3 Layer).</td>
<td>27</td>
</tr>
<tr>
<td>3.8</td>
<td>A LCB controlling three hardware neurons (2 × 3 Layer).</td>
<td>28</td>
</tr>
<tr>
<td>3.9</td>
<td>A hardware neuron being controlled by a LCB.</td>
<td>29</td>
</tr>
<tr>
<td>3.10</td>
<td>A FPGA-based NN (of size 1 × 3 × 1).</td>
<td>29</td>
</tr>
<tr>
<td>3.11</td>
<td>The trained NN for approximating a sin function.</td>
<td>30</td>
</tr>
<tr>
<td>3.12</td>
<td>The output of the Simulink simulation of the FPGA-based NN.</td>
<td>32</td>
</tr>
<tr>
<td>3.13</td>
<td>Comparing the output of hidden neurons of the FPGA-based NN and Matlab-based NN.</td>
<td>33</td>
</tr>
<tr>
<td>3.14</td>
<td>Comparing the output of the FPGA-based NN and Matlab-based NN.</td>
<td>34</td>
</tr>
</tbody>
</table>
3.15 The hardware resources used for the $1 \times 10 \times 1$ NN on the Xilinx Virtex-II Pro FPGA.
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 The values for the LUT in each PRNG cell</td>
<td>13</td>
</tr>
<tr>
<td>2.2 Achieved fitness after 1000 iterations for benchmark problems</td>
<td>17</td>
</tr>
<tr>
<td>2.3 Execution time for software and hardware PSO implementations</td>
<td>17</td>
</tr>
<tr>
<td>2.4 Hardware PSO logic requirements for the benchmark problems</td>
<td>18</td>
</tr>
<tr>
<td>3.1 The values for the hyperbolic tangent activation Function LUT where $x \in [L, U]$</td>
<td>24</td>
</tr>
</tbody>
</table>
1 INTRODUCTION

1.1 BACKGROUND

Computational Intelligence (CI) is a sub-branch of artificial intelligence. CI is the study of adaptive mechanisms to enable or facilitate intelligent behavior in complex and changing environments [1]. Many of the CI paradigms have mechanisms that exhibit the ability to adapt to new situations, to generalize, abstract, discover and associate. The main CI paradigms are artificial Neural Networks (NNs), evolutionary computing, Swarm Intelligence (SI) and fuzzy systems. Each of these paradigms has origins in some biological system. For instance, NNs are models of biological neural systems while SI models the social behavior of organisms that live in swarms or colonies.

The most common way to implement CI techniques is to write software that can be executed on a standard digital processor. One reason why this approach is the most common is because of the ease of development as developers can take advantage of the enormous number of existing software libraries. By building applications on top of existing software libraries and operating systems, the developer can implement the application without having to be concerned with the operation of the underlying digital hardware. Furthermore, there are a large number of software programming languages available, C, C++, Java, etc; for each of these languages exists multiple mature tools exist that make implementing, simulating and optimizing software applications much easier.

There are also a variety of digital processors that can be used to execute CI software, microcontrollers, digital signal processors (DSPs) and general purpose microprocessors. Microcontrollers offer the least amount of processing power but are available at very low prices and consume very little power. DSPs are designed specifically for digital signal processing applications while general purpose microprocessors offer the highest performance and flexibility. When software is compiled for execution on a specific processor, the compiler essentially converts the software into series of small instructions. These instructions are then executed in a sequential nature on the processor. However, only the simplest of processors execute instructions in a
strictly sequential manner, more advanced processors utilize different techniques to achieve instruction-level parallelism that allow the execution of multiple instructions simultaneously.

While the trend in high-performance microprocessor and DSP design is to incorporate techniques to achieve instruction-level parallelism, the processor and software compiler are left to interpret and exploit any such opportunities for parallelism. As processors are designed for general purpose computing, they cannot be designed to take advantage of the underlying parallelism in each application. Instead of executing the software on a processor, a custom digital system can be designed specifically for a given application. This custom digital system can be designed to take advantage of any level of parallelism that exists in the application being implemented. Since the custom digital system can perform a much higher degree of parallelism, a custom digital system can outperform any software-based implementation in terms of execution time. However, it is much more difficult to design a complete digital system to implement an algorithm rather than simply implement the algorithm in software.

Field-Programmable Gate Arrays (FPGAs) are essentially programmable integrated circuits. FPGAs can be reprogrammed to implement arbitrary logic functionality without having to endure the long and expensive design process required for Application Specific Integrated Circuits (ASICs). While FPGAs don’t incur the high setup costs of ASIC production, FPGAs are slower, consume more power and consume more area than their ASIC counterparts. However, the flexible architecture and lower cost of entry makes FPGAs ideal for prototyping new designs; this makes FPGAs a popular platform for research.

Specific FPGA design details vary from vendor to vendor, but generally each FPGA contains a large number of Lookup Tables (LUTs) and interconnecting logic that can be programmed. The LUTs are programmed to reflect the specific logic functions they need to perform and the interconnecting logic is programmed to properly implement the connections between the LUTs. It should be noted that this is only an abstract view of an FPGAs internals; most FPGAs contain not only LUTs, but registers, multiplexers, distributed and block memory, and dedicated circuitry for fast adders and multipliers. Recently, FPGAs have become essential components in implementing high performance digital signal processing (DSP) systems. The memory
bandwidth of a modern FPGA far exceeds that of a microprocessor or DSP processor running at clock rates two to ten times that of the FPGA. Coupled with the capability of implementing highly parallel arithmetic architectures, FPGAs are ideally suited for high-performance custom data path processors.

FPGAs have two main types of resources that are used to implement the logic for the intended application, generic programmable logic blocks and dedicated (or “embedded”) circuitry that perform fixed functions. The generic logic blocks can be used to build any arbitrary logic function. The embedded hardware blocks are essentially fixed logic functions that are available to the designer at no cost of logic blocks. The embedded hardware operations are ASIC implementations, so they are faster and require less area than the equivalent function built using logic blocks. One common embedded hardware operation is the multiply operation, which typically requires a large number of logic blocks to implement. By selecting a specific FPGA product with the proper embedded resources for the given application, some of the performance disadvantages of FPGA implementations can be reduced.

Several hardware implementations of one SI-based algorithm, Particle Swarm Optimization (PSO), has been reported in literature. Reynolds et al. have implemented a hardware version of PSO for inverting a very large neural network [2]. One Xilinx XC2V6000 was used to execute the PSO algorithm while another was used for computing the fitness. The details of the hardware PSO architecture are not reported. A multi-swarm PSO architecture for blind adaption of array antennas was proposed by Kokai et al. [3]. Each swarm optimizes a single architecture and executes in parallel with the other swarms. The authors have not described the hardware architecture in detail or provided any performance measurements.

Farmahini-Farahani et al. have implemented PSO within a system on a programmable chip framework [4]. The authors utilized a hardware implementation of the discrete version of PSO. A soft-core Altera NIOS II embedded processor was used to compute the fitness function and implemented on a Altera Stratix 1S10ES Development Kit. Performance was sacrificed in exchange for flexibility by implementing the fitness function in software.

There has been a lot of interest in hardware implementations of NNs and many different approaches have been reported in literature [5] [6] [7] [8] [9]. Several authors
have chose to design a neuroprocessor, a processor specifically developed for executing NN software [10] [11]. Danese et al. developed the NeuriCam Totem PCI board which contains two custom VLSI ICs, the NC3001. Each NC3001 implements several neurons and contains dedicated on-chip memory for storing each neuron’s weights. The speed of the processor depends on the size of the NN being implemented; a 16x16x1 NN can be evaluated in 2 \( \mu s \).

One large area of interest in hardware NN implementations is how to implement the activation function efficiently. This is a difficult problem because the non-linear activation functions cannot be efficiently implemented directly in hardware. As a result, hardware NN implementations rely on circuits that approximate the activation functions. One of the most common methods is to implement a piece-wise approximation of the activation function using LUTs. This allows the designer to easily select the desired balance between precision and circuit size. Recently, authors have shown that a genetic algorithm, another component of CI, can be used to generate an optimal spline-based approximation function [12].

1.2 THESIS OBJECTIVE

The PSO algorithm is a SI technique that has been used in a number of optimization problems. NNs are another important CI paradigm which can be used to approximate arbitrary functions. Both of these techniques are primarily implemented in software because developing hardware implementations is more difficult. However, to achieve the highest performance hardware implementations of the techniques are required. The focus of this thesis is to present the high-performance hardware implementations of these techniques on a FPGA platform.

1.3 THESIS OVERVIEW

This thesis is organized into four sections. Section 2 introduces the PSO algorithm and the developed FPGA hardware implementation. Simulation results are provided for two benchmark problems and the hardware PSO is compared to a software implementation of PSO. Section 3 introduces NNs and the developed FPGA hardware implementation. Simulation results are provided and the FPGA NN is
compared to a software NN. Section 4 concludes the thesis and provides possible directions for improvement and future research.

1.4 CONTRIBUTIONS OF THIS THESIS

In this thesis, the follow contributions have been made:

- The PSO algorithm has been implemented directly as a digital hardware design
- The hardware PSO design does not require a processor and can execute independently of any other digital hardware
- The hardware PSO design is implemented on the Xilinx Virtex-II Pro FPGA
- A high-level model that can be used to build hardware NNs in has been developed.
- The hardware NN model is implemented on the Xilinx Virtex-II Pro FPGA

1.5 RESEARCH PUBLICATIONS

A Computer DESign (CDES) conference paper has been published based on the hardware PSO implementation presented in this thesis [13]. In addition, a submission to the International Joint Conference on Neural Networks (IJCNN) 2010 will be prepared based on the hardware NN implementation presented [14].

1.6 SUMMARY

In this Section an introduction to CI and FPGAs has been presented. Also, the objectives of this work and an overview of the thesis has be presented.
2 IMPLEMENTATIONS OF PSO ALGORITHM

2.1 INTRODUCTION

Adaptive systems have become a large area of interest since many systems operate in changing, unpredictable environments. Evolutionary Algorithms (EAs) are well suited for adapting the behavior of many adaptive systems because of their simplicity; EAs only require a fitness function to provide a measure of the systems behavior. Many different variations of EAs for adapting system behavior have been extensively explored. In principle, all EAs are population-based optimization algorithms. The population consists of candidate solutions to the problem being studied and during each iteration of the algorithm a series of operators are applied to the population. After the population has been passed through the operators, the candidate solutions are evaluated and given a level of fitness that represents their degree of performance for the problem being studied. Each of the operators are based on evolution and play a role in combining and randomly modifying portions of the population. As the fitness of candidate solutions play a role in what solutions are selected to combine and modify, the population as a whole improves over time. PSO is another population-based algorithm which begins with a population of potential solutions and continually evolves the solutions until they reach a desired level of fitness [15]. While EAs and PSO are similar, PSO requires fewer operations. This is important for real-time applications where speed is critical.

PSO is a swarm intelligence based optimization algorithm that has been shown to perform very well for a large number of applications. While PSO has been applied in a large number of applications, PSO is typically executed in software. Recently, there has been interest in using PSO for real-time applications [16] [17]. However, in order to meet the time constraints of some real-time applications, PSO must be executed directly in hardware.
2.2 PARTICLE SWARM OPTIMIZATION

The PSO algorithm was developed by Kennedy and Eberhart and is based on the social behavior of bird flocking [15]. Each particle in the population has a position vector which represents a potential solution to the problem. The particles are initialized to random positions throughout the search space and for each iteration of the algorithm a velocity vector is computed and used to update each particle’s position. Each particle’s velocity is influenced by the particle’s own experience as well as the experience of its neighbors. There are two basic variants of PSO, local and global. In this study the more common global version of the PSO algorithm is applied.

The population consists of $N$ particles. For each iteration, a cost function $f$ is used to measure the fitness of each particle $i$ in the population. The position of each particle $i$ is then updated, which is influenced by three terms, the particle’s velocity from the last iteration, the difference between the particles known best position and the particle’s current position, and the difference between the swarm’s best known position and the particle’s current position. The latter two terms are each multiplied by a uniform random number in [0,1] to randomly vary the influence of each term, as well as an acceleration coefficient to scale and balance the influence of each term. The best position each particle attained is stored in the vector $p_i$, also known as $p_{best}$, while the best position attained by any particle in the population is stored in the vector $p_g$, also known as $g_{best}$. The velocity vector $v_i$ for each particle is then updated:

$$v_{i}^{t+1} = w \cdot v_{i}^{t} + c_1 r_1 \cdot (p_{i}^{t} - x_{i}^{t}) + c_2 r_2 \cdot (p_{g}^{t} - x_{i}^{t})$$  \hspace{1cm} (2.1)

where $w$, $c_1$ and $c_2$ are positive and $r_1$ and $r_2$ are uniformly distributed random numbers in [0,1]. The inertia coefficient, $w$ is used to keep the particles moving in the same direction they have been traveling. The value for $w$ is typically in [0, 1]. The term $c_1$ is called the cognitive acceleration term and $c_2$ is called the social acceleration term. These two values balance the influence between the particles own best performance and that of the population. The velocity is constrained between the parameters $V_{min}$ and $V_{max}$ to limit the maximum change in position in Equation
The position of each particle is then updated using the new velocities in Equation 2.3.

\[ x_{i}^{t+1} = x_{i}^{t} + v_{i}^{t+1} \]  

The position in each dimension is limited between the parameters \( X_{min} \) and \( X_{max} \) in Equation 2.4.

\[ x_{i}^{t+1} = \begin{cases} x_{Max} & \text{if } x_{i}^{t+1} > X_{max} \\ x_{Min} & \text{if } x_{i}^{t+1} < X_{min} \\ x_{i}^{t+1} & \text{else} \end{cases} \]  

The psuedocode for PSO is listed in Algorithm 1.

---

**Algorithm 1 PSO**

1. Initialize the positions, velocities, \( p_{best} \) and \( g_{best} \) values
2. repeat
   3. for \( i = 1 \) to NUM_PARTICLES do
      4. if \( f(x_{i}) \leq f(p_{i}) \) then
         5. \( p_{i} \leftarrow x_{i} \)
      6. if \( f(x_{i}) \leq f(p_{g}) \) then
         7. \( p_{g} \leftarrow x_{i} \)
      8. end if
   9. end if
10. for \( j = 1 \) to NUM_DIMENSIONS do
   11. \( v_{ij}^{t+1} \leftarrow w \cdot v_{ij}^{t} + c_{1}r_{1} \cdot (x_{ij}^{t} - p_{ij}^{t}) + c_{2}r_{2} \cdot (x_{ij}^{t} - p_{gj}^{t}) \)
   12. \( v_{ij}^{t+1} \in (V_{min}, V_{max}) \)
   13. \( x_{ij}^{t+1} \leftarrow x_{ij}^{t} + v_{ij}^{t+1} \)
   14. \( x_{ij}^{t+1} \in (X_{min}, X_{max}) \)
15. end for
16. end for
17. until maximum iterations reached
2.3 PSO HARDWARE IMPLEMENTATION

Software implementations of PSO often use floating-point values. However, floating-point operations typically require several times the number of logic resources for a similar fixed-point operation. In addition, it is common for FPGAs to include a number of embedded multipliers which can be used to perform fixed-point multiplications without using any of the FPGAs programmable logic. For these reasons, the hardware PSO implementation uses the fixed-point representation for all values.

For hardware implementation, the PSO algorithm is decomposed into five operations that are performed on each particle: evaluate the fitness, update the particle’s best position, update the global best position, update the velocity and update the position. Each of these five operations are implemented in a separate hardware module. It should be noted that the constraints in Equations 2.2 and 2.4 are not directly implemented; the results of the fixed-point arithmetic operations for Equations 2.1 and 2.3 are set to saturate which will indirectly constrain the values based on their width. Since updating the \( p_{\text{best}} \) and \( g_{\text{best}} \) can be performed in parallel, the five operations can be organized in a 6-stage pipeline, including the initial fetch and final write stages. The flow for an execution of a single particle is described as follows.

In the first stage, the position for particle \( i \), \( x_i \) is fetched from memory. Then in the second stage, the fitness module computes the fitness, \( f(x_i) \) based on the position of particle \( i \). The current \( p_{\text{best}} \) values, \( p_i \) and \( f(p_i) \) are also fetched from \( p_{\text{best}} \) memory. In the third stage, the \( g_{\text{best}} \) and \( p_{\text{best}} \) are updated. Both the update \( g_{\text{best}} \) and \( p_{\text{best}} \) modules are passed \( x_i \) and \( f(x_i) \). In addition, the update \( p_{\text{best}} \) module is passed \( p_i \) and \( f(p_i) \) while the update \( g_{\text{best}} \) module is passed \( p_g \) and \( f(p_g) \). Each module selects the lowest fitness and associated positions for their output, which are the now updated \( g_{\text{best}} \) and \( p_{\text{best}} \). In addition, the old velocity, \( v_i \) is fetched from the velocity memory. Now in the fourth stage of the pipeline, the new \( p_i \) and \( f(p_i) \) are stored in the \( p_{\text{best}} \) memory. The update velocity module uses \( v_i \), \( x_i \), \( p_i \), \( p_g \), \( r_1 \) and \( r_2 \) to compute the new velocity, \( v_i \). In the fifth stage, the update position module uses \( x_i \) and \( v_i \) to compute the new position, \( x_i \). The new velocity, \( v_i \) is stored in the velocity memory. In the final stage, the new position \( x_i \) is stored in the position memory. The hardware modules are shown in Figure 2.1 The position, \( p_{\text{best}} \) and velocity memory
are simply represented as 5-element registers; the hardware which stores the values could as simple as registers or as large as dynamic RAM.

![PSO Hardware Implementation](image)

**Figure 2.1.** PSO hardware implementation with 6-stage pipeline.

### 2.3.1 The Hardware Velocity Update.

In PSO, the velocity update equation involves the largest number of arithmetic operations. As shown in Equation 2.1, there are five multiplications, two additions and two subtractions. In hardware, multiplications require a large amount of logic and are usually to be avoided. Since the inertia, $w$ is typically set to 0.8, the first term of the velocity update can be simplified in two different ways. The first is to replace the term with an arithmetic shift to the right of $v_i$. This effectively changes the inertia to 0.5 and eliminates the multiplication needed. An alternative is to remove the inertia entirely and substitute $v_i$ for the first term.
The cognitive and social acceleration coefficients, \( c_1 \) and \( c_2 \) are typically set to 2.0. Performing arithmetic left shifts on \( r_1 \) and \( r_2 \) would effectively multiply each value by 2. However, since the values for \( c_1 r_1 \) and \( c_2 r_2 \) are just uniform random numbers in \([0, 2]\), the fixed-point pseudo random numbers can just be extended to fulfill the range by incorporating an additional random bit.

2.3.2 Random Number Generation. Two random numbers are needed for each velocity update. This means \( 2 \times p \times i \) random numbers are required for a PSO run of \( p \) particles and \( i \) iterations. In addition, some variants of PSO require an additional random value to vary the inertia [18]. While PSO is still able to find solutions in the absence of the random influence, it is not guaranteed PSO will converge as fast or with as high quality of solutions [2] [19]. Pseudo random numbers are generated in hardware using Pseudo Random Number Generators (PRNGs). Typically, linear feedback shift registers (LFSR) and cellular automata (CA) based PRNGs are used [20]. LFSRs are simpler to implement and are used in most hardware implementations. A LFSR is shown in Figure 2.2 the left-most bit is computed based on the previous value generated and the bits in the register are shifted to the right. However, CA-based PRNGS have been shown to offer better statistical properties. In this work, a neighborhood-of-four CA-based PRNG is used [21]. The neighborhood-of-four CA-based PRNG is selected for this work because it offers efficiency in FPGA hardware implementation as well as good statistical properties. The PRNG is organized into a grid of 8x8 cells, where each cell represents a register storing one bit and each cell receives the value from the cell above, to the left, to the right and below.

![Figure 2.2. An example of a linear-feedback shift register.](image-url)
The 8x8 PRNG structure is shown in Figure 2.3. The new value of each cell is found using a 4-input LUT, which is shown in Table 2.1. This implementation is efficient for FPGA implementation because the FPGA utilized in this work, the Xilinx Virtex-II Pro, uses 4-input LUTs to implement logic functions. Therefore, implementing the neighborhood-of-four CA-based PRNG only requires one LUT and D-type flip-flop per PRNG bit. To extract the generated number from the PRNG, the bit of each cell is concatenated together into a bit-string. As there are 64 cells, the PRNG produces a pseudorandom 64-bit string each cycle.

![Figure 2.3. The structure of the neighborhood-of-four PRNG.](image)

### 2.3.3 Control Module.

The control module is used to initialize the memory and generate the control signals for the modules. Upon reset the control module enters an *Init* state which is used to initialize the counters to their respective starting states. The control module then enters an *Init-PRNG* state to initialize the
Table 2.1. The values for the LUT in each PRNG cell.

<table>
<thead>
<tr>
<th>$x_0x_1x_2x_3$</th>
<th>$f$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>1</td>
</tr>
<tr>
<td>0011</td>
<td>1</td>
</tr>
<tr>
<td>0100</td>
<td>0</td>
</tr>
<tr>
<td>0101</td>
<td>0</td>
</tr>
<tr>
<td>0110</td>
<td>1</td>
</tr>
<tr>
<td>0111</td>
<td>0</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
</tr>
<tr>
<td>1001</td>
<td>1</td>
</tr>
<tr>
<td>1010</td>
<td>0</td>
</tr>
<tr>
<td>1011</td>
<td>1</td>
</tr>
<tr>
<td>1100</td>
<td>1</td>
</tr>
<tr>
<td>1101</td>
<td>0</td>
</tr>
<tr>
<td>1110</td>
<td>0</td>
</tr>
<tr>
<td>1111</td>
<td>0</td>
</tr>
</tbody>
</table>

PRNG. The *Init-Particles* state cycles through the particles and sets each particle's initial position, velocity and *pbest* position to a random value from the PRNG. The *gbest* position is initialized in the same manner. The next state, *Init-Pipeline* is used to prepare the module inputs. The *Execute* state is responsible for shifting each particle through the pipeline and properly passing the modules the correct particles' information from memory while storing each particle's new values as they are updated. Upon either reaching a defined fitness or number of iterations, the *Halt* state is entered and execution halts. The hardware PSO flowchart is shown in Figure 2.4.

2.4 RESULTS

All of the modules in the hardware PSO design have been designed in VHDL. The hardware PSO design has been simulated and implemented on the Xilinx Virtex-II Pro development platform [22]. In order to assess the performance of the hardware PSO implementation, the hardware implementation is compared to a software implementation developed in Matlab. The Matlab PSO implementation is executed on
Figure 2.4. The hardware PSO execution flowchart.

a 2.16 GHz Intel Core 2 Duo-based PC with 4 GB RAM. First, the performance of the two implementations is compared with respect to the lowest fitness that the implementation of able to achieve for the benchmark problems. Then the execution speed of the two implementations is compared. Finally, the logic requirements for the hardware implementation are discussed.

Two well-known benchmark optimization problems [23] have been selected for comparing the two implementations. The first benchmark problem is the sphere function:

$$f(x) = \sum_{i=1}^{n} x_i^2$$  

(2.5)

The second benchmark problem is the Rosenbrock function:

$$f(x) = \sum_{i=1}^{n-1} 100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2$$  

(2.6)

The surface of the sphere and Rosenbrock fitness functions are shown in Figures 2.5 and 2.6 respectively. The sphere function is a simple unimodal function that is
typically used to test local optimizers. The Rosenbrock function is multimodal for \( n \) of 4 and higher and is more difficult to optimize. These functions are often used to assess the performance of EAs.

![Two-dimensional view of the sphere benchmark problem fitness surface.](image)

**Figure 2.5.** A two-dimensional view of the sphere benchmark problem fitness surface.

Both the hardware and software PSO implementations are executed for 1000 iterations, with 20 particles. The achieved fitness for the hardware and software implementations of PSO on the benchmark problems is listed in Table 2.2. The execution time for the software and hardware PSO implementations iterations is listed in Table 2.3. A comparison of the fitness with respect to iteration, between the hardware and software PSO for the 10-dimensional sphere function is shown in Figure 2.7. In addition, a comparison of the fitness with respect to iteration, between the hardware and software PSO for the 10-dimensional Rosenbrock function is shown in Figure 2.8. The total execution time for the hardware PSO design can be computed as follows. After the sixth clock cycle the first particle has finished its first iteration of PSO. All particles must pass through the hardware PSO pipeline for each full iteration of PSO to be completed. Therefore, in an additional nineteen clock cycles, the 20th particle has finished, completing the first iteration of PSO. As the throughput is one
Figure 2.6. A two-dimensional view of the Rosenbrock benchmark problem fitness surface.

particle per clock cycle and each particle must pass through 1000 times, the execution time is computed as:

$$T_{\text{total}} = \frac{1}{f_{\text{clk}}} (5 + 20 \times 1000) \quad (2.7)$$

where $f_{\text{clk}}$ is the clock frequency of the PSO hardware. However, two factors can restrict the maximum clock frequency. The more mathematical operations the fitness function requires, the more time is required for the fitness module to compute the fitness. This means it will require more time to compute the fitness for the Rosenbrock function than the sphere function. In addition, as the number of dimensions increase, the number of mathematical operations required to compute the fitness increases. The utilization for Xilinx Virtex-II Pro FPGA and maximum clock frequencies are shown in Table 2.4. Since the fitness function is implemented in hardware, as the complexity of the fitness function increases, so does the number of LUTs required to implement the function in the FPGA.
Table 2.2. Achieved fitness after 1000 iterations for benchmark problems.

<table>
<thead>
<tr>
<th>Problem</th>
<th>n</th>
<th>Software PSO Fitness</th>
<th>Hardware PSO Fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sphere</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0.073</td>
<td>0.001</td>
</tr>
<tr>
<td>Rosenbrock</td>
<td>2</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0.044</td>
<td>0.085</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>8.081</td>
<td>8.615</td>
</tr>
</tbody>
</table>

Table 2.3. Execution time for software and hardware PSO implementations.

<table>
<thead>
<tr>
<th>Problem</th>
<th>n</th>
<th>Software PSO Time</th>
<th>Hardware PSO Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sphere</td>
<td>1</td>
<td>2.07 sec.</td>
<td>200 µs</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>5.79 sec.</td>
<td>338 µs</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>10.99 sec.</td>
<td>392 µs</td>
</tr>
<tr>
<td>Rosenbrock</td>
<td>2</td>
<td>2.14 sec.</td>
<td>344 µs</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>5.95 sec.</td>
<td>444 µs</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>10.91 sec.</td>
<td>800 µs</td>
</tr>
</tbody>
</table>

Figure 2.7. The hardware PSO compared with a software PSO for the sphere function of 10 dimensions.
Figure 2.8. The hardware PSO compared with a software PSO for the Rosenbrock function of 10 dimensions.

Table 2.4. Hardware PSO logic requirements for the benchmark problems.

<table>
<thead>
<tr>
<th>Problem</th>
<th>n</th>
<th>Logic Utilization</th>
<th>LUTs</th>
<th>Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sphere</td>
<td>1</td>
<td>1118</td>
<td>1523</td>
<td>100 MHz</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>4744</td>
<td>10631</td>
<td>59 MHz</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>9249</td>
<td>20873</td>
<td>51 MHz</td>
</tr>
<tr>
<td>Rosenbrock</td>
<td>2</td>
<td>2244</td>
<td>5332</td>
<td>58 MHz</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>5228</td>
<td>12513</td>
<td>45 MHz</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>9940</td>
<td>25750</td>
<td>25 MHz</td>
</tr>
</tbody>
</table>
3 IMPLEMENTATION OF NEURAL NETWORK

3.1 INTRODUCTION

NNs are universal function approximators. The structure of a multilayer perceptron, which is used in this thesis work, is given in Figure 3.1. This network has two inputs, three hidden neurons, and one output. The input and output layers are linear, while the hidden layer uses the hyperbolic tangent function. The vector \( w \) contains the weights for the input layer while the vector \( v \) contains the weights for the hidden layer. The output of the network is computed as follows:

\[
\begin{align*}
    a_i &= \sum_{j=1}^{nI} w_{i,j}x_j, i = 1, \ldots, nH \\
    d_i &= \frac{e^{a_i} - e^{-a_i}}{e^{a_i} + e^{-a_i}}, i = 1, \ldots, nH \\
    y &= \sum_{i=1}^{nH} v_id_i
\end{align*}
\] (3.1)

where \( nH \) and \( nI \) are the number of hidden neurons and inputs, respectively.

Figure 3.1. An NN with an input, hidden and output layer.
NNs are inherently parallel, with each layer of neurons processing incoming data independently of each other. While general purpose processors have reached impressive processing speeds, they still cannot fully exploit this inherent parallelism due to their sequential architecture [24]. In order to achieve the high neural network throughput needed for real-time applications, a custom hardware design is needed.

There are different ways to exploit the parallelism of NNs in hardware. For instance, each neuron in a given layer can be processed in parallel; this results in one layer being processed at a given movement. As an alternative, each of the layers could be processed simultaneously. In this design each of the neurons would be processed simultaneously. In terms of performance, the latter would yield the greatest throughput, however, logic resources on the FPGAs is limited. The most costly operation in computing an NN output is multiplication, for an NN with $I$ inputs, $H$ hidden neurons and $O$ outputs, $I \times H + H \times O$ multiplications are required. The FPGA resources required for a single multiplication operation are dependent on the width of the multiplication, i.e. a 8-bit by 8-bit multiplication or a 12-bit by 12-bit multiplication. In addition to the resources required for the multiplications, the activation function must be implemented on the FPGA. The activation function is typically a non-linear function such as the hyperbolic tangent:

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

which requires two exponentials, $e^x$ and $e^{-x}$ to be computed. These operations would also require a large amount of FPGA resources and computing them directly is usually avoided as a result. An alternative is to compute a linear piece-wise approximation and program it into the LUTs on the FPGA. Although the approximations will lose some precision the LUTs can be accessed in a single clock cycle which is far faster than computing the real solution, in addition to consuming far less FPGA resources.

### 3.2 Design Approach

Although hardware implementations of NNs offer the best performance, software implementations are far more common. As stated in Section 1, the hardware implementations are more difficult to design. More importantly, its more difficult
for engineers without digital system design backgrounds to learn how to quickly develop digital systems. Typically designing a digital system requires experience with a hardware description language (HDL) such as VHDL or Verilog.

Xilinx System Generator is a software tool for modeling and designing FPGA-based systems in MathWorks Simulink [25]. This tool presents a high level abstract view of the whole system, yet automatically maps the system to a faithful hardware implementation. System Generator allows hardware designers to design high-performance, high-level DSP systems using custom Simulink blocks. Designers can use the System Generator blocks to build a hardware system, simulate the system using Simulink and produce a bit file which can then be programmed onto a FPGA. Since Simulink is tightly integrated with MathWorks MATLAB, it becomes easier to implement complex algorithms in hardware than purely using a HDL. Furthermore, System Generator allows blocks designed using an HDL to be imported and simulated within a system designed using System Generator.

As stated in Section 3.1, NNs require a large number of multiplications and the multiplication operation is very resource consuming when implemented on a FPGA. This places a constraint on the size of the neural network; even if LUTs are used for activation computation, the FPGA must be able to provide two multipliers per neuron. Due to this constraint, a multiplier-rich FPGA platform has been selected, the Xilinx Virtex-II Pro Development System. The Virtex-II Pro boasts 136 18-bit embedded multipliers, two embedded PowerPC processors and 30,000 programmable logic cells [22]. The Virtex-II is targeted at high-performance DSP and research applications and is well suited for a FPGA-based NN implementation.

3.3 NN HARDWARE IMPLEMENTATION

The following sections discuss the design of the FPGA-based NN.

3.3.1 Implementing the Neuron MAC. The neurons are the essential components of NNs. In a feedforward NN, each neuron receives the output of each neuron in the preceding layer. The neuron model is shown in Figure 3.2. The activation of a neuron is the sum of the neurons inputs multiplied by their corresponding
weights:

\[ a = \sum_{i=0}^{N} x_i \cdot w_i \]  

(3.3)

where \( x_i \) is the output of the \( i \)th neuron in the preceding layer and \( w_i \) is the corresponding weight. This is known as performing a Multiply ACCumulate (MAC) operation. Each neuron has an activation function, such as the hyperbolic tangent function. The output of the neuron is found by applying the activation function to the activation of the neuron.

The MAC operation is modeled using a multiply and accumulate block. The MAC subsystem is provided an input \( x \), the corresponding weight \( w \) and a reset signal. The reset signal is used to reset the accumulator to 0 when all the pairs of inputs and weights have been processed. The MAC subsystem is illustrated in Figure 3.3.
3.3.2 Implementing the Activation Function LUT. As discussed in Section 3.1, the activation function is approximated using a LUT-Based Activation Function (LUTAF) module. The activation functions need to be approximated, using a simple linear function:

\[ f(x) = \alpha_2 + c_2 \times x \]  

where \( x \) is the activation of the neuron and \( \alpha_1 \) and \( \alpha_2 \) are LUT constants. To solve for the constants, the interval over which the approximations will hold must be determined. In this thesis, the interval was selected to be \( x \in [-8, 7] \). The number of linearized intervals also needs to be determined (as a power of two). In this thesis, the number of subintervals was selected to be \( 2^4 = 16 \). The linearized regions can be found in Table 3.1. Now, the values for \( \alpha_1 \) and \( \alpha_2 \) must be computed for the given activation function, the approximated interval and the respective number of subintervals. In this thesis the hyperbolic tangent activation function was used. Therefore, to compute \( \alpha_1 \) and \( \alpha_2 \) for a given region the following system must be solved:

\[ \alpha_1 + \alpha_2 \times L = \frac{e^U - e^{-U}}{e^U + e^{-U}} \]  

\[ \alpha_1 + \alpha_2 \times U = \frac{e^L - e^{-L}}{e^L + e^{-L}} \]  

where \( L \) is lower boundary in the subinterval and \( U \) is the upper boundary in the subinterval. Solving for \( \alpha_1 \) and \( \alpha_2 \) using Matlab’s Symbolic Math toolbox results in:

\[ \alpha_1 = \frac{L \times e_U - U \times e_U - U \times e_L \times e_U + L \times e_U \times e_L + U + e_L \times L - L + U \times e_L}{-U - U \times e_U - U \times e_L - U \times e_U \times e_L + L + L \times e_U + e_L \times L + L \times e_U \times e_L} \]  

\[ \alpha_2 = \frac{2 \times (e_U - e_L)}{-U - U \times e_U - U \times e_L - U \times e_U \times e_L + L + L \times e_U + e_L \times L + L \times e_U \times e_L} \]  

where \( e_U \) and \( e_L \) are constants which equal \( e^{-2U} \) and \( e^{-2L} \), respectively. The values for \( \alpha_1 \) and \( \alpha_2 \) are now computed for each subinterval using these solutions. The resulting values for this work can be found in Table 3.1. The resulting approximation is shown in Figure 3.4.
Table 3.1. The values for the hyperbolic tangent activation function LUT where \( x \in \left[ L, U \right] \).

<table>
<thead>
<tr>
<th>L</th>
<th>U</th>
<th>( \alpha_1 )</th>
<th>( \alpha_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>7</td>
<td>0.99992</td>
<td>1.0625e-005</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>0.99952</td>
<td>7.8507e-005</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>0.99701</td>
<td>0.0005799</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0.98223</td>
<td>0.0042745</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0.90197</td>
<td>0.031027</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0.55916</td>
<td>0.20243</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-2.2204e-016</td>
<td>0.76159</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0.76159</td>
</tr>
<tr>
<td>-2</td>
<td>-1</td>
<td>-0.55916</td>
<td>0.20243</td>
</tr>
<tr>
<td>-3</td>
<td>-2</td>
<td>-0.90197</td>
<td>0.031027</td>
</tr>
<tr>
<td>-4</td>
<td>-3</td>
<td>-0.98223</td>
<td>0.0042745</td>
</tr>
<tr>
<td>-5</td>
<td>-4</td>
<td>-0.99701</td>
<td>0.0005799</td>
</tr>
<tr>
<td>-6</td>
<td>-5</td>
<td>-0.99952</td>
<td>7.8507e-005</td>
</tr>
<tr>
<td>-7</td>
<td>-6</td>
<td>-0.99992</td>
<td>1.0625e-005</td>
</tr>
<tr>
<td>-8</td>
<td>-7</td>
<td>-0.99999</td>
<td>1.438e-006</td>
</tr>
</tbody>
</table>

The hardware implementation of the LUTAF is constructed from three main elements, the memory element which contains the linearized constants \( \alpha_1 \) and \( \alpha_2 \), a multiplier and an addition unit. The LUTAF subsystem can be found in Figure 3.5. The proper subinterval is determined by selecting the upper \( k \) most significant bits of the input, \( x \), where \( 2^k \) is the number of subintervals. The upper \( k \) bits are used to index the corresponding values of \( \alpha_1 \) and \( \alpha_2 \) by using the \( k \) bits as the address to the memory elements. The values for \( \alpha_2 \) and \( x \) are passed to a multiplier while \( \alpha_1 \) is latched (delayed to match the propagation of the signal \((\alpha_2 \times x))\). Finally, the terms \((\alpha_2)\) and \((\alpha_2 \times x)\) are added to produce the activation function approximation.

3.3.3 Implementing the Complete Hardware Neuron. The hardware neuron is a combination of the Neuron MAC (Section 3.3.1) and the LUTAF (Section 3.3.2) and a memory element to store the weights. The complete hardware neuron
Figure 3.4. LUTAF approximation.

Figure 3.5. LUTAF approximation subsystem.

subsystem can be seen in Figure 3.6. The hardware neuron receives the values \(x\), \(w_{\text{addr}}\), \(\text{reset}\) and \(\text{latch}_f\). The input \(x\) is the value of the current input being processed (i.e. from the preceding neuron \(i\)). The value of \(w_{\text{addr}}\) is the index of the current input being processed, \(i\). This signal is used to index the value of the corresponding weight in the memory element. The reset signal is used to reset the MAC as explained
in Section 3.3.1. The \textit{latch\_f} signal is used to indicate when the output of the MAC is valid and should be latched (since the MAC sequentially accumulates values, the values other than the last are incomplete sums). The \textit{latch\_my\_latch} is responsible for latching the result of the MAC when its valid, based on the \textit{latch\_f} signal. The latched output from the MAC is connected to the input to the LUTAF subsystem which computes the approximated activation function output.

![Image](image_url)

**Figure 3.6.** The complete hardware neuron.

### 3.3.4 Implementing the NN Layer Control Block.

As the hardware neurons process the input/weight pairs sequentially, a mechanism is needed to cycle through each of the outputs from the previous layer and provide them to the hardware neurons in the respective current layer. The LCB (Layer Control Block) receives the outputs from the neurons in the previous layer as well as a global reset signal. The LCB is responsible for providing the value of the current input being processed, the index of the current input, the MAC reset and \textit{latch\_f} signals to the hardware neurons. Since the hardware neurons process each input/weight pair in parallel, each hardware neuron receives the same signals. The LCB is implemented as a finite state machine. The finite state machine for a layer receiving two inputs with three neurons is illustrated in Figure 3.7 while the LCB and hardware neurons are shown in Figure 3.8. The operation of a hardware neuron being controlled by a LCB can be seen in Figure 3.9.
3.3.5 Implementing a Three-Layer NN. Using the components developed in Sections 3.3.3 and 3.3.4, an NN of arbitrary size can be constructed. The LCB finite state machines must have the corresponding states for the layers inputs and outputs. For a NN of size $1 \times 3 \times 1$ (a bias is included in each layer), the hardware NN is illustrated in Figure 3.10. The inputs to the FPGA-Based NN are the global reset signal, ResetSeq which synchronize the two LCBs and the $x1$ signal which is the input the NN. The FPGA-Based NN produces the signal $F$ which is the output of the NN. The Gate In and Gateway Out blocks indicate ports on the FPGA where external
signals are presented to the FPGA and received from the FPGA, respectively. The Gateway blocks are configured to specify specific I/O pins on the FPGA.

To construct a $n$-layer NN, $(n - 1)$ LCBs are required; the input layer does not require a LCB, only layers which contain neurons that receive input from multiple sources require a LCB. The number of neurons in each layer impact the number of the hardware resources consumed as well as the throughput of the NN. Layers that do not require a non-linear transfer function (linear layers) require less hardware resources to implement because there is no need for each neuron to have a LUTAF module. Therefore, a non-linear $n$-neuron hidden layer would require more hardware resources than a $n$-neuron linear output layer. The number of input patterns processed in a given time is strictly a function of the largest layer in the NN. In other words, a $1 \times 10 \times 1$ NN will process the same number of input patterns as a $5 \times 10 \times 5$ NN in a given amount of time. This is due to the fact that the LCBs must operate together, each must process the first input at the same time, one cannot begin processing a new input pattern before the other LCBs are finished for all other layers.
Figure 3.9. A hardware neuron being controlled by a LCB.

Figure 3.10. A FPGA-based NN (of size $1 \times 3 \times 1$).
3.4 RESULTS

To test the FPGA-Based NN, a simple NN has been implemented to verify its operation. A NN of size $1 \times 10 \times 1$ has been selected, utilizing the hyperbolic tangent activation function for neurons in the hidden layer and a linear activation function for the output neuron. A NN of the same architecture must first be trained in software. As an example, the following function has been selected to be approximated by the NN:

$$f(x) = \sin (5 \times x)$$  \hspace{1cm} (3.9)

In this thesis the MATLAB Neural Network is used to implement the training algorithm, however any other method could be used. The “train_nn.m” script containing the training procedure is found in Appendix B. The results of the NN training are shown in Figure 3.11. After the weights for the NN have been determined using the training algorithm some processing needs to take place before the hardware NN can be used. First the weights need to be organized into the matrices $w$ and $v$ which con-
tain the weights for the hidden and output layers, respectively. The Xilinx System Generator Single Port Read-Only Memory (ROM) is used in each hardware neuron to store its respective weights. The ROM in each neuron is configured to select the correct weight indices from the trained weights in the MATLAB workspace (from matrices $w$ and $v$), each neuron only stores the weights it needs to compute its own output. Here the weights are converted into 18.12 fixed-point representation when they are loaded into the ROMs.

In the second part of this process the values for the LUTAF must be loaded into each Neurons LUTAF ROM. A MATLAB script it used to solve for the values of $\alpha_1$ and $\alpha_2$ using the method described in Section 3.4. In order for LUTAF constants $\alpha_1$ and $\alpha_2$ to be used by the FPGA-Based NN, the values of the constants found in MATLAB must also be converted into 18.12 fixed-point.

Finally, the input patterns that were used for training are processed using a MATLAB script for presentation to the Xilinx System Generator Gateway In port, $NN\_Input$. In addition, a reset signal is prepared in the same manner for the Gateway In port $NN\_Reset$. These steps are performed by the script “nn_load.m” in Appendix B.

Now that the The FPGA-based NN model has of the necessary values converted and loaded, it can be simulated. The output of the Simulink simulation can be found in Figure 3.12. However, the output is in terms of clock cycles. In order to compare the hardware NN output against the software NN output, the software NN output must be upsampled. This is performed by the script “test_nn_model.m”, in Appendix B. The output of each hardware hidden neuron is compared with the software hidden neurons in Figure 3.13. The output of the first five hidden neurons are shown in Figure 3.13(a) while the output of the last five hidden neurons are shown in Figure 3.13(b). The output of the hardware NN is compared against the software NN in Figure 3.14. The simulations presented are not high-level simulations, they are cycle-accurate simulations of the synthesized hardware NN design executing on the Xilinx Virtex-II Pro FPGA platform.

---

18.12 fixed-point representation indicates that a total of 18 bits are used and 12 of the bits are dedicated to the fractional portion of the value, leaving 6 bits for the integer portion of the value.
The design is then synthesized using Xilinx ISE. The FPGA resources used can be found in Figure 3.15. It should be noted that the 1x10x1 hardware NN only requires approximately 6% of the Xilinx Virtex II Pro’s logic resources. The critical path has a delay of 15.4 ns, resulting in a maximum clock rate of 64 MHz. Using a newer FPGA, such as the Xilinx Virtex-4 or Xilinx Virtex-5 will result in even higher clock rates. The logic utilization will vary if another FPGA is used, depending on the size of the FPGA being targeted.
Figure 3.13. Comparing the output of hidden neurons of the FPGA-based NN and Matlab-based NN.

(a) The output of hidden neurons 1-5.

(b) The output of hidden neurons 6-10.
Figure 3.14. Comparing the output of the FPGA-based NN and Matlab-based NN.

Figure 3.15. The hardware resources used for the $1 \times 10 \times 1$ NN on the Xilinx Virtex-II Pro FPGA.
This thesis has presented two independent hardware designs, a hardware implementation of the PSO algorithm and a hardware implementation of an NN.

A pipelined hardware implementation of PSO has been presented. The hardware PSO design implemented on a Xilinx Virtex-II Pro FPGA is shown to perform well on two standard benchmark problems when compared to a common software implementation of PSO in Matlab. When compared to the software implementation, the hardware implementation is between 6,220 - 28,935 times faster. The system is targeted for real-time applications where minimizing PSO execution time is critical. One such application is real-time neural network training.

A high-performance Xilinx Virtex-II Pro FPGA-based NN architecture has been presented. The system allows feed-forward NNs to be implemented on FPGAs resulting in very high throughput. The hardware NN is developed using a model-based methodology which is easier for researchers who are not knowledgeable in a HDL to modify for their specific application.

The PSO hardware design in this thesis did not utilize any explicit memory such as RAM for storing the PSO variables\(^2\) instead, all of the variables were simply stored in registers. This approach is not the most efficient for problems which have fitness function with a large number of dimensions. Training a large NN would be such a case, where the number of weights could be in the hundreds or thousands. For these applications, the PSO hardware design presented could be augmented with a memory interface that used a RAM module to store PSO variables.

The hardware NN presented in this thesis was of the feed-forward architecture. None of the outputs or outputs from the hidden layer were passed back as inputs to the NN. These recurrent architectures have been shown to offer more capabilities for approximating problems with temporally related data. The hardware NN design presented here could be extended to incorporate recurrent architectures by modifying the LCBs.

---

\(^2\)While FPGAs utilize memory components as LUTs to implement logic functions, they also have a number of block RAMs available that are more area-efficient for storing large amounts of data.
APPENDIX A

HARDWARE PSO SOURCE CODE
The main portions of the VHDL source code developed for the hardware PSO implementation has been selected for inclusion in this appendix for reference. All of the VHDL source code was compiled and simulated using Mentor Graphics ModelSim 6.5. The source code was synthesized using Xilinx ISE 10.1.
PSO.vhd

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

LIBRARY ieee_proposed;
USE ieee_proposed.math_utility_pkg.ALL;
USE ieee_proposed.fixed_pkg.ALL;
USE WORK.pso_package.ALL;

— Uncomment the following library declaration if instantiating
— any Xilinx primitives in this code.
—library UNISIM;
—use UNISIM.VComponents.all;

entity PSO is
  PORT(
    SYSTEM_CLK : in std_logic;
    SWITCHES : in STD_LOGIC_VECTOR(1 downto 0);
    LEDS : out STD_LOGIC_VECTOR(1 downto 0);
    GBEST : out STD_LOGIC_VECTOR(7 downto 0)
  );
end PSO;

architecture Behavioral of PSO is
  signal DCM_IBUF_OUT, DCM_CLK0_OUT, DCM_LOCKED : std_logic;
  signal clk : std_logic;
  signal gbest_position : total_position;
  signal gbest_fitness : sfixed(FITNESS_BITS_LEFT downto –
  FITNESS_BITS_RIGHT);
  signal PSO_reset, PSO_finished : std_logic;
  signal iteration : unsigned(ITERATION_COUNT_BITS–1 downto 0);
  signal gbest_fitness_tmp : std_logic_vector(FITNESS_BITS_LEFT +
  FITNESS_BITS_RIGHT downto 0);
COMPONENT dcm0
PORT(
  CLKIN_IN : IN std_logic;
  CLKDV_OUT : OUT std_logic;
  CLKIN_IBUFG_OUT : OUT std_logic;
  CLK0_OUT : OUT std_logic;
  LOCKED_OUT : OUT std_logic
);
END COMPONENT;

COMPONENT pso_top
PORT(
  clk, reset : in std_logic;
  gbest : total_position;
  gbest_e : out sfixed(FITNESS_BITS_LEFT downto FITNESS_BITS_RIGHT);
  iteration : out unsigned(ITERATION_COUNT_BITS-1 downto 0);
  finished : out std_logic);
END COMPONENT;

begin

Inst_dcm0 : dcm0 PORT MAP(
  CLKIN_IN => SYSTEM_CLK,
  CLKDV_OUT => clk,
  CLKIN_IBUFG_OUT => DCM_IBUFG_OUT,
  CLK0_OUT => DCM_CLK0_OUT,
  LOCKED_OUT => DCM_LOCKED
);

Inst_pso_top0 : pso_top PORT MAP(
  clk => clk,
  reset => PSO_reset,
  gbest => gbest_position,
  gbest_e => gbest_fitness,
  iteration => iteration,
  finished => PSO_finished
);
PSO.reset <= SWITCHES(0);
LEDs(0) <= NOT PSO.reset;
LEDs(1) <= NOT PSO.finished;
gbest_fitness_tmp <= to_slv(gbest_fitness);

—GBEST <= gbest_position(7 downto 0);
GBEST <= gbest_fitness_tmp(7 downto 0);
end Behavioral;

LIBRARY floatfixlib;
USE floatfixlib.math_utility_pkg.ALL;
USE floatfixlib.fixed_pkg.ALL;

LIBRARY ieee_proposed;
USE ieee_proposed.math_utility_pkg.ALL;
USE ieee_proposed.fixed_pkg.ALL;

PACKAGE pso_package is
  constant NUM_PARTICLES : natural := 20;
  constant RN_BITS : natural := 64;
  constant NUM_DIMENSIONS : natural := 10;
  constant POSITION_BITS_LEFT : natural := 7;
  constant POSITION_BITS_RIGHT : natural := 9;
  constant FITNESS_BITS_LEFT : natural := POSITION_BITS_LEFT;
  constant FITNESS_BITS_RIGHT : natural := POSITION_BITS_RIGHT;
  constant FITNESS_BITS_LEFT : natural := POSITION_BITS_LEFT + POSITION_BITS_LEFT + NUM_DIMENSIONS;
  constant FITNESS_BITS_RIGHT : natural := POSITION_BITS_RIGHT + POSITION_BITS_RIGHT;
  constant FITNESS_BITS_LEFT : natural := 43+NUM_DIMENSIONS + 6;
  constant FITNESS_BITS_RIGHT : natural := 36;
  constant C_BITS_LEFT : natural := 2;
  constant C_BITS_RIGHT : natural := 2;
  constant R_BITS_LEFT : natural := 0;
  constant R_BITS_RIGHT : natural := 9;
  constant W_BITS_LEFT : natural := 0;
  constant W_BITS_RIGHT : natural := 3;
  constant VELOCITY_BITS_LEFT : natural := POSITION_BITS_LEFT+1+R_BITS_LEFT+C_BITS_LEFT+2;
constant VELOCITY_BITS_RIGHT : natural := POSITION_BITS_RIGHT +
R_BITS_RIGHT + C_BITS_RIGHT;

constant VELOCITY_BITS_LEFT : natural := POSITION_BITS_LEFT;
constant VELOCITY_BITS_RIGHT : natural := POSITION_BITS_RIGHT + 1;

constant PARTICLE_COUNT_BITS : natural := 5;
constant DIMENSION_COUNT_BITS : natural := 4;
constant ITERATION_COUNT_BITS : natural := 10;
constant MAX_ITERATIONS : natural := 1000;

type total_position is array (0 to NUM_DIMENSIONS-1) of sfixed(
POSITION_BITS_LEFT downto -POSITION_BITS_RIGHT);

type total_velocity is array (0 to NUM_DIMENSIONS-1) of sfixed(
VELOCITY_BITS_LEFT downto -VELOCITY_BITS_RIGHT);

END pso_package;
USE WORK.pso_package.ALL;
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

LIBRARY floatfixlib;
—USE floatfixlib.math_utility_pkg.ALL;
—USE floatfixlib.fixed_pkg.ALL;

LIBRARY ieee_proposed;
USE ieee_proposed.math_utility_pkg.ALL;
USE ieee_proposed.fixed_pkg.ALL;

ENTITY pso_top IS
PORT (clk, reset : in std_logic;
    gbest : out total_position;
    gbest_e : out sfixed(FITNESS_BITS_LEFT downto FITNESS_BITS_RIGHT);
    iteration : out unsigned(ITERATION_COUNT_BITS-1 downto 0);
    finished : out std_logic);
END ENTITY pso_top;

ARCHITECTURE behavioral OF pso_top IS

— Reset Signals
signal reset_modules : std_logic;
— Signals to RNG
signal reset_rng, rng_enable : std_logic;
— Signals from RNG
signal rn : std_logic_vector(RN_BITS-1 downto 0);
— Signals to Fitness module
signal position_to_eval : total_position;
— Signals from Fitness module to Best modules
signal fitness_from_eval : sfixed(FITNESS_BITS_LEFT downto FITNESS_BITS_RIGHT);
— Signals to Best modules
signal position_to_best : total_position;
signal pbest_position, gbest_position : total_position;
signal pbest_fitness, gbest_fitness : sfixed(FITNESS_BITS_LEFT
downto −FITNESS_BITS_RIGHT);

— Signals from Best modules
signal new_pbest_position, new_gbest_position : total_position;
signal new_pbest_fitness, new_gbest_fitness : sfixed(
FITNESS_BITS_LEFT downto −FITNESS_BITS_RIGHT);

— Signals to Update Velocity module
signal velocity_to_update_velocity : total_velocity;
signal position_to_update_velocity : total_position;
signal rn1, rn2 : sfixed(R_BITS_LEFT−1 downto −R_BITS_RIGHT);

— Signals to Update Position module
signal new_velocity : total_velocity;

— Signals from Update Position module
signal new_position : total_position;

— Particle Memory
type particle_position_mem is array (0 to NUM_PARTICLES−1) of
total_position;
type particle_velocity_mem is array (0 to NUM_PARTICLES−1) of
total_velocity;
type particle_fitness_mem is array (0 to NUM_PARTICLES−1) of sfixed(
FITNESS_BITS_LEFT downto −FITNESS_BITS_RIGHT);

signal position_mem : particle_position_mem;
signal velocity_mem : particle_velocity_mem;
signal pbest_position_mem : particle_position_mem;
signal pbest_fitness_mem : particle_fitness_mem;
signal gbest_position_mem : total_position;
signal gbest_fitness_mem : sfixed(FITNESS_BITS_LEFT downto −
FITNESS_BITS_RIGHT);
signal init_fitness_to : sfixed(FITNESS_BITS_LEFT downto −
FITNESS_BITS_RIGHT);

— PSO Top FSM
TYPE states IS (idle, init_rng, init_particles1, init_particles2,
init_particles3, init_pipeline1, init_pipeline2, exec, done);
signal state : states;
signal dimension_cnt : unsigned(DIMENSION_COUNT_BITS–1 downto 0);
signal particle_cnt_init : unsigned(PARTICLE_COUNT_BITS–1 downto 0);
signal particle_cnt_stage1 : unsigned(PARTICLE_COUNT_BITS–1 downto 0);
signal particle_cnt_stage2 : unsigned(PARTICLE_COUNT_BITS–1 downto 0);
signal particle_cnt_stage3 : unsigned(PARTICLE_COUNT_BITS–1 downto 0);
signal particle_cnt_stage4 : unsigned(PARTICLE_COUNT_BITS–1 downto 0);
signal particle_cnt_stage5 : unsigned(PARTICLE_COUNT_BITS–1 downto 0);
signal iteration_cnt : unsigned(ITERATION_COUNT_BITS–1 downto 0);
signal init_rng_cnt : unsigned(4 downto 0);
BEGIN
rng : ENTITY work.RandomNumberGenerator(General)
PORT MAP (clk, rng_enable, reset_rng, rn);
delay_positions : for i in 0 to NUM_DIMENSIONS–1 generate
delay_position1 : ENTITY work.delay_position(behavioral)
PORT MAP (clk, position_to_eval(i), position_to_best(i));
delay_position2 : ENTITY work.delay_position(behavioral)
PORT MAP (clk, position_to_best(i), position_to_update_velocity(i));
end generate delay_positions;

—fitness_eval : ENTITY work.fitness_dummy(behavioral)
— PORT MAP (clk, position_to_eval, fitness_from_eval);
—fitness_eval : ENTITY work.fitness_squared(behavioral)
— PORT MAP (clk, position_to_eval, fitness_from_eval);
fitness_eval : ENTITY work.fitness_rosenbrock(behavioral)
PORT MAP (clk, position_to_eval, fitness_from_eval);

update_pbest : ENTITY work.update_best(behavioral)
PORT MAP (clk, reset_modules, position_to_best, fitness_from_eval,
pbest_position, pbest_fitness,
    new_pbest_position, new_pbest_fitness);
update_gbest : ENTITY work.update_best(behavioral)
PORT MAP (clk, reset_modules, position_to_best, fitness_from_eval,
new_gbest_position, new_gbest_fitness,
    new_gbest_position, new_gbest_fitness);

pso_dimensions : for i in 0 to NUM_DIMENSIONS-1 generate
    pso_dimension : ENTITY work.pso_dimension(behavioral)
        PORT MAP (clk, reset_modules, position_to_update_velocity(i),
            velocity_to_update_velocity(i), new_pbest_position(i),
            new_gbest_position(i),
            rn1, rn2, new_velocity(i), new_position(i))
    end generate pso_dimensions;

iteration <= iteration_cnt;

PSO_main : PROCESS(clk, reset)
BEGIN
    — Go into 'idle' state upon reset
    IF reset='1' THEN
        state <= idle;
        — Reset modules
        reset_modules <= '1';
        — Reset RNG
        reset_rng <= '1';
        rng_enable <= '0';
        finished <= '0';
        iteration_cnt <= (others => '0');
    ELSIF (clk'EVENT AND clk='1') THEN
        CASE state IS
            WHEN idle =>
                state <= init_rng;
                particle_cnt_init <= (others => '0');
                dimension_cnt <= (others => '0');
                iteration_cnt <= (others => '0');
                finished <= '0';
                — Maximum (worst) fitness value
                init_fitness_to <= (FITNESS_BITS_LEFT => '0', others => '1')
            ;
                init_rng_cnt <= (others => '0');
particle_cnt_stage1 <= (others => '0');
particle_cnt_stage2 <= to_unsigned(NUM_PARTICLES-2,
   PARTICLE_COUNT_BITS);
particle_cnt_stage3 <= to_unsigned(NUM_PARTICLES-3,
   PARTICLE_COUNT_BITS);
particle_cnt_stage4 <= to_unsigned(NUM_PARTICLES-4,
   PARTICLE_COUNT_BITS);
particle_cnt_stage5 <= to_unsigned(NUM_PARTICLES-5,
   PARTICLE_COUNT_BITS);

WHEN init_rng =>  
  -- Cycle through some RNG values so 
  -- they are 'more random'
  reset_rng <= '0';
  rng_enable <= '1';
  if(init_rng_cnt < "11111") then
    init_rng_cnt <= init_rng_cnt + 1;
  else
    state <= init_particles1;
  end if;

WHEN init_particles1 =>  
  -- Initialize all particle
  memory to random values
  position_mem(to_integer(particle_cnt_init))(to_integer(
     dimension_cnt)) <= to_sfixed(rn(POSITION_BITS_LEFT +
     POSITION_BITS_RIGHT downto 0), POSITION_BITS_LEFT, -
     POSITION_BITS_RIGHT);
  velocity_mem(to_integer(particle_cnt_init))(to_integer(
     dimension_cnt)) <= to_sfixed(0, VELOCITY_BITS_LEFT, -
     VELOCITY_BITS_RIGHT);

  if(dimension_cnt < NUM_DIMENSIONS-1) then
    dimension_cnt <= dimension_cnt + 1;
  else
    dimension_cnt <= (others => '0');
    if(particle_cnt_init < NUM_PARTICLES-1) then
      particle_cnt_init <= particle_cnt_init + 1;
    else
      particle_cnt_init <= (others => '0');
      state <= init_particles2;
WHEN init_particles2 =>
    — Initialize all particle memory to random values
    pbest_position_mem(to_integer(particle_cnt_init))(to_integer(dimension_cnt)) <= to_sfixed(rn(POSITION_BITS_LEFT + POSITION_BITS_RIGHT downto 0), POSITION_BITS_LEFT, −POSITION_BITS_RIGHT);
    pbest_position_mem(to_integer(particle_cnt_init))(to_integer(dimension_cnt)) <= to_sfixed(0, POSITION_BITS_LEFT, −POSITION_BITS_RIGHT);
    — Initialize fitness to largest (worst) value
    pbest_fitness_mem(to_integer(particle_cnt_init)) <= init_fitness_to;

if(dimension_cnt < NUM_DIMENSIONS−1) then
    dimension_cnt <= dimension_cnt + 1;
else
    dimension_cnt <= (others => '0');
    if(particle_cnt_init < NUM_PARTICLES−1) then
        particle_cnt_init <= particle_cnt_init + 1;
    else
        particle_cnt_init <= (others => '0');
        state <= init_particles3;
    end if;
end if;

WHEN init_particles3 =>
    — Initialize all particle memory to random values
    gbest_position_mem(to_integer(dimension_cnt)) <= to_sfixed(rn(POSITION_BITS_LEFT + POSITION_BITS_RIGHT downto 0), POSITION_BITS_LEFT, −POSITION_BITS_RIGHT);
    gbest_position_mem(to_integer(dimension_cnt)) <= to_sfixed(0, POSITION_BITS_LEFT, −POSITION_BITS_RIGHT);
    — Initialize fitness to largest (worst) value
    gbest_fitness_mem <= init_fitness_to;
if (dimension\_cnt < NUM\_DIMENSIONS−1) then
    dimension\_cnt <= dimension\_cnt + 1;
else
    dimension\_cnt <= (others => '0');
if (particle\_cnt\_init < NUM\_PARTICLES−1) then
    particle\_cnt\_init <= particle\_cnt\_init + 1;
else
    particle\_cnt\_init <= (others => '0');
    state <= init\_pipeline1;
end if;
end if;

— Pass the particles positon to the fitness module
position\_to\_eval <= position\_mem(to\_integer(
    particle\_cnt\_stage1));

WHEN init\_pipeline1 ⇒

— Pass the particles positon to the fitness module
position\_to\_eval <= position\_mem(to\_integer(
    particle\_cnt\_stage1));

— Pass PBest to PBest Update Module
pbest\_position <= pbest\_position\_mem(to\_integer(
    particle\_cnt\_stage2));
pbest\_fitness <= pbest\_fitness\_mem(to\_integer(
    particle\_cnt\_stage2));

— Pass GBest to GBest Update Module
gbest\_position <= gbest\_position\_mem;
gbest\_fitness <= gbest\_fitness\_mem;

— Old velocity to Update Velocity module
velocity\_to\_update\_velocity <= velocity\_mem(to\_integer(
    particle\_cnt\_stage3));

— Random numbers to Update Velocity module
rn1 <= to_sfixed(rn(R_BITS_LEFT-1 + R_BITS_RIGHT downto 0),
         R_BITS_LEFT-1, -R_BITS_RIGHT);
rn2 <= to_sfixed(rn(2*(R_BITS_LEFT-1 + R_BITS_RIGHT)+1
downto (R_BITS_LEFT-1 + R_BITS_RIGHT)+1), R_BITS_LEFT-1,
        -R_BITS_RIGHT);

       — state <= init_pipeline2;

       —WHEN init_pipeline2 =>
       state <= exec;
       reset_modules <= '0';
       report "Executing";

WHEN exec =>
     — Execute PSO algorithm
     — Pass the particles position to the fitness module
     position_to_eval <= position_mem(to_integer(
             particle_cnt_stage1));

     — Pass PBest to PBest Update Module
     pbest_position <= pbest_position_mem(to_integer(
             particle_cnt_stage2));
     pbest_fitness <= pbest_fitness_mem(to_integer(
             particle_cnt_stage2));

     velocity_to_update_velocity <= velocity_mem(to_integer(
             particle_cnt_stage2));

     — Random numbers to Update Velocity module
     rn1 <= to_sfixed(rn(R_BITS_LEFT-1 + R_BITS_RIGHT downto 0),
                     R_BITS_LEFT-1, -R_BITS_RIGHT);
     rn2 <= to_sfixed(rn(2*(R_BITS_LEFT-1 + R_BITS_RIGHT)+1
downto (R_BITS_LEFT-1 + R_BITS_RIGHT)+1), R_BITS_LEFT-1,
                 -R_BITS_RIGHT);

if((iteration_cnt > 0) ) then
     — assert new_gbest_fitness <= gbest_fitness;
--- report "E@PSO_TOP: GBestE increased from " & integer' image(to_integer(signed(gbest_fitness_mem(FITNESS_BITS_LEFT downto 0)))) & " to " & integer' image(to_integer(signed(new_gbest_fitness(FITNESS_BITS_LEFT downto 0))))

--- severity Warning;

--- report "Iteration : " & integer' image(to_integer(unsigned(iteration_cnt)))) & " GBestE: " & integer' image(to_integer(signed(new_gbest_fitness(FITNESS_BITS_LEFT downto 0)))) & "/" & to_string(new_gbest_fitness);

--- Store new PBest

pbest.position_mem(to_integer(particle_cnt_stage3)) <=
new.pbest_position;
pbest.fitness_mem(to_integer(particle_cnt_stage3)) <=
new.pbest_fitness;

--- Store new GBest
gbest.position_mem <= new_gbest_position;
gbest.fitness_mem <= new_gbest_fitness;

--- Store new Velocity
velocity_mem(to_integer(particle_cnt_stage5)) <=
new.velocity;

--- Store new Position
position_mem(to_integer(particle_cnt_stage5)) <=
new_position;

else

end if;

--- Restart the sequence when the last particle has been passed to the fitness module

IF (particle_cnt_stage1 = NUM_PARTICLES-1) THEN

particle_cnt_stage1 <= (others => '0');

iteration_cnt <= iteration_cnt + 1;
Write current gbest position/fitness to output ports

```
gest <= new_gbest_position;
gbest_e <= new_gbest_fitness;
```

ELSE

```
particle_cnt_stage1 <= particle_cnt_stage1 + 1;
```

END IF;

Update counters

```
IF particle_cnt_stage2 = NUM_PARTICLES-1 THEN
  particle_cnt_stage2 <= (others => '0');
ELSE
  particle_cnt_stage2 <= particle_cnt_stage2 + 1;
END IF;
```

```
IF particle_cnt_stage3 = NUM_PARTICLES-1 THEN
  particle_cnt_stage3 <= (others => '0');
ELSE
  particle_cnt_stage3 <= particle_cnt_stage3 + 1;
END IF;
```

```
IF particle_cnt_stage4 = NUM_PARTICLES-1 THEN
  particle_cnt_stage4 <= (others => '0');
ELSE
  particle_cnt_stage4 <= particle_cnt_stage4 + 1;
END IF;
```

```
IF particle_cnt_stage5 = NUM_PARTICLES-1 THEN
  particle_cnt_stage5 <= (others => '0');
ELSE
  particle_cnt_stage5 <= particle_cnt_stage5 + 1;
END IF;
```

```
IF (iteration_cnt = MAX_ITERATIONS) THEN
  state <= done;
END IF;
```

WHEN done =>

```
  -- Reached maximum number of iterations
  -- LED On
  finished <= '1';
```
WHEN others =>

END CASE;

END IF;

END PROCESS PSO_main;

END behavioral;
USE WORK.pso_package.ALL;
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

LIBRARY floatfixlib;
—USE floatfixlib.math_utility_pkg.ALL;
—USE floatfixlib.fixed_pkg.ALL;

LIBRARY ieee_proposed;
USE ieee_proposed.math_utility_pkg.ALL;
USE ieee_proposed.fixed_pkg.ALL;

USE ieee.std_logic_textio.ALL;
USE std.textio.ALL;

entity pso_tb is
end entity pso_tb;

architecture bench of pso_tb is
  signal clk, reset : std_logic;
  signal gbest_position : total_position;
  signal gbest_fitness : sfixed(FITNESS_BITS_LEFT downto FITNESS_BITS_RIGHT);
  —signal gbest_fitness_prev : sfixed(FITNESS_BITS_LEFT downto FITNESS_BITS_RIGHT);
  signal finished : std_logic;
  signal iteration : unsigned(ITERATION_COUNT_BITS-1 downto 0);
begin
  pso : ENTITY work.pso_top(behavioral) PORT MAP(clk, reset, gbest_position, gbest_fitness, iteration, finished);

  clock : PROCESS
  BEGIN
    clk <= '0';
    wait for 5 ns;
clk <= '1';
wait for 5 ns;

END PROCESS clock;

stimulus : PROCESS
BEGIN
reset <= '1';
wait until clk'event AND clk = '1';
reset <= '0';
wait until clk'event AND clk = '1';

wait;
end process stimulus;

write_fitness : process (iteration)
BEGIN
file OUTFILE : text is out "rosenbrock_dim10.txt";
variable WRBUFFER : line;
variable fitness : real;

fitness := to_real(gbest_fitness);
write (WRBUFFER, string "ModelSimGBestE()" );
write (WRBUFFER, integer 'image(to_integer(iteration)));
write (WRBUFFER, string "=" );
write (WRBUFFER, real 'image(fitness));
write (WRBUFFER, string ";");
writeln(OUTFILE,WRBUFFER);
report "GBestE : " & real 'image(fitness);
end process;
end bench;
pso_tb.vhd

USE WORK.pso_package.ALL;
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

LIBRARY floatfixlib;
—USE floatfixlib.math_utility_pkg.ALL;
—USE floatfixlib.fixed_pkg.ALL;

LIBRARY ieee_proposed;
USE ieee_proposed.math_utility_pkg.ALL;
USE ieee_proposed.fixed_pkg.ALL;

USE ieee.std_logic_textio.ALL;
USE std.textio.ALL;

entity pso_tb is
end entity pso_tb;

architecture bench of pso_tb is
signal clk, reset : std_logic;
signal gbest_position : total_position;
signal gbest_fitness : sfixed(FITNESS_BITS_LEFT downto FITNESS_BITS_RIGHT);
—signal gbest_fitness_prev : sfixed(FITNESS_BITS_LEFT downto FITNESS_BITS_RIGHT);
signal finished : std_logic;
signal iteration : unsigned(ITERATION_COUNT_BITS-1 downto 0);

begin
  pso : ENTITY work.pso_top(behavioral) PORT MAP(clk, reset, gbest_position, gbest_fitness, iteration, finished);

clock : PROCESS
BEGIN
  clk <= '0';
  wait for 5 ns;
clk <= '1';
wait for 5 ns;

END PROCESS clock;

stimulus : PROCESS
BEGIN
reset <= '1';
wait until clk'event AND clk = '1';
reset <= '0';
wait until clk'event AND clk = '1';

wait;
end process stimulus;

write_fitness : process(iteration)

file OUTFILE : text is out "rosenbrock_dim10.txt";
variable WRBUFFER : line;
variable fitness : real;

begin
fitness := to_real(gbest_fitness);
write (WRBUFFER, string "ModelSimGBestE()");
write (WRBUFFER, integer image(to_integer(iteration)));
write (WRBUFFER, string "=");
write (WRBUFFER, real image(fitness));
write (WRBUFFER, string ";");
writeln(OUTFILE,WRBUFFER);
report "GBestE : " & real image(fitness);
end process;
end bench;
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

LIBRARY floatfixlib;
USE floatfixlib.math_utility_pkg.ALL;
USE floatfixlib.fixed_pkg.ALL;

LIBRARY ieee_proposed;
USE ieee_proposed.math_utility_pkg.ALL;
USE ieee_proposed.fixed_pkg.ALL;

LIBRARY work;
USE work.pso_package.ALL;

ENTITY fitness_squared IS
  PORT (clk : in std_logic;
      x : in total_position;
      fitness : out sfixed(FITNESS_BITS_LEFT downto -FITNESS_BITS_RIGHT));
END ENTITY fitness_squared;

ARCHITECTURE behavioral OF fitness_squared IS
BEGIN
  behavior : PROCESS(clk) IS
  BEGIN
    IF(rising_edge(clk)) THEN
      fitness <= x(0)*x(0);
      fitness <= x(0)*x(0) + x(1)*x(1);
      fitness <= x(0)*x(0) + x(1)*x(1) + x(2)*x(2);
      fitness <= x(0)*x(0) + x(1)*x(1) + x(2)*x(2) + x(3)*x(3);
      fitness <= x(0)*x(0) + x(1)*x(1) + x(2)*x(2) + x(3)*x(3) + x(4)*x(4);
      fitness <= x(0)*x(0) + x(1)*x(1) + x(2)*x(2) + x(3)*x(3) + x(4)*x(4) + x(5)*x(5);
  END PROCESS;
END behavioral;
fitness <= \( x(0) \times x(0) + x(1) \times x(1) + x(2) \times x(2) + x(3) \times x(3) + x(4) \times x(4) + x(5) \times x(5) + x(6) \times x(6) + x(7) \times x(7) \);

fitness <= \( x(0) \times x(0) + x(1) \times x(1) + x(2) \times x(2) + x(3) \times x(3) + x(4) \times x(4) + x(5) \times x(5) + x(6) \times x(6) + x(7) \times x(7) + x(8) \times x(8) + x(9) \times x(9) \);

fitness <= \( x(0) \times x(0) + x(1) \times x(1) + x(2) \times x(2) + x(3) \times x(3) + x(4) \times x(4) + x(5) \times x(5) + x(6) \times x(6) + x(7) \times x(7) + x(8) \times x(8) + x(9) \times x(9) + x(10) \times x(10) + x(11) \times x(11) \);

fitness <= \( x(0) \times x(0) + x(1) \times x(1) + x(2) \times x(2) + x(3) \times x(3) + x(4) \times x(4) + x(5) \times x(5) + x(6) \times x(6) + x(7) \times x(7) + x(8) \times x(8) + x(9) \times x(9) + x(10) \times x(10) + x(11) \times x(11) + x(12) \times x(12) + x(13) \times x(13) + x(14) \times x(14) + x(15) \times x(15) + x(16) \times x(16) + x(17) \times x(17) + x(18) \times x(18) + x(19) \times x(19) \);

END IF;

END PROCESS behavior;

END behavioral;
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

LIBRARY floatfixlib;
—USE floatfixlib.math_utility_pkg.ALL;
—USE floatfixlib.fixed_pkg.ALL;

LIBRARY ieee_proposed;
USE ieee_proposed.math_utility_pkg.ALL;
USE ieee_proposed.fixed_pkg.ALL;

LIBRARY work;
USE work.pso_package.ALL;

ENTITY fitness_rosenbrock IS
  PORT (clk : in std_logic;
         x : in total_position;
         fitness : out fixed(FITNESS_BITS_LEFT downto –FITNESS_BITS_RIGHT));
END ENTITY fitness_rosenbrock;

ARCHITECTURE behavioral OF fitness_rosenbrock IS
BEGIN
  behavior : PROCESS(clk) IS
BEGIN
    IF(rising_edge(clk)) THEN
      —fitness <= (100*(x(1)-x(0)*x(0))*(x(1)-x(0)*x(0))+(x(0)-1)*(x(0)-1));
      —fitness <= (100*(x(1)-x(0)*x(0))*(x(1)-x(0)*x(0))+(x(0)-1)*(x(0)-1)) + (100*(x(2)-x(1)*x(1))*(x(2)-x(1)*x(1))+(x(1)-1)*(x(1)-1)) + (100*(x(3)-x(2)*x(2))*(x(3)-x(2)*x(2))+(x(2)-1)*(x(2)-1)) + (100*(x(4)-x(3)*x(3))*(x(4)-x(3)*x(3))+(x(3)-1)*(x(3)-1));
    END IF;
  END PROCESS behavior;
END behavioral;
fitness <= (100*(x(1)-x(0)*x(0)))*(x(1)-x(0)*x(0))+(x(0)-1)*(x(0)
-1)) + (100*(x(2)-x(1)*x(1)))*(x(2)-x(1)*x(1))+(x(1)-1)*(x(1)
-1)) + (100*(x(3)-x(2)*x(2)))*(x(3)-x(2)*x(2))+(x(2)-1)*(x(2)
-1)) + (100*(x(4)-x(3)*x(3)))*(x(4)-x(3)*x(3))+(x(3)-1)*(x(3)
-1)) + (100*(x(5)-x(4)*x(4)))*(x(5)-x(4)*x(4))+(x(4)-1)*(x(4)
-1)) + (100*(x(6)-x(5)*x(5)))*(x(6)-x(5)*x(5))+(x(5)-1)*(x(5)
-1)) + (100*(x(7)-x(6)*x(6)))*(x(7)-x(6)*x(6))+(x(6)-1)*(x(6)
-1)) + (100*(x(8)-x(7)*x(7)))*(x(8)-x(7)*x(7))+(x(7)-1)*(x(7)
-1)) + (100*(x(9)-x(8)*x(8)))*(x(9)-x(8)*x(8))+(x(8)-1)*(x(8)
-1));
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

LIBRARY work;
USE work.pso_package.ALL;

LIBRARY floatfixlib;
—USE floatfixlib.math_utility_pkg.ALL;
—USE floatfixlib.fixed_pkg.ALL;

LIBRARY ieee_proposed;
USE ieee_proposed.math_utility_pkg.ALL;
USE ieee_proposed.fixed_pkg.ALL;

ENTITY delay_position IS
PORT (clk : in std_logic;
    x : in sfixed(POSITION_BITS_LEFT downto -POSITION_BITS_RIGHT);
    x_delayed : out sfixed(POSITION_BITS_LEFT downto -POSITION_BITS_RIGHT));
END ENTITY delay_position;

ARCHITECTURE behavioral OF delay_position IS
BEGIN
behavior : PROCESS(clk) IS
BEGIN
IF (rising_edge(clk)) THEN
    x_delayed <= x;
END IF;
END PROCESS behavior;
END behavioral;
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

LIBRARY floatfixlib;
—USE floatfixlib.math_utility_pkg.ALL;
—USE floatfixlib.fixed_pkg.ALL;

LIBRARY ieee_proposed;
USE ieee_proposed.math_utility_pkg.ALL;
USE ieee_proposed.fixed_pkg.ALL;

LIBRARY work;
USE work.pso_package.ALL;

ENTITY update_best IS
PORT (clk, reset : in std_logic;
  x : in total_position;
  fitness : in sfixed(FITNESS_BITS_LEFT downto –
    FITNESS_BITS_RIGHT);
  best_position : in total_position;
  best_fitness : in sfixed(FITNESS_BITS_LEFT downto –
    FITNESS_BITS_RIGHT);
  new_best_position : out total_position;
  new_best_fitness : out sfixed(FITNESS_BITS_LEFT downto –
    FITNESS_BITS_RIGHT));
END ENTITY update_best;

ARCHITECTURE behavioral OF update_best IS
BEGIN
  behavior : PROCESS(clk, reset) IS
    BEGIN
      IF(rising_edge(clk)) THEN
        IF reset = '1' THEN
          ...
        END IF;
      END IF;
    END PROCESS behavior;
END behavioral;
new best position(0) <= (POSITION_BITS_LEFT => '0', others => '1');
new best fitness <= (FITNESS_BITS_LEFT => '0', others => '1');
ELSE
IF fitness < best fitness THEN
—record new position/fitness
new best position <= x;
new best fitness <= fitness;
assert fitness <= best fitness
report "UPDATE_BEST: Best Fitness increased from " &
  to_string(best fitness) & " to " & to_string(fitness);
ELSE
—keep old position/fitness
new best position <= best position;
new best fitness <= best fitness;
END IF;
END IF;
END IF;
END PROCESS behavior;
END behavioral;
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

LIBRARY work;
USE work.pso_package.ALL;

LIBRARY floatfixlib;
—USE floatfixlib.math_utility_pkg.ALL;
—USE floatfixlib.fixed_pkg.ALL;

LIBRARY ieee_proposed;
USE ieee_proposed.math_utility_pkg.ALL;
USE ieee_proposed.fixed_pkg.ALL;

ENTITY update_velocity IS
PORT (clk : in std_logic;
    x : in sfixed(POSITION_BITS_LEFT downto -POSITION_BITS_RIGHT);
    v : in sfixed(VELOCITY_BITS_LEFT downto -VELOCITY_BITS_RIGHT);
    pbest, gbest : in sfixed(POSITION_BITS_LEFT downto -POSITION_BITS_RIGHT);
    rn1, rn2 : in sfixed(R_BITS_LEFT-1 downto -R_BITS_RIGHT);
    new_v : out sfixed(VELOCITY_BITS_LEFT downto -VELOCITY_BITS_RIGHT));
END ENTITY update_velocity;

ARCHITECTURE behavioral OF update_velocity IS
constant c1 : sfixed(C_BITS_LEFT downto -C_BITS_RIGHT) := to_sfixed(2.0, C_BITS_LEFT, -C_BITS_RIGHT);
constant c2 : sfixed(C_BITS_LEFT downto -C_BITS_RIGHT) := to_sfixed(2.0, C_BITS_LEFT, -C_BITS_RIGHT);
constant w : sfixed(W_BITS_LEFT downto -W_BITS_RIGHT) := to_sfixed(0.5, W_BITS_LEFT, -W_BITS_RIGHT);

BEGIN
PROCESS(clk) IS
BEGIN
IF(rising_edge(clk)) THEN

  -- report "@update_velocity: x=" & to_string(x) & " v=" & to_string(v) & " pbest=" & to_string(pbest) & " gbest=" & to_string(gbest) & " rn1=" & to_string(rn1) & " rn2=" & to_string(rn2);

  -- new_v <= c1*rn1*(x - pbest) + c2*rn2*(x - gbest);

  -- new_v <= resize((rn1+rn2)*v + c1*rn1*(x - pbest) + c2*rn2*(x - gbest), new_v'high, new_v'low);
  new_v <= resize((v sra 1) + c1*rn1*(x - pbest) + c2*rn2*(x - gbest), new_v'high, new_v'low);

  -- new_v <= resize((v sra 1) + rn1*(x - pbest) + rn2*(x - gbest), new_v'high, new_v'low);

END IF;

END PROCESS;

END behavioral;
The Matlab scripts used for the hardware NN have been selected for inclusion in this appendix for reference. All of the Matlab source code was created using Matlab 2007b. The Xilinx Simulink System Generator 10.1 toolbox was used to implement the NN hardware models. Xilinx ISE 10.1 was used to synthesize the NN hardware design, once exported from Xilinx System Generator.
nn_train.m

```matlab
clear

numInput = 1;
numHidden = 10;
numOutput = 1;

func_type = 1;

fixed_delta = 2^-12;

p = zeros(1,200);
for i = 1:length(p)
    for j = 1:numInput
        p(j,i) = 1*j*1.5*(i)/length(p);
    end
end
t = zeros(numOutput,length(p));
for i = 1:length(t)
    for j = 1:numOutput
        %t(i) = p(i)^2;
        t(j,i) = 1*j*sin(p(j,i)*4.20) + 1;
        %t(i) = p(i)*1.5;
    end
end

[p2,ps] = mapminmax(p);
[t2,ts] = mapminmax(t);

% Don't scale input/output
p2 = p;
t2 = t;

if(0)
    % figure(1)
    % plot(1:length(p),p2);
    % legend('Input');
```
figure (2)
plot (p2(:,t2));
legend (’Target’);

[trainV,val,test] = dividevec(p2,t2,0.20,0.20);

if (func_type == 0)
    net = newff(minmax(p2),[numHidden 1],’logsig’ ’purelin’);
else
    net = newff(minmax(p2),[numHidden numOutput],’tansig’ ’purelin’);
end
[net,tr] = train(net,trainV.P,trainV.T,[],[],val,test);
a2 = sim(net,p2);
a = mapminmax(’reverse’,a2,ts);
[m,b,r] = postreg(a,t);
for i = 1:numOutput
    figure(2+i)
    plot (1:length(t2),t2(i,:),1:length(a2),a2(i,:));
    legend (’Target’, ’Output’);
end
nn_load.m

```matlab
1 tmp = net.IW(1);
2 IW = tmp{1,1};
3 tmp = net.b(1);
4 BI = tmp{1,1};
5
6 OW = net.LW{2,1};
7 BH = net.b{2,1};
8
9 W = zeros(numHidden, numInput+1);
10 for h = 1:numHidden
11   for i = 1:numInput
12     W(h,i) = IW(h,i);
13   end
14   W(h,numInput+1) = BI(h);
15 end
16
17 V = zeros(numOutput, numHidden+1);
18 for o = 1:numOutput
19   for h = 1:numHidden
20     V(o,h) = OW(o,h);
21   end
22   V(o,numHidden+1) = BH(o);
23 end
24
25 %NN_Input = zeros(length(p2)*(numInput+2),2);
26 NN_Input = zeros(length(p2)*(numHidden+2),numInput+1);
27 for p = 1:length(p2)
28   NN_Input((p-1)*(numHidden+2)+1:(p-1)*(numHidden+2)+(numHidden+2),2:
29     numInput+1) = ones((p-1)*(numHidden+2)+(numHidden+2)-(p-1)*(
30       numHidden+2),1)*p2(:,p)';
31 end
32 NN_Input(length(NN_Input)+1:2*length(NN_Input),:) = NN_Input(:,:);
33 NN_Input(:,1) = 1:length(NN_Input);
34
35 NN_Input1(1:length(NN_Input),1) = NN_Input(:,1);
36 NN_Input1(1:length(NN_Input),2) = NN_Input(:,2);
37 if(numInput > 1)
38```

NN_Input2(1:length(NN_Input),1) = NN_Input(:,1);
NN_Input2(1:length(NN_Input),2) = NN_Input(:,3);
end
if(numInput > 2)
NN_Input3(1:length(NN_Input),1) = NN_Input(:,1);
NN_Input3(1:length(NN_Input),2) = NN_Input(:,4);
end
if(numInput > 3)
NN_Input4(1:length(NN_Input),1) = NN_Input(:,1);
NN_Input4(1:length(NN_Input),2) = NN_Input(:,5);
end

k = 8;
[LUT_C1, LUT_C2] = generate_act_lut(k,func_type);
sample_length = length(NN_Input);
NN_Reset(:,1) = 1:length(NN_Input);
NN_Reset(:,2) = 1;
NN_Reset(1,2) = 0;
NN_Write(:,1) = 1:length(NN_Input);
NN_Write(:,2) = 0;
NN_Write_DATA(:,1) = 1:length(NN_Input);
NN_Write_DATA(:,2) = 0;
NN_Write_ADDR(:,1) = 1:length(NN_Input);
NN_Write_ADDR(:,2) = 0;

for i = 1:length(V)
  a = fi(V(i), 1, 18, 12);
disp(['num2str(i),'' = 0x',dec2hex(bin2dec(a.bin))]);
end
test_nn_model.m

tmp = net.IW(1);
IW = tmp{1,1};
tmp = net.b(1);
BI = tmp{1,1};

OW = net.LW{2,1};
BH = net.b{2,1};
clear O;

X = NN_Input(:,2:numInput+1)';

for p = 1:length(X)
    %hidden
    a = zeros(numHidden,1);
    for h = 1:numHidden
        for i = 1:numInput
            a(h) = a(h) + X(i,p)*IW(h,i);
        end
    end
    a = a + BI;

    if(func_type == 0)
        dh = logsig(a);
    else
        dh = tansig(a);
    end

    hidden_preoutput(p,:) = a(:);
    hidden_output(p,:) = dh(:);

    %output
    a = zeros(numOutput,1);
    for o = 1:numOutput
        for h = 1:numHidden
            a(o) = a(o) + dh(h)*OW(o,h);
        end
    end
38     end
39     a = a + BH;
40
41     %do = logsig(a);
42     do = a;
43
44     output_output(p,:) = do(:);
45
46     O(p,:) = do;
47 end
48
49 if(1)
50     for n = 1:length(hidden_output)
51         for i = 1:5
52             %SIM_NN_Hidden(n,i) = SIM_NN_Hidden_Output.signals(i).values((n-1)*3+1+11);
53             SIM_NN_Hidden(n,i) = SIM_NN_Hidden_Output1.signals(i).values(n);
54         end
55     end
56     if(numHidden > 5)
57         for n = 1:length(hidden_output)
58             for i = 1:5
59                 %SIM_NN_Hidden(n,i) = SIM_NN_Hidden_Output.signals(i).values((n-1)*3+1+11);
60                 SIM_NN_Hidden(n,i+5) = SIM_NN_Hidden_Output2.signals(i).values(n);
61             end
62         end
63     end
64     end
65
66     for i = 1:numHidden/5
67         figure(4+i)
%plot(1:length(hidden_output),hidden_output(:,1),1:length(hidden_output),hidden_output(:,2),1:length(hidden_output),hidden_output(:,3))
%legend('1','2','3')
for sp = 1:5
    subplot(5,1,sp);
    delay = numHidden+9;
    %plot(1:length(hidden_output),hidden_output(:,(i-1)*5+sp),1:length(hidden_output),SIM_NN_Hidden(:,(i-1)*5+sp));
    plot(1:length(hidden_output)-delay,hidden_output(1:length(hidden_output)-delay,(i-1)*5+sp),1:length(hidden_output)-delay,SIM_NN_Hidden(delay:length(SIM_NN_Hidden)-1,(i-1)*5+sp));
    axis([1,length(hidden_output),-1.5,1.5]);
end
legend('Floating-Point','Hardware');
end

SIM_NN_Output = zeros(1:length(O),numOutput);
for n = 1:length(hidden_output)
    for i = 1:numOutput
        %SIM_NN_Output(n,i) = SIM_NN_Output_Output.signals(i).values((n-1)*3+1+19);
        SIM_NN_Output(n,i) = SIM_NN_Output_Output.signals(i).values(n);
    end
end

for i = 1:numOutput
    figure(10+i)
    %plot(1:length(O),O,1:length(O),SIM_NN_Output(:,1));
    delay = numHidden+9+numHidden+5;
    plot(1:length(O)-delay,O(1:length(O)-delay,i),1:length(O)-delay,SIM_NN_Output(delay:length(SIM_NN_Output)-1,i));
    legend('Floating-Point','Hardware');
end
end


VITA

Parviz Michael Palangpour is the son of Parviz and Elaine Palangpour. He was born in Columbia, Missouri on July 14, 1983. He graduated with a Bachelor of Science in Computer Engineering from the Missouri University of Science and Technology (then known as the University of Missouri-Rolla) in 2007. Immediately after obtaining his bachelor’s degree, he began his graduate studies at the Missouri University of Science and Technology in Computer Engineering.