Spring 2010

FPGA implementation of PSO algorithm and neural networks

Parviz Palangpour

Department:

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

LIST OF ILLUSTRATIONS

LIST OF TABLES

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 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 μs.

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 particles position. Each particles velocity is in influenced by the particles 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 particles velocity from the last iteration, the difference between the particles known best position and the particles current position, and the difference between the swarms best known position and the particles current position. The latter two terms are each multiplied by a uniform random number in [0,1] to randomly vary the in influence of each term, as well as an acceleration coefficient to scale and balance the in influence of each term. The best position each particle attained is stored in the vector p_i, also known as *pbest*, while the best position attained by any particle in the population is stored in the vector p_g, also known as *gbest*. The velocity vector v_i for each particle is then updated:

$$v_i^{t+1} = w \cdot v_i^t + c_1 r_1 \cdot (p_i^t - x_i^t) + c_2 r_2 \cdot (p_g^t - x_i^t) \tag{2.1}$$

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

2.2.

$$v_i^{t+1} = \begin{cases} v_{Max} & \text{if } v_i^{t+1} > V_{max} \\ v_{Min} & \text{if } v_i^{t+1} < V_{min} \\ v_i^{t+1} & \text{else} \end{cases} \tag{2.2}$$

The position of each particle is then updated using the new velocities in Equation 2.3.

$$x_i^{t+1} = x_i^t + v_i^{t+1} \tag{2.3}$$

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

$$x_i^{t+1} = \begin{cases} x_{Max} & \text{if } x_i^{t+1} > X_{max} \\ x_{Min} & \text{if } x_i^{t+1} < X_{min} \\ x_i^{t+1} & \text{else} \end{cases} \tag{2.4}$$

The psuedocode for PSO is listed in Algorithm 1.

Algorithm 1 PSO

Initialize the positions, velocities, *pbest* and *gbest* values
repeat
  for $i = 1$ to NUM_PARTICLES **do**
    if $f(x_i) \leq f(p_i)$ **then**
      $p_i \leftarrow x_i$
      if $f(x_i) \leq f(p_g)$ **then**
        $p_g \leftarrow x_i$
      end if
    end if
    for $j = 1$ to NUM_DIMENSIONS **do**
      $v_{ij}^{t+1} \leftarrow w \cdot v_{ij}^t + c_1 r_1 \cdot (x_{ij}^t - p_{ij}^t) + c_2 r_2 \cdot (x_{ij}^t - p_{gj}^t)$
      $v_{ij}^{t+1} \in (V_{min}, V_{max})$
      $x_{ij}^{t+1} \leftarrow x_{ij}^t + v_{ij}^{t+1}$
      $x_{ij}^{t+1} \in (X_{min}, X_{max})$
    end for
  end for
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 *pbest* and *gbest* 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 *pbest* values, p_i and $f(p_i)$ are also fetched from *pbest* memory. In the third stage, the *gbest* and *pbest* are updated. Both the update *gbest* and *pbest* modules are passed x_i and $f(x_i)$. In addition, the update *pbest* module is passed p_i and $f(p_i)$ while the update *gbest* 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 *gbest* and *pbest*. 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 *pbest* 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, *pbest* 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.

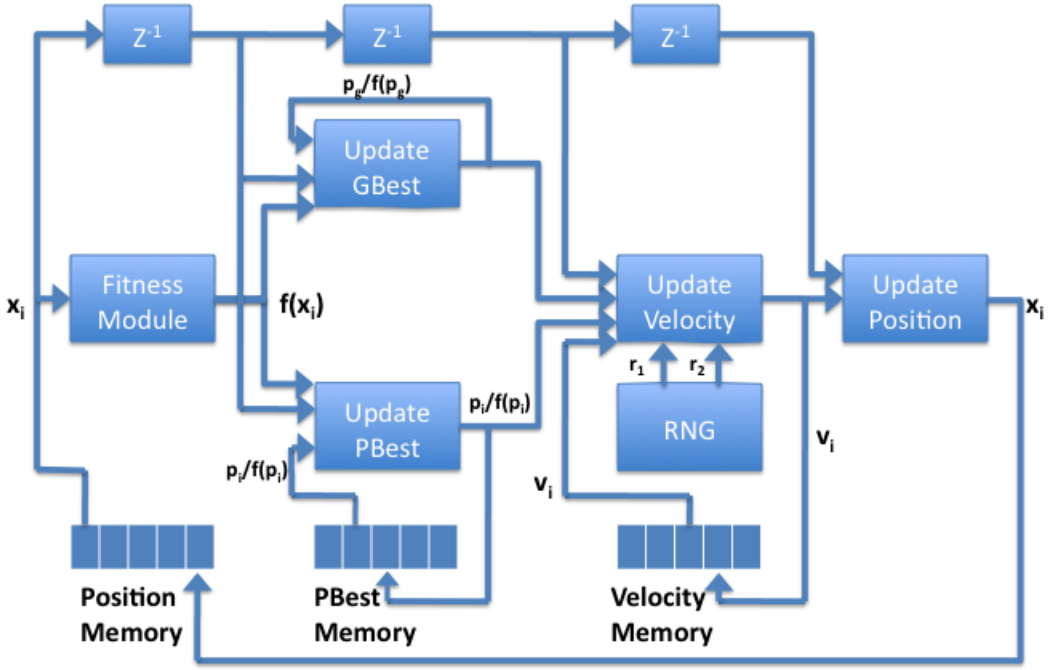


Figure 2.1. PSO hardware implementation with 6-stage pipeline.

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^t. 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^t 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

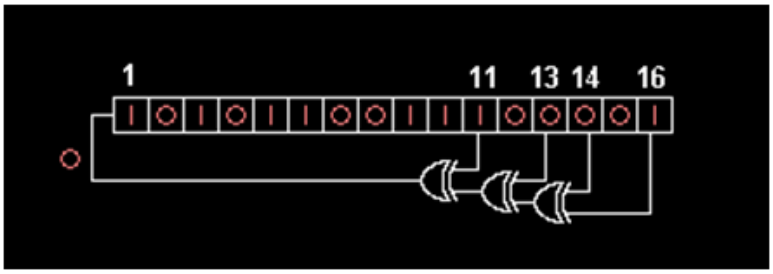


Figure 2.2. An example of a linear-feedback shift register.

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.

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.

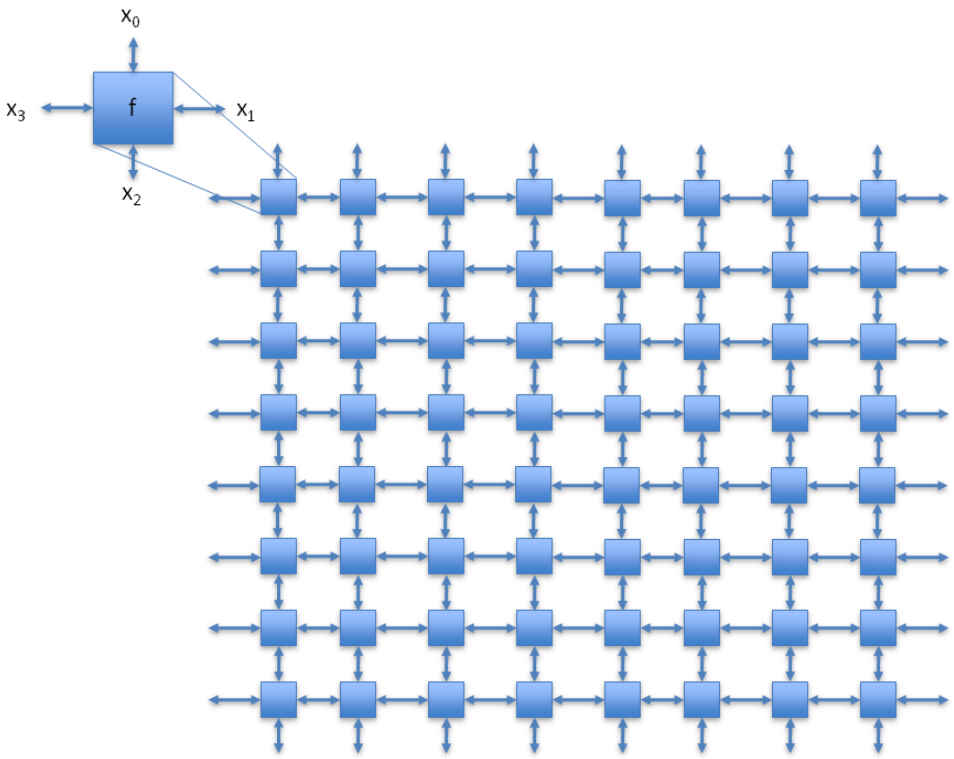


Figure 2.3. The structure of the neighborhood-of-four PRNG.

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.

$x_0x_1x_2x_3$	f
0000	0
0001	1
0010	1
0011	1
0100	0
0101	0
0110	1
0111	0
1000	1
1001	1
1010	0
1011	1
1100	1
1101	0
1110	0
1111	0

PRNG. The *Init-Particles* state cycles through the particles and sets each particles 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 particles 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

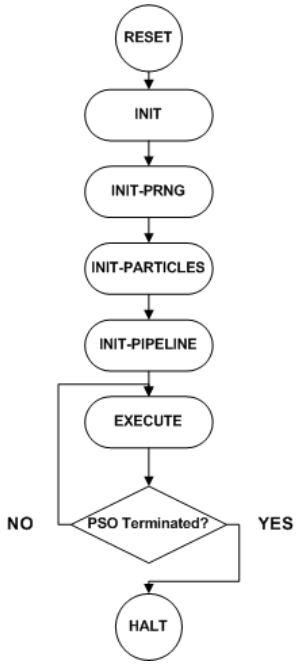


Figure 2.4. The hardware PSO execution flowchart.

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:

$$f(x) = \sum_{i=1}^{n} x_i^2 \tag{2.5}$$

The second benchmark problem is the Rosenbrock function:

$$f(x) = \sum_{i=1}^{n-1} 100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2 \tag{2.6}$$

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.

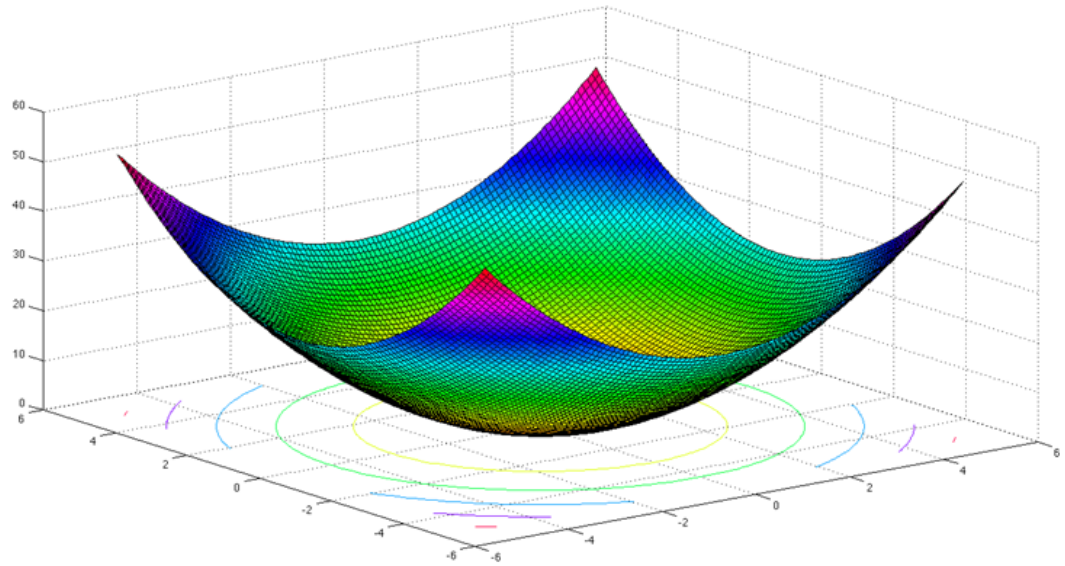


Figure 2.5. A two-dimensional view of the sphere benchmark problem fitness surface.

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

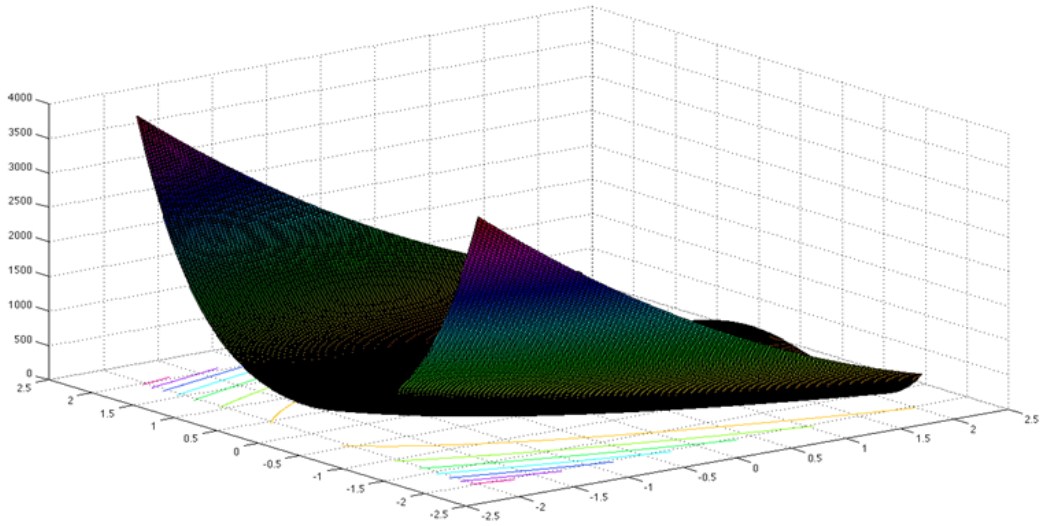


Figure 2.6. A two-dimensional view of the Rosenbrock benchmark problem fitness surface.

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

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

where f_{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.

Problem	n	Software PSO Fitness	Hardware PSO Fitness
Sphere	1	0.00	0.00
	5	0.00	0.00
	10	0.073	0.001
Rosenbrock	2	0.00	0.00
	5	0.044	0.085
	10	8.081	8.615

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

Problem	n	Software PSO Time	Hardware PSO Time
Sphere	1	2.07 sec.	200 μs
	5	5.79 sec.	338 μs
	10	10.99 sec.	392 μs
Rosenbrock	2	2.14 sec.	344 μs
	5	5.95 sec.	444 μs
	10	10.91 sec.	800 μs

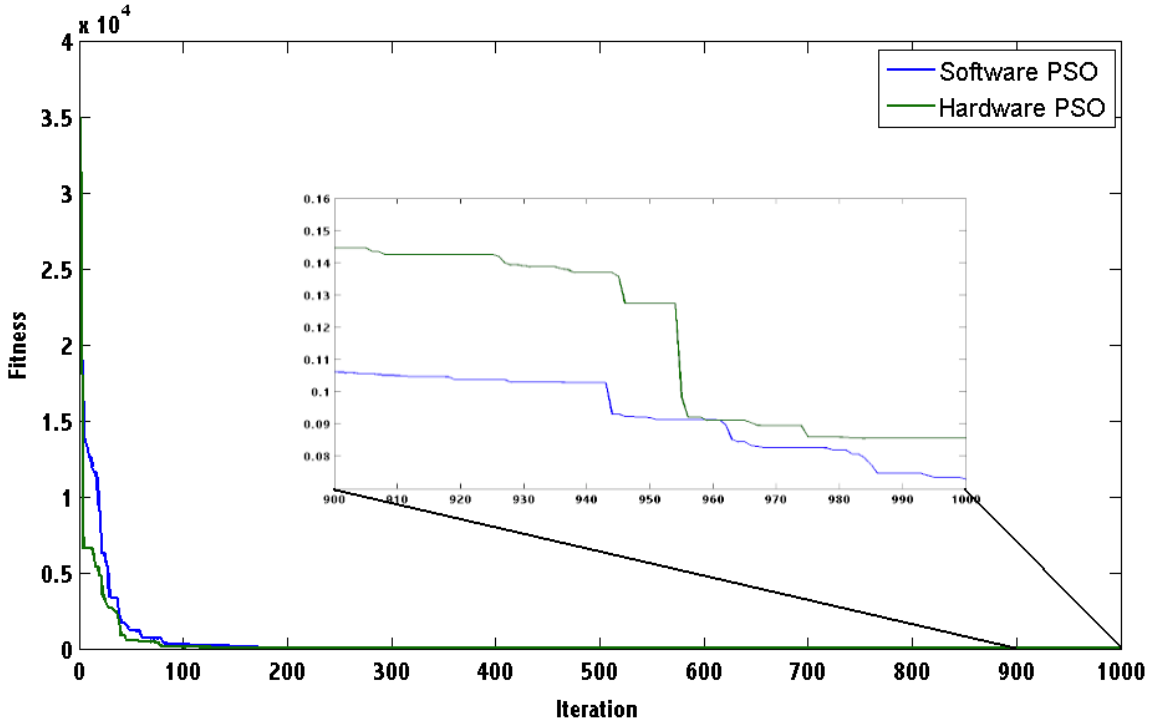


Figure 2.7. The hardware PSO compared with a software PSO for the sphere function of 10 dimensions.

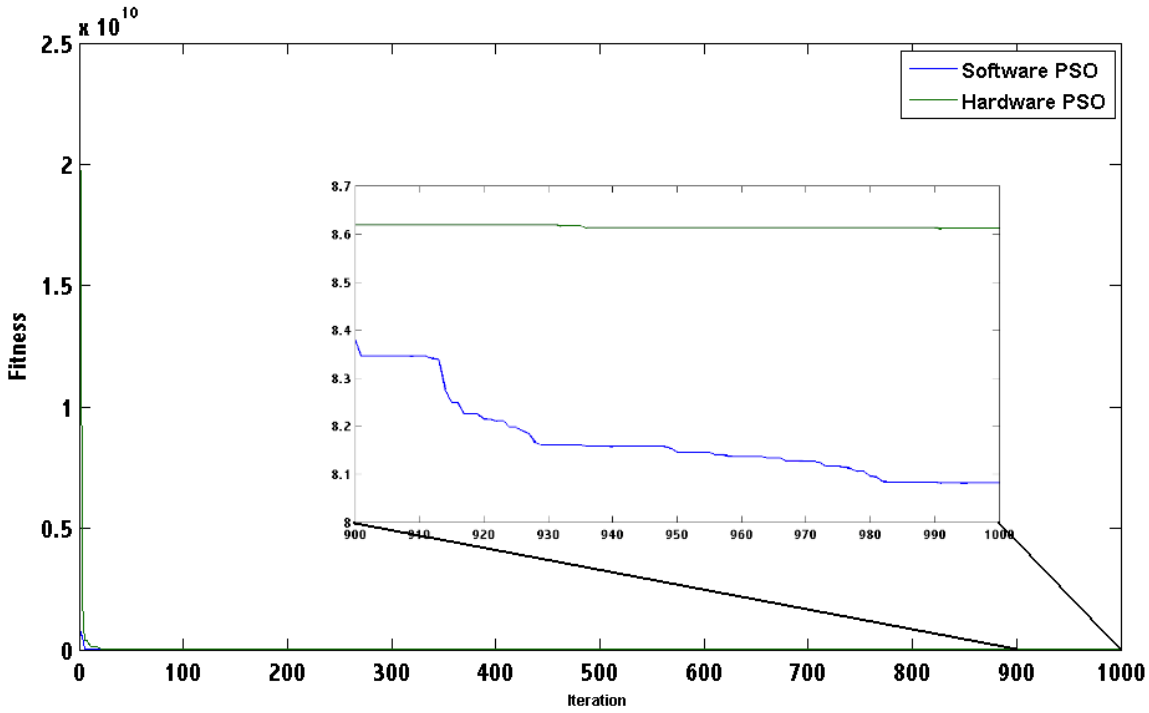


Figure 2.8. The hardware PSO compared with a software PSO for the Rosenbrock function of 10 dimensions.

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

Problem	n	Logic Utilization	LUTs	Clock
Sphere	1	1118	1523	100 MHz
	5	4744	10631	59 MHz
	10	9249	20873	51 MHz
Rosenbrock	2	2244	5332	58 MHz
	5	5228	12513	45 MHz
	10	9940	25750	25 MHz

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:

$$\begin{aligned} a_i &= \sum_{j=1}^{nI} w_{i,j} x_{j,i}, i = 1, ..., nH \\ d_i &= \frac{e^{a_i} - e^{-a_i}}{e^{a_i} + e^{-a_i}}, i = 1, ..., nH \\ y &= \sum_{i=1}^{nH} v_i d_i \end{aligned} \tag{3.1}$$

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

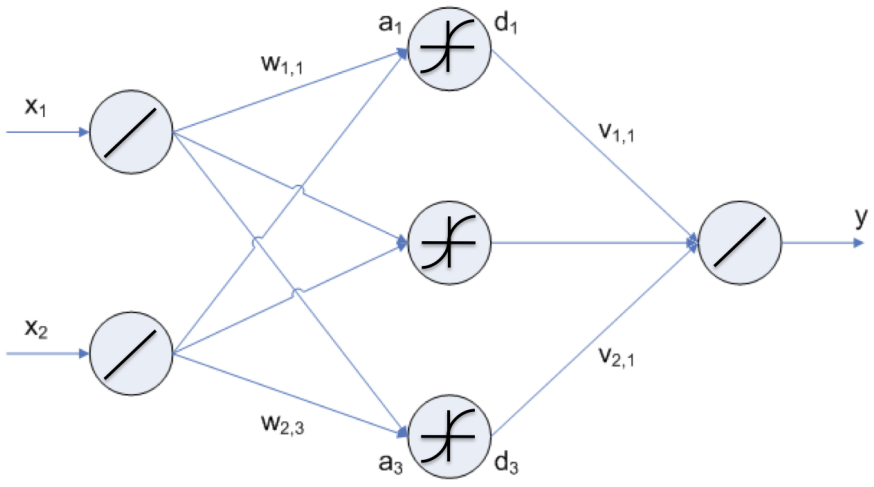


Figure 3.1. An NN with an input, hidden and output layer.

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 need 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:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3.2}$$

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:

$$a = \sum_{i=0}^{N} x_i \cdot w_i \tag{3.3}$$

where x_i is the output of the ith 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.

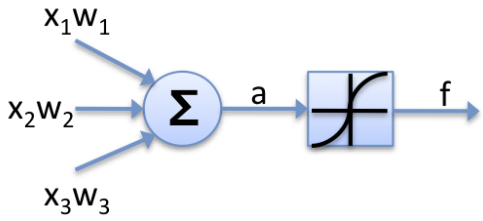


Figure 3.2. A diagram of the neuron model.

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.

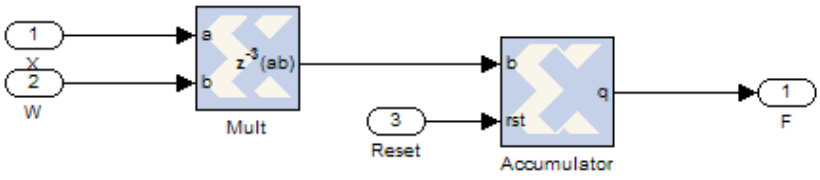


Figure 3.3. MAC hardware subsystem.

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:

$$f(x) = \alpha_2 + c_2 * x \tag{3.4}$$

where x is the activation of the neuron and α_1 and α_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 α_1 and α_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 α_1 and α_2 for a given region the following system must be solved:

$$\alpha_1 + \alpha_2 * L = \frac{e^U - e^{-U}}{e^U + e^{-U}} \tag{3.5}$$

$$\alpha_1 + \alpha_2 * U = \frac{e^L - e^{-L}}{e^L + e^{-L}} \tag{3.6}$$

where L is lower boundary in the subinterval and U is the upper boundary in the subinterval. Solving for α_1 and α_2 using Matlab's Symbolic Math toolbox results in:

$$\alpha_1 = \frac{L * e_U - U * e_U - U * e_U * e_L + L * e_U * e_L + U + e_L * L - L + U * e_L}{-U - U * e_U - U * e_L - U * e_U * e_L + L + L * e_U + e_L * L + L * e_U * e_L} \tag{3.7}$$

$$\alpha_2 = \frac{2 * (e_U - e_L)}{-U - U * e_U - U * e_L - U * e_U * e_L + L + L * e_U + e_L * L + L * e_U * e_L} \tag{3.8}$$

where e_U and e_L are constants which equal e^{-2*U} and e^{-2*L}, respectively. The values for α_1 and α_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 $x \in [L, U]$.

L	U	α_1	α_2
6	7	0.99992	1.0625e-005
5	6	0.99952	7.8507e-005
4	5	0.99701	0.0005799
3	4	0.98223	0.0042745
2	3	0.90197	0.031027
1	2	0.55916	0.20243
0	1	-2.2204e-016	0.76159
-1	0	0	0.76159
-2	-1	-0.55916	0.20243
-3	-2	-0.90197	0.031027
-4	-3	-0.98223	0.0042745
-5	-4	-0.99701	0.0005799
-6	-5	-0.99952	7.8507e-005
-7	-6	-0.99992	1.0625e-005
-8	-7	-0.99999	1.438e-006

The hardware implementation of the LUTAF is constructed from three main elements, the memory element which contains the linearized constants α_1 and α_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 α_1 and α_2 by using the k bits as the address to the memory elements. The values for α_2 and x are passed to a multiplier while α_1 is latched (delayed to match the propagation of the signal $(\alpha_2 \times x)$). Finally, the terms (α_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

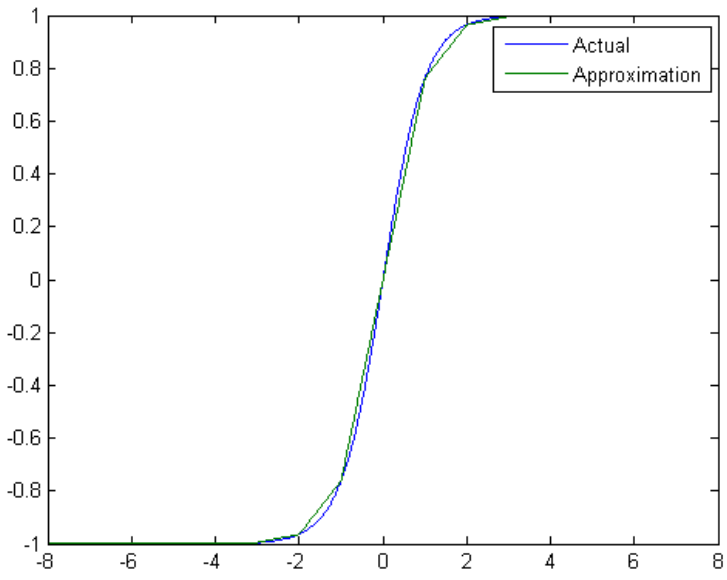


Figure 3.4. LUTAF approximation.

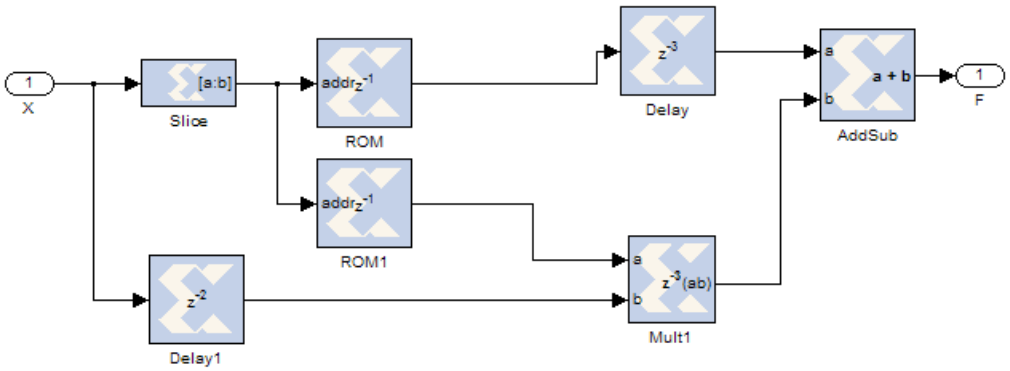


Figure 3.5. LUTAF approximation subsystem.

subsystem can be seen in Figure 3.6. The hardware neuron receives the values x, *w_addr*, *reset* and *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_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 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 latch *my_latch* is responsible for latching the result of the MAC when its valid, based on the 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.

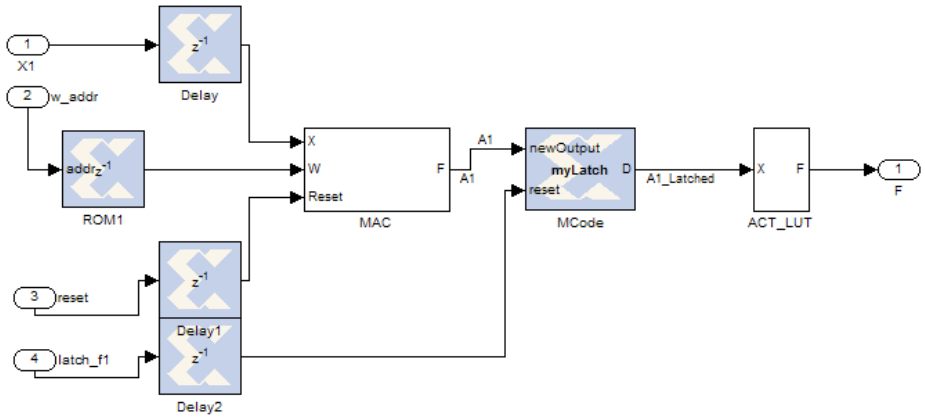


Figure 3.6. The complete hardware neuron.

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 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.

```
function [X, W_Addr, ResetMac, Latch_F] = neuronContro_1_3(ResetSeq, x1, x2)
init = 0;
persistent state, state = xl_state(init, {xlUnsigned, 3, 0});
W_Addr_Width = 2;
switch double(state)
    case 0
        X = xfix({xlSigned,18,14},0);
        W_Addr = xfix({xlUnsigned,W_Addr_Width,0},2);
        ResetMac = xfix({xlBoolean},0);
        Latch_F = xfix({xlBoolean},0);
    case 1
        X = x1;
        W_Addr = xfix({xlUnsigned,W_Addr_Width,0},0);
        ResetMac = xfix({xlBoolean},1);
        Latch_F = xfix({xlBoolean},1);
    case 2
        X = x2;
        W_Addr = xfix({xlUnsigned,W_Addr_Width,0},1);
        ResetMac = xfix({xlBoolean},0);
        Latch_F = xfix({xlBoolean},0);
    case 3
        X = 0;
        W_Addr = xfix({xlUnsigned,W_Addr_Width,0},2);
        ResetMac = xfix({xlBoolean},0);
        Latch_F = xfix({xlBoolean},0);
    case 4
        X = 0;
        W_Addr = xfix({xlUnsigned,W_Addr_Width,0},2);
        ResetMac = xfix({xlBoolean},0);
        Latch_F = xfix({xlBoolean},0);
    otherwise
        X = 0;
        W_Addr = xfix({xlUnsigned,W_Addr_Width,0},2);
        ResetMac = xfix({xlBoolean},1);
        Latch_F = xfix({xlBoolean},0);
end
if(state == 4)
    state = 0;
else
    state = state + 1;
end
```

Figure 3.7. The LCB finite state machine (2×3 Layer).

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

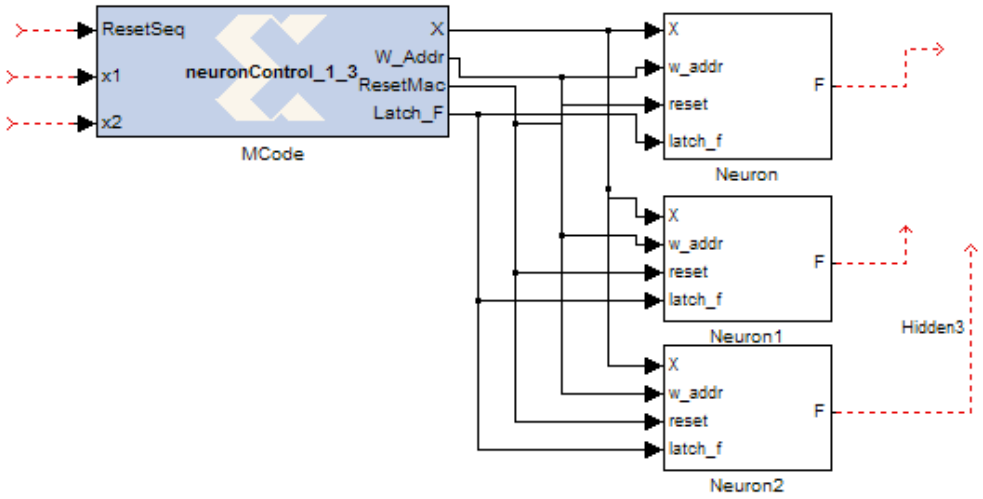


Figure 3.8. A LCB controlling three hardware neurons (2×3 Layer).

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.

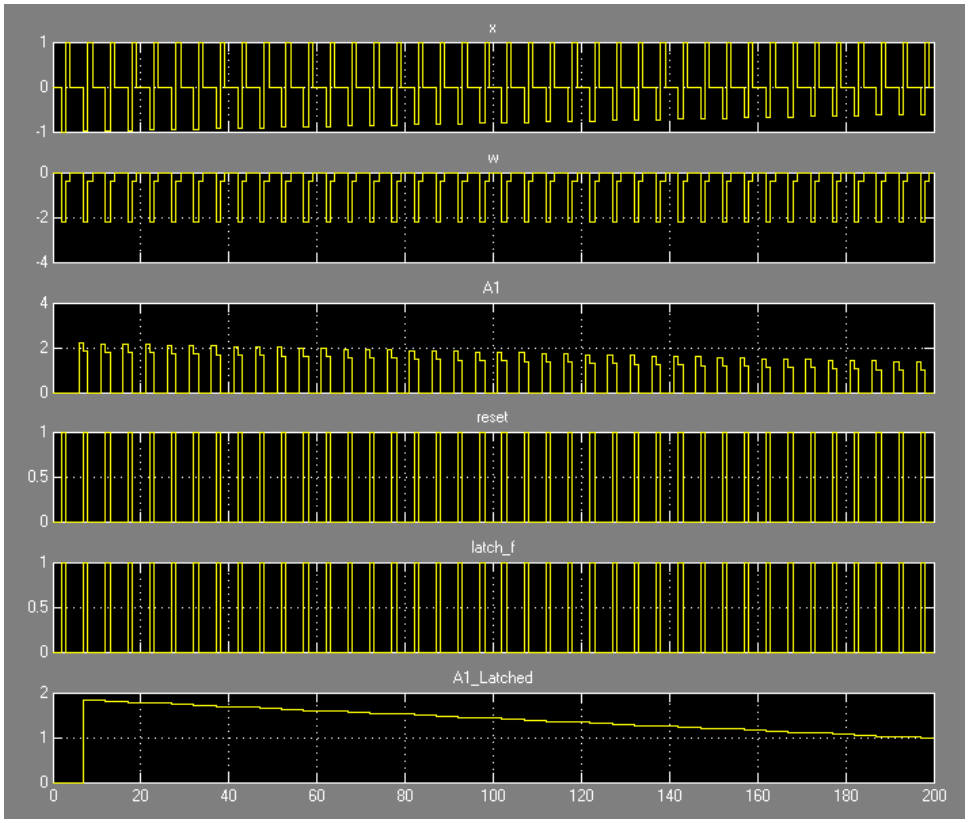


Figure 3.9. A hardware neuron being controlled by a LCB.

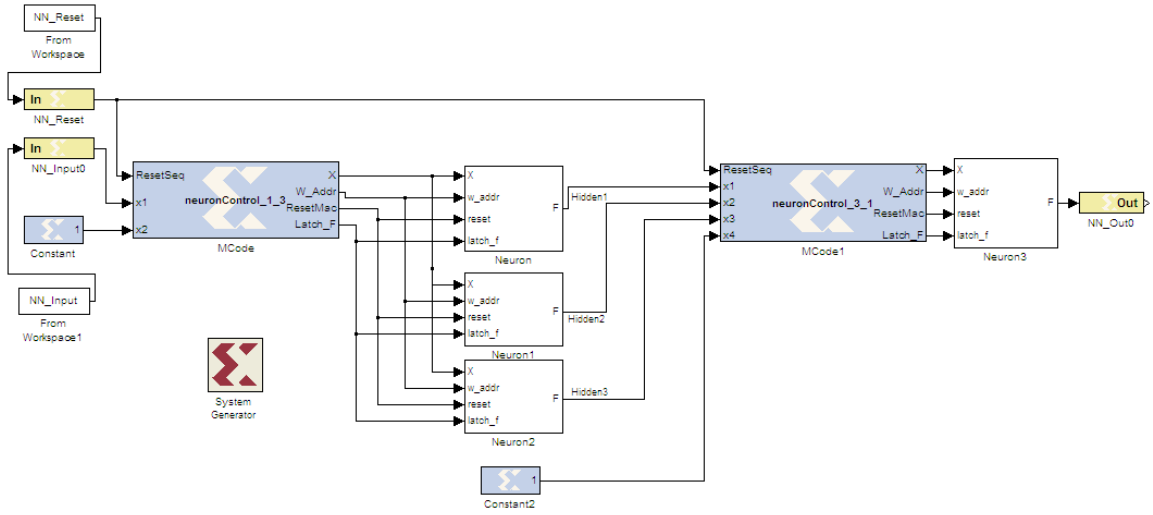


Figure 3.10. A FPGA-based NN (of size $1 \times 3 \times 1$).

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:

$$f(x) = \sin(5 * x) \tag{3.9}$$

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-

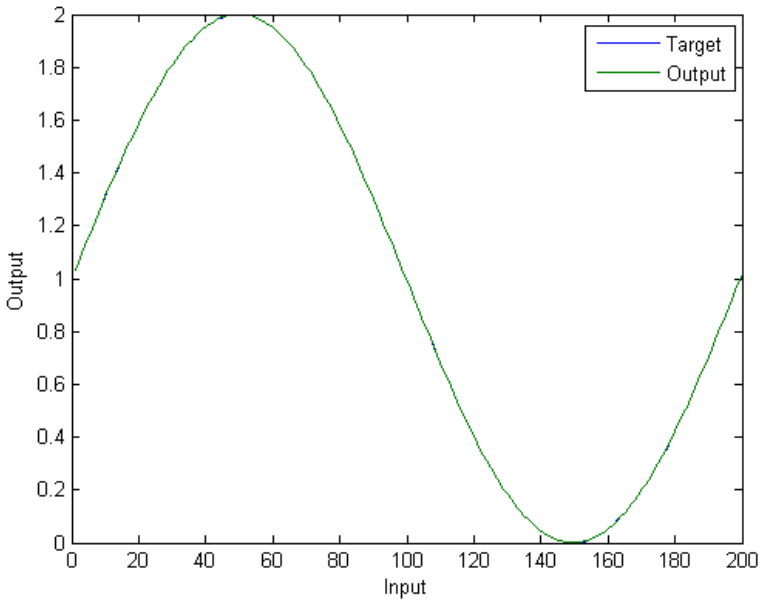


Figure 3.11. The trained NN for approximating a sin function.

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[1] 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 α_1 and α_2 using the method described in Section 3.4. In order for LUTAF constants α_1 and α_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.

[1] 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.

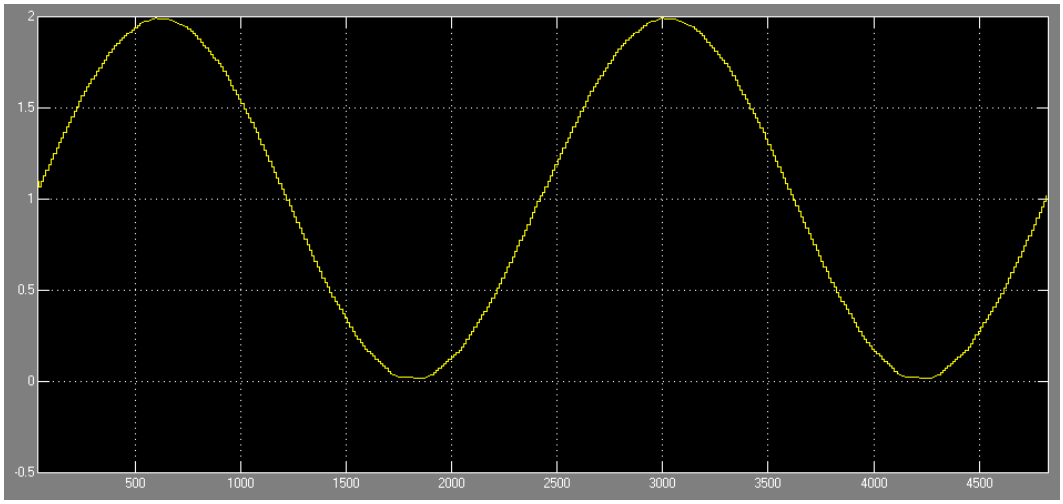
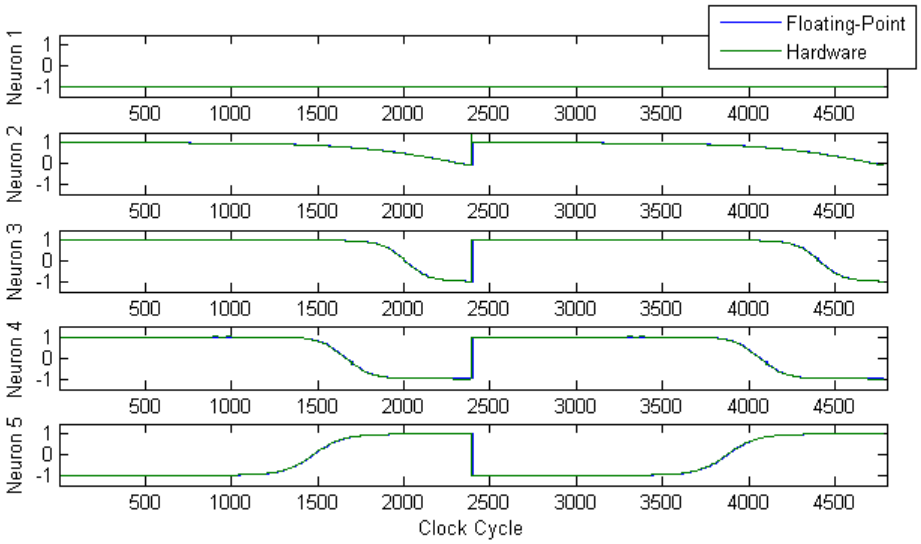
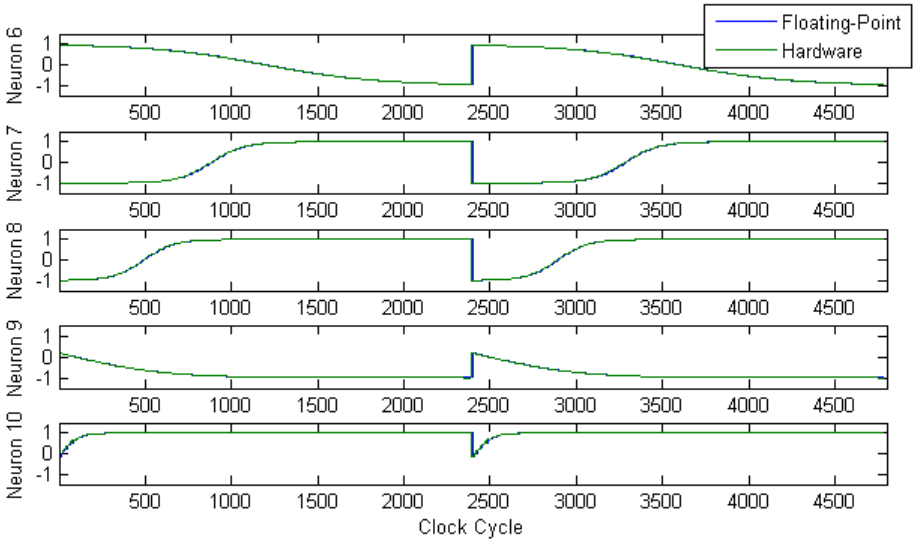


Figure 3.12. The output of the Simulink simulation of the FPGA-based NN.

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.



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



(b) The output of hidden neurons 6-10.

Figure 3.13. Comparing the output of hidden neurons of the FPGA-based NN and Matlab-based NN.

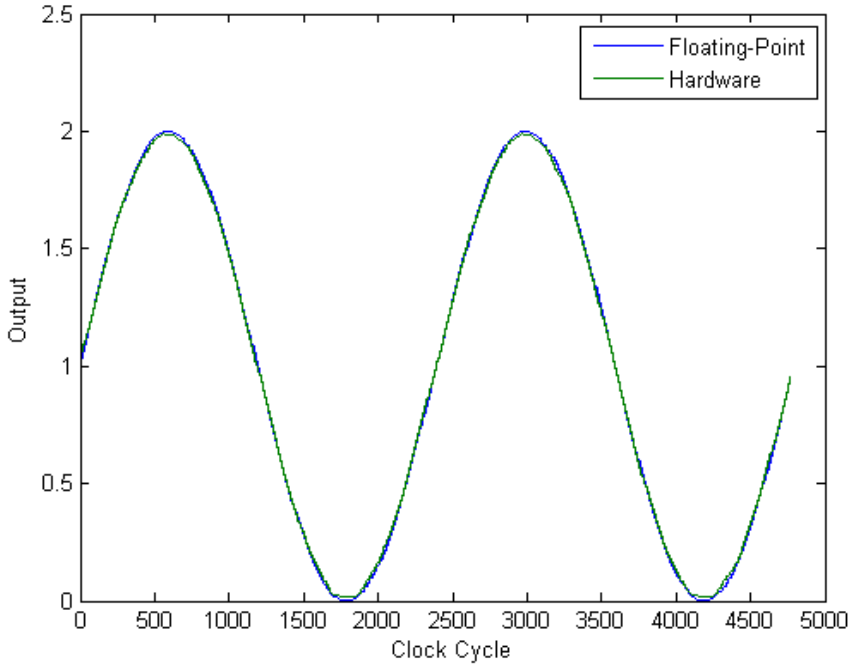


Figure 3.14. Comparing the output of the FPGA-based NN and Matlab-based NN.

Device Utilization Summary				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	1,388	27,392	5%	
Number of 4 input LUTs	1,701	27,392	6%	
Logic Distribution				
Number of occupied Slices	1,245	13,696	9%	
Number of Slices containing only related logic	1,245	1,245	100%	
Number of Slices containing unrelated logic	0	1,245	0%	
Total Number of 4 input LUTs	1,701	27,392	6%	
Number used as logic	1,312			
Number used as Shift registers	389			
Number of bonded IOBs	36	416	8%	
Number of RAMB16s	20	136	14%	
Number of MULT18X18s	21	136	15%	
Number of BUFGMUXs	1	16	6%	
Number of RPM macros	21			

Figure 3.15. The hardware resources used for the $1 \times 10 \times 1$ NN on the Xilinx Virtex-II Pro FPGA.

4 CONCLUSIONS AND FUTURE WORK

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

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_ARITH.ALL;
4  use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6  LIBRARY ieee_proposed;
7  USE ieee_proposed.math_utility_pkg.ALL;
8  USE ieee_proposed.fixed_pkg.ALL;
9
10 USE WORK.pso_package.ALL;
11
12 ---- Uncomment the following library declaration if instantiating
13 ---- any Xilinx primitives in this code.
14 --library UNISIM;
15 --use UNISIM.VComponents.all;
16
17 entity PSO is
18   PORT(
19     SYSTEM_CLK : in std_logic;
20     SWITCHES : in STD_LOGIC_VECTOR(1 downto 0);
21     LEDS : out STD_LOGIC_VECTOR(1 downto 0);
22     GBEST : out STD_LOGIC_VECTOR(7 downto 0)
23     );
24 end PSO;
25
26 architecture Behavioral of PSO is
27
28   signal DCM_IBUFG_OUT, DCM_CLK0_OUT, DCM_LOCKED : std_logic;
29   signal clk : std_logic;
30   signal gbest_position : total_position;
31   signal gbest_fitness : sfixed(FITNESS_BITS_LEFT downto -
       FITNESS_BITS_RIGHT);
32   signal PSO_reset, PSO_finished : std_logic;
33   signal iteration : unsigned(ITERATION_COUNT_BITS-1 downto 0);
34
35   signal gbest_fitness_tmp : std_logic_vector(FITNESS_BITS_LEFT +
       FITNESS_BITS_RIGHT downto 0);
```

```
36
37   COMPONENT dcm0
38   PORT(
39      CLKIN_IN : IN std_logic;
40      CLKDV_OUT : OUT std_logic;
41      CLKIN_IBUFG_OUT : OUT std_logic;
42      CLK0_OUT : OUT std_logic;
43      LOCKED_OUT : OUT std_logic
44      );
45   END COMPONENT;
46
47   COMPONENT pso_top
48   PORT (
49      clk, reset : in std_logic;
50      gbest : total_position;
51      gbest_e : out sfixed(FITNESS_BITS_LEFT downto -FITNESS_BITS_RIGHT);
52      iteration : out unsigned(ITERATION_COUNT_BITS-1 downto 0);
53      finished : out std_logic);
54   END COMPONENT;
55
56 begin
57
58   Inst_dcm0: dcm0 PORT MAP(
59      CLKIN_IN => SYSTEM_CLK,
60      CLKDV_OUT => clk,
61      CLKIN_IBUFG_OUT => DCM_IBUFG_OUT,
62      CLK0_OUT => DCM_CLK0_OUT,
63      LOCKED_OUT => DCM_LOCKED
64   );
65
66   Inst_pso_top0: pso_top PORT MAP(
67      clk => clk,
68      reset => PSO_reset,
69      gbest => gbest_position,
70      gbest_e => gbest_fitness,
71      iteration => iteration,
72      finished => PSO_finished
73   );
74
```

```
75    PSO_reset <= SWITCHES(0);
76
77    LEDS(0) <= NOT PSO_reset;
78    LEDS(1) <= NOT PSO_finished;
79
80    gbest_fitness_tmp <= to_slv(gbest_fitness);
81
82    --GBEST <= gbest_position(7 downto 0);
83    GBEST <= gbest_fitness_tmp(7 downto 0);
84
85 end Behavioral;
```

pso_package.vhd

```
1  --LIBRARY floatfixlib;
2  --USE floatfixlib.math_utility_pkg.ALL;
3  --USE floatfixlib.fixed_pkg.ALL;
4
5  LIBRARY ieee_proposed;
6  USE ieee_proposed.math_utility_pkg.ALL;
7  USE ieee_proposed.fixed_pkg.ALL;
8
9  PACKAGE pso_package is
10   constant NUM_PARTICLES : natural := 20;
11   constant RN_BITS : natural := 64;
12   constant NUM_DIMENSIONS : natural := 10;
13
14   constant POSITION_BITS_LEFT : natural := 7;
15   constant POSITION_BITS_RIGHT : natural := 9;
16
17   --constant FITNESS_BITS_LEFT : natural := POSITION_BITS_LEFT;
18   --constant FITNESS_BITS_RIGHT : natural := POSITION_BITS_RIGHT;
19   --constant FITNESS_BITS_LEFT : natural := POSITION_BITS_LEFT +
         POSITION_BITS_LEFT + NUM_DIMENSIONS;
20   --constant FITNESS_BITS_RIGHT : natural := POSITION_BITS_RIGHT +
         POSITION_BITS_RIGHT;
21   constant FITNESS_BITS_LEFT : natural := 43+NUM_DIMENSIONS + 6;
22   constant FITNESS_BITS_RIGHT : natural := 36;
23
24   constant C_BITS_LEFT : natural := 2;
25   constant C_BITS_RIGHT : natural := 2;
26
27   constant R_BITS_LEFT : natural := 0;
28   constant R_BITS_RIGHT : natural := 9;
29
30   constant W_BITS_LEFT : natural := 0;
31   constant W_BITS_RIGHT : natural := 3;
32
33   --constant VELOCITY_BITS_LEFT : natural := POSITION_BITS_LEFT+1+
         R_BITS_LEFT+C_BITS_LEFT+2;
```

```
34    --constant VELOCITY_BITS_RIGHT : natural := POSITION_BITS_RIGHT+
          R_BITS_RIGHT+C_BITS_RIGHT;
35    constant VELOCITY_BITS_LEFT : natural := POSITION_BITS_LEFT;
36    constant VELOCITY_BITS_RIGHT : natural := POSITION_BITS_RIGHT+1;
37
38    constant PARTICLE_COUNT_BITS : natural := 5;
39    constant DIMENSION_COUNT_BITS : natural := 4;
40
41    constant ITERATION_COUNT_BITS : natural := 10;
42    constant MAX_ITERATIONS : natural := 1000;
43
44    type total_position is array (0 to NUM_DIMENSIONS-1) of sfixed(
          POSITION_BITS_LEFT downto -POSITION_BITS_RIGHT);
45    type total_velocity is array (0 to NUM_DIMENSIONS-1) of sfixed(
          VELOCITY_BITS_LEFT downto -VELOCITY_BITS_RIGHT);
46
47 END pso_package;
```

pso_top.vhd

```
1  USE WORK.pso_package.ALL;
2  LIBRARY IEEE;
3  USE IEEE.std_logic_1164.ALL;
4  USE IEEE.numeric_std.ALL;
5
6  --LIBRARY floatfixlib;
7  --USE floatfixlib.math_utility_pkg.ALL;
8  --USE floatfixlib.fixed_pkg.ALL;
9
10 LIBRARY ieee_proposed;
11 USE ieee_proposed.math_utility_pkg.ALL;
12 USE ieee_proposed.fixed_pkg.ALL;
13
14 ENTITY pso_top IS
15   PORT (clk, reset : in std_logic;
16         gbest : out total_position;
17         gbest_e : out sfixed(FITNESS_BITS_LEFT downto -
               FITNESS_BITS_RIGHT);
18         iteration : out unsigned(ITERATION_COUNT_BITS-1 downto 0);
19         finished : out std_logic);
20 END ENTITY pso_top;
21
22 ARCHITECTURE behavioral OF pso_top IS
23
24     -- Reset Signals
25     signal reset_modules : std_logic;
26     -- Signals to RNG
27     signal reset_rng, rng_enable : std_logic;
28     -- Signals from RNG
29     signal rn : std_logic_vector(RN_BITS-1 downto 0);
30     -- Signals to Fitness module
31     signal position_to_eval : total_position;
32     -- Signals from Fitness module to Best modules
33     signal fitness_from_eval : sfixed(FITNESS_BITS_LEFT downto -
           FITNESS_BITS_RIGHT);
34     -- Signals to Best modules
35     signal position_to_best : total_position;
```

```
36      signal pbest_position, gbest_position : total_position;
37      signal pbest_fitness, gbest_fitness : sfixed(FITNESS_BITS_LEFT
            downto -FITNESS_BITS_RIGHT);
38      -- Signals from Best modules
39      signal new_pbest_position, new_gbest_position : total_position;
40      signal new_pbest_fitness, new_gbest_fitness : sfixed(
            FITNESS_BITS_LEFT downto -FITNESS_BITS_RIGHT);
41      -- Signals to Update Velocity module
42      signal velocity_to_update_velocity : total_velocity;
43      signal position_to_update_velocity : total_position;
44      signal rn1, rn2 : sfixed(R_BITS_LEFT-1 downto -R_BITS_RIGHT);
45      -- Signals to Update Position module
46      signal new_velocity : total_velocity;
47
48      -- Signals from Update Ppsition module
49      signal new_position : total_position;
50
51      -- Particle Memory
52      type particle_position_mem is array (0 to NUM_PARTICLES-1) of
            total_position;
53      type particle_velocity_mem is array (0 to NUM_PARTICLES-1) of
            total_velocity;
54      type particle_fitness_mem is array (0 to NUM_PARTICLES-1) of sfixed(
            FITNESS_BITS_LEFT downto -FITNESS_BITS_RIGHT);
55
56      signal position_mem : particle_position_mem;
57      signal velocity_mem : particle_velocity_mem;
58      signal pbest_position_mem : particle_position_mem;
59      signal pbest_fitness_mem : particle_fitness_mem;
60      signal gbest_position_mem : total_position;
61      signal gbest_fitness_mem : sfixed(FITNESS_BITS_LEFT downto -
            FITNESS_BITS_RIGHT);
62      signal init_fitness_to : sfixed(FITNESS_BITS_LEFT downto -
            FITNESS_BITS_RIGHT);
63
64      -- PSO Top FSM
65      TYPE states IS (idle, init_rng, init_particles1, init_particles2,
            init_particles3, init_pipeline1, init_pipeline2, exec, done);
66      signal state : states;
```

```
67      signal dimension_cnt : unsigned(DIMENSION_COUNT_BITS-1 downto 0);
68      signal particle_cnt_init : unsigned(PARTICLE_COUNT_BITS-1 downto 0);
69      signal particle_cnt_stage1 : unsigned(PARTICLE_COUNT_BITS-1 downto
            0);
70      signal particle_cnt_stage2 : unsigned(PARTICLE_COUNT_BITS-1 downto
            0);
71      signal particle_cnt_stage3 : unsigned(PARTICLE_COUNT_BITS-1 downto
            0);
72      signal particle_cnt_stage4 : unsigned(PARTICLE_COUNT_BITS-1 downto
            0);
73      signal particle_cnt_stage5 : unsigned(PARTICLE_COUNT_BITS-1 downto
            0);
74      signal iteration_cnt : unsigned(ITERATION_COUNT_BITS-1 downto 0);
75      signal init_rng_cnt : unsigned(4 downto 0);
76  BEGIN
77      rng : ENTITY work.RandomNumberGenerator(General)
78        PORT MAP (clk, rng_enable, reset_rng, rn);
79
80      delay_positions : for i in 0 to NUM_DIMENSIONS-1 generate
81        delay_position1 : ENTITY work.delay_position(behavioral)
82          PORT MAP (clk, position_to_eval(i), position_to_best(i));
83        delay_position2 : ENTITY work.delay_position(behavioral)
84          PORT MAP (clk, position_to_best(i), position_to_update_velocity(
                i));
85      end generate delay_positions;
86
87      --fitness_eval : ENTITY work.fitness_dummy(behavioral)
88      --  PORT MAP (clk, position_to_eval, fitness_from_eval);
89      --fitness_eval : ENTITY work.fitness_squared(behavioral)
90      --  PORT MAP (clk, position_to_eval, fitness_from_eval);
91      fitness_eval : ENTITY work.fitness_rosenbrock(behavioral)
92        PORT MAP (clk, position_to_eval, fitness_from_eval);
93
94      update_pbest : ENTITY work.update_best(behavioral)
95        PORT MAP (clk, reset_modules, position_to_best, fitness_from_eval,
              pbest_position, pbest_fitness,
96                    new_pbest_position, new_pbest_fitness);
97
98      update_gbest : ENTITY work.update_best(behavioral)
```

```
99        PORT MAP (clk, reset_modules, position_to_best, fitness_from_eval,
              new_gbest_position, new_gbest_fitness,
100               new_gbest_position, new_gbest_fitness);
101
102     pso_dimensions : for i in 0 to NUM_DIMENSIONS-1 generate
103       pso_dimension : ENTITY work.pso_dimension(behavioral)
104         PORT MAP (clk, reset_modules, position_to_update_velocity(i),
              velocity_to_update_velocity(i), new_pbest_position(i),
              new_gbest_position(i),
105           rn1, rn2, new_velocity(i), new_position(i)
106         );
107     end generate pso_dimensions;
108
109     iteration <= iteration_cnt;
110
111     PSO_main : PROCESS(clk, reset)
112     BEGIN
113       -- Go into 'idle' state upon reset
114       IF reset='1' THEN
115         state <= idle;
116         -- Reset modules
117         reset_modules <= '1';
118         -- Reset RNG
119         reset_rng <= '1';
120         rng_enable <= '0';
121         finished <= '0';
122         iteration_cnt <= (others => '0');
123       ELSIF (clk'EVENT AND clk='1') THEN
124         CASE state IS
125           WHEN idle =>                -- Was reset
126             state <= init_rng;
127             particle_cnt_init <= (others => '0');
128             dimension_cnt <= (others => '0');
129             iteration_cnt <= (others => '0');
130             finished <= '0';
131             --Maximum (worst) fitness value
132             init_fitness_to <= (FITNESS_BITS_LEFT => '0', others => '1')
                  ;
133             init_rng_cnt <= (others => '0');
```

```
134
135             particle_cnt_stage1 <= (others => '0');
136             particle_cnt_stage2 <= to_unsigned(NUM_PARTICLES-2,
                    PARTICLE_COUNT_BITS);
137             particle_cnt_stage3 <= to_unsigned(NUM_PARTICLES-3,
                    PARTICLE_COUNT_BITS);
138             particle_cnt_stage4 <= to_unsigned(NUM_PARTICLES-4,
                    PARTICLE_COUNT_BITS);
139             particle_cnt_stage5 <= to_unsigned(NUM_PARTICLES-5,
                    PARTICLE_COUNT_BITS);
140
141           WHEN init_rng =>          -- Cycle through some RNG values so
                they are 'more random'
142             reset_rng <= '0';
143             rng_enable <= '1';
144             if(init_rng_cnt < "11111") then
145               init_rng_cnt <= init_rng_cnt + 1;
146             else
147               state <= init_particles1;
148             end if;
149           WHEN init_particles1 =>             -- Initalize all particle
                memory to random values
150             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);
151             velocity_mem(to_integer(particle_cnt_init))(to_integer(
                    dimension_cnt)) <= to_sfixed(0, VELOCITY_BITS_LEFT, -
                    VELOCITY_BITS_RIGHT);
152
153             if(dimension_cnt < NUM_DIMENSIONS-1) then
154               dimension_cnt <= dimension_cnt + 1;
155             else
156               dimension_cnt <= (others => '0');
157               if(particle_cnt_init < NUM_PARTICLES-1) then
158                 particle_cnt_init <= particle_cnt_init + 1;
159               else
160                 particle_cnt_init <= (others => '0');
161                 state <= init_particles2;
```

```
162               end if;
163             end if;
164
165           WHEN init_particles2 =>               -- Initalize all particle
                memory to random values
166             --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);
167             --pbest_position_mem(to_integer(particle_cnt_init))(
                  to_integer(dimension_cnt)) <= to_sfixed(0,
                  POSITION_BITS_LEFT, -POSITION_BITS_RIGHT);
168
169             --Initialize fitness to largest (worst) value
170             pbest_fitness_mem(to_integer(particle_cnt_init)) <=
                  init_fitness_to;
171
172             if(dimension_cnt < NUM_DIMENSIONS-1) then
173               dimension_cnt <= dimension_cnt + 1;
174             else
175               dimension_cnt <= (others => '0');
176               if(particle_cnt_init < NUM_PARTICLES-1) then
177                 particle_cnt_init <= particle_cnt_init + 1;
178               else
179                 particle_cnt_init <= (others => '0');
180                 state <= init_particles3;
181               end if;
182             end if;
183           WHEN init_particles3 =>               -- Initalize all particle
                memory to random values
184             --gbest_position_mem(to_integer(dimension_cnt)) <= to_sfixed
                  (rn(POSITION_BITS_LEFT + POSITION_BITS_RIGHT downto 0),
                  POSITION_BITS_LEFT, -POSITION_BITS_RIGHT);
185             --gbest_position_mem(to_integer(dimension_cnt)) <= to_sfixed
                  (0, POSITION_BITS_LEFT, -POSITION_BITS_RIGHT);
186
187             --Initialize fitness to largest (worst) value
188             gbest_fitness_mem <= init_fitness_to;
189
```

```
190             if(dimension_cnt < NUM_DIMENSIONS-1) then
191               dimension_cnt <= dimension_cnt + 1;
192             else
193               dimension_cnt <= (others => '0');
194               if(particle_cnt_init < NUM_PARTICLES-1) then
195                 particle_cnt_init <= particle_cnt_init + 1;
196               else
197                 particle_cnt_init <= (others => '0');
198                 state <= init_pipeline1;
199
200
201               end if;
202             end if;
203
204             -- Pass the particles positon to the fitness module
205             position_to_eval <= position_mem(to_integer(
                    particle_cnt_stage1));
206           WHEN init_pipeline1 =>
207
208
209             -- Pass the particles positon to the fitness module
210             position_to_eval <= position_mem(to_integer(
                    particle_cnt_stage1));
211
212             -- Pass PBest to PBest Update Module
213             pbest_position <= pbest_position_mem(to_integer(
                    particle_cnt_stage2));
214             pbest_fitness <= pbest_fitness_mem(to_integer(
                    particle_cnt_stage2));
215
216             -- Pass GBest to GBest Update Module
217             gbest_position <= gbest_position_mem;
218             gbest_fitness <= gbest_fitness_mem;
219
220             -- Old velocity to Update Velocity module
221             velocity_to_update_velocity <= velocity_mem(to_integer(
                    particle_cnt_stage3));
222
223             -- Random numbers to Update Velocity module
```

```
224             rn1 <= to_sfixed(rn(R_BITS_LEFT-1 + R_BITS_RIGHT downto 0),
                    R_BITS_LEFT-1, -R_BITS_RIGHT);
225             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);
226
227             --state <= init_pipeline2;
228
229           --WHEN init_pipeline2 =>
230             state <= exec;
231             reset_modules <= '0';
232             report "Executing";
233           WHEN exec =>                  -- Execute PSO algorithm
234             -- Pass the particles positon to the fitness module
235             position_to_eval <= position_mem(to_integer(
                    particle_cnt_stage1));
236
237             -- Pass PBest to PBest Update Module
238             pbest_position <= pbest_position_mem(to_integer(
                    particle_cnt_stage2));
239             pbest_fitness <= pbest_fitness_mem(to_integer(
                    particle_cnt_stage2));
240
241             velocity_to_update_velocity <= velocity_mem(to_integer(
                    particle_cnt_stage2));
242
243             -- Random numbers to Update Velocity module
244             rn1 <= to_sfixed(rn(R_BITS_LEFT-1 + R_BITS_RIGHT downto 0),
                    R_BITS_LEFT-1, -R_BITS_RIGHT);
245             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);
246
247             if((iteration_cnt > 0) ) then
248               --assert new_gbest_fitness <= gbest_fitness;
```

```
249             --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))))
250             --severity Warning;
251
252             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);
253
254             -- Store new PBest
255             pbest_position_mem(to_integer(particle_cnt_stage3)) <=
                    new_pbest_position;
256             pbest_fitness_mem(to_integer(particle_cnt_stage3)) <=
                    new_pbest_fitness;
257
258             -- Store new GBest
259             gbest_position_mem <= new_gbest_position;
260             gbest_fitness_mem <= new_gbest_fitness;
261
262             -- Store new Velocity
263             velocity_mem(to_integer(particle_cnt_stage5)) <=
                    new_velocity;
264
265             -- Store new Position
266             position_mem(to_integer(particle_cnt_stage5)) <=
                    new_position;
267           else
268
269           end if;
270
271
272           -- Restart the sequence when the last particle has been
                  passed to the fitness module
273           IF (particle_cnt_stage1 = NUM_PARTICLES-1) THEN
274             particle_cnt_stage1 <= (others => '0');
275             iteration_cnt <= iteration_cnt + 1;
```

```
276                 --Write current gbest position/fitness to output ports
277                 gbest <= new_gbest_position;
278                 gbest_e <= new_gbest_fitness;
279               ELSE
280                 particle_cnt_stage1 <= particle_cnt_stage1 + 1;
281               END IF;
282
283               -- Update counters
284               IF particle_cnt_stage2 = NUM_PARTICLES-1 THEN
285                 particle_cnt_stage2 <= (others => '0');
286               ELSE
287                 particle_cnt_stage2 <= particle_cnt_stage2 + 1;
288               END IF;
289
290               IF particle_cnt_stage3 = NUM_PARTICLES-1 THEN
291                 particle_cnt_stage3 <= (others => '0');
292               ELSE
293                 particle_cnt_stage3 <= particle_cnt_stage3 + 1;
294               END IF;
295
296               IF particle_cnt_stage4 = NUM_PARTICLES-1 THEN
297                 particle_cnt_stage4 <= (others => '0');
298               ELSE
299                 particle_cnt_stage4 <= particle_cnt_stage4 + 1;
300               END IF;
301
302               IF particle_cnt_stage5 = NUM_PARTICLES-1 THEN
303                 particle_cnt_stage5 <= (others => '0');
304               ELSE
305                 particle_cnt_stage5 <= particle_cnt_stage5 + 1;
306               END IF;
307
308               IF (iteration_cnt = MAX_ITERATIONS) THEN
309                 state <= done;
310               END IF;
311             WHEN done =>                    -- Reached maximum number of
                  iterations
312               --LED On
313               finished <= '1';
```

```
314             WHEN others =>
315
316           END CASE;
317         END IF;
318       END PROCESS PSO_main;
319 END behavioral;
```

pso_tb.vhd

```
1  USE WORK.pso_package.ALL;
2  LIBRARY IEEE;
3  USE IEEE.std_logic_1164.ALL;
4  USE IEEE.numeric_std.ALL;
5
6  --LIBRARY floatfixlib;
7  --USE floatfixlib.math_utility_pkg.ALL;
8  --USE floatfixlib.fixed_pkg.ALL;
9
10 LIBRARY ieee_proposed;
11 USE ieee_proposed.math_utility_pkg.ALL;
12 USE ieee_proposed.fixed_pkg.ALL;
13
14 USE ieee.std_logic_textio.ALL;
15 USE std.textio.ALL;
16
17 entity pso_tb is
18 end entity pso_tb;
19
20 architecture bench of pso_tb is
21     signal clk, reset : std_logic;
22     signal gbest_position : total_position;
23     signal gbest_fitness : sfixed(FITNESS_BITS_LEFT downto -
           FITNESS_BITS_RIGHT);
24     --signal gbest_fitness_prev : sfixed(FITNESS_BITS_LEFT downto -
           FITNESS_BITS_RIGHT);
25     signal finished : std_logic;
26     signal iteration : unsigned(ITERATION_COUNT_BITS-1 downto 0);
27
28 begin
29     pso : ENTITY work.pso_top(behavioral) PORT MAP(clk, reset,
           gbest_position, gbest_fitness, iteration, finished);
30
31     clock : PROCESS
32     BEGIN
33       clk <= '0';
34       wait for 5 ns;
```

```
35        clk <= '1';
36        wait for 5 ns;
37
38
39      END PROCESS clock;
40
41      stimulus : PROCESS
42      BEGIN
43          reset <= '1';
44          wait until clk'event AND clk = '1';
45          reset <= '0';
46          wait until clk'event AND clk = '1';
47
48
49          wait;
50      end process stimulus;
51
52      write_fitness: process(iteration)
53
54          file       OUTFILE         : text is out "rosenbrock_dim10.txt";
55          variable   WR_BUFFER       : line;
56          variable   fitness         : real;
57
58          begin
59              fitness := to_real(gbest_fitness);
60              write (WR_BUFFER, string'("ModelSimGBestE("));
61              write (WR_BUFFER, integer'image(to_integer(iteration)));
62              write (WR_BUFFER, string'(")="));
63              write (WR_BUFFER, real'image(fitness));
64              write (WR_BUFFER, string'(";"));
65              writeline(OUTFILE,WR_BUFFER);
66              report "GBestE : " & real'image(fitness);
67          end process;
68  end bench;
```

pso_tb.vhd

```
1  USE WORK.pso_package.ALL;
2  LIBRARY IEEE;
3  USE IEEE.std_logic_1164.ALL;
4  USE IEEE.numeric_std.ALL;
5
6  --LIBRARY floatfixlib;
7  --USE floatfixlib.math_utility_pkg.ALL;
8  --USE floatfixlib.fixed_pkg.ALL;
9
10 LIBRARY ieee_proposed;
11 USE ieee_proposed.math_utility_pkg.ALL;
12 USE ieee_proposed.fixed_pkg.ALL;
13
14 USE ieee.std_logic_textio.ALL;
15 USE std.textio.ALL;
16
17 entity pso_tb is
18 end entity pso_tb;
19
20 architecture bench of pso_tb is
21     signal clk, reset : std_logic;
22     signal gbest_position : total_position;
23     signal gbest_fitness : sfixed(FITNESS_BITS_LEFT downto -
           FITNESS_BITS_RIGHT);
24     --signal gbest_fitness_prev : sfixed(FITNESS_BITS_LEFT downto -
           FITNESS_BITS_RIGHT);
25     signal finished : std_logic;
26     signal iteration : unsigned(ITERATION_COUNT_BITS-1 downto 0);
27
28 begin
29     pso : ENTITY work.pso_top(behavioral) PORT MAP(clk, reset,
           gbest_position, gbest_fitness, iteration, finished);
30
31     clock : PROCESS
32     BEGIN
33       clk <= '0';
34       wait for 5 ns;
```

```
35        clk <= '1';
36        wait for 5 ns;
37
38
39    END PROCESS clock;
40
41    stimulus : PROCESS
42    BEGIN
43        reset <= '1';
44        wait until clk'event AND clk = '1';
45        reset <= '0';
46        wait until clk'event AND clk = '1';
47
48
49        wait;
50    end process stimulus;
51
52    write_fitness: process(iteration)
53
54        file      OUTFILE        : text is out "rosenbrock_dim10.txt";
55        variable  WR_BUFFER      : line;
56        variable  fitness        : real;
57
58        begin
59            fitness := to_real(gbest_fitness);
60            write (WR_BUFFER, string'("ModelSimGBestE("));
61            write (WR_BUFFER, integer'image(to_integer(iteration)));
62            write (WR_BUFFER, string'(")="));
63            write (WR_BUFFER, real'image(fitness));
64            write (WR_BUFFER, string'(";"));
65            writeline(OUTFILE,WR_BUFFER);
66            report "GBestE : " & real'image(fitness);
67        end process;
68 end bench;
```

fitness_squared.vhd

```
1  LIBRARY IEEE;
2  USE IEEE.std_logic_1164.ALL;
3  USE IEEE.numeric_std.ALL;
4
5  --LIBRARY floatfixlib;
6  --USE floatfixlib.math_utility_pkg.ALL;
7  --USE floatfixlib.fixed_pkg.ALL;
8
9  LIBRARY ieee_proposed;
10 USE ieee_proposed.math_utility_pkg.ALL;
11 USE ieee_proposed.fixed_pkg.ALL;
12
13 LIBRARY work;
14 USE work.pso_package.ALL;
15
16 ENTITY fitness_squared IS
17     PORT (clk : in std_logic;
18           x : in total_position;
19           fitness : out sfixed(FITNESS_BITS_LEFT downto -
                 FITNESS_BITS_RIGHT));
20 END ENTITY fitness_squared;
21
22 ARCHITECTURE behavioral OF fitness_squared IS
23
24 BEGIN
25     behavior : PROCESS(clk) IS
26     BEGIN
27       IF(rising_edge(clk)) THEN
28         fitness <= x(0)*x(0);
29         --fitness <= x(0)*x(0) + x(1)*x(1);
30         --fitness <= x(0)*x(0) + x(1)*x(1) + x(2)*x(2);
31         --fitness <= x(0)*x(0) + x(1)*x(1) + x(2)*x(2) + x(3)*x(3);
32         --fitness <= x(0)*x(0) + x(1)*x(1) + x(2)*x(2) + x(3)*x(3) + x
               (4)*x(4);
33         --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);
```

```
34          --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) + x(6)*x(6) + x(7)*x(7);
35          --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) + x(6)*x(6) + x(7)*x(7) + x(8)*x(8) + x
                (9)*x(9);
36          --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) + x(6)*x(6) + x(7)*x(7) + x(8)*x(8) + x
                (9)*x(9) + x(10)*x(10) + x(11)*x(11);
37          --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) + x(6)*x(6) + x(7)*x(7) + x(8)*x(8) + x
                (9)*x(9) + x(10)*x(10) + x(11)*x(11) + x(12)*x(12) + x(13)*x
                (13) + x(14)*x(14) + x(15)*x(15) + x(16)*x(16) + x(17)*x(17)
                 + x(18)*x(18) + x(19)*x(19);
38        END IF;
39      END PROCESS behavior;
40  END behavioral;
```

fitness_rosenbrock.vhd

```
1  LIBRARY IEEE;
2  USE IEEE.std_logic_1164.ALL;
3  USE IEEE.numeric_std.ALL;
4
5  --LIBRARY floatfixlib;
6  --USE floatfixlib.math_utility_pkg.ALL;
7  --USE floatfixlib.fixed_pkg.ALL;
8
9  LIBRARY ieee_proposed;
10 USE ieee_proposed.math_utility_pkg.ALL;
11 USE ieee_proposed.fixed_pkg.ALL;
12
13 LIBRARY work;
14 USE work.pso_package.ALL;
15
16 ENTITY fitness_rosenbrock IS
17     PORT (clk : in std_logic;
18           x : in total_position;
19           fitness : out sfixed(FITNESS_BITS_LEFT downto -
                 FITNESS_BITS_RIGHT));
20 END ENTITY fitness_rosenbrock;
21
22 ARCHITECTURE behavioral OF fitness_rosenbrock IS
23
24 BEGIN
25     behavior : PROCESS(clk) IS
26     BEGIN
27       IF(rising_edge(clk)) THEN
28         --fitness <= (100*(x(1)-x(0)*x(0))*(x(1)-x(0)*x(0))+(x(0)-1)*(x
               (0)-1));
29         --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));
```

```
30          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));
31
32        END IF;
33      END PROCESS behavior;
34  END behavioral;
```

delay_position.vhd

```
1  LIBRARY IEEE;
2  USE IEEE.std_logic_1164.ALL;
3  USE IEEE.numeric_std.ALL;
4
5  LIBRARY work;
6  USE work.pso_package.ALL;
7
8  --LIBRARY floatfixlib;
9  --USE floatfixlib.math_utility_pkg.ALL;
10 --USE floatfixlib.fixed_pkg.ALL;
11
12 LIBRARY ieee_proposed;
13 USE ieee_proposed.math_utility_pkg.ALL;
14 USE ieee_proposed.fixed_pkg.ALL;
15
16 ENTITY delay_position IS
17   PORT (clk : in std_logic;
18         x : in sfixed(POSITION_BITS_LEFT downto -POSITION_BITS_RIGHT);
19         x_delayed : out sfixed(POSITION_BITS_LEFT downto -
               POSITION_BITS_RIGHT));
20 END ENTITY delay_position;
21
22 ARCHITECTURE behavioral OF delay_position IS
23 BEGIN
24   behavior : PROCESS(clk) IS
25   BEGIN
26     IF(rising_edge(clk)) THEN
27       x_delayed <= x;
28     END IF;
29   END PROCESS behavior;
30 END behavioral;
```

update_best.vhd

```
1  LIBRARY IEEE;
2  USE IEEE.std_logic_1164.ALL;
3  USE IEEE.numeric_std.ALL;
4
5  --LIBRARY floatfixlib;
6  --USE floatfixlib.math_utility_pkg.ALL;
7  --USE floatfixlib.fixed_pkg.ALL;
8
9  LIBRARY ieee_proposed;
10 USE ieee_proposed.math_utility_pkg.ALL;
11 USE ieee_proposed.fixed_pkg.ALL;
12
13 LIBRARY work;
14 USE work.pso_package.ALL;
15
16 ENTITY update_best IS
17     PORT (clk, reset : in std_logic;
18           x : in total_position;
19           fitness : in sfixed(FITNESS_BITS_LEFT downto -
                 FITNESS_BITS_RIGHT);
20           best_position : in total_position;
21           best_fitness : in sfixed(FITNESS_BITS_LEFT downto -
                 FITNESS_BITS_RIGHT);
22           new_best_position : out total_position;
23           new_best_fitness : out sfixed(FITNESS_BITS_LEFT downto -
                 FITNESS_BITS_RIGHT));
24 END ENTITY update_best;
25
26 ARCHITECTURE behavioral OF update_best IS
27
28 BEGIN
29
30     behavior : PROCESS(clk, reset) IS
31     BEGIN
32       IF(rising_edge(clk)) THEN
33        IF reset = '1' THEN
```

```
34          --new_best_position(0) <= (POSITION_BITS_LEFT => '0', others =>
                '1');
35          new_best_fitness <= (FITNESS_BITS_LEFT => '0', others => '1');
36        ELSE
37          IF fitness < best_fitness THEN
38            --record new position/fitness
39            new_best_position <= x;
40            new_best_fitness <= fitness;
41            assert fitness <= best_fitness
42              report "E@UPDATE_BEST: Best Fitness increased from " &
                    to_string(best_fitness) & " to " & to_string(fitness);
43
44          ELSE
45            --keep old position/fitness
46            new_best_position <= best_position;
47            new_best_fitness <= best_fitness;
48          END IF;
49        END IF;
50      END IF;
51      END PROCESS behavior;
52  END behavioral;
```

update_velocity.vhd

```
1  LIBRARY IEEE;
2  USE IEEE.std_logic_1164.ALL;
3  USE IEEE.numeric_std.ALL;
4
5  LIBRARY work;
6  USE work.pso_package.ALL;
7
8  --LIBRARY floatfixlib;
9  --USE floatfixlib.math_utility_pkg.ALL;
10 --USE floatfixlib.fixed_pkg.ALL;
11
12 LIBRARY ieee_proposed;
13 USE ieee_proposed.math_utility_pkg.ALL;
14 USE ieee_proposed.fixed_pkg.ALL;
15
16 ENTITY update_velocity IS
17   PORT (clk : in std_logic;
18         x : in sfixed(POSITION_BITS_LEFT downto -POSITION_BITS_RIGHT);
19         v : in sfixed(VELOCITY_BITS_LEFT downto -VELOCITY_BITS_RIGHT);
20         pbest, gbest : in sfixed(POSITION_BITS_LEFT downto -
               POSITION_BITS_RIGHT);
21         rn1, rn2 : in sfixed(R_BITS_LEFT-1 downto -R_BITS_RIGHT);
22         new_v : out sfixed(VELOCITY_BITS_LEFT downto -
               VELOCITY_BITS_RIGHT));
23 END ENTITY update_velocity;
24
25 ARCHITECTURE behavioral OF update_velocity IS
26   constant c1 : sfixed(C_BITS_LEFT downto -C_BITS_RIGHT) := to_sfixed
       (2.0, C_BITS_LEFT, -C_BITS_RIGHT);
27   constant c2 : sfixed(C_BITS_LEFT downto -C_BITS_RIGHT) := to_sfixed
       (2.0, C_BITS_LEFT, -C_BITS_RIGHT);
28   constant w : sfixed(W_BITS_LEFT downto -W_BITS_RIGHT) := to_sfixed
       (0.5, W_BITS_LEFT, -W_BITS_RIGHT);
29
30 BEGIN
31   PROCESS(clk) IS
32   BEGIN
```

```
33      IF(rising_edge(clk)) THEN
34        --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);
35
36        --new_v <= c1*rn1*(x - pbest) + c2*rn2*(x - gbest);
37
38        --new_v <= resize((rn1+rn2)*v + c1*rn1*(x - pbest) + c2*rn2*(x -
              gbest), new_v'high, new_v'low);
39        new_v <= resize( (v sra 1) + c1*rn1*(x - pbest) + c2*rn2*(x -
              gbest), new_v'high, new_v'low);
40        --new_v <= resize( (v sra 1) + rn1*(x - pbest) + rn2*(x - gbest),
              new_v'high, new_v'low);
41
42
43      END IF;
44    END PROCESS;
45  END behavioral;
```

APPENDIX B

HARDWARE NN SOURCE CODE

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

```
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');
```

```
38
39      figure(2)
40      plot(p2(:),t2);
41      legend('Target');
42  end
43
44  [trainV,val,test] = dividevec(p2,t2,0.20,0.20);
45
46  if(func_type == 0)
47      net = newff(minmax(p2),[numHidden 1],{'logsig' 'purelin'});
48  else
49      %net = newff(minmax(p2),[numHidden 1],{'tansig' 'tansig'});
50      net = newff(minmax(p2),[numHidden numOutput],{'tansig' 'purelin'});
51      %net = newff(minmax(p2),[numHidden 1],{'purelin' 'purelin'});
52  end
53  [net,tr]=train(net,trainV.P,trainV.T,[],[],val,test);
54
55  a2 = sim(net,p2);
56  %a = mapminmax('reverse',a2,ts);
57  %[m,b,r] = postreg(a,t);
58  for i = 1:numOutput
59      figure(2+i)
60      plot(1:length(t2),t2(i,:),1:length(a2),a2(i,:));
61      legend('Target','Output');
62  end
```

nn_load.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};

W = zeros(numHidden, numInput+1);
for h = 1:numHidden
    for i = 1:numInput
        W(h,i) = IW(h,i);
    end
    W(h,numInput+1) = BI(h);
end

V = zeros(numOutput, numHidden+1);
for o = 1:numOutput
    for h = 1:numHidden
        V(o,h) = OW(o,h);
    end
    V(o,numHidden+1) = BH(o);
end

%NN_Input = zeros(length(p2)*(numInput+2),2);
NN_Input = zeros(length(p2)*(numHidden+2),numInput+1);
for p = 1:length(p2)
    NN_Input((p-1)*(numHidden+2)+1:(p-1)*(numHidden+2)+(numHidden+2),2:
        numInput+1) = ones((p-1)*(numHidden+2)+(numHidden+2)-(p-1)*(
        numHidden+2),1)*p2(:,p)';
end
NN_Input(length(NN_Input)+1:2*length(NN_Input),:) = NN_Input(:,:);
NN_Input(:,1) = 1:length(NN_Input);

NN_Input1(1:length(NN_Input),1) = NN_Input(:,1);
NN_Input1(1:length(NN_Input),2) = NN_Input(:,2);
if(numInput > 1)
```

```
36 NN_Input2(1:length(NN_Input),1) = NN_Input(:,1);
37 NN_Input2(1:length(NN_Input),2) = NN_Input(:,3);
38 end
39 if(numInput > 2)
40 NN_Input3(1:length(NN_Input),1) = NN_Input(:,1);
41 NN_Input3(1:length(NN_Input),2) = NN_Input(:,4);
42 end
43 if(numInput > 3)
44 NN_Input4(1:length(NN_Input),1) = NN_Input(:,1);
45 NN_Input4(1:length(NN_Input),2) = NN_Input(:,5);
46 end
47
48 k = 8;
49 [LUT_C1, LUT_C2] = generate_act_lut(k,func_type);
50
51 sample_length = length(NN_Input);
52
53 NN_Reset(:,1) = 1:length(NN_Input);
54 NN_Reset(:,2) = 1;
55 NN_Reset(1,2) = 0;
56
57 NN_Write(:,1) = 1:length(NN_Input);
58 NN_Write(:,2) = 0;
59 %NN_Write(sample_length/2:sample_length/2+2,2) = 1;
60
61 NN_Write_Data(:,1) = 1:length(NN_Input);
62 NN_Write_Data(:,2) = 0;
63 %NN_Write_Data(sample_length/2:sample_length/2+2,2) = 0.0356;
64
65 NN_Write_Addr(:,1) = 1:length(NN_Input);
66 NN_Write_Addr(:,2) = 0;
67 %NN_Write_Addr(sample_length/2:sample_length/2+2,2) = 0;
68
69 for i = 1:length(V)
70     a = fi(V(i), 1, 18, 12);
71     disp([num2str(i),' = 0x',dec2hex(bin2dec(a.bin))]);
72 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
```

```
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  %figure(4)
49  %plot(1:length(O),O);
50  %plot(1:length(O),O,1:length(t2),a2);
51  %legend('Man. Output','Matlab Output');
52
53  if(1)
54      for n = 1:length(hidden_output)
55          for i = 1:5
56              %SIM_NN_Hidden(n,i) = SIM_NN_Hidden_Output.signals(i).values
                    ((n-1)*3+1+11);
57              SIM_NN_Hidden(n,i) = SIM_NN_Hidden_Output1.signals(i).values
                    (n);
58          end
59      end
60      if(numHidden > 5)
61      for n = 1:length(hidden_output)
62          for i = 1:5
63              %SIM_NN_Hidden(n,i) = SIM_NN_Hidden_Output.signals(i).values
                    ((n-1)*3+1+11);
64              SIM_NN_Hidden(n,i+5) = SIM_NN_Hidden_Output2.signals(i).
                    values(n);
65          end
66      end
67      end
68
69      for i = 1:numHidden/5
70          figure(4+i)
```

```
71          %plot(1:length(hidden_output),hidden_output(:,1),1:length(
                hidden_output),hidden_output(:,2),1:length(hidden_output),
                hidden_output(:,3))
72          %legend('1','2','3')
73          for sp = 1:5
74              subplot(5,1,sp);
75              delay = numHidden+9;
76              %plot(1:length(hidden_output),hidden_output(:,(i-1)*5+sp),1:
                    length(hidden_output),SIM_NN_Hidden(:,(i-1)*5+sp));
77              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));
78              axis([1, length(hidden_output), -1.5, 1.5]);
79          end
80          legend('Floating-Point','Hardware');
81      end
82
83      SIM_NN_Output = zeros(1:length(O),numOutput);
84      for n = 1:length(hidden_output)
85          for i = 1:numOutput
86              %SIM_NN_Output(n,i) = SIM_NN_Output_Output.signals(i).values
                    ((n-1)*3+1+19);
87              SIM_NN_Output(n,i) = SIM_NN_Output_Output.signals(i).values(
                    n);
88          end
89      end
90
91      for i = 1:numOutput
92          figure(10+i)
93          %plot(1:length(O),O,1:length(O),SIM_NN_Output(:,1));
94          delay = numHidden+9+numHidden+5;
95          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));
96          legend('Floating-Point','Hardware');
97      end
98  end
```

BIBLIOGRAPHY

[1] A. P. Engelbrecht, *Computational Intelligence: An Introduction.* John Wiley and Sons Ltd., Chichester, West Sussex, England, 2002.

[2] P. D. Reynolds, R. W. Duren, M. L. Trumbo, and I. Marks, R. J., "FPGA implementation of particle swarm optimization for inversion of large neural networks," in *Swarm Intelligence Symposium, 2005. SIS 2005. Proceedings 2005 IEEE*, Jun. 2005, pp. 389–392.

[3] G. Kokai, T. Christ, and H. H. Frhauf, "Using hardware-based particle swarm method for dynamic optimization of adaptive array antennas," in *Adaptive Hardware and Systems, 2006. AHS 2006. First NASA/ESA Conference on*, Istanbul, Jun. 2006, pp. 51–58.

[4] A. Farmahini-Farahani, S. M. Fakhraie, and S. Safari, "SOPC-based architecture for discrete particle swarm optimization," in *Electronics, Circuits and Systems, 2007. ICECS 2007. 14th IEEE International Conference on*, Marrakech, Dec. 2007, pp. 1003–1006.

[5] B. Widrow, D. E. Rumelhart, and M. A. Lehr, "Neural networks: Applications in industry, business and science," *Communications of the ACM*, vol. 37, no. 3, pp. 93–105, 1994.

[6] M. Valle, "Analog vlsi implementation of artificial neural networks with supervised on-chip learning," *Analog Integr. Circuits Signal Process.*, vol. 33, no. 3, pp. 263–287, 2002.

[7] I. D. dos Santos Miranda and A. I. A. Cunha, "Asic design of a novel high performance neuroprocessor architecture for multi layered perceptron networks," in *SBCCI '09: Proceedings of the 22nd Annual Symposium on Integrated Circuits and System Design.* New York, NY, USA: ACM, 2009, pp. 1–6.

[8] D. Lettnin, A. Braun, M. Bodgan, J. Gerlach, and W. Rosenstiel, "Synthesis of embedded systemc design: A case study of digital neural networks," in *DATE '04: Proceedings of the conference on Design, automation and test in Europe.* Washington, DC, USA: IEEE Computer Society, 2004, p. 30248.

[9] T. Szabó and G. Horváth, "An efficient hardware implementation of feed-forward neural networks," in *IEA/AIE '01: Proceedings of the 14th International conference on Industrial and engineering applications of artificial intelligence and expert systems.* London, UK: Springer-Verlag, 2001, pp. 300–313.

[10] G. Danese, F. Leporati, and S. Ramat, "A parallel neural processor for real-time applications," *IEEE Micro*, vol. 22, no. 3, pp. 20–31, 2002.

[11] P. Ferreira, P. Ribeiro, A. Antunes, and F. M. Dias, *Artificial Neural Networks Processor A Hardware Implementation Using a FPGA*. Springer Berlin / Heidelberg, 2004, pp. 1084–1086.

[12] D. Larkin, A. Kinane, V. Muresan, and N. E. O'Connor, "An efficient hardware architecture for a neural network activation function generator," *ISNN 2006 International Symposium on Neural Networks*, vol. 3973, pp. 1319–1327, 2006.

[13] P. Palangpour, G. K. Venayagamoorthy, and S. C. Smith, "Particle swarm optimization: A hardware implementation," in *2009 International Conference on Computer Design*, Las Vegas, 2009, pp. 134–139.

[14] P. Palangpour and G. K. Venayagamoorthy, "Xilinx system generator implementation of neural network," in *Neural Networks, 2010. IJCNN 2010. (IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, 2010.

[15] R. Eberhart and J. Kennedy, "A new optimizer using particle swarm theory," in *Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, vol. 1, Nagoya, Japan, Oct. 1995, pp. 39–43.

[16] L. Liu, W. Liu, D. A. Cartes, and N. Zhang, "Real time implementation of particle swarm optimization based model parameter identification and an application example," in *Evolutionary Computation, 2008. CEC 2008. (IEEE World Congress on Computational Intelligence). IEEE Congress on*, Hong Kong, Jun. 2008, pp. 3480–3485.

[17] K. H. S. Hla, Y. Choi, and J. S. Park, "Applying particle swarm optimization to prioritizing test cases for embedded real time software retesting," in *Computer and Information Technology Workshops, 2008. CIT Workshops 2008. IEEE 8th International Conference on*, Sydney, QLD, Jul. 2008, pp. 527–532.

[18] Y. Feng, Y.-M. Yao, and A.-X. Wang, "Comparing with chaotic inertia weights in particle swarm optimization," in *Machine Learning and Cybernetics, 2007 International Conference on*, vol. 1, Hong Kong, Aug. 2007, pp. 329–333.

[19] M. Rodgers, "Random numbers and their effect on particle swarm optimization."

[20] H. Zhang, Y. Wang, B. Wang, and X. Wu, "Evolutionary random sequence generators based on lfsr," *Wuhan University Journal of Natural Sciences*, vol. 12, no. 1, pp. 75–78, 2007.

[21] B. Shackleford, M. Tanaka, R. J. Carter, and G. Snider, "FPGA implementation of neighborhood-of-four cellular automata random number generators," in *FPGA '02: Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*. New York, NY, USA: ACM, 2002, pp. 106–112.

[22] "XUPV2P Documentation," December 2009, http://www.xilinx.com/univ/xupv2p.html.

[23] P. W. Moore and G. K. Venayagamoorthy, "Empirical study of an unconstrained modified particle swarm optimization," in *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, 2006, pp. 1477–1482.

[24] A. R. Omandi and J. C. Rajapakse, *FPGA Implementations of Neural Networks*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[25] "System Generator for DSP User Guide," March 2008, http://www.xilinx.com/support/sw_manuals/sysgen_user.pdf.

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.