A DSP-BASED COMPUTATIONAL ENGINE FOR A BRAIN-MACHINE INTERFACE

By

SCOTT A. MORRISON

A THESIS PRESENTED TO THE GRADUATE SCHOOL OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF ENGINEERING

UNIVERSITY OF FLORIDA

2003
This work is dedicated to the Engineers, Neurobiologists, Scientists, and Researchers—students, professors, and professionals—who pursue the amazing goal of linking mind with machine.
ACKNOWLEDGMENTS

I thank Dr. José C. Principe for his leadership in developing signal processing methods to understand the way the brain encodes motor movement. I owe much gratitude to Dr. Karl Gugel for bringing this work to me and for pursuing great advances in DSP technology. Dr. Michael Nechyba and Dr. John Harris contributed much direction for this project.

The BMI consortium is headed by Dr. Miguel A.L. Nicolelis of Duke University, and funded by the Defense Advanced Research Project Agency. Many thanks go to Dr. Nicolelis and his collaborators for pioneering the research that may one day help so many people. Dr. Nicolelis assembled a top-notch team including MIT, SUNY, and Plexon Incorporated, as well as the Duke BME Department and Medical Center.

Jeremy Parks, my friend and co-designer of the C33 DSP Board, was instrumental in making such a useful and powerful DSP tool. Shalom Darmanjian, Joel Fuster, and Andy Lin helped support all aspects of this project.

I thank Justin Sanchez, Deniz Erdogmus, and the other CNEL researchers for their contributions to understanding how the brain encodes motor movement, and their kind patience in explaining it to me.

Many thanks go to Ellie Goodwin and Janet Holman who were responsible for getting all the parts, documents, and arrangements that I needed to do this work.

I thank my parents for instilling in me the admiration of science, and for encouraging me to achieve such high goals throughout my lifetime. My brothers—Lee,
Dean, and Casey—have taught me that success is a choice, and that to be successful, one must work hard and be patient.

And I thank my wife, Kate, whose love and dedication to me have made possible all of these achievements. Kate has supported me through many nights and weekends of work throughout my school career. She is my past, present, and future, and the reason that I do what I do.
# TABLE OF CONTENTS

ACKNOWLEDGMENTS ........................................................................................................ iv

LIST OF TABLES ............................................................................................................. ix

LIST OF FIGURES ......................................................................................................... x

ABSTRACT ..................................................................................................................... xii

CHAPTER

1 INTRODUCTION ........................................................................................................ 1

Motivation ..................................................................................................................... 1
Brain-Machine Interface ............................................................................................... 5
Duke BMI Project ........................................................................................................ 8
Data Properties ......................................................................................................... 9
Neural-to-Motor Mapping Algorithms .................................................................... 11
Thesis Goals ............................................................................................................... 12

2 ALGORITHMS FOR NEURAL-TO-MOTOR TRANSLATION ............................. 15

Notation and Terminology .......................................................................................... 16
Linear Filter Topology ................................................................................................. 17
Linear Filter Training ................................................................................................ 19
Wiener Filter Theory ................................................................................................. 20
LMS Theory ............................................................................................................... 21
NLMS Theory ........................................................................................................... 24
NLMS Equations ....................................................................................................... 25
RMLP Topology ......................................................................................................... 26
Neural Networks in General .................................................................................... 27
Multi-Layer Perceptron ............................................................................................ 29
Recursive Multi-Layer Perceptron .......................................................................... 30
Training an RMLP using RTRL ................................................................................ 33
RTRL Equations ......................................................................................................... 35
RTRL Comparison with BPTT .................................................................................. 37
NLMS and RTRL Requirements Compared ............................................................. 39
Memory Requirements ............................................................................................. 39
Precision Effects ....................................................................................................... 41
Summary ..................................................................................................................... 43
3 DSP-BASED COMPUTATIONAL ENGINE ...........................................................44

   System Requirements .................................................................................................44
   Portability Considerations ...............................................................................................45
   DSP Development versus Deployment ...........................................................................46

Component Selection ..........................................................................................................47
   TMS320VC33 DSP ........................................................................................................47
   PLX PCI 9030 SMARTarget I/O Accelerator .............................................................52
   External SRAM ..............................................................................................................55
   Bootable EEPROM .......................................................................................................56
   Power .............................................................................................................................57
   CPLD Control ...............................................................................................................58
   Future Add-on Hardware .............................................................................................59

Printed Circuit Board Fabrication ..................................................................................60

Control Software .............................................................................................................64
   PC Software ................................................................................................................65

DSP Operating System ................................................................................................67

Summary ..........................................................................................................................68

4 DSP ALGORITHM IMPLEMENTATION AND PERFORMANCE RESULTS....69

Comparison Metrics .........................................................................................................69
   Method of Coding .........................................................................................................69
   Test Data ......................................................................................................................70
   Weight Initialization ....................................................................................................73
   Data I/O .......................................................................................................................73
   Training and Performance Comparisons ......................................................................75

Timing in the DSP ............................................................................................................78

Linear Transversal Filter trained with NLMS in DSP .........................................................79
   Data Flow and Program Organization ...........................................................................79
   Algorithm Verification and Performance ......................................................................82
   NLMS Timing Results .................................................................................................86

Recursive Multi-Layer Perceptron trained with RTRL in DSP ........................................89
   Data Flow and Program Organization ...........................................................................89
   RTRL in C .....................................................................................................................90
   RTRL in C compared with BPTT in NeuroSolutions ....................................................91
   RTRL in DSP ...............................................................................................................94
   RTRL Timing Results .................................................................................................97

Discussion ........................................................................................................................100

5 CONCLUSION AND FUTURE WORK .................................................................102

DSP-Based Computational Engine ................................................................................102
DSP Algorithm Implementation ......................................................................................103
Future Work ....................................................................................................................105
## APPENDIX

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A</strong> ACRONYMS AND ABBREVIATIONS</td>
<td>108</td>
</tr>
<tr>
<td><strong>B</strong> SCHEMATICS AND LAYOUT OF DSP BOARD</td>
<td>110</td>
</tr>
<tr>
<td>Schematics</td>
<td></td>
</tr>
<tr>
<td>Layout</td>
<td></td>
</tr>
<tr>
<td>Photo of Finished DSP Board</td>
<td>117</td>
</tr>
<tr>
<td><strong>C</strong> DSP BOARD CONTROL SOFTWARE</td>
<td>120</td>
</tr>
<tr>
<td>C Console Software</td>
<td></td>
</tr>
<tr>
<td>VHDL Code for the CPLD</td>
<td>120</td>
</tr>
<tr>
<td>DSP Operating System Code</td>
<td>123</td>
</tr>
<tr>
<td>Main DSP Operating System Code</td>
<td></td>
</tr>
<tr>
<td>Subroutines for the DSP Operating System</td>
<td>124</td>
</tr>
<tr>
<td>Global Definitions</td>
<td>126</td>
</tr>
<tr>
<td><strong>D</strong> CODE FOR DSP ALGORITHMS</td>
<td>127</td>
</tr>
<tr>
<td>Linear Transversal Filter trained with NLMS</td>
<td>127</td>
</tr>
<tr>
<td>RTRL in C</td>
<td>132</td>
</tr>
<tr>
<td><strong>LIST OF REFERENCES</strong></td>
<td>141</td>
</tr>
<tr>
<td><strong>BIOGRAPHICAL SKETCH</strong></td>
<td>145</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>Notation and terminology used to describe adaptive filter theory.</td>
</tr>
<tr>
<td>2-2</td>
<td>Equations for the Normalized Least-Mean-Square Adaptation Algorithm</td>
</tr>
<tr>
<td>2-3</td>
<td>Computational complexity and storage requirements of RTRL and BPTT</td>
</tr>
<tr>
<td>2-4</td>
<td>Memory requirements compared for NLMS and RTRL</td>
</tr>
<tr>
<td>3-1</td>
<td>C Console functions to control the DSP board.</td>
</tr>
<tr>
<td>3-2</td>
<td>PLX Windows API functions used to communicate with the PCI9030.</td>
</tr>
<tr>
<td>3-3</td>
<td>Opcodes used in PC-to-DSP communication</td>
</tr>
<tr>
<td>4-1</td>
<td>Performance comparisons between PC- and DSP-implementations</td>
</tr>
<tr>
<td>4-2</td>
<td>Comparison of PC-trained versus DSP-trained NLMS filters.</td>
</tr>
<tr>
<td>4-3</td>
<td>The difference of weights produced by PC- and DSP-NLMS.</td>
</tr>
<tr>
<td>4-4</td>
<td>Comparison of RTRL- versus BPTT-trained filters</td>
</tr>
<tr>
<td>4-5</td>
<td>The difference of weights produced by PC- and DSP-RTRL</td>
</tr>
<tr>
<td>4-6</td>
<td>Comparison of the RTRL algorithm running the PC and DSP</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-2</td>
<td>A sampling of 100ms binned spike counts shows data sparseness.</td>
</tr>
<tr>
<td>1-3</td>
<td>BMI DSP System overview showing the C33 DSP Board.</td>
</tr>
<tr>
<td>2-1</td>
<td>Linear transversal filter.</td>
</tr>
<tr>
<td>2-2</td>
<td>Linear Combiner.</td>
</tr>
<tr>
<td>2-3</td>
<td>Multi-dimensional adaptive linear transversal filter.</td>
</tr>
<tr>
<td>2-4</td>
<td>Model of a neuron used in neural networks for signal processing.</td>
</tr>
<tr>
<td>2-5</td>
<td>Nonlinear activation functions used in neural networks.</td>
</tr>
<tr>
<td>2-6</td>
<td>A Multi-Layer Perceptron having J PEs.</td>
</tr>
<tr>
<td>2-7</td>
<td>Topology of a fully-connected Recursive Multi-Layer Perceptron.</td>
</tr>
<tr>
<td>2-8</td>
<td>Notation used to explain the training of an RMLP using RTRL.</td>
</tr>
<tr>
<td>2-9</td>
<td>Memory requirements of NLMS and RTRL compared.</td>
</tr>
<tr>
<td>3-2</td>
<td>Features of the TMS320VC33 DSP from the C33 datasheet.</td>
</tr>
<tr>
<td>3-3</td>
<td>Memory map of the C33 DSP showing internal SRAM blocks.</td>
</tr>
<tr>
<td>3-4</td>
<td>Example of a C33 parallel-addressing instruction.</td>
</tr>
<tr>
<td>3-5</td>
<td>PCI9030 features for interfacing with the PCI Local Bus.</td>
</tr>
<tr>
<td>3-6</td>
<td>Interfacing the C33 with the PCI bus through Dual-Port SRAM.</td>
</tr>
<tr>
<td>3-7</td>
<td>512K by 32-bit External SRAM Architecture.</td>
</tr>
<tr>
<td>3-8</td>
<td>Two different methods to boot the C33: EEPROM or DPM.</td>
</tr>
<tr>
<td>3-9</td>
<td>C33 DSP Board component integration.</td>
</tr>
</tbody>
</table>
Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Engineering

A DSP-BASED COMPUTATIONAL ENGINE FOR A
BRAIN-MACHINE INTERFACE

By
Scott A. Morrison
May 2003

Chair: José C. Príncipe
Major Department: Electrical and Computer Engineering

The fields of neurobiology and electrical engineering have come together to pursue
an integrated Brain-Machine Interface (BMI). Signal processing methods are used to find
mapping algorithms between motor cortex neural firing rate and hand position. This
cognitive extension could help patients with quadriplegia regain some independence
using a thought-controlled robot arm. Current signal processing methods to achieve real-
time neural-to-motor translation involve large, multi-processor systems to produce motor
control parameters. Eventually, software running in a portable signal processing system is
needed to allow for the patient to have the BMI in a backpack or attached to a wheelchair.

This thesis presents a DSP-Based Computational Engine for a Brain-Machine
Interface. The development of a DSP Board based on the Texas Instruments
TMS320VC33 DSP will be presented, along with implementations of two digital filters
and their training methods: 1) FIR trained with Normalized Least Mean Square Adaptive
Filter (NLMS) and 2) Recurrent Multi-Layer Perceptron (RMLP) trained with Real-Time
Recurrent Learning (RTRL). The requirements of the DSP Board, component selection and integration, and control software are discussed. The DSP implementations of the digital filters are presented, along with performance and timing analysis in real data collected from an Owl Monkey at Duke University.

The weights of the FIR-NLMS filter converged similarly on the DSP as they did in MATLAB. Likewise, the weights of the RMLP-RTRL filter converged similarly on the DSP as they did using the Backpropagation Through Time method in NeuroSolutions.

The custom DSP Board and two digital algorithms implemented in this thesis create a starting point for an integrated, portable, real-time signal processing solution for a Brain-Machine Interface.
CHAPTER 1
INTRODUCTION

Real-time direct interfaces between the brain and electronic and mechanical devices could one day be used to restore sensory and motor functions lost through injury or disease. Hybrid brain-machine interfaces also have the potential to enhance our perceptual, motor and cognitive capabilities by revolutionizing the way we use computers and interact with remote environments.


Motivation

Humans are the most dexterous and intelligent beings to inhabit Earth. The legs and arms are the tools the body uses to perform amazing feats such as cooking a meal, playing the piano, running marathons, planting a garden, and assembling model airplanes. The body’s sensory system—including vision, hearing, and touch—provide feedback and control of these routine and sometimes extreme tasks.

The human brain is an adaptive collection of input and output controls for sensors, motors, and levers. These systems are synchronized and controlled by the brain to perform voluntary tasks—speaking, lifting, and manipulating objects—and involuntary tasks—heartbeat, breathing, and digestion. The brain and spinal cord make up the Central Nervous System (CNS) which processes the information received from the peripheral nervous system extended over the entire body [1]. The peripheral nervous system acts as an electrical-chemical data bus carrying information to and from the CNS. This information includes messages from sense receptors for taste, smell, hearing, sight, and balance, as well as muscles for motor control. Without the nervous system, our muscles would be useless, unable to move or coordinate their actions.
Not often are we aware of the complexity of coordinating these input sensors and output controls. With the exception of sleep, the human brain is continuously processing video and audio, an uncountable number of touch sensors, sampling the smell of the air and taste of food, and performing other thought tasks. Additionally, the brain produces control parameters for muscles, facial expressions, and keeps the body alive through digestion, breathing, and chemical balance.

Altogether the human brain, peripheral nervous system, and sensors across the body form a contiguous biological computer having distributed processing power. No single component—especially the brain—may be removed or replaced without affecting the entire system. In this respect, the brain and nervous system are unlike a digital computer that has a central processing unit (CPU) directly responsible for controlling interchangeable peripherals. This is the key idea behind the brain as a biological computer—processing is distributed throughout the brain and nervous system, right down to local sensors and nerves.

Patients who have suffered damage to the brain or nervous system through injury or disease sometimes lose voluntary control over their arms, legs, and even speech. The brain as a biological computer is no longer able to communicate with or control these “peripherals.” How can we use the fields of neuroscience and electrical engineering to overcome the brain’s abilities by providing a method for a thought-controlled neuroprosthesis? How do we enable the brain to welcome this prosthesis into its peripheral representation space as a cognitive extension of the body? Can we create a Brain-Machine Interface (BMI) that would take as input neural firing patterns from the brain to produce control parameters for a series of actuators and neuroprostheses?
To answer these questions, we must first acknowledge the two paradigms—biological computers and digital computers—are fundamentally different in almost every respect. Digital computer components are usually interconnected using standard interfaces such as a parallel port, USB, or PCI bus. Engineers may add components easily by writing device driver software and adhering to the interface specifications. On the contrary, interfacing with a biological computer is not standard in any way. Even with the limited knowledge we have about the way in which neurons encode motor movement, methods for capturing and using this information to control a motorized prosthesis is still very experimental.

We do know that the brain is an electrophysiological entity consisting of some $10^{12}$ neurons arranged in various reconfigurable single- or multi-purpose networks. An even greater number of synapses are connected to the input of these neurons, providing massive webs of processing networks. Having this many neurons and synapses gives the brain a tremendous amount of processing power. More importantly, this gives the brain the advantage of plasticity. For example, sometimes the brain is able to repair damaged tissue and restore control to areas affected by a stroke [2]. This feature will be important to help the brain assimilate a neuroprosthesis into its cognitive space. In this way, thought-controlled peripherals could become part of a person’s cognitive realm, managed by the brain as easily as speech or hand motions.

A Brain-Machine Interface is sought to mimic and possibly extend the brain’s ability to control peripherals as well as received feedback from new sources. Concepts from digital as well as biological computer paradigms will be used to bridge mind with machine. Using spatial and temporal neuron firing information collected from multiple
regions in the brain, we seek a method to process this data in real-time to control a number of peripherals such as a robot arm. To establish a method for integrating the mind with machine would break down conventional I/O barriers, allowing the restoration of speech and motor control to patients with communication or motor disabilities. To do this, there is a need for real-time digital signal processing methods that would take as input neural activity, compute the parameters for peripheral control, and continually enhance the performance of such a system by learning from mistakes.

Even with advanced signal processing methods to decode the neural firing patterns, a BMI will still depend—even in a small way—on the brain’s plasticity to help integrate the two systems. In this respect, a BMI will have two adaptive signal processing systems, 1) the patient’s brain that receives visual and tactile feedback by interacting with a thought-controlled peripheral, and 2) the digital signal processing algorithm that computes motor control parameters from the patient’s neural activity and is adjusted to minimize control error.

Having presented the motivation for a BMI, I present a small step in this direction—a custom-made DSP Board in which adaptive and neural algorithms were implemented to aid the neural-to-motor translation required to interface mind with machine. The requirements of the DSP Board will be discussed, as well as the selection and integration of the components. Two popular digital signal processing algorithms—one linear, the other non-linear—were implemented in DSP in real-time to allow for thought-to-motor translation. Lastly, performance of the DSP Board and algorithms will be presented, as well as discussion on the next steps in this work.
Next, I present a description of the BMI research group, as well as the goals for the real-time digital signal processing hardware and algorithms.

**Brain-Machine Interface**

The collection and processing of neuronal firing data in a manner that extends the brain’s normal output pathways of nerves and muscles is called a Brain-Machine Interface (BMI). BMIs are sought to provide a cognitive extension of the body, allowing for thought-controlled peripherals such as computers and motorized prostheses. Patients with conditions causing severe degenerative neural disease may benefit from BMI technology. Similarly, a person with quadriplegia may also benefit from BMI technology to control a motorized arm by thought, thus restoring some independence to the patient.

Neuroscientists throughout the world are trying to learn more about how the brain encodes motor movement and processes sensory information for motor control. Kettner *et al.* showed that motor movement in primates is encoded by the collective activation of large populations of neurons in the primary motor cortex (M1) [3]. Monkeys with micro-wires implanted into the arm areas of their motor cortex were made to move their arms according to eight different directions in 3D space. Analysis of the neuron data showed that changes in single-neuron firing rates were significant to show unique information concerning the direction of the motor movement in 3D space; in fact, eight population vectors were calculated using the whole population of cells, each was close to the direction of the corresponding arm movement. Similar research involving single-cell recordings in monkeys was conducted by Zhang and colleagues, who concluded that ample information to control a neuroprosthesis could be extracted from large populations of cortical neurons [4].
Nicolelis and colleagues have extended the research involving non-human primates, focusing on recording large populations of neuronal firings while subjects perform motor tasks such as reaching for food and manipulating objects. During these experiments, single-neuron firings as well as hand position were recorded simultaneously. Linear as well as non-linear digital signal processing methods were applied to the data and resulted in highly significant predictions of hand movement trajectories over several months [5]. These results are significant for several reasons. First, there is enough information contained in hundreds—rather than millions—of single-cell neuron recordings to adequately control a robotic device. This is reiterated by Scott, Kalaska, and colleagues [6]. Secondly, day-to-day neuron recordings showed that a significant number of the same neurons were “usable” for long periods of time, even as long as two years. Additionally, it was found that once an adaptive or neural model had been found, it would require continual updates approximately every 1 to 10 minutes in order to improve accuracy of the robotic control. This can be attributed to the dynamic changes in the relationship between neuronal activity and movement, therefore making the input to the models non-stationary.

It was also found that the accuracy of robot movements was proportional to the number of neurons sampled—however, there was an upper limit beyond which an increased number of neurons would cease to help prediction [7]. Therefore, as stated earlier, an ideal BMI would need to sample hundreds of neurons, not millions. Nicolelis and Chapin suggested that a distributed network of neurons within each cortical area was responsible for producing the parameters for hand movement. Thus, should one part of the brain be damaged, the remaining parts would most likely be able to continue
controlling the motor movement. This redundancy within the brain suggests that patients who have suffered brain damage—such as stroke—could still benefit from BMI technology because most likely the remaining healthy parts of the brain could be “interfaced” for a BMI.

While the technology is still in its infancy, most closed-loop BMI systems use the procedure listed below.

1. **Extracting brain-derived signals.** There are two general ways to collect brain activity, Non-Invasive and Invasive:
   a. **Non-invasive.** Using methods that have been around for years—such as Electroencephalogram (EEG), Positron Emission Tomography (PET), and functional Magnetic Resonance Imagining (fMRI)—brain activity is monitored without physical encumbrance to the patient. The information obtained from neurons using these techniques is almost negligible compared with the number of active neurons inside the head [8]. Additionally, these surface techniques suffer from low signal-to-noise (SNR) ratio as well as poor neural activity resolution.
   b. **Invasive.** Neurosurgery is required to implant electrodes capable of touching single or multiple neurons. In many cases, multiple-electrode arrays are used to collect information from hundreds of neurons simultaneously. Even though neurosurgery techniques have been around for many years now, the risks involved with exposing the brain to infection are present with this invasive technique. In most cases, connectors to the micro-wires have to be reachable from outside the body and therefore require a connector to be mounted through the skull. This further increases the risks of post-op infection because of possible imperfections in the connector seal. The invasive technique is often preferred by BMI researchers because of its increase resolution over EEG methods, as well as higher SNR.

2. **Pre-processing of brain-derived signals.** Once brain activity has been sampled, signal processing techniques are used to “sort” the data into individual neuron spike trains. This spike-timing information is then used as the input to prosthetic control algorithms.

3. **Translating neural spike patterns into control commands.** In order to control a motorized prosthesis, the neural spike patterns have to be translated to control commands according to the will of the patient. Adaptive filters and neural networks, among others, are trained using behavioral data collected while the patient performs certain controlled tasks. Once the mathematical models have been
trained using the patient’s neural spike patterns, the neuroprosthesis may then be controlled by thought alone.

4. **Providing feedback to the patient.** As the neuroprosthesis is controlled by the patient’s thoughts, visual feedback will allow for the patient’s brain to modulate spike activity to improve the prosthetic control. Additional feedback information is also wanted. For example in the case of a motorized arm, grip force and arm torque information should be made aware to the patient in the form of haptic feedback. This could be in the form of a pressure cuff on the patient’s other arm, or a display on a computer screen [9].

The above list of BMI components is only a guideline that is repeated throughout the literature in one form or another [10, 11]. The BMI pursued by Nicolelis and colleagues—for which the work of this thesis was done—is described below.

**Duke BMI Project**

Researchers at Duke University in North Carolina—headed by Dr. Miguel A. L. Nicolelis, M.D., Ph.D.—are seeking a Closed-Loop Brain-Machine Interface for Augmenting Motor Performance. Among several partnered institutions, The University of Florida (UF) complements the Duke BMI team by providing signal processing methods for neural-to-motor translation. The goal of the project is to provide information about the way the brain encodes motor movement and use this information to make a brain-controlled motorized prosthesis [12]. The system architecture of this particular BMI is shown in Figure 1-1.

The cortical regions with motor associations in Macaca Mulatta monkeys—also called a *Rhesus* or *Rhesus Macaque* monkey—as well as Owl Monkeys are implanted with micro-electrode arrays for simultaneous single-cell neural recordings [5]. These electrodes connect to large Multichannel Acquisition Processor (MAP) boxes that digitize and sort the single-neuron channels for use in digital filters [13]. While the monkeys perform certain motor tasks, the neural as well as behavioral information is collected
simultaneously and processed using large PC workstations for real-time neural-to-motor mapping.

![Diagram](image)

Figure 1-1: General architecture of the Duke Brain-Machine Interface.

To make possible a small and portable processing solution—other than several PCs linked together—a DSP-based computational engine is needed for use in research labs, hospitals, and on portable assistive equipment such as a wheelchair. The UF DSP Board should be small enough to fit into a backpack carried by the patient or stowed on the wheelchair. Researchers at the Duke University Department of Biomedical Engineering are developing an implantable neural acquisition device that wirelessly transmits neural spike data. Whether the DSP Board interfaces with the MAP or the wireless neural telemetry device, this portable and fast computational engine will replace the cluster of computers currently used in the BMI research system.

**Data Properties**

The maximum firing rate of a neuron is less than 1000 times per second due to the refractory period required after a neuron fires [2]. In practice though, neurons rarely
sustain their maximum firing rate, and therefore the raw spike data is extraordinarily sparse. In different experimental setups, the data collected by our BMI group is sampled at least as fast as 31kHz, which provides ample information to detect and sort all the neuron spike events. After spike-sorting by the MAP, the neuron firings are collected into non-overlapping windows of 100ms—also called binning. This helps to reduce the sparseness of the data [14].

This particular bin size, 100ms, was chosen arbitrarily. Experiments using different bin sizes—from 5ms to 200ms—give varying results. For most experiments though, using the 100ms bin size produces adequate neural-to-motor translation. Therefore, all experiments in this thesis use 100ms non-overlapping bins. An example of the 100ms neural spike counts are shown in the Figure 1-2. Ten seconds of data (100 bins) are shown for 104 neurons. As you can see, only a handful of neurons show significant activity while the rest are relatively “quiet.”

![Figure 1-2: A sampling of 100ms binned spike counts shows the sparseness of the data.](image)

From Figure 1-2 it is observed that neuron number 100 does not fire at all during the 10 seconds shown in the figure. On the other hand, neurons number 42 and 43 are
very active. Even still, most neurons fire only occasionally and therefore the whole data set is exceptionally sparse.

**Neural-to-Motor Mapping Algorithms**

A wide range of signal processing methods are used in current research to translate the neural spike patterns into control commands. Finite Impulse Response (FIR) filters, Artificial Neural Networks (ANNs), Tap-Delay Neural Networks (TDNNs), and Wiener Filters are just a few methods used to translate the neural information into usable motor control. Sanchez and colleagues have continued the work by Nicolelis, showing that the FIR and Gamma Model—developed by De Vries and Principe [15]—are capable of capturing general movements, but are not as fine-tuned as the Recursive Multi-Layer Perceptron (RMLP). The advantage of the RMLP over the FIR or Gamma Model is its flexible memory and fewer free parameters. Although the linear FIR models are easier to train using the Least-Mean-Square (LMS) method, the RMLP is capable of learning more complex hand trajectories at the expense of increased computational complexity in training [16]. The Real-Time Recurrent Learning (RTRL) training method for an RMLP is considerably more complicated than the LMS method is for training an FIR [17].

This complexity may be seen differently when considering the size of the input space. Given a large input space—for example 104 neurons—the number of free parameters with a 20-tap FIR is precisely $104 \times 20 = 2080$. Multiplied by the number of outputs—XYZ is 3 outputs in this case—we get 6240 free parameters. For the RMLP, Sanchez frequently used no more than 10 Processing Elements (PEs) in the hidden state. This yields $104 \times 10 = 1040$ parameters in the first layer, added to $10 \times 3 = 30$ parameters in the second layer, plus $10 \times 10 = 100$ feedback weights, results in approximately 1170 free parameters. The RMLP has fewer free parameters than the FIR trained with LMS
because the memory is not in the input layer—as is with the FIR—but in the hidden layer. This decreases the memory utilization in the CPU, which will be a key factor as Nicolelis pursues increased numbers of recorded neurons.

**Thesis Goals**

 Currently, research institutions employ fast, multi-processor systems to compute the digital filter parameters for neural-to-motor translation. But for persons with motor disabilities that are confined to a wheelchair, a small and portable processing system is required. The need arises to have a small computational engine that is both portable and supports the digital filtering demands for the BMI.

This thesis will present a DSP-Based Computational Engine for a Brain-Machine Interface. In order to implement the algorithms mentioned previously, the development of a DSP Board based on the Texas Instruments TMS320VC33 DSP will be presented, along with implementations of two digital filters and their training methods: 1) FIR trained with Normalized Least Mean Square Adaptive Filter (NLMS) and 2) Recurrent Multi-Layer Perceptron (RMLP) trained with Real-Time Recurrent Learning (RTRL).

Together, the hardware and software presented in this thesis provides a fast and portable digital signal processing solution for a BMI. The technology of this BMI will, one day, provide rehabilitative and assistive devices for patients who have lost the ability to control limbs. In addition, this same technology could be used to extend the cognitive realm of the human brain, creating thought-based links to computers and machinery.

The Duke BMI system is composed of five main parts, 1) neural acquisition, 2) behavioral acquisition, 3) signal processing, 4) prosthetic control, and 5) haptic feedback. The focus of this thesis is highlighted in the Figure 1-3 which shows the BMI system overview.
Figure 1-3: BMI system overview showing the DSP Board discussed in this thesis.

The top-right corner of Figure 1-3 shows the DSP Board that was developed for this research. Implemented in the DSP were two neural-to-motor mapping algorithms. The scope of this thesis is limited to the development of the DSP Board as well as the implementation of the mapping algorithms. Direct connections to the neural hardware, robot arm, and haptic feedback devices are not included.

This thesis is organized into five chapters:

1. Introduction
2. Algorithms for Neural-to-Motor Translation
3. DSP-Based Computational Engine
4. DSP Algorithm Implementation and Performance Results
5. Conclusion and Future Work

In the next chapter, the two algorithms for neural-to-motor translation are discussed. Chapter 3 covers the design and manufacturing of the DSP Board. Chapter 4 discusses implementation and performance of the two algorithms executed in the DSP. Finally, Chapter 5 summarizes and concludes the outcome of this research, and talks about the future direction of this work.
Please note that Appendix A contains a list of acronyms and abbreviations used in this thesis. Should you come across an unfamiliar term, please consult this list to help you understand the material.
CHAPTER 2
ALGORITHMS FOR NEURAL-TO-MOTOR TRANSLATION

This chapter covers the theory and topology of the two algorithms implemented in the DSP for neural-to-motor translation. It is important to understand how the structure of these filters affects their implementation in digital hardware. Whether linear or non-linear, most algorithms contain two basic parts: 1) a parametric model to compute filter output, and 2) a method to find the parameters of this model (called an adaptation or learning method). This thesis will focus on two models: the Linear Transversal Filter (LTF) and the Recursive Multi-Layer Perceptron (RMLP). The former, a linear model, is trained with the Normalized Least-Mean-Square (NLMS) Adaptation Algorithm. The latter, a non-linear model, is trained with Real-Time Recurrent Learning (RTRL).

Because the focus of this research is DSP implementation rather than selection of the algorithms, only a high-level overview of each algorithm will be given. Several papers listed in the Reference Section go into great detail on why these algorithms were selected for neural-to-motor mapping. Additionally, the papers contain detailed comparisons of each method.

The two algorithms discussed in this chapter—as well as their corresponding training methods—represent only two of several algorithms identified by BMI researchers as appropriate for neural-to-motor translation. Therefore, it should not be assumed that all future systems are limited to these two algorithms.

A brief overview of each algorithm pair—LTF-NLMS and RMLP-RTRL—will be presented along with their advantages and disadvantages. Additionally, the equations for
each model will be given. The following chapter will present the DSP implementations through memory organization, data flow diagrams, as well as coding schemes used to maximize performance in the DSP.

**Notation and Terminology**

Before continuing in a discussion using different variables, equations, and terminology to explain these adaptive models, the notation used in the following sections is described in Table 2-1. This notation is used throughout this work unless exceptions are noted.

| Table 2-1: Notation and Terminology used to describe adaptive filter theory. |
|-----------------------------|----------------------------------|
| **Formula** | **Description** |
| $n \in \{0,1,2,\ldots,N-1\}$ | Discrete time index with $N$ total number of points. |
| $L$ | Filter order represented by the number of taps (Time dimension). |
| $M$ | Number of inputs (Spatial dimension). |
| $x = [x(0), x(1), \ldots, x(N-1)]^T$ | Single-dimensional input data with $N$-1 samples. Column vectors are standard, hence the vector transpose ($^T$). $x(n)$ is the input at time $n$, while $x(n)$ is the $L$-by-1 input vector starting at time $n$. |
| $w = [w_0, w_1, \ldots, w_{L-1}]^T$ | Single-dimensional weight vector with $L$ values corresponding to $L$ delays in time. |
| $y(n)$ | Scalar output at time $n$. |
| $e(n)$ | Scalar error at time $n$. |
| $d(n)$ | Scalar desired response at time $n$. |
| $v(n)$ | Scalar noise at time $n$, usually Additive White Gaussian Noise (AWGN) with zero mean and constant variance. |
| $R = E[x(n)x(n)^T]$ | The $L$-by-$L$ auto-correlation matrix of the input data. |
| $p = E[x(n)d(n)]$ | The $L$-by-1 cross-correlation vector of the input and desired data. |
| $w_o = R^{-1}p$ | Wiener-Hopf equation for the optimum tap-weights |
| $J = E[e^T e] = \frac{1}{N} \sum_{k=0}^{N-1} e^2(k)$ | Mean-Square-Error (MSE) Cost Function |

Please note that all methods discussed in this thesis apply to real-valued data only and therefore the formulas presented are not for complex-valued data. Explanations will
first be given in terms of one-dimensional input data—meaning that a single measurement from one process is used as input. This is noted as $x = [x(0), x(1), \ldots, x(N-1)]^T$. The neural data collected from the primate experiments are multi-dimensional, meaning that multiple measurements from different cortical areas are used as inputs. Specifically, the 100ms binned spike counts from hundreds of neurons are used as the input to these models—in this case, each neuron is represented by the spatio-temporal series $x^0 = [x^0(0), x^0(1), \ldots, x^0(N-1)]^T$, $x^1 = [x^1(0), x^1(1), \ldots, x^1(N-1)]^T$, and so forth. Similarly, the weight parameters will also have time and spatial dimensions.

**Linear Filter Topology**

At the heart of many discrete-time linear filters, the Linear Transversal Filter produces an additive linear combination of the tap-delayed filter input. The filter parameters $w = [w_0, w_1, \ldots, w_{L-1}]^T$ are chosen depending on the application. In some instances, the filter parameters are solved explicitly using well-defined equations such as for a low-pass filter. In other instances the filter parameters will be adapted to achieve echo cancellation or system identification. The standard linear transversal filter is shown in Figure 2-1.

![Linear Transversal Filter: L lags, 1 dimension](image)

Figure 2-1: Linear transversal filter. This model is one of the most basic forms of a Finite Impulse Response filter.
The output of the linear filter in Figure 2-1 is given by Equation 2-1:

\[ y(n) = w^T x(n) \]  

(2-1)

Note that this filter is both stable and causal, and has a finite impulse response due to the lack of feedback. This simple structure may be re-drawn to operate on multiple streams of data rather than a time-delayed sequence of data. This is done by a standard Linear Combiner, as shown in Figure 2-2.

![Linear Combiner: 1 lag, M dimensions](image.png)

Figure 2-2: Linear Combiner. This model combines the input from multiple sensors to produce the time series \( y(n) \).

The Linear Combiner is similar to a Linear Transversal Filter, except the dimension on which the operation is performed is spatial rather than temporal. Merging these two models to accommodate a linear filter operating across multiple time lags on a multiple-sensor input, we get the Linear Transversal Filter shown in Figure 2-3 in which the weights are adapted. This filter configuration supports the multi-dimensional neural input with a tap-delay for each neuron. As the number of neural inputs grows, the number of weight parameters grows linearly. This is the filter configuration that was implemented in the DSP, as will be discussed in Chapter 4.
Figure 2-3: Multi-dimensional adaptive linear transversal filter. This model was used in the BMI project, trained by the NLMS adaptation method.

With the M-by-L tap-delay input matrix $\mathbf{x}(n)$, scalar desired signal $d(n)$, and M-by-L weight matrix $\mathbf{w}(n)$, the output and error of the multi-dimensional linear filter is given by the following equations:

$$y(n) = \mathbf{w}(n)^T \mathbf{x}(n) \quad (2-2)$$

$$e(n) = d(n) - y(n) \quad (2-3)$$

Again, this linear feed-forward model is both stable and causal—furthermore, its simple topology is easily implemented in any coding language, including DSP. The weights of this model are adapted using the NLMS update rule, the method by which is explained in the following sections.

**Linear Filter Training**

The NLMS Adaptive Filter is a modification of the popular Least-Mean-Square (LMS) Adaptive Filter. Acclaimed for its simplicity of implementation, the LMS filter was pioneered by Bernard Widrow and it has been proven a robust and widely-used method for adaptive noise cancellation, adaptive line enhancement, adaptive beam forming, and many other applications [18]. In order to arrive at the NLMS adaptation
method, the theory behind Wiener Filtering, Method of Steepest-Descent, and the LMS adaptation method will be discussed first.

As mentioned earlier, an adaptive filter trained with NLMS consists of two basic components:

- **Linear transversal filter.** As shown in the previous section, the transversal filter is simply a linear combination of the input data.

- **Adaptation process.** Mathematical method to modify the weights for achieving the desired output.

In order to gauge the performance of the filter in order to adapt the parameters, the Mean-Square-Error (MSE) performance metric is often used because of its simplicity. For the set of weights \( w \) and for the filter error sequence \( e = [e(0), e(1), ..., e(N)] \), the MSE cost function \( J(w) \) is defined to be:

\[
J(w) = E[e^T e] = \frac{1}{N} \sum_{k=0}^{N-1} e^2(k)
\]  
\[(2-4)\]

An optimum filter in the MSE-sense is defined as having the weight parameters \( w_o \) such that for any other set of weights \( w_i \), the cost is minimized: \( J(w_o) < J(w_i) \).

The NLMS update rule is used to adapt the filter weights toward the optimum weights \( w_o \) to produce an output that is as close to the desired signal as possible. The method for finding the optimum weights is called the Weiner Solution.

**Wiener Filter Theory**

The Wiener-Hopf method provides the optimal weights for linear filtering. Given input data \( x = [x(0), x(1), ..., x(N-1)]^T \) and desired data \( d = [d(0), d(1), ..., d(N-1)]^T \) from single realizations of a jointly wide-sense stationary stochastic process, the MSE-optimal filter parameters are given by the Wiener-Hopf equations. With the L-by-L auto-
correlation matrix \( \mathbf{R} = \mathbb{E}[\mathbf{x}\mathbf{x}^T] \) and the L-by-1 cross-correlation vector \( \mathbf{p} = \mathbb{E}[\mathbf{x}\mathbf{d}] \), the optimal weights are \( \mathbf{w}_{o} = \mathbf{R}^{-1}\mathbf{p} \) [19]. This off-line method involves taking the matrix inverse of a sometimes large auto-correlation matrix \( \mathbf{R} \). Additionally, applications requiring strictly on-line filtering may not benefit from a Wiener filter given that finding the optimal weights requires post-analysis of the entire data set.

The weights found by the Wiener Solution do not necessarily provide output having zero error. This is caused by many reasons including sub-optimal choice of filter length L, non-stationary data, or input data that was created through a non-linear process. In any case, the optimum weights \( \mathbf{w}_{o} \) will have an MSE of \( J(\mathbf{w}_{o}) \geq 0 \). The weights \( \mathbf{w}_{i} \) produced by any other means—including the LMS adaptation method—will produce an error no less than the optimal solution. This is stated by the following equation:

\[
J(\mathbf{w}_{i}) \geq J(\mathbf{w}_{o}) \geq 0
\] (2-5)

Wiener Filter theory is used as the basis for LMS, which is an on-line method that converges to the weights \( \mathbf{w}_{o} \), minus some misadjustment which is defined as the difference between the MSE of both methods:

\[
\text{Misadjustment} = J(\mathbf{w}_{i}) - J(\mathbf{w}_{o})
\] (2-6)

**LMS Theory**

The LMS on-line method for adapting the weight parameters is based on the method of steepest-descent to approach the minimum point on the error-performance surface, \( J(\mathbf{w}_{o}) \). This works even in the case where input data statistics change over time [19]. The method of steepest-descent allows for a sample-by-sample *estimate* of the
optimum weights without having to compute the Wiener-Hopf equations each time the statistics change.

Remember that the Wiener Solution requires finding the auto- and cross-correlation parameters $\mathbf{R}$ and $\mathbf{p}$ after the full data set is collected. This is a deterministic method that is not suitable for on-line implementation. For true online updates, the LMS algorithm avoids the deterministic method of computing $\mathbf{R}$ and $\mathbf{p}$. Using a stochastic gradient algorithm rather than a recursive computation of the Wiener filter for stochastic inputs, the LMS algorithm uses the statistics of the input data to achieve an estimate of the gradient. This is done through instantaneous estimates of $\mathbf{R}$ and $\mathbf{p}$ based on the sample values of the tap-input vector and desired response, rather than the whole data. Namely, the stochastic estimates for $\mathbf{R}$ and $\mathbf{p}$ become:

\[
\hat{\mathbf{R}}(n) = \mathbf{x}(n)\mathbf{x}^T(n) \tag{2-7}
\]

\[
\hat{\mathbf{p}}(n) = \mathbf{x}(n)d(n) \tag{2-8}
\]

At each instance in time, an adjustment is applied to the weight vector $\mathbf{w}(n)$ in the direction of the steepest-descent on the performance surface—which is precisely in the direction opposite to the gradient vector of the cost function $J(\mathbf{w}(n))$. Using the stochastic estimates $\hat{\mathbf{R}}$ and $\hat{\mathbf{p}}$ as defined in Equations 2-7 and 2-8, the estimate of the gradient is described by the following equation:

\[
\hat{\nabla}J(\mathbf{w}(n)) = \frac{\partial J(\mathbf{w}(n))}{\partial \mathbf{w}(n)} = -2\hat{\mathbf{p}}(n) + 2\hat{\mathbf{R}}(n)\mathbf{w}(n) \tag{2-9}
\]

Remember that the gradient estimate $\hat{\nabla}J(\mathbf{w}(n))$ is based on the tap-weight vector $\mathbf{w}(n)$ at time $n$, and incorporates estimates of $\mathbf{R}$ and $\mathbf{p}$ using only the limited
information in the tap-input rather than the whole data. By this, the expectation operator is removed from the computation of the gradient. Consequently, the recursive computation of each tap weight in the LMS algorithm suffers from gradient noise introduced by this estimate.

Even though the gradient estimates used by the LMS algorithm might not be near those produced by a deterministic algorithm, over time the effects of gradient noise will be filtered out and the weights will converge to near the optimum solution $w_o$, minus some misadjustment which is controlled by the step size, $\mu$ [19].

From the gradient estimate in Equation 2-9, the current weight vector $w(n)$ may be updated to approach the optimum solution $w_o$ with the following steepest-descent update rule:

$$w(n + 1) = w(n) - \frac{1}{2} \mu \hat{J}(w(n)) = w(n) + \mu\left(\hat{p}(n) - \hat{R}(n)w(n)\right)$$ (2-10)

Substituting Equations 2-7 and 2-8 into the Eq. 2-10, we get:

$$w(n + 1) = w(n) + \mu\left(x(n)d(n) - x(n)x^T(n)w(n)\right)$$ (2-11)

Rearranging Equation 2-11, the LMS weight update equation becomes:

$$w(n + 1) = w(n) + \mu x(n)e(n)$$ (2-12)

An important thing to note about the LMS weight update equation above is that each weight is updated according to the local input. This is better seen by rewriting Equation 2-12 using subscripts denoting local values. With the input number represented by $i$, and the tap-delay stage represented by $j$, the LMS update equation for a single-output LTF is rewritten in local form:

$$w_i(n + 1) = w_i(n) + \mu x_i(n)e(n)$$ (2-13)
The LMS weight update written in its local form in Equation 2-13 shows that the weight update could be performed using a loop over \( i \) and \( j \)—hence, the regularity of this weight update is easily implemented in the DSP as will be discussed in Chapter 4.

**NLMS Theory**

As seen in Equation 2-13, the weight update is directly proportional to the local input \( x_{ij}(n) \), which is a member of the linear transversal filter input \( x(n) \). Consequently, when \( x_{ij}(n) \) is large, the gradient noise is amplified in the weight update. This is especially seen with bursty data such as the sparse neural data. The Normalized Least-Mean-Square method counters this problem by normalizing the update with respect to the power of the tap-input \( x(n) \), which is seen as the square of the Euclidean norm, \( \|x(n)\|^2 \). This has the effect of reducing the size of the update when the power of input is large.

The NLMS weight update equation is:

\[
\mathbf{w}(n+1) = \mathbf{w}(n) + \frac{\mu}{\|x(n)\|^2} \mathbf{x}(n)e(n)
\]  

(2-14)

A problem is encountered during “quiet” periods when the input vector has zero-power. In this case, the weight update in Equation 2-14 will be undefined due to the zero-denominator. A simple modification is made so that zero-power input will still receive a slight update according to the following equation:

\[
\mathbf{w}(n+1) = \mathbf{w}(n) + \frac{\mu}{\delta + \|x(n)\|^2} \mathbf{x}(n)e(n) , \text{ with } \delta > 0 .
\]  

(2-15)

The constant \( \delta \) is set to a small value usually in the range of \( 0 < \delta \leq 1 \). Because the NLMS method modifies the size of the update as a function of input power, the NLMS update is considered equivalent the LMS update having a time-varying step-size
denoted by $\mu(n)$. With a constant step-size of $\bar{\mu}$, the time-varying step-size $\mu(n)$ is shown in Equation 2-16:

$$\mu(n) = \frac{\bar{\mu}}{\delta + \|x(n)\|^2}$$  \hspace{1cm} (2-16)

To discourage large weight parameters which might lead to instability, as well as to keep the model flexible for changes of input statistics, one additional modification is made to the weight update rule in Equation 2-15. A weight decay parameter $\alpha$ is used to deflate the weights slowly over time. Equation 2-15 becomes:

$$w(n + 1) = w(n) + \frac{\mu}{\delta + \|x(n)\|^2} x(n)e(n) - \alpha w(n)$$  \hspace{1cm} (2-17)

The weight decay parameter $\alpha$ is usually small, typically in the range of $0 \leq \alpha << 1$. With $\alpha = 0$, Equation 2-17 is precisely the NLMS update equation without weight decay.

**NLMS Equations**

Following the previous discussion, the equations for the Linear Transversal filter trained with NLMS are as summarized in Table 2-2.

<table>
<thead>
<tr>
<th>Topology</th>
<th>Input</th>
<th>$x(n) = [x(n), x(n-1), \ldots, x(n-L-1)]^T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tap-Weights</td>
<td>$w(n) = [w_0(n), w_1(n), \ldots, w_{L-1}(n)]^T$</td>
<td></td>
</tr>
<tr>
<td>Output</td>
<td>$y(n) = w^T(n)x(n)$</td>
<td></td>
</tr>
<tr>
<td>Error</td>
<td>$e(n) = d(n) - y(n)$</td>
<td></td>
</tr>
<tr>
<td>Training</td>
<td>Weight Update</td>
<td>$w(n + 1) = w(n) + \frac{\mu}{\delta + |x(n)|^2} x(n)e(n) - \alpha w(n)$</td>
</tr>
<tr>
<td>Step Size Restriction</td>
<td>$0 &lt; \bar{\mu} &lt; \frac{2}{\lambda_{\text{max}}} \delta$ and $\delta &gt; 0$</td>
<td></td>
</tr>
</tbody>
</table>
The weight update Equation 2-17 may be re-written in the following form:

\[ w(n+1) = (1 - \alpha)w(n) + \frac{\mu x(n)e(n)}{\delta + \|x(n)\|^2} \]  

Equation (2-18)

The weight decay parameter \( \alpha \) acts as a gain control over the amount retained of the current weight. In this form, the NLMS algorithm was implemented into the DSP as will be discussed later.

**RMLP Topology**

Considering the original goal of interfacing mind with machine, we must realize that, in essence, we are trying to mimic the brain’s process of translating neural information into motor control signals. The FIR trained with NLMS was able to output a decent estimate of the hand position—however, complicated hand movements were not reproduced as well as simple hand movements [16]. In trying to produce a model that is capable of learning more complicated hand trajectories, neurobiology is used as a guide. The BMI researchers have identified the RMLP as capable of learning more complex hand trajectories. Once trained with sufficient data for those hand movements, the model was capable of operating in feed-forward mode to produce better results than the linear FIR trained with NLMS.

Before discussing the RMLP specifically, a description of neural networks in general will be given, including discussion of the MLP and RMLP topologies. The gradient is recursively estimated over the RMLP topology to yield the Real-Time Recurrent Learning method. The equations for RTRL are presented later in this chapter, and are specific to the fully-connected RMLP used in this research.
Neural Networks in General

Theory behind a neural network is based upon modeling of a neuron as an information-processing unit, having components that aid input-output modeling. The most widely-used model of a neuron has the following components:

- **Input synapses.** Synapses apply a weight to each input according to the importance of the input as a part of the input-output model.

- **Summing junction.** All of the weighted synaptic inputs are summed as input to the activation function.

- **Activation function.** In order to limit the amplitude of the output of a neuron, the activation function *squashes* the output signal into a finite value, usually on the order of [0,1] or [-1,1].

The above three components make up a neuron as used in a neural network for signal processing. This combination of synaptic inputs, summing junction, and activation function is a model loosely based on a real biological neuron. This neuron is typically represented by the picture in Figure 2-4.

![Model of a Neuron](image)

**Figure 2-4: Model of a neuron used in neural networks for signal processing.**

With synaptic input \( \mathbf{x}(n) = [x_0(n), x_1(n), ..., x_{M-1}(n)]^T \), synaptic weights \( \mathbf{w} = [w_0, w_1, ..., w_{M-1}]^T \), input bias \( b \), and summed local field \( v \), the neuron input-output model is described by Equations 2-19 and 2-20.
\[ v(n) = w^T x(n) + b \]  
\[ y(n) = f(v(n)) = f(w^T x(n) + b) \]

The nonlinear activation function \( f(\bullet) \) is usually a smooth sigmoid function \([17]\).

Typical sigmoid functions are the logistic function defined by
\[ y(n) = f(v(n)) = \frac{1}{1 + \exp(-a \times v(n))} \] and the hyperbolic tangent defined by
\[ y(n) = f(v(n)) = \tanh(a \times v(n)) \]. In both cases, the term \( a \) is used to control the slope of the nonlinear function. These functions saturate given large positive or negative inputs. Saturation helps the filter to maintain stability because the output is limited no matter what values the synaptic weights take on. Graphical examples of these sigmoid functions are shown in Figure 2-5.

This mathematical model of a neuron is the fundamental unit for creating larger neural networks, and thus is called a *processing element* (PE) or *perceptron*. Arranging multiple PEs into a single layer that maps a set of inputs into a set of outputs is called a single-layer neural network. This model is useful in some areas of signal processing,
requiring simple nonlinear mapping such as the Rosenblatt’s famous pattern-recognition machine [20].

In a single-layer network, an input layer of source nodes projects onto an output layer of neurons. Increasing the number of PEs in the network allows for more complicated input-output mapping functions. In fact, the single-layer neural network having infinite PEs has been proven to approximate any mathematical function [17]. Having infinite PEs is hardly realizable, so an alternate method is preferred.

**Multi-Layer Perceptron**

It is common to have two layers of PEs in which the output of the first layer is the input to the second layer—this yields a Multi-Layer Perceptron (MLP). The middle layer is called a *hidden layer* because neither the input nor the output is directly connected to this layer. The PEs of the hidden layer are called *hidden PEs*, and they provide an intermediate mapping between the input and output layers that is able to generate more complex surfaces than a single layer can achieve. An example of a Multi-Layer Perceptron is shown in Figure 2-6.

![Multi-Layer Perceptron](image)

Figure 2-6: A Multi-Layer Perceptron having $J$ PEs in the hidden layer and a single PE in the output layer.
The input vector is mapped onto a space provided by the hidden layer, whose output is then used as the input to the next layer. The output of the final layer—called the output layer—is then considered the overall response of the network. In this manner, multiple layers of perceptrons provide nonlinear projections from layer-to-layer, resulting in mappings that are more complicated than those produced by a linear filter. It has been shown that an MLP with a finite number of PEs can approximate any mathematical function [20].

Important to the BMI project is the learning of hand motion trajectories. Learning a sequence of inputs is beyond the capability of an MLP because its topology lacks time-dependence. Therefore, a different structure is required that is capable of capturing time-dependent data. One method is to add a tap-delay to the front-end of the MLP, creating a Tap-Delay Neural Network (TDNN). This allows simultaneous “viewing” of several ordered input vectors—thus the network may be trained to look for time-dependent sequences similar to that of the FIR trained with LMS. This time-dependence comes at a price, though. Each tap has an associated weight and therefore an $M$-dimensional input vector using $L$ taps has $M \times L$ weights—the increased number of weights makes training and generalization more difficult. The LTF discussed earlier also has a large number of free parameters and, therefore, suffers from the same problems. Hence, a TDNN with a large memory depth is not desired.

**Recursive Multi-Layer Perceptron**

Another way to add memory to a neural network is to add recurrence to the hidden layer, such as the fully-connected RMLP (Figure 2-7). This provides two main advantages. First, the number of free parameters is reduced dramatically by eliminating the tap-delay on the input layer. In doing so, the memory is moved from the input layer to
the hidden layer, now called the “state” of the system. A set of feedback weights $w^f$ controls the “amount” of feedback to each PE. The number of free parameters to achieve this memory is reduced from $M \times L$ to $J \times J$, where $J$ is the number of PEs in the hidden layer. The RMLP will have approximately $M \times J$ additional weight parameters for each layer such as the input and output layers.

![Recurrent Multi-Layer Perceptron](image)

Figure 2-7: Topology of a fully-connected Recursive Multi-Layer Perceptron with linear outputs.

Using neural data from the BMI project, the number of free parameters to achieve a certain memory depth can be compared. Common values for $M$ and $L$ are $M = 30 \rightarrow 200$ and $L = 10 \rightarrow 30$. This gives $M \times L = 300 \rightarrow 6000$ free parameters for the TDNN. For the RMLP, the number of hidden PEs is usually $J = 3 \rightarrow 10$ which yields approximately $(J \cdot J) + (M \cdot J) = 99 \rightarrow 2100$ free parameters. Having fewer free parameters, the RMLP has a smaller storage requirement than the TDNN and also a greater ability to generalize.

The second and possibly more profound advantage of the RMLP over the TDNN is the ability of the RMLP to have simultaneous, varying time resolutions. Different weights in the feedback matrix $w^f$ allow the network to simultaneously use different memory
depths for different PEs. A larger feedback weight increases the amount of “history” used by that particular PE. Likewise, a smaller feedback weight creates a shorter memory. This ability to have multiple simultaneous time-scales is contrary to the fixed memory used by the TDNN, which gives every PE the same time resolution. With fewer free parameters and the ability to have multiple, simultaneous time-resolutions, the RMLP has the ability to learn more complicated hand trajectories that are required for a BMI.

The feedback within the RMLP is not instantaneous—rather, the feedback is delayed by one time step using an ideal unit delay. Note that auto-feedback is allowed in the topology of Figure 2-7—some networks are created without auto-feedback and this choice is based on the required application.

The following equations describe the topology of the RMLP in Figure 2-7:

\[
\text{Output of first layer: } y^1(n) = f\left( w^1x(n) + w^F y^1(n-1) + b^1 \right) \tag{2-21}
\]

\[
\text{Output of Network: } y^2(n) = w^2y^1(n) + b^2 \tag{2-22}
\]

The nonlinearity in Equation 2-21 is represented by \( f(\bullet) \). This could be the logistic, hyperbolic tangent, or other nonlinearity as chosen by the user. Because the output layer in this case has linear PEs, the calculation of \( y^2(n) \) does not use \( f(\bullet) \).

One detail that cannot be stressed enough is the importance of the network state \( y^1(n) \). The RMLP only “knows” two things at each iteration, 1) the current input and 2) the previous state. The state vector \( y^1(n) \) as well as the feedback weights \( w^F \) control the memory of the network—and because a network with memory (RMLP) behaves very differently than a network without memory (MLP), one must be careful when choosing the correct topology as required by the application. Given that spatial and temporal sequences of neural data are required to reproduce hand trajectories, and knowing how
complicated the neural encoding is, it is concluded that the single-hidden-layer RMLP topology is an appropriate choice over the MLP and the TDNN.

The RMLP topology is defined by the number of inputs, hidden PEs, and outputs—this is commonly noted as \textit{INPUTS:PEs:OUTPUTS}, such as 104:5:3 for 104 inputs, 5 hidden PEs, and three outputs. The parameters of the RMLP are defined by the set of weights \( \{w^1, w^2, w^r, b^1, b^2\} \). Real-Time Recurrent Learning is the method used to train these parameters to translate 100ms spike bins into 3D hand position. The next section discusses training an RMLP using RTRL. Due to the complexity of such a discussion, only a broad view of the training algorithm is presented.

\textbf{Training an RMLP using RTRL}

As its name implies, Real-Time Recurrent Learning allows for real-time weight updates to a recurrent network. By using recursive estimates of the gradient over a trajectory of data, RTRL avoids the non-causal gradient calculation problem incurred by the backpropagation through time (BPTT) method [20].

![Notation for RMLP trained with RTRL](image)

\textbf{Figure 2-8:} Notation used to explain the training of an RMLP using RTRL.

The method of \textit{static backpropagation} is used to train an MLP. Problems are encountered when the method of static backpropagation is applied to a recurrent network
because the recurrence prevents the direct backpropagation of the error. Therefore RTRL relies on recursively estimating the gradient over a *trajectory length* of $T$ samples. A trajectory represents a sequence of inputs to produce a single output. A suitable trajectory length is chosen according to the properties of the data. In the BMI case, the trajectory length was determined based on the primate motor reaching tasks. In one experiment, for example, each reaching task seemed to take no more than 3 seconds which includes time for motion planning, execution, and termination. In light of this, a trajectory length of $T = 30$ is chosen to capture $100\text{ms} \times 30 = 3\text{s}$ of data.

As each sample is passed through the network, RTRL updates each weight according to the error’s sensitivity to that weight. The chain rule is applied directly to the cost function to achieve sensitivity equations for the weight parameters
\[
\{w^1, w^2, w^F, b^1, b^2\}.
\]
These sensitivities are calculated using Equation 2-23:
\[
\frac{\partial J}{\partial w_{ij}} = -\sum_{n=1}^{N} \sum_{k=1}^{n_k} e_k(n) \frac{\partial y_k(n)}{\partial w_{ij}} \quad \text{for all weights } w_{ij}.
\] (2-23)

The key point in the above equation is the calculation of the sensitivity of the output $y_k(n)$ with respect to the change of each weight $w_{ij}$ in the form of $\frac{\partial y_k(n)}{\partial w_{ij}}$. Accumulating this sensitivity over an entire trajectory of data allows the weights to be updated with respect to the “amount” of error produced by each weight. The weights are updated according to the steepest decent rule:
\[
w_{ij}(n + 1) = w_{ij}(n) - \frac{1}{2} \mu \hat{y} J(w_{ij}(n))
\] (2-24)

The general weight update rule above is applied to all weights and biases in the set
\[
\{w^1, w^2, w^F, b^1, b^2\}.
\]
Remember that these weight updates are estimates of the gradient
over a single trajectory—being stochastic rather than deterministic, the weight updates
suffer from gradient noise that will be largely filtered out as time goes on. To lessen the
effects of gradient noise in the weight updates, the updates may be accumulated over
several trajectories before updating the weights.

**RTRL Equations**

The RTRL algorithm was used to train the RMLP in the DSP according to the
following equations. These equations assume a fully-connected RMLP with \( n_0 \) inputs, \( n_i \)
hidden PEs, and \( n_2 \) outputs which is denoted as \( n_0:n_i:n_2 \). The RMLP topology is
described by the following equations:

Input vector: \( \mathbf{x}(t) = [x_{0}(t), x_{1}(t), \ldots, x_{n_{o}-1}(t)]^{T} \) \hspace{1cm} (2-25)

Output of hidden layer: \( \mathbf{y}^1(t) = [y_{1}^{1}(t), y_{2}^{1}(t), \ldots, y_{n_{i}}^{1}(t)]^{T} \) \hspace{1cm} (2-26)

Output of output layer: \( \mathbf{y}^2(t) = [y_{1}^{2}(t), y_{2}^{2}(t), \ldots, y_{n_{o}}^{2}(t)]^{T} \) \hspace{1cm} (2-27)

First layer weights: \( \mathbf{w}^1 = \begin{bmatrix} w_{11}^{1} & w_{12}^{1} & \cdots & w_{n_{o}n_{i}}^{1} \\ w_{21}^{1} & w_{22}^{1} & \cdots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_{i}1}^{1} & \cdots & \cdots & w_{n_{i}n_{o}}^{1} \end{bmatrix} \), size \( n_0 \) by \( n_i \) \hspace{1cm} (2-28)

First layer bias: \( \mathbf{b}^1 = [b_{1}^{1} \ b_{2}^{1} \ \cdots \ b_{n_{i}}^{1}]^{T} \), size \( n_i \) by 1 \hspace{1cm} (2-29)

Second layer weights: \( \mathbf{w}^2 = \begin{bmatrix} w_{11}^{2} & w_{12}^{2} & \cdots & w_{n_{i}n_{i}}^{2} \\ w_{21}^{2} & w_{22}^{2} & \cdots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_{i}1}^{2} & \cdots & \cdots & w_{n_{i}n_{i}}^{2} \end{bmatrix} \), size \( n_i \) by \( n_i \) \hspace{1cm} (2-30)

Output layer bias: \( \mathbf{b}^2 = [b_{1}^{2} \ b_{2}^{2} \ \cdots \ b_{n_{o}}^{2}]^{T} \), size \( n_2 \) by 1 \hspace{1cm} (2-31)

Feedback weights: \( \mathbf{w}^{F} = \begin{bmatrix} w_{11}^{F} & w_{12}^{F} & \cdots & w_{n_{i}n_{i}}^{F} \\ w_{21}^{F} & w_{22}^{F} & \cdots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_{i}1}^{F} & \cdots & \cdots & w_{n_{i}n_{i}}^{F} \end{bmatrix} \), size \( n_i \) by \( n_i \) \hspace{1cm} (2-32)
The following equations describe the input-output mapping of the RMLP as well as the mean-square-error cost function $J$.

Input to hidden layer: $y^1(n) = f \left( w^1 x(n) + w^F y^1(n-1) + b^1 \right)$ \hspace{1cm} (2-33)

Linear output of network: $y^2(t) = w^2 y^1(t) + b^2$ \hspace{1cm} (2-34)

Cost function (MSE): $J = \frac{1}{2N} \sum_{k=1}^{n_2} \sum_{t=1}^{N} \left( d_k(t) - y^2_k(t) \right)^2$ \hspace{1cm} (2-35)

Using the topology described above, the Real-Time Recurrent Learning algorithm uses the equations below to update the weights using a recursive estimate of the gradient.

The “repmat” function is used to simplify the written equations and is defined below.

$$
\text{repmat} \left( \begin{bmatrix} a & b \\ c & d \end{bmatrix}, 2, 3 \right) = 
\begin{bmatrix}
  a & b & a & b & a & b \\
  c & d & c & d & c & d \\
  a & b & a & b & a & b \\
  c & d & c & d & c & d 
\end{bmatrix}
$$ \hspace{1cm} (2-36)

The following sensitivities are calculated for each input-output pair:

$$
\frac{\partial J}{\partial w^1_{ji}} = -\frac{1}{N} \sum_{k=1}^{n_2} \sum_{t=1}^{N} e_k(t) w^2_{kj} g_{ji}(t) \text{ for, } j = 1, \ldots, n_1, \ i = 1, \ldots, n_0
$$ \hspace{1cm} (2-37)

$$
\frac{\partial J}{\partial b^1_j} = -\frac{1}{N} \sum_{k=1}^{n_2} \sum_{t=1}^{N} e_k(t) w^2_{kj} c_{ji}(t) \text{ for, } j = 1, \ldots, n_0
$$ \hspace{1cm} (2-38)

$$
\frac{\partial J}{\partial w^2_k} = -\frac{1}{N} \sum_{t=1}^{N} e_k(t) y^1_j(t) \text{ for } k = 1, \ldots, n_2, \ j = 1, \ldots, n_1
$$ \hspace{1cm} (2-39)

$$
\frac{\partial J}{\partial b^2_k} = -\frac{1}{N} \sum_{t=1}^{N} e_k(t) \text{ for } k = 1, \ldots, n_2
$$ \hspace{1cm} (2-40)

$$
\frac{\partial J}{\partial w^F_{jl}} = -\frac{1}{N} \sum_{k=1}^{n_1} \sum_{t=1}^{N} e_k(t) w^2_{kj} h_{jl}(t) \text{ for } j = 1, \ldots, n_1, \ l = 1, \ldots, n_1
$$ \hspace{1cm} (2-41)

For three-dimensional arrays, the triple-subscript notation $A_{i12}$ is used, where 1, 2, and 3, represent the indices for their respective dimensions. The colon “:” is used to specify all indices in a certain dimension. Additionally, the k-by-k identity matrix is
notated as $I_k$. Using this notation, the recursive gradients are defined by Equations 2-42, 2-43, and 2-44.

\[
c(t) = \text{repmat}(f'(z^1(t)), 1, n_1) \ast (I_{n_1} + w^F c(t-1))
\]

(2-42)

\[
g_{\gamma \beta}(t) = \text{repmat}(f'(z^1(t)), 1, n_1) \ast \left( I_{n_1} x_{\beta}(t) + w^F g_{\gamma \beta}(t-1) \right) \text{ for } \beta = 1, ..., n_0
\]

(2-43)

\[
h_{\alpha \alpha}(t) = \text{repmat}(f'(z^1(t)), 1, n_1) \ast \left( I_{n_1} y_{\alpha}^1(t-1) + w^F h_{\alpha \alpha}(t-1) \right) \text{ for } \alpha = 1, ..., n_1
\]

(2-44)

The above recursive gradient estimates are accumulated over the entire trajectory of 30 samples. At the end of the trajectory, the gradients are reset to zero to prepare for the next trajectory. In this way sequences of neural patterns are learned by the network to reproduce the 3D hand trajectory.

**RTRL Comparison with BPTT**

The RTRL algorithm was chosen over the BPTT algorithm for DSP implementation because of its causal nature that allows for on-line learning. The BPTT method—which is non-causal—relies on creating a static network from a recurrent network by *unfolding in time* the recurrence in the network. Over a trajectory length $T$, a feed-forward network is created having $T$ layers, each layer represented by a copy of the original network. The input to each layer is applied as a vector across time. The outputs of each layer are computed in forward-time order as functions of the previous layer’s output. All local activations are stored during this process. At the end of the trajectory, all outputs have been computed and the BPTT algorithm begins. The error at the output layer (the $T$th layer) is backpropagated one layer at a time using the static backpropagation method until the first layer is reached. Once all the local errors exist at each PE, the weights are updated using the steepest descent method.

The important thing to note about the BPTT method is that it computes weight updates in backward-time order (non-causal). This requires saving all outputs for the
whole trajectory—once the end of the trajectory is reached, the sensitivities are propagated backwards in time to the beginning of the trajectory. At that time the weights are updated.

Unlike the RTRL method, the BPTT method requires waiting $T$ time ticks until the trajectory data has been processed in forward-mode. Also, the activations of each PE in the unfolded network are saved during this process, hence BPTT may have excessive memory requirements for large networks with long trajectories. The RTRL method requires having only the current input as well as the previous estimate of the gradient and is therefore a causal method.

Furthermore, the BPTT algorithm is local in space, meaning that only the error local to that PE is used to update the corresponding weight, not the error of the entire unfolded network. The BPTT algorithm is not local in time, however. This arises from the fact that the errors used in weight updates were derived from future error (meaning toward the end of the trajectory). Hence, this is not an on-line algorithm because it requires calculation of $T$ samples before going backwards to updates the weights.

The RTRL algorithm is local in time because the current output and error are used to change the current set of weights (rather than error at some future time, such as the end of a trajectory). On the other hand, RTRL is not local in space—all of the weights across the entire topology are changed according to a global error, not error local to each PE. The locality in time makes RTRL a truly on-line method—however, it is computationally burdensome compared with BPTT [20]. The computational complexity, storage requirement, and time- and space-locality of BPTT and RTRL are compared in Table 2-3.
Table 2-3: Computational complexity and storage requirements of RTRL and BPTT compared. \( P \) is the number of PEs. \( T \) is the trajectory length.

<table>
<thead>
<tr>
<th></th>
<th>RTRL</th>
<th>BPTT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space Complexity</td>
<td>( O(P^3) )</td>
<td>( O(PT) )</td>
</tr>
<tr>
<td>Time Complexity</td>
<td>( O(P^4T) )</td>
<td>( O(P^2T) )</td>
</tr>
<tr>
<td>Space Locality</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Time Locality</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 2-3 shows without a doubt that RTRL is more computationally burdensome. As the network grows larger by increasing the number of processing elements (\( P \) increasing), both the computational and memory requirements grow geometrically. For smaller networks, however, the disparity is less noticeable.

**NLMS and RTRL Requirements Compared**

The two algorithms presented in this section—NLMS adaptive filter and RMLP trained with RTRL—have very different memory and precision requirements. In order to implement these algorithms in DSP, the memory utilization and minimum precision must be understood.

**Memory Requirements**

Over the years, multi-electrode arrays for brain implantation have decreased in size and have increased in observed SNR—thus, the ability to implant hundreds or even thousands of microwires into the brain has greatly increased the ability to observe thousands of neuronal firings simultaneously [21].

When considering neural-to-motor translation algorithms, this large-dimensional input space greatly increases the computational burden of the processor. Neuron sensitivity analysis could help to select the most important neurons, restrict the filter inputs to only those neurons, and therefore reduce the size of the network [22]. Additionally, various pruning techniques such as Sensitivity Based Pruning (SBP) may
be used to automatically reduce the size of the network. It is still unclear whether or not 
online sensitivity analysis to choose subsets of neurons will be used in the final BMI.

Given the number of neural inputs $n_0$, the number of hidden PEs $n_1$ (for RMLP 
only), and the number of outputs $n_2$, the memory requirements of the $L$-tap Linear 
Transversal Filter and the Recursive Multi-Layer Perceptron may be calculated. In the 
DSP, one integer or floating-point value is stored in one 32-bit word. Therefore, Table 2- 
4 shows the memory requirements in words for both filters.

Table 2-4: Memory requirements compared for NLMS and RTRL.

<table>
<thead>
<tr>
<th></th>
<th>Memory Requirement Formula</th>
<th>Example using</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear Transversal Filter Trained with NLMS</td>
<td>$n_2((n_0 + 1)L)$</td>
<td>$n_0 = 104$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3D output: $n_2 = 3$</td>
</tr>
<tr>
<td>RMLP trained with RTRL</td>
<td>$3\left[ (n_0n_1) + (n_1n_2) + n_1 + n_2 \right]$</td>
<td>Hidden PEs: $n_1 = 5$</td>
</tr>
</tbody>
</table>

The memory utilization shown in Table 2-4 includes memory for weights as well as 
for gradient estimates and temporary update storage. For the implementation in this 
thesis, the LTF trained with NLMS requires 3150 words of storage. On the other hand, 
the RMLP trained with RTRL requires 1704 words of storage. This considerable 
difference is shown for various filter parameters in Figure 2-9.

The trade-off between increasing the number of neural inputs verses algorithm size 
and complexity will need constant attention. Although the NLMS algorithm has a smaller 
memory requirement, it is easier to implement in DSP (as will be discussed in Chapter 4). 
But as the number of neural inputs increases beyond 500, the memory requirement for 
NLMS may become too cumbersome. On the contrary, the RTRL algorithm has a smaller 
memory requirement, but is mathematically complicated when compared with NLMS.
Precision Effects

Algorithms coded in MATLAB or C are subject to the limited precision of the digital computer on which they are executed. Commonly, MATLAB uses a 64-bit floating-point notation—however, floating-point operations are still subject to the floating-point register size inside the CPU. Frequently, the Intel processors have 64-bit floating-point registers for direct support of 64-bit arithmetic. High-speed and low-power DSPs are usually limited to 16- or 32-bit word length, sometimes having 40-bit floating-point registers for intermediary values. Because the precision of the digital hardware directly affects algorithms performance, it was important to consider how precision affects linear and non-linear algorithms.

First, the literature suggests that recurrent functions (RMLP) tend to be more sensitive to precision-limitations than non-recurrent functions (NLMS) [17]. This is in part due to the well-know issues involving feedback and stability. In the case of the RMLP, however, stability itself is not an issue—it is more of a problem with staying in
the saturation regions of the nonlinearity. The RMLP is highly sensitive to the linear region of the nonlinear function (Figure 2-5).

Specifically for linear-output models—as opposed to squashed-output functions having a limited range imposed by the nonlinearities—RMLPs perform significantly better with an increased number of bits of precision [23]. Holt and Baker suggest that this is not so much the case when RMLPs are used for classification, as lower-precision fixed-point arithmetic usually works well. In this case, however, the mapping algorithms are producing continuous hand trajectories for motorized prosthetic control and therefore needs the increased precision of floating-point arithmetic.

The benefits of floating-point arithmetic over fixed-point arithmetic are seen when complicated mathematical operations are performed, such as the Hyperbolic Tangent (TANH) frequently used in RMLPs (Figure 2-5). Even with subroutines to compute the real output of the TANH—rather than using a lookup table—fixed-point arithmetic introduces significant constraints when compared with floating-point arithmetic. Fixed-point algorithms require scaling and overflow control not needed by counterpart floating-point algorithms—this adds increased overhead and may limit the choice of BMI neural-to-motor translation algorithms. Xie and Jabri show that ill-effects from quantization limitations in training neural networks are sometimes fatal [24], but can be overcome with algorithm-specific tweaking. They assert that accommodating a neural network with increased precision—especially with the wider dynamic range of floating-point—allows for the greatest amount of flexibility in changing algorithms. For these reasons, a floating point DSP—the Texas Instruments TMS320VC33—was chosen over a fixed-point DSP.
Summary

Two filter methods identified by BMI researchers for neural-to-motor translation were presented—the NLMS Adaptive Filter and the Recursive Multi-Layer Perceptron trained by Real-Time Recurrent Learning. The NLMS update method for a linear transversal filter is a simple and widely-used algorithm for online learning. The number of free parameters grows considerably as the input dimension expands or the memory depth increases. The parallel multiply-accumulate operation in the C33 DSP makes the NLMS method suitable for DSP implementation as will be seen later.

For learning of more complicated hand trajectories, the RMLP provides an enhanced ability to learn trajectories of data while using fewer free parameters than the NLMS filter. Unlike the transversal filter used by the NLMS method, the RMLP is able to provide multiple memory depth resolutions to different inputs and PEs through the feedback weights. The recurrence of the hidden layer provides IIR-like features that are made stable by the nonlinear activation function.

Training of the RMLP using RTRL is a mathematically complicated method—however the features of RTRL such as online updates using estimates of the gradient make it suitable for real-time DSP implementation.

The following chapter discusses the development of the custom DSP Board based on the Texas Instruments C33 DSP.
CHAPTER 3
DSP-BASED COMPUTATIONAL ENGINE

The DSP Board was created to provide portable, high-performance digital signal processing for a Brain-Machine Interface. This chapter outlines the requirements of the DSP Board, selection and integration of the components, as well as the software that was developed to interface the DSP with a PC.

System Requirements

The previous chapter described two signal processing algorithms suitable for neural-to-motor mapping, 1) Linear Transversal Filter trained with the Normalized Least-Mean-Square (NLMS) adaptation method, and 2) the Recursive Multi-Layer Perceptron (RMLP) trained with Real-Time Recurrent Learning (RTRL). Because additional algorithms would be needed in the future, the DSP Board needed to be a flexible architecture capable of supporting linear and non-linear, as well as large and small filter configurations. Eventually, the need might arise to have multiple models—both linear and non-linear—running simultaneously in the DSP. In this case, a large amount of fast, Static Random Access Memory (SRAM) is needed for increased data and program storage.

The list of possible translation algorithms is long and diverse. This is not to say that every algorithm works equally as well or is as computationally efficient as the next. Even though every effort is made to identify parsimonious versions of BMI mapping algorithms, there will not always be the luxury of lean and simple filters. Therefore, the DSP Board must be capable of supporting all types of digital algorithms—such as those
having large tap-delay lines as well as non-linear functions—and it is the goal of the DSP Board architecture to support as many as possible. The Texas Instruments TMS320VC33 floating-point DSP provides fast, precise, and low-power signal processing to accomplish this goal.

**Portability Considerations**

Besides the computational requirements of the neural-to-motor translation algorithms, the physical aspects of the DSP Board must be considered. In doing so, three possible scenarios are described.

- **Case 1.** Computational hardware is carried by a freely roaming subject. Most likely a battery powered device, the DSP Board will have a wired or wireless interface to the neural microwires and will be strapped to the body or carried in a backpack. In this case, reducing the weight of the hardware would be top priority as well as conserving battery life.

- **Case 2.** Computational hardware is located on an assistive vehicle, such as a wheelchair. Because the hardware is not directly carried by the subject, weight is not a major consideration. Long battery life would still be a priority given that a non-tethered system is ideal.

- **Case 3.** Computational hardware is located at a central station, possibly in the same room as the subject. For research purposes, computational ability as well as reconfigurability would be more important than size and battery life. In this case, a wired or wireless neural acquisition system would be connected to the DSP Board, and all experiments would be performed within the research lab. This lessens the need for portability, but would be for initial experimental purposes and not for a final product.

The Owl and Rhesus monkeys used in BMI research are fairly small creatures, not capable of carrying more than a kilogram of weight [25]. Therefore, since Case #1 listed above requires a completely portable system, the weight of the DSP Board, neural acquisition hardware, and batteries would have to be less than one kilogram.

Alternatively, in Cases #2 and #3 above, the weight could be more than that amount because the device is not directly carried by the subject. Mounting the device on a
wheelchair would still impose some restriction on weight and size—as a result, the DSP Board should be designed as small as possible to accommodate all of these cases.

**DSP Development versus Deployment**

To this point, the computational and portability requirements of the DSP board have been discussed. These requirements will be further segmented into three categories.

- **PC-Development.** In order to design and test algorithms in the DSP, many iterations of writing and compiling code will be required to verify the implemented algorithms. In this case, the DSP Board will not be required to be deployed with a subject. Instead, the DSP Board will be connected to a PC for transferring data and running diagnostics.

- **PC-Deployment.** In portability Case #3, the DSP Board will be used for real-time signal processing while connected to a PC-based system rather than carried by the subject. Real-time neural telemetry will provide inputs to the processing algorithm running in the DSP. Thus, fast PC-to-DSP data transfer is required in this mode.

- **Field-Deployment.** Once the development of the DSP algorithms has passed all tests in PC-Development mode, the DSP Board will be deployed for use with a subject and placed in a portable backpack or on a wheelchair. In this mode, fast data transfer and PC diagnostics are not required. Therefore, this becomes a “standalone mode” apart from a PC.

During the requirements elicitation, several scenarios were discussed that required the DSP Board to have fast PC-to-DSP data transfers in PC-Deployment mode. This would be in conjunction with the Plexon Multiple Acquisition Processor (MAP) or the Duke Wireless Backpack. In this configuration, streaming neural data comes through a wired or wireless network—such as IEEE 802.11b in the case of the backpack. This data is spike-sorted in the hardware-MAP or in the software-MAP made by Plexon [13]. In both cases, the spike-sorted data would be streamed to the DSP Board attached to the computer. The neural-to-motor translation algorithms would be trained, and the DSP would output the motor control parameters.
Realizing the need for portability as well as PC-based applications, it was decided to use the fast PCI bus to interface the DSP [26]. The PCI bus offers 32- or 64-bit connectivity—running at 33MHz or 66MHz—with maximum transfer speeds of 132MB/s for 32-bit/33MHz transfers all the way up to 528MB/s for 64-bit/66MHz transfers. After surveying various PCI interface chips, and because of Printed Circuit Board (PCB) requirements and limitations, it was decided that the 32-bit/33MHz PCI bus connection would be more than adequate to fulfill the data bandwidth requirement. Additionally, many PCs do not have 64-bit PCI slots, so the 32-bit/33MHz PCI bus was selected.

Component Selection

With the requirements of the portable, high-performance DSP Board set, the components used to fulfill these requirements were chosen. In deciding which parts to select, the following criteria were used:

- **Performance.** Does the chip or component meet or exceed the requirements for the high-performance DSP Board?
- **Packaging.** Is the form factor small enough to make a light-weight board?
- **Power.** What is the operating as well as the standby power consumption?
- **Availability.** Is this component available through the regular distribution channels, or is it a hard-to-get item?
- **Price.** Can we obtain samples of this product, or is there a reasonable price to buy in small quantities?

After considering these criteria, the following components were selected.

**TMS320VC33 DSP**

The Texas Instruments TMS320VC33 (C33) is a fast, floating-point DSP capable of up to 150 MFLOPS. It was chosen because it fulfills the algorithmic requirements of
the BMI project. In addition, there is an abundance of existing support and code available through online development channels [27]. The C33 pinout and actual size is shown in Figure 3-1.

Figure 3-1: C33 DSP pinout and packaging.

The C33 is a low-power DSP that has both 1.8V core and 3.3V I/O supply voltages. This allows for the processor to consume less power at full operational speed, while supporting 3.3V peripherals. Unfortunately, like many low-power devices, the C33 does not have 5V-tolerant I/O pins. Given the overwhelming number of 5V parts that might be interfaced with the DSP in the future, it was determined that 3.3V-translating level shifters will be incorporated into the design. The Texas Instruments SN74CBTD3384DBQR 10-bit level shifters were selected because they are fast (0.25ns) and are available in small 24-pin SSOP packaging [28]. Because 42 signals required
level-shifting—including data and control inputs to the DSP—five of these parts were needed.

Incorporating a range of integrated peripherals—such as four hardware interrupts, two 32-bit timers, and a DMA controller—the C33 provided sufficient flexibility for the BMI system requirements now and in the future. Figure 3-2 shows a list of the key features of the C33 DSP [29].

- High-Performance Floating-Point Digital Signal Processor (DSP):
  - TMS320VC33-150
  - 13-ns Instruction Cycle Time
  - 150 Million Floating-Point Operations Per Second (MFLOPS)
  - 75 Million Instructions Per Second (MIPS)
  - TMS320VC33-120
  - 17-ns Instruction Cycle Time
  - 120 MFLOPS
  - 60 MIPS
- 34K × 32-Bit (1.1-Mbit) On-Chip Words of Dual-Access Static Random-Access Memory (SRAM) Configured in 2 × 16K Plus 2 × 1K Blocks to Improve Internal Performance
- x5 Phase-Locked Loop (PLL) Clock Generator
- Very Low Power: < 200 mW @ 150 MFLOPS
- 32-Bit High-Performance CPU
- 16-/32-Bit Integer and 32-/40-Bit Floating-Point Operations
- Four Internally Decoded Page Strobes to Simplify Interface to I/O and Memory Devices
- Boot-Program Loader
- EDGEMODE Selectable External Interrupts
- 32-Bit Instruction Word, 24-Bit Addresses
- Eight Extended-Precision Registers
- On-Chip Memory-Mapped Peripherals:
  - One Serial Port
  - Two 32-Bit Timers
  - Direct Memory Access (DMA) Coprocessor for Concurrent I/O and CPU Operation
- Fabricated Using the 0.18-μm (L_{eff}-Effective Gate Length) Timeline™ Technology by Texas Instruments (TI)
- 144-Pin Low-Profile Quad Flatpack (LQFP) (PGE Suffix)
- Two Address Generators With Eight Auxiliary Registers and Two Auxiliary Register Arithmetic Units (ARAU)
- Two Low-Power Modes
- Two- and Three-Operand Instructions
- Parallel Arithmetic/Logic Unit (ALU) and Multiplier Execution in a Single Cycle
- Block-Repeat Capability
- Zero-Overhead Loops With Single-Cycle Branches
- Conditional Calls and Returns
- Interlocked Instructions for Multiprocessing Support
- Bus-Control Registers Configure Strobe-Control Wait-State Generation
- 1.8-V (Core) and 3.3-V (I/O) Supply Voltages
- On-Chip Scan-Based Emulation Logic, IEEE Std 1149.1† (JTAG)

Figure 3-2: Features of the TMS320VC33 DSP from the C33 datasheet [29].

The C33 has four automatic peripheral strobes—PAGE0-PAGE3—allowing for easily memory-mapped devices. The 24-bit address bus creates a usable address range from 0x000000–0xFFFFFFFF with over 16 million addresses. Only a fraction of these
addresses are used by the C33 for internal memory-mapped peripherals, as shown in the memory map of Figure 3-3.

![Figure 3-3: Memory map of the C33 DSP showing internal SRAM blocks.](image)

In order to achieve maximum performance when running the BMI DSP algorithms in the C33, efficient use of the internal memory blocks will be necessary. Because the C33 has a dedicated multiplier in addition to an ALU, it can compute two mathematical operations per cycle, one of which is a floating-point or integer multiplication. The dual address generators allow for simultaneous RAM access as long as both locations do not reside within the same RAM block. For example, a parallel load operation can be made in a single cycle if one load is from RAM Block 2 and the other load is from RAM Block 3—or vice versa. On the contrary, two simultaneous loads may not be made from a single RAM block.

In the case of $A \oplus B = C$, where “$\oplus$” represents either addition, subtraction, or multiplication, every effort should be made to place arguments A and B in separate data
blocks. This allows for simultaneous reading of A and B and therefore the operation can be completed in a single cycle. Figure 3-4 shows an example of how different RAM blocks may be used for parallel instructions.

![Example of Parallel Addressing: Compute and element-wise vector multiplication](image)

Figure 3-4: Example of a C33 parallel-addressing instruction. This multiply-then-store instruction performs two reads, one multiply, and one store in a single cycle.

The example in Figure 3-4 shows the importance of the user allocating memory appropriately. This is especially important for multiply-accumulate operations as required by FIR filters in the form of dot products. All BMI neural-to-motor translation algorithms should be written to use this parallel architecture whenever possible. As will be discussed in Chapter 4, manually writing assembly code—versus using the C compiler—is the best way to ensure high-performance applications that fully exploit these hardware resources.

Given this parallel architecture and plentiful internal SRAM and peripheral control features, the C33 DSP provides fast, floating-point computations for the complicated BMI neural-to-motor mapping algorithms.
PLX PCI 9030 SMARTarget I/O Accelerator

In order to interface the DSP Board to a personal computer for PC-Development and PC-Deployment, a PCI bridge chip is necessary. After a survey of PCI bridging products, the PLX PCI 9030 SMARTarget I/O Accelerator was chosen [30]. Providing a 32-bit, 33-MHz PCI Bus Target Interface, the PCI9030 provides high-speed data transfers between the DSP Board and the PC host computer for high-bandwidth computation such as real-time data acquisition and filtering. The important features of the PCI9030 are listed in Figure 3-5.

- PCI Local Bus Specification r2.2-compliant 32-bit, 33 MHz Bus Target Interface Device enabling PCI Burst Transfers up to 132 MB/s
- PCI Bus Power Management Interface Specification r1.1 compliant
- PCI Local Bus Specification r2.2 Vital Product Data (VPD) configuration support
- PICMG 2.1, R2.0, CompactPCI Hot Swap Specification, Hot Swap Silicon
  - Programming Interface 0 (PI = 0)
  - Precharge Bias Voltage Support
  - Early Power Support
- PCI Target Programmable Burst Management
- PCI Target Read Ahead mode
- PCI Target Delayed Read mode
- PCI Target Delayed Write mode
- Programmable Interrupt Generator/Controller
- Two programmable FIFOs for zero wait state burst operation
- Flexible Local Bus runs up to 60 MHz
- 3.3/5V tolerant PCI and Local signaling supports Universal PCI Adapter designs
- Flexible Local Bus provides 32-bit Multiplexed or Non-Multiplexed Protocol for 8-, 16-, or 32-bit Peripheral and Memory devices
- Serial EEPROM Interface
- Nine programmable General Purpose I/O (GPIOs)
- Five programmable Local Address spaces
- Four programmable independent Chip Selects
- Programmable Local Bus wait states
- Programmable Local Read prefetch mechanism
- Local Bus can run asynchronously to the PCI Bus
- Two programmable Local-to-PCI interrupts
- Endian Byte Swapping
- 3.3V Core, Low-Power CMOS in 176-pin PQFP or 180-pin μBGA
- Industrial Temp Range operation

Figure 3-5: PCI9030 features for interfacing with the PCI Local Bus.

Both the C33 and PCI9030 are “master” type devices, meaning they control their own address and data buses and therefore are not directly connectable. It was decided to use Dual-Port RAM (DPM) as an intermediary between the C33 and the PCI9030. With the address and data buses of the C33 and PCI9030 devices connected to each side of the DPM, respectively, both devices can act as “master” devices by reading and writing to
the memory. In this way data or code is easily transferred between devices just like any memory-mapped peripheral.

Both the C33 and PCI9030 have 32-bit data buses and, therefore, ideally we would like to have a 32-bit wide DPM. In searching for such a device, it was found that 8- and 16-bit wide DPMs were more common and therefore had a better availability. It was elected to use two 16-bit DPMs together, giving a 32-bit wide memory.

Another aspect to consider is the speed of the DPM. The C33 operates on a 75MHz (13.3ns) clock and the internal C33 SRAM operates at full speed—meaning that one read or write can be performed within 13.3ns. According to the C33 data sheet, approximately 5ns of setup time is required for accessing external memory [29]. In conjunction with the setup and hold times inherent to the external memory itself, the external memory should be capable of speeds of \(13.3\text{ns} - 5\text{ns} \approx 8\text{ns}\). This is extraordinarily fast SRAM and is difficult at this point in time to find.

In lieu of 8ns SRAM, the number of wait-states to access external memory will be increased. In setting the C33 to use one wait state, a \(13.3\text{ns} \times 2 = 26.6\text{ns}\) cycle time is achieved. This will be capable of accessing memory no slower than \(26.6\text{ns} - 5\text{ns} \approx 21\text{ns}\). Of course, the number of wait-states could be continually increased—to seven, in fact—in order to accommodate all sorts of slow, external memory. On the other hand, that would be contrary to obtaining the highest performance possible.

In evaluating the available DPM, the Cypress CY7C024 Dual-Port Static Ram was chosen because it has a 15ns access time, which is faster than the requirement of 21ns. This chip was also widely available, as many distributors were frequently out-of-stock on other items. The CY7C024 has the following characteristics:
- 4K by 16-bits organization
- High-Speed 15ns access time
- Special message-passing mailboxes that generate hardware interrupts
- Master/Slave chip select allowing for pairing of devices to achieve 32-bits wide
- 100-pin TQFP package

With two of these devices linked using the Master/Slave pins, a 32-bit wide memory is created for C33-to-PCI9030 communication. The DPM will also make easier high-speed transfers to real-time DSP applications. For instance, the DPM could act as a working memory, not just a data-passing memory. DSP algorithms could keep configuration parameters—such as step size, memory depth, and stopping criterion—in the DPM accessible to both the C33 and PC. The user could monitor performance and adjust parameters kept in the DPM all the while the DSP continues processing.

Using the two Cypress dual-port memories, the DSP Board communication is represented in the following figure.

![Figure 3-6: Interfacing the C33 with the PCI bus through Dual-Port SRAM.](image)

The PCI9030 is used in the “PCI Target Direct Slave” configuration. In this mode, the PCI Target operations originate on the PCI bus, go through the PCI9030, and then pass onto the Local Bus. The PCI9030 is a PCI Bus target and a Local Bus master. In this
way, the PLX interface software (running on the PC) is used to read/write data through
the PCI9030 to the DPM on the Local Bus. Configuration registers on the PCI9030 are
modified at bootup using a serial EEPROM interfacing the PLX device (MicroChip Part
Number 93LC56B). Parameters for the DPM timing and chip select as well as the Local
Bus address range are modified from data loaded from the serial EEPROM. In all, this
makes the PCI9030 transparent to the user and fulfills the goal of linking the DSP with
the PC in a high-speed and flexible manner.

External SRAM

The C33 DSP has 34K by 32-bits internal high-speed SRAM. As mentioned earlier,
the BMI algorithmic requirements including executable code, weight parameters, and
data, may increase beyond this limit. Therefore, additional external SRAM is required.

Interfacing the C33 address/data bus to SRAM is a relatively straight-forward task.
Keeping in mind the previous discussion of wait states for external memory interfaces,
fast SRAM is required—less than 21ns access time. Additionally, a 32-bit wide memory
would be ideal to connect to the C33 data bus. However, as seen with the dual-port
memory, common SRAM widths are 8- and 16-bits. A single-chip 32-bit memory is rare,
at least at the time that this board was first designed. Because large 16-bit memories were
also scarce, it was decided to use four 8-bit memories, namely the Cypress CY7C1049B-
15VC was chosen having the characteristics below [31].

- High speed 15 ns access time
- Low active power, 1320 mW (max.)
- Low CMOS standby power, 2.75 mW (max.)
- Automatic power-down when deselected
- TTL-compatible inputs and outputs
- Easy memory expansion with /CE and /OE features
Having four CY7C1049B parts yields a total of 512k by 32-bits or 4MB of external memory. These four parts will be incorporated using the same Chip Enable line connected to different bytes of the data bus, giving the appearance of a single 32-bit memory. The following figure shows the configuration of these SRAM chips.

![512K by 32-bit External SRAM Architecture](image)

Figure 3-7: 512K by 32-bit External SRAM Architecture. These four SRAM chips have a 15ns access time, requiring one wait state by the C33 DSP.

**Bootable EEPROM**

When the C33 boots, the boot address from which code will be read is chosen based on the interrupt line that was pulled low at the time of reset. Specifically, when /INT2 is asserted at reset, the C33 will boot from address 0xFFF000. Similarly, when /INT1 is asserted at reset, the C33 will boot from 0x400000.

As stated earlier, the DSP Board was intended for use in one of three configurations—two of which require a PC-interface, and the third requiring standalone capability. When in the standalone mode, the C33 will boot from non-volatile memory such as an EEPROM. Once the boot code is established and burnt into the EEPROM, the DSP operating system and BMI algorithm will bootup when power is applied, and then the BMI system will be operational. There is no need to change the boot code when the system is deployed on a subject, hence there is no need to re-burn the EEPROM.
frequently. For this standalone requirement, an EEPROM (Atmel AT29C256) is placed at address 0xFFF000 and selected using /INT2 at boot time.

When in Development Mode using a PC, the user might want to change the C33 boot code on-the-fly in order to test new and different configurations. Having to re-burn EEPROMs in this case would be tedious. Therefore, the DPM “between” the C33 and the PCI9030 is used by the C33 to boot instead of the EEPROM. This involves placing the DPM in the C33 memory map at address 0x400000 and grounding /INT1 at reset. In this manner, the user may choose one of several boot or configuration routines for the C33 without having to burn an EEPROM. The boot mode—EEPROM or DPM—is selectable by a jumper on the DSP Board. Figure 3-8 depicts the two methods for booting the C33 DSP.

![Booting the C33 DSP](image)

Figure 3-8: Two different methods to boot the C33: EEPROM or DPM.

**Power**

The parts selected for the DSP Board have three different power requirements—1.8V, 3.3V, and 5V. According to the specifications, the PCI Local Bus provides both 5V and 3.3V power [26]. Because both 1.8V and 3.3V are needed to power the C33, Texas Instruments offers feature-filled DSP power regulators that offer both voltages from a single chip. The TI TPS70351 Dual-Output LDO Voltage Regulator operates on 5V power and outputs 3.3V and 1.8V for the DSP as well as other parts on the board [32].
This chip also provides power-up sequencing required by the DSP where the core voltage (1.8V) is turned on first, followed by the I/O voltage (3.3V). Additionally, two active-low inputs are used for manual reset signals, making this chip ideal for use on the DSP Board.

Choosing this power regulator to provide 3.3V and 1.8V also eliminates the need to draw 3.3V from the PCI Bus. Consequently, only 5V will be drawn from the PCI Bus (or from an external power supply when used in standalone mode). Having one required voltage input source is an advantage for the Deployment Mode since one 5V battery supply is necessary.

**CPLD Control**

The DSP Board now has six major components: 1) C33 DSP, 2) PCI9030, 3) DPM, 4) External SRAM, 5) Bootable EEPROM, and 6) Power Regulator. To provide the “glue logic” between all these components—such as Write Enable, Read Enable, Chip Select, and Reset signals—a Complex Programmable Logic Device (CPLD) was selected to perform the following functions:

- Provide RESET inputs to the Power Regulator and DSP as functions of RESET signals coming from a manual push-button or the PCI9030.
- Further decode the address spaces created for the DPM, EEPROM, and SRAM.
- Provide advanced interrupt control to the DSP based on the real-time operating system implemented in the DSP.
- Moderate communication between the C33 and the PCI9030.
- Aid the interfacing of expansion hardware that will be connected in the future.

Given the above requirements for the CPLD control and the number of interface signals needed, the Altera EPM3032 CPLD was chosen [33]. A member of the Altera MAX family, this 44-pin PLCC chip provides 32-pins of I/O with 600 usable gates and
32 Macrocells. Most notably, the fast 4.5ns pin-to-pin logic delays are suitable for controlling the fast 15ns SRAM and DPM.

To control the devices above, VHDL code was written to provide the necessary control signals for each component on the board. Once the VHDL code is compiled, the CPLD is programmed through the 10-pin ByteBlaster port. In this manner, the reconfigurable CPLD provides for a flexible architecture as the BMI requirements change. The VHDL code is included in Appendix C.

**Future Add-on Hardware**

The requirements of the BMI project call for high-performance digital signal processing of large samplings of neural activity. This primary requirement is kept at the forefront of this discussion. In addition to this requirement, it is known that the system will be used in a portable configuration. However, as stated earlier, there may be several stages of portability and different requirements along the way. Therefore, the DSP Board must accommodate an unknown number or type of peripherals. These peripherals may include:

- Wireless interface for receiving neural data and transmitting hand coordinates
- FPGA for implementations of FFTs or FIR filters
- Analog-to-Digital and Digital-to-Analog Converters
- Direct connection to a robotic arm or other motorized prosthesis
- Connection to additional DSPs for parallel processing

Given this wide-ranging list of devices that might be connected in the future, a general-purpose interface for connecting any (or several) of these devices was sought. To meet this requirement, two 40-pin connectors were created to contain all necessary interfacing signals—including connections to the CPLD to help meet the interfacing requirements of the future hardware.
The external add-on connectors include the following signals.

- 32 data lines
- 24 address lines
- DSP Clocks H1 and H3
- General DSP interface signals: /HOLD, /HOLDA, /PAGEx, SXF1, SXF0, R/W
- Two CPLD I/O signals for controlling the external devices

These two 40-pin headers will accommodate future hardware connected to the DSP Board, allowing for flexibility as the BMI technology evolves.

**Printed Circuit Board Fabrication**

To this point, all of the DSP Board components have been described, including the reasons for their selection. Integrating these components into a larger design requires much electrical engineering and fabrication knowledge. The literature was consulted for help to design an electrically-sound PCB [34]. The integrated DSP Board architecture may be seen in Figure 3-9.

![Figure 3-9: C33 DSP Board component integration. The top box represents the block diagram of the DSP Board. The bottom box represents the PCI Local Bus socket in the host PC.](image-url)
The Protel 99SE Schematic Capture and PCB Design Suite was used to design the DSP Board [35]. Different schematic sheets were created each containing a different subsystem. All interconnections were made after careful study of the component datasheets. The schematics created were as follows:

- Master Sheet—The top-level design incorporating all schematics
- DPM and CPLD
- C33 DSP and EEPROM, also including the Level Shifters
- Power Circuit, including the 3.3V-1.8V regulator
- PCI9030 and PCI Connector
- External SRAMS
- Add-on connectors, DSP Serial Port, and DSP JTAG Header

These schematics were compiled to place the components onto a PCB layout. Because of the relative high-speed (75MHz, 60MHz, and 33MHz) required by the board components, it was necessary to have at least a four-layer board, where the two internal planes are used for power and ground. However, the high number of component interconnections forced the addition of two more signal layers, bringing the total number of layers to six.

With the need for three different supply voltages—5V, 3.3V, and 1.8V—the “split plane” feature in Protel was used. The internal power copper plane was split into three regions according to the power requirements of the components. The power plane was connected to 5V, and two split regions were created on this 5V copper plane. The separate regions were connected to 1.8V and 3.3V, respectively. Creating the split plane is a manual operation using a freestyle tool to draw the edge of the plane. It is important to have a wide track width to achieve good electrical separation between the split planes. The six layers of the PCB are shown in Figure 3-10.
Figure 3-10: Layer stack of the 6-layer Printed Circuit Board. The two innermost layers are for power and ground, with the power plane split for three different voltages—5V, 3.3V, and 1.8V.

Having the internal power plan split into three different voltages presented a routing problem for the DSP. The 3.3V and 1.8V pins of the DSP are scattered, and therefore the split 1.8V split plane had to be drawn very carefully to accommodate the position of the 1.8V pins without restricting access to the 3.3V plane for the 3.3V pins. Ideally, each voltage would have its own power plane—increasing the total number of planes to seven or eight. Unfortunately, the price of the PCB manufacturing increases greatly beyond 6 layers. Because the DSP was the only device to use 1.8V, it was reasoned that having a split-plane would be both economical and reasonable to support the few 1.8V pins. As a result, the Figure 3-11 shows the split-plane arrangement that was used in the final design.

The main 5V power from the PCI Bus is connected from several 5V PCI fingers to the 5V power plane. Alternatively, the board may be powered using a separate 5V connector when used in standalone mode. The TI power chip uses 5V as input to produce 1.8V and 3.3V for those respective split planes.
Figure 3-11: Split power planes in the C33 DSP Printed Circuit Board. All three voltages—5V, 3.3V, and 1.8V—share the inner power plane.

In order to have high-quality electrical characteristics for the component interconnections, several design rules in Protel were created. These include *minimum trace widths* of 7mils with preferred widths of 10mils. Power and ground traces were routed using a minimum width of 10mils with a preferred width of 12mils. In the case of power connections from the PCI finger connectors, much wider traces were used to assure a solid connection to 5V and Ground through the PCI bus. Other design rules included *trace length matching* to ensure the length of data and address buses matched within a certain tolerance.

The Protel auto-router was used to complete the routing of the board. The six-layer board accommodated the routing very nicely, and all connections were manually compared to the schematic sheets to ensure accuracy. Diagrams of the PCB layout as well as the schematics may be found in Appendix B.
After having the PCBs manufactured, they were sent to a third-party company for soldering of the surface-mount devices (except for the discrete capacitors). This was necessary to ensure electrically-sound solder joints as most of the surface-mount components had very small pins. All other discrete components were soldered by hand and tested incrementally.

After assembling the DSP Board, the next step was to configure the PC software used to interface the DSP.

**Control Software**

There are levels of software in the DSP Board environment: 1) PC Software, 2) DSP Operating System, and 3) DSP Algorithms. This section will describe in brief the operation of the PC Software and the DSP Operating System. The DSP Algorithms are application-dependent and are discussed in Chapter 4. In order to understand how the interface software works, first consider the DSP Board memory map shown in Figure 3-12.

![Figure 3-12: DSP Board memory map for SRAM, DPM, and EEPROM.](image)
The PLX Local Bus configuration registers were set to access the Dual-Port Memory at address 0x000000 through 0x000FFF. Similarly, the DSP was setup to access the DPM using address 0x400000 through 0x400FFF. Remember, the PLX and DSP are accessing different “sides” of the DPM. The following two sections describe the C-console program interfacing the PLX as well as the DSP Operating System.

**PC Software**

Using the PLX Application Programming Interface (API), a C-console program was written to interface the DSP through the DPM. The console program calls C-functions that, in turn, call the PLX API functions. Thus, the console functions act as wrappers for the PLX API functions. In this manner, the user appears to be communicating directly with the DSP because the PLX and DPM devices are transparent.

The following table describes functions used in this console.

Table 3-1: C-console functions to control the DSP board. These functions are wrappers for the PLX API functions and provide transparent communication to the DSP.

<table>
<thead>
<tr>
<th>Console Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>resetDSP()</td>
<td>Reset the C33 DSP, load the default operating system.</td>
</tr>
<tr>
<td>resetPLXBoard()</td>
<td>Reset the PLX and C33 DSP, load the default operating system.</td>
</tr>
<tr>
<td>loadOS()</td>
<td>Load a new operating system into the DSP, then reset the DSP.</td>
</tr>
<tr>
<td>readMailbox()</td>
<td>Read from the DPM DSP-to-PC mailbox.</td>
</tr>
<tr>
<td>writeMailbox()</td>
<td>Write to the DPM PC-to-DSP mailbox.</td>
</tr>
<tr>
<td>fetchOpcode()</td>
<td>Create an opcode for communication with the DSP operating system.</td>
</tr>
<tr>
<td>readLocalReg()</td>
<td>Read from a PLX Local Bus configuration register.</td>
</tr>
<tr>
<td>writeLocalReg()</td>
<td>Write to a PLX Local Bus configuration register.</td>
</tr>
<tr>
<td>readPCIReg()</td>
<td>Read from a PLX PCI Bus configuration register.</td>
</tr>
<tr>
<td>writePCIReg()</td>
<td>Write to a PLX PCI Bus configuration register.</td>
</tr>
<tr>
<td>writeDSPMem()</td>
<td>Write to the DSP memory space (Internal or External SRAM)</td>
</tr>
<tr>
<td>readDSPMem()</td>
<td>Read from the DSP memory space (Internal or External SRAM)</td>
</tr>
<tr>
<td>writeDPMem()</td>
<td>Write to the DPM through the PLX.</td>
</tr>
<tr>
<td>readDPMem()</td>
<td>Read from the DPM through the PLX.</td>
</tr>
<tr>
<td>loadDSP()</td>
<td>Download a program to the DSP.</td>
</tr>
<tr>
<td>runDSPProg()</td>
<td>Execute a DSP program at any memory location.</td>
</tr>
</tbody>
</table>
In order to make the communication with the DSP work, the C-console program needs to “talk” to the PCI9030 chip via the PCI bus. The drivers for the PCI9030 support Windows 98, 2000, and XP, and support the following function calls, as well as many others that were not used for this implementation.

Table 3-2: PLX Windows API functions used to communicate with the PCI9030.

<table>
<thead>
<tr>
<th>PLX API Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PlxPciBoardReset()</td>
<td>Reset PLX chip and load parameters from EEPROM.</td>
</tr>
<tr>
<td>PlxRegisterRead()</td>
<td>Read from a PLX Local Bus configuration register.</td>
</tr>
<tr>
<td>PlxRegisterWrite()</td>
<td>Write to a PLX Local Bus configuration register.</td>
</tr>
<tr>
<td>PlxPciConfigRegisterRead()</td>
<td>Read from a PLX PCI Bus configuration register.</td>
</tr>
<tr>
<td>PlxPciConfigRegisterWrite()</td>
<td>Write to a PLX PCI Bus configuration register.</td>
</tr>
<tr>
<td>PlxBusIopWrite()</td>
<td>Write to the PLX Local Bus, which writes to the DPM.</td>
</tr>
<tr>
<td>PlxBusIopRead()</td>
<td>Read from PLX Local Bus, which reads from the DPM.</td>
</tr>
</tbody>
</table>

Using the console functions, which are based on the PLX API functions, the communication to the DSP is represented in the following figure.

![Figure 3-13: PC software interface to the DSP using the PLX API functions. The C program is used to control the DSP through the DPM.](image)

Using the console program, the user may modify DSP configuration registers, download and execute DSP code, and view and edit memory locations. The C code is easily changed to accommodate different testbench requirements, such as different methods for streaming data to the DSP.
DSP Operating System

A simple and expandable protocol was created for PC-to-DSP communication. This protocol involves a series of 32-bit commands containing different opcodes. The DSP operating system was written to support these series of commands, which include the ability to read from the DSP, write to the DSP, and execute a program in DSP memory. The following table shows the different opcodes supported by the DSP operating system.

Table 3-3: Opcodes used in PC-to-DSP communication. These opcodes are passed through the DPM along with data associated with the operation.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Write data to DSP memory.</td>
</tr>
<tr>
<td>0x01</td>
<td>Read data from DSP memory.</td>
</tr>
<tr>
<td>0x02</td>
<td>Execute DSP code.</td>
</tr>
</tbody>
</table>

When the PC wants to initiate a message to the DSP, the corresponding opcode is written to a certain memory location in the DPM (Table 3-3). If additional data is needed for the requested operation—such as data to be written to the DSP memory—the data is also placed in the DPM starting at address 0x001. A DSP interrupt is triggered using a GPIO pin on the PLX. This causes the DSP OS to read the opcode from DPM, perform the requested action, and then return a confirmation message to the PC. After completing the requested operation, the DSP returns to the idle state to wait for the next command.

This simple DSP OS is scalable in the number of unused opcodes—only 3 of the 16 possible opcodes are currently used. Because the DSP OS may be changed on-the-fly without having to burn the bootable EEPROM, the OS may be changed to accommodate more complicated DSP environments involving multi-tasking. The code for the DSP OS is included in Appendix C.
Summary

The DSP Board was developed to provide real-time digital signal processing for a Brain-Machine Interface. In keeping with this goal, a scalable DSP Board architecture was created with high-speed components, having ample room for algorithm parameters as well as high-speed floating-point processing for complicated mathematical models.

The dual-mode DSP Board provides a lightweight and portable option for standalone clinical as well as research use. In addition, for use in a PC-based research system, the PCI Local Bus provides the DSP Board with high-speed data transfer capabilities and on-the-fly swapping of the operating system.

The flexibility of the architecture is supported by the reconfigurable, high-speed CPLD as well as the general-purpose add-on connectors for future hardware. These aspects make the DSP Board a malleable platform for signal processing research.
CHAPTER 4
DSP ALGORITHM IMPLEMENTATION AND PERFORMANCE RESULTS

Two filter topologies and their training methods were implemented in the C33 DSP: 1) Linear Transversal Filter (LTF) trained with Normalized Least-Mean-Square Adaptive Filter (NLMS), and 2) Recurrent Multi-Layer Perceptron (RMLP) trained with Real-Time Recurrent Learning (RTRL). This chapter outlines the organization and coding of these algorithms in terms of data-flow diagrams, DSP memory utilization to achieve maximum performance, and timing analysis as a function of varying filter parameters.

Comparison Metrics

Each DSP filter implementation was verified for mathematical equivalence to the intended algorithm. The two algorithms were coded differently, as explained below.

Method of Coding

The linear transversal filter trained with NLMS was manually coded in C33 assembly language to achieve maximum performance. The linear filter represents a model that is ideal for C33 implementation because of its multiply-accumulate nature. For an N-element vector, the C33 parallel instructions allow for a vector-times-a-vector operation of $O(N)$. This approach was used whenever possible in the DSP implementation of NLMS.

The mathematics of RTRL to train the RMLP are far more complex than the NLMS algorithm. Specifically, the equations for recursively estimating the gradient are cumbersome operations that were best coded in C rather than assembly. The RTRL
algorithm was coded in C and then compiled into DSP code using the Texas Instruments Code Composer Studio [36]. The environment for RTRL in DSP was kept similar to that of RTRL in the PC—however, a few changes were made such as the removal of print statements. In addition, the memory was allocated differently to support data-passing between the PC and the DSP.

**Test Data**

The NLMS and RTRL algorithms were tested using neural and hand-position data collected from experiments involving an Owl Monkey at Duke University. About 38 minutes of data were collected while the monkey performed a reaching task having three distinct regions: 1) hand at rest, 2) reaching for food, 3) bringing food to mouth. The neural firings were collected into 100ms non-overlapping bins, and the hand position was down-sampled from 200Hz to 10hz (100ms) 3D coordinates. The data was segmented into two sets—a training set containing 20100 samples and a testing set containing 3000 samples. An example of the neural and hand position data is shown in Figure 4-1.

![Figure 4-1: Monkey neural and hand position data collected during a 3D reaching task.](image)

There were 104 neurons recorded in tandem with three hand position coordinates. Therefore the input dimension was 104 and the output dimension was 3. In the case of
NLMS, a tap-delay line was used to capture multiple 104-dimensional input vectors across time. By contrast, the RMLP used a single input vector of 104 neurons because the memory used by a RMLP is contained in feedback weights of the hidden layer rather than a large tap-delay line.

Training an adaptive filter or neural network using a PC involves passing over the entire data set multiple times. Each pass over the entire training set is called an *epoch*. Multiple epochs are necessary to learn the data well—this is also a substitute for having larger data sets. Generally, there is a direct relationship between the number of epochs and better filter performance—however, passing over the data too many times will cause the filter to lose generalization and might not perform as well on novel testing data. Even still, multiple-epoch learning is usually an offline method performed after large amounts of data have been collected.

Thinking ahead to a real-time BMI involving streaming neural and hand-position data, the subject must not wait until a half-hour of data is collected before training the filter parameters for motor control. Instead, a method of online learning is used that requires smaller epochs of data. This *semi-batch* method is illustrated in Figure 4-2.

![Figure 4-2: The BMI requires online training that is implemented using small epochs of streaming data.](image-url)
The DSP method for online training using 300-sample epochs is shown in Figure 4-2. Of course, the epoch size will vary depending on the application, and the epoch size is chosen according to a number of factors listed below.

- Data streaming rate
- Number of neurons in the input vector
- I/O time to load the data into the DSP
- Hand movement trajectory length
- Memory capacity of the DSP

For the experiments listed in the following section, an epoch size of 300 samples (30s) was determined to be appropriate. This was chosen for several reasons. A single reaching task took no more than 3s, and therefore the trajectory length for RTRL was set to 3 seconds (Figure 4-3). Updating the weights after a single trajectory—especially if the trajectory contains anomalies in the data—may lead to a very noisy learning path. To counter this noise, the weight updates are accumulated over multiple trajectories. For this particular data, the gradients were accumulated over ten trajectories (300 samples) before updating the weights.

![Figure 4-3: An example of a single hand movement trajectory observed in 30 samples of data (3 seconds).](image-url)
Although the epoch length was fixed to 300 samples for all DSP tests in this thesis, it is certainly a parameter that is open to adjustment depending on the specific algorithm as well as the reasons listed above. For uniformity, epochs of 300-samples were used for RTRL as well as NLMS, even though NLMS updates the weights after every sample.

**Weight Initialization**

For both algorithms, the weights were initialized to small random numbers uniformly distributed on the interval $-0.1 < w < 0.1$. These initializations were generated by MATLAB and stored into a file to be processed by the TMS compiler. Therefore, these initializations were fixed and do not vary each time the algorithm is re-started. This was done for simplicity in design. By contrast, a simple pseudo-random number generator was implemented in DSP code that used the free-running counter to seed the generator [37]. It was found that the filter would perform no better or worse using the DSP-based random initialization versus the fixed MATLAB initialization. Therefore, the fixed initialization was kept for simplicity and speed.

**Data I/O**

The data input and output requirements for an integrated BMI system are dictated by the neural acquisition hardware and robotic arm. To process this data, the DSP will have three primary functions:

- **Main function.** Train on the current 300-sample set of data as many times as possible before the next set is loaded.

- **Interrupt for data input.** When a new sample of data is ready, the main function will be interrupted to read the new data. If 300 samples have been collected, the pointer to the training data is changed to the new data.

- **Interrupt for data output.** When it is time to send control parameters to the motorized prosthesis, the main function is interrupted to compute and transmit the filter output.
The data processing functions for an integrated real-time BMI are shown graphically in Figure 4-4.

**Figure 4-4: Program and data flow sequence for a real-time BMI.**

The scenario in Figure 4-4 is not realistic right now given that the BMI system is still in its infancy. The neural and hand position data used in this thesis was taken from files in the BMI data repository. For testing purposes, 300-samples of the input and desired data were loaded into the external SRAM using C console program. After processing of the data for a fixed number of epochs, a new set of data was loaded. This scenario is shown in Figure 4-5.

It is understood that future implementations of an integrated real-time system, such as that of Figure 4-4, will require additional I/O time for streaming data. Having to service interrupts in the DSP for data input and output could greatly reduce the
computation time devoted to filter training. This thesis, however, is limited to the
testbench shown in Figure 4-5.

![Digital Signal Processing Testbench](image)

Figure 4-5: Program and data flow sequence used in this thesis.

**Training and Performance Comparisons**

The first step in comparing a PC- to DSP-implementation of NLMS was to check
weight update and intermediate values at incremental steps. After coding both algorithms
for PC and DSP, weight updates and filter output for 1, 10, and 100 epochs were
compared for PC- and DSP-output. These comparisons showed an average mathematical
difference on the order of 10e-7 which was attributed to the finite-precision difference
between the PC and DSP. Because the mathematical equivalence was confirmed, the
fundamental comparison was between PC-offline training using 20100-sample epochs
versus DSP-online training using 300-sample epochs. This is summarized in Table 4-1.

The mathematical equations for implementing RTRL may vary by
implementation—specifically, the equations for recursively estimating the gradient are
complicated and topology-dependent. To verify the RTRL equations used in this thesis would indeed train the network properly, the RTRL results were compared using the NeuroSolutions software package with an equivalent network trained with Backpropagation Through Time (BPTT). Although RTRL and BPTT are very different methods for training the parameters of an RMLP, in theory they will converge to the same solution given the same initial condition and trajectory length. This gave validity to the specific RTRL equations used for DSP implementation.

After verifying the RTRL equations in C would train an RMLP properly, the PC and DSP RTRL implementations were compared using 300-sample epochs. In theory, both the PC and DSP methods should produce the same arithmetic result given they were compiled from the same code. It should be noted that the adaptive step size method could be subject to the finite-precision difference between the PC and the DSP. Specifically, the adaptive step size method involves comparing the current and previous MSE to determine change in the step size. In comparing the current and previous MSE using a greater-than or less-than operation, the finite-precision difference may cause the program to branch differently, resulting in different step sizes at the same point in the algorithm. While this scenario is unlikely to happen often, occurring at least once may cause difference in the PC and DSP results. As will be shown later, this difference was insignificant and still allowed for conclusions to be drawn regarding the success of the implementations. Table 4-1 summarizes the PC- and DSP-implementation comparisons made in this chapter.

Again, it should be noted that the mathematical equivalence was verified for the PC and DSP implementations of NLMS using 300-sample epochs. Therefore, the PC-offline
and DSP-online methods were compared—which use differently-sized epochs—to show the feasibility of online learning using NLMS in the DSP.

Table 4-1: Summary of performance comparisons between PC- and DSP-implementations.

<table>
<thead>
<tr>
<th></th>
<th>Methods compared</th>
<th>Reason for comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NLMS in MATLAB using 20100-sample epochs</td>
<td>Compare PC-offline NLMS versus DSP-online NLMS training performance.</td>
</tr>
<tr>
<td></td>
<td>NLMS in DSP using 300-sample epochs</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>RTRL in C using 20100-sample epochs</td>
<td>Verify that the RTRL equations and methods would train a network as well as BPTT in NeuroSolutions. This gives validity to the RTRL equations.</td>
</tr>
<tr>
<td></td>
<td>BPTT in NeuroSolutions using 20100-sample epochs</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>RTRL in C using 300-sample epochs</td>
<td>Compare PC-online RTRL versus DSP-online RTRL.</td>
</tr>
<tr>
<td></td>
<td>RTRL in DSP using 300-sample epochs</td>
<td></td>
</tr>
</tbody>
</table>

In order to make the comparisons listed in Table 4-1, the following metrics were calculated:

- Signal-to-Error Ratio (SER) between filter output and error
- Average correlation coefficient between the filter output and the actual hand position averaged over a sliding window of 40 samples.
- Movement “hits” versus “misses”

The test data set contains ten “rest-food-mouth” movements. The rest of the data is considered “non-movement.” For comparison purposes, if the filter output captured 70% of the movement trajectory, it was considered a “hit.” Likewise, if the filter output failed to achieve 70% of the movement trajectory, it was considered a “miss.” For increased resolution, comparison metrics were calculated for the movement regions and the non-movement regions separately. For example, all of the windowed correlation coefficients were averaged over the movement regions and the non-movement regions separately—
this provided two correlation coefficients, one each for movement and non-movement. Similarly, the SER was calculated for both regions as well.

Graphical plots of the filter output versus the actual hand position will show how well the filters reproduce hand trajectory from neural input. To compare the weight parameters produced by the DSP and PC, the mean and standard deviation were calculated for the error between the weight matrix elements.

Both the PC- and DSP-trained filters provide 3D coordinate outputs. These outputs are compared with the actual monkey hand position to determine the size of the error at each point. Based on the filter output, the probability may be calculated for errors of different magnitude. For example, the probability of the filter producing an error greater than 10mm may be calculated. The same can be done for 20mm, and so forth. By plotting the probability of different errors versus the range of error, the results are an additional way of comparing filter performance, namely the Cumulative Error Measure (CEM). The CEM will be shown for each filter as they are compared later in this chapter.

Even though any single comparison metric is not sufficient to determine whether the DSP implementation was correct, using all of these metrics together provides a more thorough understanding of algorithm performance. After discussing the DSP implementation of each filter, timing measurements for each will be given as functions of filter size and memory depth. This information will be used as a guide in the future to help select adequate neural-to-motor mapping algorithms that are both fast and accurate.

**Timing in the DSP**

In order to obtain accurate timing for each algorithm, a general-purpose output pin XF0 was toggled at the end of each iteration. Using a Tektronix TDS3034 digital
A square-wave was timed with a half-period representing a single iteration. An example of the oscilloscope display is shown in Figure 4-6.

![Oscilloscope screen capture showing the method for algorithm timing.](image)

With the code running in an infinite loop over 300 samples of data, the algorithm computation time was measured and recorded. Using these timing measurements, we may calculate the number of epochs possible in 30 seconds. For example, if 300 samples of data may be processed 70 times during 30 seconds, then we say that the training algorithm was able to compute 70 epochs of data in real-time. Generally, an increased number of epochs provide more time to learn from that particular group of data.

**Linear Transversal Filter trained with NLMS in DSP**

The Normalized Least-Mean-Square Adaptive Filter was the first of the two algorithms to be implemented in the DSP. The first step in manually coding this adaptive filter was to understand the data flow and program organization.

**Data Flow and Program Organization**

The DSP code is organized into two parts: 1) linear filter topology providing estimated hand position output, and 2) adaptation method to update the weights as a function of error. The input to the linear transversal filter is a 104-element vector of
100ms binned spike counts. The 3D filter output is computed, and then the error is used to update the weights. This process is illustrated in the data flow diagram of Figure 4-7.

![Data flow diagram for training a linear transversal filter using the NLMS adaptation method.](image)

The output of the filter is given by the summation of the element-wise matrix multiplication \( y(n) = \mathbf{w}(n)^T \mathbf{x}(n) \), which is better seen in the following form:

\[
y(n) = \sum_{i=0}^{M-1} \sum_{j=0}^{L-1} w_{ij}(n)x'(n-j)
\]

(4.1)

The DSP multiply-accumulate instructions are used to implement Equation 4.1 by using three pointers to each of the weight matrices \( \mathbf{w}_x, \mathbf{w}_y, \mathbf{w}_z \) and one pointer to the M-by-L input matrix, \( \mathbf{X} \). Because only four of the eight floating point registers may be used for parallel instructions, the coding example is limited to calculating \( y_x(n) \) and \( y_y(n) \) using parallel multiply-and-add instructions. The output \( y_z(n) \) uses two individual instructions to accomplish the same result. A single hardware-controlled loop over the entire M-by-L input matrix makes this a fast computation.
Figure 4-8: DSP Code example for parallel instruction use and hardware loop control to calculate the output of the NLMS adaptive filter.

The make the above code example run as fast as possible, the weight matrices were placed in a separate DSP memory block than the tap-input matrix $X$. Additionally, the executable code was placed in another separate block. This gives the C33 simultaneous access to all three locations to maximize performance by avoiding pipeline stalls and additional wait-states for memory access [38, 39]. This was accomplished by using the memory map in Figure 4-9.

![Figure 4-9: Memory map](image)

Having calculated the filter output, the filter error is used to update the weight matrices. From Chapter 2, the NLMS weight update equation with step size $\mu$, weight decay parameter $\alpha$, and regularization term $\delta$ is shown below for convenience:

$$w(n + 1) = (1 - \alpha)w(n) + \frac{\mu x(n)e(n)}{\delta + \|x(n)\|^2} (4.2)$$

Given a neuron having fired zero times in a particular 100ms bin, the weight update in Equation 4.2 would be zero. We could assume, therefore, that computation time could be saved by not calculating weight updates for zero-input. In reality, however, checking for zero-input requires additional “if” statements placed in the loop of Figure 4-8. This modification interrupts the regularity of the loop and thus the DSP performs in a non-
optimal fashion. Given this result, the code shown in Figure 4-8 was not modified to check for zero-input.

![Diagram of DSP Memory Organization for NLMS adaptive filter]

Figure 4-9: Placement of NLMS code and data into the DSP memory map for maximum parallelism.

**Algorithm Verification and Performance**

To gauge the possible difference in precision as well as to check the mathematical-equivalence of the hand-coded DSP NLMS algorithm, 1, 10, and 100 epochs of 300-sample data were trained in the DSP. The weight matrix produced in the DSP was compared to that produced using MATLAB. The mean squared difference between the elements of the two weight matrices was 2.78e-14 with a standard deviation of 4.35e-14. Given the small MSE between the weight matrices, it was concluded that the algorithms were mathematically identical. Furthermore, the precision difference between MATLAB and the DSP was not significant enough to be noticeable after 100 data samples. However, this effect could be amplified after many additional epochs of data.

Switching to the epoch scheme illustrated in Figure 4-2, the two NLMS adaptive filters—one in the PC, the other in the DSP—were trained using 100 epochs of the 20100-sample training set. Remember that, according to Figure 4-2, the PC filter is trained using 100 epochs of length 20100, and the DSP filter is trained using 100 epochs...
of length 300 for each of 67 sequences of data. With this scheme, all of the trained data was used exactly the same number of times. The only difference is the arrangement of the epochs.

The output of both the PC- and DSP-trained NLMS filters is shown in Figure 4-10 along with the actual monkey hand position.

![Figure 4-10: Output comparison between the PC-trained NLMS filter and the DSP-trained NLMS filter.](image)

**Table 4-2: Comparison of PC-trained versus DSP-trained NLMS filters.**

<table>
<thead>
<tr>
<th>Metric</th>
<th>PC-Trained NLMS Filter</th>
<th>DSP-Trained NLMS Filter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Movement Hits</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>Movement Misses</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Total Movements</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Correlation Coefficient (movements)</td>
<td>Average = 0.8172</td>
<td>Average = 0.8305</td>
</tr>
<tr>
<td></td>
<td>Std. Dev. = 0.1012</td>
<td>Std. Dev. = 0.0884</td>
</tr>
<tr>
<td>Correlation Coefficient (non-movements)</td>
<td>Average = 0.0838</td>
<td>Average = 0.0866</td>
</tr>
<tr>
<td></td>
<td>Std. Dev. = 0.3053</td>
<td>Std. Dev. = 0.2737</td>
</tr>
<tr>
<td>SER (movements)</td>
<td>Average = 5.1802 dB</td>
<td>Average = 5.4634 dB</td>
</tr>
<tr>
<td></td>
<td>Std. Dev. = 1.3198 dB</td>
<td>Std. Dev. = 1.6377 dB</td>
</tr>
<tr>
<td>SER (non-movements)</td>
<td>Average = 3.1013 dB</td>
<td>Average = 2.1080 dB</td>
</tr>
<tr>
<td></td>
<td>Std. Dev. = 2.8899 dB</td>
<td>Std. Dev. = 2.8327 dB</td>
</tr>
</tbody>
</table>
As seen in Figure 4-10, the testing data set contained ten “rest-food-mouth” movements. The PC-output was almost identical to the DSP-output, and a more precise comparison is seen in the Table 4-2. The average movement-region SER for the PC-output and the DSP output was 5.1802 dB and 5.4634 dB, respectively. Likewise, the non-movement regions had average SERs of 3.1013 dB and 2.1080 dB. These measurements had similar standard deviations. The similarity of the SER values shows that the small-epoch learning scheme used in the DSP produced similar results to the traditional-epoch scheme used by the PC. Additionally, the correlation coefficients for movement regions of 0.8712 and 0.8305 for the PC and DSP, respectively, uphold this conclusion.

Notice the poor correlation coefficients produced for the non-movement regions (Table 4-2). The correlation coefficient calculated over the non-movement regions is poor because of the filter output shown in Figure 4-10. The linear transversal filter did an inadequate job at capturing the steady non-movement regions, therefore causing a low correlation coefficient. This is partly caused by the LTF itself, but mostly caused by limited training of the filter. Ideally, the filter would be trained over and over again until the optimum parameters are found—such as step size, weight decay parameter, and more epochs over the data. Because the goal of this research was implementation, the filters were not exhaustively trained to achieve the best possible output.

In some respects, the small-epoch training scheme produced better results than the traditional-epoch scheme. This is supported by the 7/10 hits produced by the DSP-trained filter compared with the 5/10 hits produced by the PC-trained filter. Likewise, the DSP-trained filter had a higher correlation coefficient during movements and non-movements.
Figure 4-11 shows the Cumulative Error Measure for each of the PC- and DSP-trained filters.

![Cumulative Error Measure to compare PC- and DSP-Trained NLMS Filters](image)

For $X$ ranging from 0mm to 100mm, the graphs in Figure 4-11 may be interpreted as follows: “What is the probability that the difference of filter output versus actual hand position is greater than $X$mm?” Using this metric, both the PC- and DSP-trained filters have error probabilities that are similar. The top graph shows that the DSP-trained filter has a slightly greater probability of producing an error less than 30mm. On the other hand, during movement regions only, the bottom graph shows that the PC-trained filter has a slightly greater probability of producing an error less than 30mm for the movement regions. Remember that these filters are identical with exception for the online- versus offline-training scheme (Figure 4-2).
The DSP filter weights for the three matrices \((\mathbf{w}_x, \mathbf{w}_y, \mathbf{w}_z)\) were compared with the PC-trained filter weights. The mean and standard deviation of the difference of the matrix elements was calculated and shown in Table 4-3.

Table 4-3: The difference of weights produced by PC- and DSP-NLMS.

<table>
<thead>
<tr>
<th></th>
<th>(\mathbf{w}_x)</th>
<th>(\mathbf{w}_y)</th>
<th>(\mathbf{w}_z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>3.0243e-6</td>
<td>4.7112e-6</td>
<td>9.8010e-7</td>
</tr>
<tr>
<td>Std. Dev.</td>
<td>3.4999e-5</td>
<td>4.2366e-5</td>
<td>4.0249e-5</td>
</tr>
</tbody>
</table>

The elements of the weight matrices were similar in value, having a small mean difference. Even though the algorithms may exhibit similar MSE performance, the weight matrices are not guaranteed to be the same because they use different epoch lengths (DSP uses 300-samples and PC uses 20100 samples). Therefore, the weight matrix comparison is not by itself sufficient proof that the algorithm was a successful implementation. All of the algorithm performance measurements must be considered to make this determination.

When considered altogether, these performance metrics as well as the filter output suggest that the DSP NLMS algorithm was coded correctly and is able to train at least as well as the NLMS algorithm coded in MATLAB. The timing results are discussed in the following section.

**NLMS Timing Results**

With the code running in an infinite loop over 300 samples of data, the algorithm computation time was measured and recorded. Having an input of 104 neurons with 10 taps in the delay line, one iteration to produce a 3D output and update the weights took 576us. Likewise, 300 weight updates across the 30s data set took 0.173 seconds. This implies that up to 173 epochs over the 300-sample data set may be computed before the next 30s set of data is loaded.
It is important to know how filter size, both size of the input vector and depth of the tap-delay line, affects the speed of the algorithm. The relationship between filter parameters and speed will directly affect algorithm choice in the future of the project. Timing analysis was done for varying input sizes as well as memory depths. For each trial, only one parameter was changed and the timing captured. For the purposes of timing input sizes larger than 104 neurons, surrogate data was used to increase the size of the input space. This was done for timing purposes only, and the output of these trials was ignored. The timing results of varying memory depth are shown in Figure 4-12.

![Figure 4-12: Timing analysis of the NLMS filter versus memory depth. For filter orders greater than 15, the weights are stored in external memory.](image)

The memory depth was varied from 2 taps to 50 taps. Large weight matrices for depths of 15 taps and greater needed to be stored in the external memory. The top diagram in Figure 4-12 shows the linear increase in computation time for increasing memory depth. This is true whether the weights are stored internally or externally,
although there are different linear relationships for each memory configuration (as denoted with the color red versus yellow).

The bottom diagram in Figure 4-12 shows the number of epochs able to be computed within 30s. With 50 taps, only 21 epochs were computed during 30s. Conversely, as the filter size decreases to 5 taps, almost 400 epochs may be computed during 30s. Similar results are shown for varying input size as shown in Figure 4-13.

Figure 4-13: Timing analysis of the NLMS filter versus input size. For input vectors larger than 150 neurons, the weights are stored in external memory.

From the beginning of this project, it was made clear that the number of microwires implanted into the cortical regions of the subject’s brain would increase with time. As mentioned earlier, the data used for testing has recordings from 104 neurons. As the number of neurons approaches 500, Figure 4-13 shows a linear increase in computation time. This result is expected given the linear increase in the size of the weight matrix and required computations. With 500 neurons over 10 taps, as many as 22 epochs maybe computed over 30s of data. Likewise, decreasing the number of input neurons to as little
as 30 allows for the computation of 630 epochs on 30s of data. This makes clear the effects of increasing the number of sampled neurons, an effect that will have to be accounted for when considering the topology of the mapping algorithm.

**Recursive Multi-Layer Perceptron trained with RTRL in DSP**

The Real-Time Recurrent Learning algorithm to train the Recursive Multi-Layer Perceptron is more complex than the NLMS algorithm discussed in the previous section. The recursive gradient estimate equations shown in Chapter 2 involve multiple nested loops over all the input- and hidden-layer weights. Furthermore, the RTRL algorithm calls for maintaining a larger collection of intermediary update values, such as the recursive gradient estimate, the previous weight update, and the current cumulative weight update. Therefore, the task of maintaining this list of memory pointers and nested loop variables is prohibitively large. Because of this, manually writing the RTRL algorithm into DSP assembly code was too cumbersome. The Texas Instruments C3x/C4x Code Composer Studio was used instead to produce DSP code from compiled C code.

**Data Flow and Program Organization**

While the RMLP topology is relatively easy to implement in DSP, the RTRL training method is more complicated. For a 104-dimensional input, the filter output is calculated using the RMLP topology. The sensitivities are accumulated over a trajectory length of $T = 30$. In order to smooth the gradient noise that might result from a single, bad trajectory, the weights were updated every $U$ trajectories where $U$ is a user-selected parameter called “trajectories per update.” This was a slight modification over the standard RTRL algorithm which calls for updates after every trajectory. The number of trajectories per update was set at $U = 10$ for all experiments. Given the trajectory length
of $T = 30$, the weights were updated every 300 samples or 30 seconds. This data flow description is summarized in Figure 4-14.

![Figure 4-14: State diagram representing the RMLP topology and RTRL training method that was implemented in C and the DSP.](image)

The RMLP topology and RTRL training method were first coded in C and then compiled for DSP as described next.

**RTRL in C**

The first step in implementing RTRL in the DSP was to code the algorithm in C and then test it on a PC. Using the equations from Chapter 2, the RTRL algorithm was coded in a C style that was directly compatible with the DSP. The C-compiler in Code Composer is able to generate more efficient machine code if certain guidelines are followed [36, 40]. For example, passing arguments to functions using pointers instead of values greatly increases performance. Further performance increases are realized by
avoiding function calls if possible, such as creating functions for major rather than minor tasks.

In addition to the equations listed in Chapter 2, a simple adaptive step size was implemented to speed up learning by encouraging a larger step size. The MSE was calculated after passing over 300 samples of data. If the error was less than the previous error, the step size was increased. Likewise, if the error was greater than the previous error, the step size was reduced. This helped to ensure quick learning through a larger step size at the beginning of the training, and then a smaller step size toward the end of training.

Using the small-epoch scheme shown in Figure 4-EPOCH, the C code was used to train a 104:5:3 fully-connected RMLP. Each 300-sample data set was processed 50 times which was equivalent to 400 epochs on the whole data set. The weights of the RMLP were used to produce output of the testing set.

**RTRL in C compared with BPTT in NeuroSolutions**

It was important to know whether the RTRL algorithm coded in C was adequate to use in place of the BPTT algorithm used by BMI researchers through NeuroSolutions. For comparison, NeuroSolutions was used to train a similar 104:5:3 RMLP using the Backpropagation Through Time method, as discussed in Chapter 3. The same step sizes and slope of the nonlinearity were used in NeuroSolutions as in the RTRL trial. Remember that, even though the same initial conditions were used, the two algorithms will produce different results—the purpose of comparing RTRL with BPTT was to ensure the RTRL equations were implemented correctly. The output of both filters as well as the monkey’s actual hand position are shown together in Figure 4-15.
The output of both the RTRL- and BPTT-trained filters are compared with the monkey’s actual hand position (Figure 4-15). The figure shows that the output of both filters is similar and captures the hand trajectories well, but not perfectly. The performance of these two filters is better compared with metrics listed in Table 4-4.

Table 4-4: Comparison of RTRL- versus BPTT-trained filters.

<table>
<thead>
<tr>
<th>Metric</th>
<th>RTRL in C</th>
<th>BPTT in NeuroSolutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Movement Hits</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>Movement Misses</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Total Movements</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Correlation Coefficient (movements)</td>
<td>Average = 0.8128, Std. Dev. = 0.1187</td>
<td>Average = 0.8406, Std. Dev. = 0.1394</td>
</tr>
<tr>
<td>Correlation Coefficient (non-movements)</td>
<td>Average = 0.1067, Std. Dev. = 0.2422</td>
<td>Average = 0.0661, Std. Dev. = 0.2574</td>
</tr>
<tr>
<td>SER (movements)</td>
<td>Average = 5.9860 dB, Std. Dev. = 1.6858 dB</td>
<td>Average = 6.5717 dB, Std. Dev. = 1.6385 dB</td>
</tr>
<tr>
<td>SER (non-movements)</td>
<td>Average = 4.5212 dB, Std. Dev. = 2.9271 dB</td>
<td>Average = 5.5540 dB, Std. Dev. = 3.5701 dB</td>
</tr>
</tbody>
</table>
The RTRL-trained filter captured one additional “hit” over the BPTT-trained filter. On the other hand, the BPTT-trained filter had a slightly better correlation coefficient for movement regions, 0.8406 versus 0.8128 for the RTRL. Similarly, the BPTT had a slightly better SER of 6.5717 dB versus 5.9860 dB for the RTRL. Remember, these filters were not exhaustively trained to achieve the optimum output—therefore, the non-movement regions suffered from poor correlation coefficients. Overall, the performance metrics of both these filters were very similar, as well as the hand position output as seen in Figure 4-15. One additional comparison was made through the Cumulative Error Measure seen in Figure 4-16.

Both filters exhibit a similar error probability for the entire test trajectory. However, the CEM for movement regions shows different filter performance. The BPTT-trained filter has a higher probability of producing errors less than 30mm. Conversely, the RTRL-trained filter has a higher probability of producing errors less than 60mm. This may be interpreted to mean that, in general, the RTRL-trained filter produces a narrower
range of errors. By contrast, the BPTT-trained filter produces a wider range of errors, but is more likely to produce smaller errors than the RTRL-trained filter.

The interpretation of these performance metrics is not, by any means, an exhaustive comparison between the learning abilities of RTRL versus BPTT. It was simply a way to show that the C-coded RTRL algorithm was able to train a RMLP in a way that was comparable to BPTT. Having shown that the output of these filters was sufficiently close, the C-coded RTRL algorithm was ported to DSP code.

**RTRL in DSP**

The C-code written for RTRL was loaded into the TI C33 Code Composer Studio. Although the C-code was written to be simple in nature and directly compatible with DSP code, the following modifications were made:

- Print statements were removed
- Weight update and cumulative gradient parameters were no longer dynamically allocated by C. Instead, all allocations were made in an assembly file to control placement in DSP memory. The assembly file was linked with the C-program.
- Weight initialization files were converted to assembly and linked with the C-program.

Code Composer was used to compile the C-code, and the entry point was placed at address 0x800100. Similar to the NLMS environment, the PC console program was modified to download 300 samples of data, execute the RTRL program, and then retrieve the MSE from the DSP. This loop was executed over the entire training set using 300-sample epochs. After training, the weights were used to produce the output shown in Figure 4-17.
The weights produced by the RTRL algorithm running in the DSP were compared with those produced by RTRL running on a PC, and the results are shown below.

Table 4-5: The difference of weights produced by PC- and DSP-RTRL.

<table>
<thead>
<tr>
<th></th>
<th>$w_1$</th>
<th>$w_2$</th>
<th>$b_1$</th>
<th>$b_2$</th>
<th>$w^F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>1.1501e-4</td>
<td>7.2970e-5</td>
<td>1.4401e-5</td>
<td>4.6349e-6</td>
<td>0.00010</td>
</tr>
<tr>
<td>Std. Dev.</td>
<td>1.8714e-4</td>
<td>9.3994e-5</td>
<td>1.0220e-5</td>
<td>1.0117e-5</td>
<td>0.00007</td>
</tr>
</tbody>
</table>

As expected, the difference was small between the weight parameters of the PC and DSP. The largest difference was seen in the feedback weights with an average difference of 0.0001. These small differences may be attributed to two reasons. First, the precision with which the floating-point intermediary values are stored. The Pentium III processor uses 64-bit floating-point registers. Conversely, the C33 DSP has 40-bit floating-point registers. Both processors store the data back to memory in 32-bit notation, and therefore the increased precision of the Pentium processor is only “seen” during data manipulation. Secondly, this finite precision difference may cause the adaptive step size to take a
different direction on the PC than on the DSP. Again, this difference was seen to be negligible, and both filters were trained to adequately reproduce the hand position.

Table 4-6: Comparison of the RTRL algorithm running the PC and DSP.

<table>
<thead>
<tr>
<th>Metric</th>
<th>RTRL in the PC</th>
<th>RTRL in the DSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Movement Hits</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Movement Misses</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Total Movements</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Correlation Coefficient (movements)</td>
<td>Average = 0.8128 Std. Dev. = 0.1187</td>
<td>Average = 0.8096 Std. Dev. = 0.1196</td>
</tr>
<tr>
<td>Correlation Coefficient (non-movements)</td>
<td>Average = 0.1067 Std. Dev. = 0.2422</td>
<td>Average = 0.1067 Std. Dev. = 0.2417</td>
</tr>
<tr>
<td>SER (movements)</td>
<td>Average = 5.9860 dB Std. Dev. = 1.6858 dB</td>
<td>Average = 5.8284 dB Std. Dev. = 1.6698 dB</td>
</tr>
<tr>
<td>SER (non-movements)</td>
<td>Average = 4.5212 dB Std. Dev. = 2.9271 dB</td>
<td>Average = 4.3345 dB Std. Dev. = 2.8896 dB</td>
</tr>
</tbody>
</table>

For hand movements within the test data, the DSP algorithm produced output having a slightly lower SER than the PC algorithm (5.9860 dB versus 5.8284 dB). Likewise, the correlation coefficients were off by no more than 0.1. These performance metrics suggest that the DSP implementation matched the performance of the PC algorithm.

Figure 4-18: The CEM for the PC- and DSP-RTRL algorithms are shown against the BPTT algorithm using NeuroSolutions.
One additional comparison between the PC- and DSP-trained algorithms is shown through the Cumulative Error Measure in Figure 4-18. As shown by the green and blue lines, the cumulative error measure of the PC- and DSP-RTRL algorithms are virtually identical. Overall, the DSP- and PC-trained RTRL parameters were close enough for the DSP implementation to be considered successful. The possible difference in step-size caused by the finite-precision difference between the PC and the DSP was apparently not an issue. Having shown the correct algorithm implementation, the next step was to capture timing information for various filter configurations.

**RTRL Timing Results**

The DSP implementation of the RTRL algorithm was timed similarly to that of the NLMS algorithm. The digital oscilloscope was used to accurately observe the toggling of the XF0 pin with the code running in an infinite loop over 300 samples of data. Having an input of 104 neurons with 5 PEs, one pass over 300 samples of data (with 10 weight updates) took 536ms. This implies that up to 55 epochs over the 300-sample data set may be computed before the next 30s set of data is loaded.

Just as it was important to learn how different NLMS filter topologies affected speed in the DSP, the RMLP topology was varied in two ways to view corresponding changes in performance. Both the size of the input vector and number of hidden PEs were changed and then the algorithm timed. Again, surrogate data was used to increase the size of the input space beyond the 104 neuronal data that was available. This was done for timing purposes only, and the output of these trials was ignored. The timing results of varying input size are shown below in Figure 4-19.
As seen in Figure 4-19, a linear increase in computation time is observed as the number of neuron inputs increases. For input sizes larger than 500 neurons, the input vector was loaded into the external SRAM. This caused the computation time to increase at a pace faster than keeping the weights in internal memory. This is caused by the additional wait-state required for accessing external memory.

Increasing the input size well beyond 104 neurons greatly changes the number of epochs that are possible during the 30 seconds. For instance, one 300-sample epoch over 1000 neurons requires almost 7 seconds for processing. Consequently, only 4 epochs are able to be calculated within 30s.

As important as the size of the input vector, the number of PEs in the hidden layer greatly affects the capabilities of the neural network. As discussed in Chapter 2, an increased number of PEs are needed to learn more complex hand trajectories and,
therefore, may be needed in the future. By varying the number of PEs while keeping the input size the same, the following diagram shows the timing results in the DSP.

Figure 4-20: Timing analysis of the RTRL algorithm versus number of PEs in the hidden layer. All weights were stored in internal memory for this experiment.

The size of the hidden layer was varied to have 3 to 10 processing elements. The top diagram in Figure 4-20 shows the non-linear increase in computation time for increasing the number of PEs. By increasing the size of the hidden layer, both the input- and output-layer computations are affected—all three weight matrices and two bias vectors are increased in size. This explains the non-linear increase in computation time by adding PEs to the hidden layer.

The bottom diagram in Figure 4-20 shows the number of epochs computed within 30 seconds. With 3 PEs, the DSP is capable of 160 passes of the 300-sample data. Increasing the hidden layer to 10 PEs allows only 12 epochs—this great increase in computation time should be considered when BMI researchers choose an appropriate RMLP topology.
Discussion

This Chapter described the implementation and performance of two neural-to-motor translation algorithms in the DSP, namely the Linear Transversal Filter trained by the Normalized Least-Mean-Square adaptation algorithm, and the Recursive Multi-Layer Perceptron trained by Real-Time Recurrent Learning. Both DSP algorithms produced comparable results to PC implementations using MATLAB, C, or NeuroSolutions.

The data used to test these algorithms was collected from 104 neurons across multiple regions in an Owl Monkey. Sets of 300 samples (30 seconds) were downloaded to the DSP and then the algorithm computed as many epochs over the 300-samples as possible within 30 seconds.

With this many neurons, the 10-tap NLMS algorithm was able to produce an output and update the weights in 576us, allowing for the computation of 176 epochs within 30 seconds. The RTRL algorithm was able to estimate the gradient over 10 trajectories (300 samples) and update the weights within 536ms, allowing for the computation of 55 epochs within 30 seconds.

By varying the size of each filter, timing analysis showed a linear increase in computation time by increasing the number of input neurons. However, this linear increase may be problematic as the number of neurons increases beyond 500, at which point the RTRL algorithm is limited to computing less than 10 epochs per 30 seconds. According to the literature as well as experience, it is generally better to have as many epochs as possible.

Varying the NLMS filter depth also produced a linear increase in computation time. By contrast, adding PEs to the RMLP hidden layer had a non-linear effect on
computation time. This was consistent with the RTRL time complexity of $O(p^4T)$ that increases geometrically with the number of PEs. Compounded with a larger number of input neurons, an increased number of PEs will greatly limit the ability to compute multiple epochs over the streaming data. These results highlight the importance of carefully selecting a lean filter topology.

Together with the custom C33 DSP Board, these algorithm implementations provide a powerful, portable DSP-based method for neural-to-motor translation. The following chapter provides additional discussion about an integrated BMI system, its use in the future, as well as the conclusion to this thesis.
CHAPTER 5
CONCLUSION AND FUTURE WORK

The quest to link mind with machine has brought together the disciplines of neurobiology and electrical engineering to pursue an integrated Brain-Machine Interface. Current methods for sampling and processing multiple cortical areas involve large, multi-processor systems to compute the digital filter parameters for neural-to-motor translation. Beyond these massive neural processing systems, a portable system is desired for persons with motor disabilities that are confined to a wheelchair. Therefore, there is a need to have a small computational engine that is both portable and supports the digital filtering demands for the BMI.

This thesis presented a custom, portable BMI computational engine based on the Texas Instruments TMS320VC33 DSP. Two digital filters and their training methods were implemented in the DSP for neural-to-motor translation, 1) FIR trained with Normalized Least Mean Square adaptation method, and 2) Recurrent Multi-Layer Perceptron trained with Real-Time Recurrent Learning. The performance and timing results of these algorithms running in DSP were presented.

The following three sections discuss the DSP Board, DSP algorithms, and the future work that is forecasted to move toward an integrated BMI.

**DSP-Based Computational Engine**

A custom DSP Board was created to provide a multi-purpose, re-configurable signal processing solution for real-time neural-to-motor translation. At the center of this board is the 75MHz C33 DSP. Providing 150 MFLOPS, the C33 is able to provide
efficient, precise floating-point computational power for a real-time BMI. It was shown that the C33 40-bit floating-point representation was sufficient to support training of both a linear and non-linear filters.

Connected to a PC through the high-speed PCI bus, the DSP Board may be used for PC-based neural-to-motor mapping with a subject in a research lab. In this configuration, filter parameters may be tweaked, training performance observed, and DSP program code updated through the PCI bus.

For portable signal processing, the DSP Board may be used in standalone mode for direct connection to the neural acquisition hardware through the external add-on connectors. The light-weight DSP Board may be carried in a backpack by the subject, or mounted on an assistive vehicle such as a wheelchair.

The 512k words of off-chip SRAM provides ample storage room for data and program code. Additionally, this storage may eventually be used to implement larger filter topologies, perhaps multiple filters running in parallel.

**DSP Algorithm Implementation**

Two algorithms, identified by BMI researchers as appropriate for neural-to-motor translation, were successfully implemented in the C33 DSP. These filter implementations were trained and tested using real data collected from an Owl Monkey at Duke University. Specifically, 104 neurons were sampled from various cortical regions simultaneously while the monkey performed a reaching task.

The 10-tap linear filter trained with NLMS provided a fast output and weight-update in 576us. Using a 300-sample epoch of data (30 seconds), the DSP computed 176 epochs in the time before receiving the next 300-sample set. In other words, this DSP filter implementation is 173 times faster than real-time. This speed varied, of course, as
the filter topology was changed. With as many as 500 neurons, the filter output and weight updates were computed in 4.38ms, allowing up to 22 epochs over the 300-sample data set. This was a linear increase in computation time over the 104-input topology.

Varying the filter depth also exhibited a linear relationship in the computation time. With 104-inputs and a filter depth of 50-taps, the filter output and weight updates were computed in 4.60ms, allowing for up to 41 epochs over the 300-sample data set. Overall, the linear filter trained with NLMS reproduced the hand trajectory in an acceptable fashion. Improvements in hand trajectory output come with more complex filter topologies, such as the RMLP.

The benefits of using the RMLP over the Linear Transversal Filter are twofold: 1) reduced number of free parameters by moving the memory to the hidden layer, and 2) increased ability to learn complex hand trajectories through non-linear projection spaces and multiple, simultaneous time resolutions. For the 104-dimensional neural data, the RMLP having five PEs had 568 free parameters. This is compared with the 3120 free parameters of the FIR trained with NLMS. However, unlike the linear filter, the RTRL algorithm for training the RMLP is far complex and time-consuming.

With a 104:5:3 fully-connected RMLP, one weight update after 10 trajectories (300 samples of data) was computed in 536ms. This allowed for up to 55 epochs on the 300-sample data set. Multiple data sets were streamed to the DSP to mimic a real-time experiment. The network trained in the DSP was compared a similar network trained on a PC using NeuroSolutions. Both sets of weights provided adequate output with similar signal-to-error ratios and correlation coefficients. This helped to verify that the 300-sample epoch training scheme would converge to the same solution.
RMLP topologies having a larger input size required much more processing time. For example, 6.58 seconds was needed for a network with an input of 1000 neurons. Consequently, only four epochs were possible using this configuration. A geometric increase in computation time was observed by increasing the number of PEs in the hidden layer.

It is concluded that BMI researchers should select only the largest network topology that is necessary to fulfill a particular application. Reducing the number of neuron inputs as well as having only the minimum number of processing elements will maximize the number of possible epochs.

Another important conclusion is that neural-to-motor mapping algorithms should be coded with as much regularity as possible. Algorithms having a large number of conditional branches slow down the execution speed. This was experienced by trying to eliminate weight-updates for zero-input data in both the linear and non-linear algorithms. Placing “if” statements in the main code loops increased the computational burden for each pass of the loop—this resulted in increased computational time compared with not checking for zero-input data. What started out as a speed-saving modification resulted in slower execution. Therefore, because DSP execution speed is coupled with code regularity, all future algorithms should be written to minimize the number of operations computed during large loops.

**Future Work**

It is understood that the test environment used in this thesis was ideal in many ways and did not necessarily represent a future, integrated BMI system. The purpose of this thesis was to develop a first-attempt at a portable, high-speed computational engine for
neural-to-motor translation. The method for taking this DSP Board and incorporating it into future BMI systems is still largely under deliberation.

The testbench for processing neural and hand position data was ideal in this research. The data was pre-processed and readily available in a file. Future work will require a method of direct-connection to the neural acquisition hardware. This hardware may be a wired system in which the DSP Board resides in a backpack carried by the subject. On the other hand, the system may transmit the data wirelessly to the DSP Board in the backpack or across the room. In either case, the external add-on connectors will be used to interface any additional hardware. Additionally, collecting the data in real-time increases the computational burden of the DSP—therefore, the DSP will have less time to train the filter parameters. Much care should be taken to link the DSP with neural acquisition hardware in a manner that minimizes the I/O time and computational burden required of the DSP.

It is clear that advances in neurobiology and microwire devices will eventually allow for the simultaneous recording of thousands of neurons. As mentioned earlier, filters with many free parameters caused by large input dimensions tend to generalize poorly. Furthermore, a large input dimension reduces the number of epochs the DSP may compute over the training data in real-time. For these reasons, methods for real-time neuron sensitivity analysis should be pursued. Given a large number of neuronal inputs, the DSP should somehow select a subset of neurons for training the filter. Current research with Hidden Markov Models might allow for real-time subset selection, or possibly the selection of multiple, smaller filters running in parallel.
Operating the algorithm on a limited 300-sample epoch greatly limits the scope of error detection. For instance, offline analysis typically involves processing as large a data set as possible, sometimes tens of thousands of samples. Using this large data set, the MSE may be calculated over the entire training set after each weight update. The MSE for the current and previous weights may be compared to determine algorithm performance, adjust the step size, or decide whether to keep or reject that particular update. Real-time systems do not have the luxury of checking the error caused by weight updates against the entire data set—only the current, small set of data may be considered. If the current data set happens to be a “bad data set,” the RTRL algorithm may spend several epochs (55 in this case) learning “bad data.” Therefore, future work should find methods to estimate performance based on past knowledge, and possibly change the step size in order to reject weight updates learned from bogus data.

Overall, the custom DSP Board and real-time algorithm implementations presented in the thesis create a stepping stone for a future, integrated BMI system that may one day help link the mind with machine.
APPENDIX A
ACRONYMS AND ABBREVIATIONS

ALU – Arithmetic/Logic Unit
API – Application Programming Interface
BMI – Brain-Machine Interface
C33 – Texas Instruments TMS320VC33 Digital Signal Processor
CE – Chip Enable
CEM – Cumulative Error Measure
CPLD – Complex Programmable Logic Device
DSP – Digital Signal Processor
DPRAM – Dual-Port Random Access Memory
EEPROM – Electrically Erasable Programmable Read Only Memory
FIR – Finite-duration Impulse Response filter
FPGA – Field Programmable Gate Array
HMM – Hidden Markov Model
IIR – Infinite-duration Impulse Response filter
INT – Interrupt
JTAG – Joint Test Action Group, a standard connector used for debugging a component
LMS – Least-Mean-Square adaptive filter.
MAP – Multiple Acquisition Processor, developed by Plexon Incorporated.
MFLOPS – Millions of Floating Point Operations per Second
MIPS – Millions of Instructions per Second
MLP – Multi-Layer Perceptron
MSE – Mean Square Error
OS – Operating System
PE – Processing Element
PCB – Printed Circuit Board
PCI – Peripheral Component Interconnect
PLCC – Plastic Lead Chip Carrier, a type of IC packaging usually placed in a socket.
RE – Read Enable
RW – Read/Write
RMLP – Recursive Multi-Layer Perceptron
TANH – Hyperbolic Tangent
TDNN – Tap-Delay Neural Network
TI – Texas Instruments Incorporated (Dallas, TX)
WE – Write Enable
VHDL – VHSIC Hardware Description Language
VHSIC – Very High Speed Integrated Circuit
APPENDIX B
SCHEMATICS AND LAYOUT OF DSP BOARD

Schematics

The following six schematics were created using the Protel 99 SE Schematic and Layout software package [35].
This schematic has the following reserved ranges:

Capacitors C1 - C30

Resistors R1 - R15

This diagram includes the DSP Oscillator and Data Bus Level Shifters.
Main Memory
Four 512Kx8 SRAM => 512Kx32

External Memory (SRAM)

Note: 2 bypass caps per SRAM chip
This schematic has the following reserved ranges:
Capacitors C115 - Cxxx
Resistors R46 - Rxx
Layout

Top-layer components.
Bottom-layer components. Note the picture is mirrored left-to-right so that the component labels may be read.
APPENDIX C
DSP BOARD CONTROL SOFTWARE

C Console Software

The C code for the console program is located in the following files:

- **dslave.c**  Executable code
- **dslave.h**  Header file with constants and definitions
- **load_dsp.c**  Method for reading a COFF file and write to DSP
- **tms_ieee.c**  Methods for converting between floating point formats

VHDL Code for the CPLD

-- EPM3032 VHDL Code for the C33 DSP Board - v2
-- Updated Spring 2002

-- University of Florida
-- Jeremy Parks, jeremyp@ufl.edu
-- Scott Morrison, scott@cnel.ufl.edu

library ieee;
use ieee.std_logic_1164.all;

entity epm3032 is port (  
-- Reset for C33 Board
DSPRESET_L : buffer std_logic; -- (pin 40) reset for DSP board
PWR_RESET_L : in std_logic;  -- (pin 41) reset output from power

-- PLX signals
CS0_L : in  std_logic;  -- (pin 37)
WR_L : in  std_logic;  -- (pin 34)
BCLKo : in  std_logic;  -- (pin 43) main clock
GPIO8 : in  std_logic;  -- (pin  4) output from PLX to signal
GPIO4 : out std_logic;  -- (pin  5) PLX I/O
GPIO1 : out std_logic;  -- (pin  6) PLX I/O

-- C33 signals
PAGE0_L : in  std_logic;  -- (pin 24)
PAGE1_L : in  std_logic;  -- (pin 25)
PAGE2_L : in  std_logic;  -- (pin 26)
PAGE3_L : in  std_logic;  -- (pin 27)
DSPRW : in  std_logic;  -- (pin 28)
DSPINT3_L : out std_logic;  -- (pin 19)
XF0 : out std_logic;  -- (pin 33) DSP I/O
A21 : in  std_logic;  -- (pin 2)
A20 : in  std_logic;  -- (pin 1)
A19 : in  std_logic;  -- (pin 44)

-- Control Signals (CE, R/W)
PLXDP_CE_L : out std_logic;  -- (pin 11)
C33DP_CE_L : out std_logic;  -- (pin 12)

end epm3032;
SRAM_CE_L  : out std_logic;  -- (pin 14)
PRM_CE_L   : out std_logic;  -- (pin 16)
--EXT_IO1  : out std_logic;  -- (pin 13) CPLD_TMS
EXT_IO2   : out std_logic;  -- (pin 39)
PLXDP_RW  : out std_logic;  -- (pin 29) HEADER3
--PLXDP_RW  : out std_logic;  -- (pin  7) CPLD_TDI
C33DP_RW  : out std_logic;  -- (pin  9) HEADER2
--C33DP_RW  : out std_logic;  -- (pin 38) CPLD_TDO
SRAM_WE_L : out std_logic;  -- (pin  8) HEADER1
--SRAM_WE_L : out std_logic;  -- (pin 32) CPLD_TCK
-- DPRAM Interrupts
DPM_INTL_L : in  std_logic;  -- (pin 21)
DPM_INTR_L : in  std_logic;  -- (pin 20)
-- Other Signals
--HEADER1  : out std_logic;  -- (pin  8)
--HEADER2  : out std_logic;  -- (pin  9)
--HEADER3  : out std_logic;  -- (pin 29)
HEADER4   : in std_logic;  -- (pin 18)
USER_LED  : out std_logic); -- (pin 31)
end epm3032;

architecture behavior of epm3032 is

-- the following is for a clean reset
Type state_type is (S0, S1, S2, S3, S4);
signal next_state: state_type;
begin

-- State Machine to control DSP /RESET and DSP_INT3
dspReset: process (BCLKo, GPIO8, DPM_INTL_L)
begin
if (GPIO8 = '0') then  -- asynchronous reset
  next_state <= S0;
elsif (BCLKo'event and BCLKo = '1') then
  case next_state is
  when S0 =>
    next_state <= S1;
  when S1 =>
    next_state <= S2;
  when S2 =>
    next_state <= S3;
  when S3 =>
    if (DPM_INTL_L = '1') then
      next_state <= S3;
    else
      next_state <= S4;
    end if;
  end case;
end if;
when S4 =>
  next_state <= S4;
when others =>
  next_state <= S0;
end if;
end process dspReset;

-- Signal Assignments to control DSP /RESET and DSP_INT3
dspResetOutputs: process (BCLKo, next_state)
begin
if (next_state = S3) then
  DSPRESET_L <= PWR_RESET_L AND GPIO8;
  DSPINT3_L  <= '1';  -- do not assert INT3
end if;
end process dspResetOutputs;

end behavior;
elsif (next_state = S4) then
  DSPRESET_L <= PWR_RESET_L AND GPIO8;
  DSPINT3_L  <= DPM_INTR_L;  -- allow INT3 to be asserted
else
  DSPRESET_L <= '0';  -- keep DSP in reset
  DSPINT3_L  <= '1';  -- do not assert INT3
end if;
end process dspResetOutputs;

-- Control Signals
-- Note: not (not A AND not B) = A OR B
PLXDP_CE_L  <= CS0_L;
C33DP_CE_L  <= PAGE1_L OR NOT DSPRESET_L;
  -- Protect C33DP_CE during reset
SRAM_CE_L   <= PAGE0_L OR NOT DSPRESET_L;
  -- Protect SRAM_CE_L during reset;
PROM_CE_L   <= PAGE3_L OR NOT DSPRESET_L;
  -- Protect PROM_CE_L during reset and PROM Emulator Programming;
PLXDP_RW    <= 'Z' when HEADER4 = '1' else (WR_L OR CS0_L);
  -- ensure R/W is high during address transitions
C33DP_RW    <= 'Z' when HEADER4 = '1' else (DSPRW OR PAGE1_L);
  -- ensure R/W is high during address transitions
SRAM_WE_L   <= 'Z' when HEADER4 = '1' else (DSPRW OR PAGE0_L);
  -- ensure WE is high during address transitions

-- Turn on User LED when safe to program the CPLD with the ByteBlaster
USER_LED    <= '1' when HEADER4 = '1' else '0';

-- Work-Around: Use HEADER4 to control three pins shared on TCK/TDO/TDI
-- Apply input to EXT_IO1:
--   +3.3V ==> Connect /SRAM_WE, C33DP_RW, and PLXDP_RW (Header1-3 outputs)
--   GND ==> High-Z Header 1-3 outputs, which will not drive TCK/TDO/TDI

-- 'Z' all unused outputs
--EXT_IO1    <= 'Z';
EXT_IO2     <= 'Z';
GPIO4       <= DPM_INTL_L;
GPIO1       <= 'Z';
XF0         <= 'Z';
--HEADER1   <= 'Z';
--HEADER2   <= 'Z';
--HEADER3   <= 'Z';

end behavior;
DSP Operating System Code

The DSP Operating system code is segmented into three parts: 1) main OS code, 2) supporting subroutines, 3) global definitions.

Main DSP Operating System Code

*(c) 2001 University of Florida
*************************************************************************
* filename = c33_os.asm
* initializations for the C33 DSP Board
* this program also sets up the main command interrupt
*************************************************************************
.sect ".prog"
.include "c33_glb.asm" ;global variables file
init
*** initialize the C33 ***
and 0h,ie ; disable all interrupts
and 0h,if ; clear interrupt flags

* read from DPRAM mailbox just in case it is already set
ldp dpm_addr ; load from dpram
ldi @mb_pc2dsp,ar0 ; read pc to dsp mailbox, discard this data
ldi 8098h,sp ; *** set the stack at 0809800h (RAM Block 0)
lsh 8,sp
or 0800h,st ; *** enable cache
ldp memi_addr ; load page with 0800000h (regs in internal SRAM)
ldi 0048h,ar0 ; Change Wait States (0048 => 2 ws, 0028 => 1 ws)
sti ar0,08064h ; Primary Bus Control Reg: WTCNT=1, SWW=1

* set up int3 vector (this is the system command interrupt)
* put a branch to interrupt subroutine address at location 809FC4h
ldi 06080h,ar0 ; set up the unconditional branch
lsh 16,ar0 ; '06' is the opcode, '80' is the msb
ldi int3_isr,ar1 ; byte of operand (branch address)
or ar1,ar0 ; and the lower two bytes of operand
sti ar0,080FC4h ; are obtained from assembler (int3_isr label)

* signal to pc that reset has completed
ldp dpm_addr ; load dpram page
xor ar0,ar0 ; clear ar0
or 0FAB4h,ar0 ; load confirm message
sti ar0,0mb_dsp2pc ; store message to mailbox for pc

* enable interrupts
or 8,ie ; enable int3 in int enable register
or 02000h,st ; set the global int. enable active
; enables cpu to receive an interrupt

*** wait for command interrupt from pc ***
wfi
or 08h,ie ; *** enable int3 so new command interrupts
can fire upon returning from one.
br wfi

* End of main routine
* command interrupt routine - issue commands (i.e. run specific subroutines) *
******************************************************************************

int3_isr
and OFFFFh,ie ; disable int3 to prevent nested ints

ldp dpm_addr ; load from dprom
ldi @mb_pc2dsp,ar0 ; read pc to dsp mailbox
ldi ar0,ir0 ; move opcode into index register
lsh -24,ir0 ; shift address right to leave opcode only
ldi 00FFH,ar1
lsh 16,ar1 ; O0FFFFF AND [8-bit opcode][24-bit address]
or 00FFFH,ar1 ; result: [00h][24-bit address]
and ar1,ar0 ; mask off opcode to leave address only in ar0

ldi memi_addrh,ar1 ; load 8 bits of sram address and
lsh 16,ar1 ; shift up to 3rd byte then or with first addr.
or disptable,ar1 ; in dispatch table. fetch the subroutine
ldi *+ar1(ir0),ar2 ; address based on a command displacement.
callu ar2 ; jump to the subroutine

* signal to pc that operation has completed
ldp dpm_addr ; load dprom page
xor ar0,ar0 ; clear ar0
or 0FAB4h,ar0 ; load confirm message
sti ar0,@mb_dsp2pc ; store message to mailbox for pc

ldi 00FFH,ar7 ; clear int3 flag only
lsh 16,ar7 ; O0FFFF7h is the clear int3 flag
or 00FFFH,ar7
and ar7,if
reti

****************************************************************************** variables section******************************************************************************
.data

* Command displacement table (used by int3 to jump to a particular routine).
disptable
.word write_mem,read_mem,execute,user_command ; system commands 0,1,2

* store opcode during subroutine
opcode
.word 0h ; reserve one word

* Mask for inverse routine
MSK .word OFFFFF7FFFH

Subroutines for the DSP Operating System

* (c) 2001 University of Florida
******************************************************************************
*************** command #0 - write to dsp memory ***************
* Register ar0 contains start address
* DPRAM address 0h contains NumWords (total 32-bit words to write)
* Register ar3 used for temp storage of data

write_mem
push ar1 ; preserve registers
push ar2
push ar3

ldi dpm_addrh,ar2 ; ar2 = 00000040h
lsh 16,ar2 ; ar2 = 00400000h
or dpm_data,ar2 ; load ar2 with dpram start address

ldp dpm_addr ; load dpram page
ldi @dpm_numwords,rc ; load NumWords from DPRAM address 0h
subi 1,rc ; do this for rptb count to be correct

rptb end_wr ; setup for repeat loop

ldi *ar2++(1),ar3 ; retrieve data from dpram

end_wr sti ar3,*ar0++(1) ; write to dsp memory & post increment

pop ar3 ; restore registers
pop ar2
pop ar1

retsu

*************** command #1 - read from dsp memory ***************
* Register ar0 contains start address
* DPRAM address 0h contains NumWords (total 32-bit words to read)
* Register ar3 used for temp storage of data

read_mem
push ar1 ; preserve registers
push ar2
push ar3

ldi dpm_addrh,ar2 ; ar2 = 00000040h
lsh 16,ar2 ; ar2 = 00400000h
or dpm_data,ar2 ; load ar2 with dpram start address

ldp dpm_addr ; load dpram page
ldi @dpm_numwords,rc ; load NumWords from DPRAM address 0h
subi 1,rc ; do this for rptb count to be correct

rptb end_rd ; setup for repeat loop

ldi *ar0++(1),ar3 ; retrieve data from dsp mem

end_rd sti ar3,*ar2++(1) ; write to dpram memory & post increment

pop ar3 ; restore registers
pop ar2
pop ar1

retsu

*************** command #2 - execute program ***************
* Call a program somewhere in memory

execute
bu ar0 ; call the program in memory
; note, the user must have included "retsu"
; at the end of their code

******************************************************************************
Global Definitions
******************************************************************************
* (c) 2001 University of Florida
* filename = c33_glb.asm
* global declaration file
******************************************************************************
* global constants related to ext./int. memory map

dpm_addr .set 0400000h ; dpmem start address
dpm_addrh .set 0040h ; msb of dpmem start address
mb_dsp2pc .set 0ffeh ; dpram dsp to pc mailbox (left-side)
mb_pc2dsp .set 0fffh ; dpram pc to dsp mailbox (right-side)
dpm_numwords .set 0h ; dpram addr containing no. of words to read/write
dpm_data .set 1h ; dpram read/write data start address
inv_start .set 4000h ; inverse start memory location
sram_addr .set 0100000h ; sram start address
sram_addrh .set 0010h ; msb of sram start address
eprom_addr .set 0FFF000h ; eprom start address (boot 3)
eprom_addrh .set 00FFh ; msb of eprom start address
ext_addr .set 0900000h ; ext start address
ext_addrh .set 0090h ; msb of ext start address
memi_addr .set 0800000h ; dsp internal sram start address
memi_addrh .set 0080h ; msb of internal sram start address

* global constants for dsp gpio manipulation
xf0_output .set 0004h ; xf0 output data
xf0_input .set 0008h ; xf0 input data
xf0_dir_out .set 0002h ; iof or xf0_dir_out => iof, make xf0 output
xf0_dir_in .set 0FFFDh ; iof and xf0_dir_in => iof, make xf0 input
xf0_high .set 0004h ; iof or xf0_high => xf0=1
xf0_low .set 0FFFBh ; iof and xf0_low => xf0=0

* global constants for invert subroutine
ONE .SET 1.0
TWO .SET 2.0

* system command routines
.global write_mem, read_mem, execute, user_command, fpinv, MSK

**********************************************************************

APPENDIX D
CODE FOR DSP ALGORITHMS

Linear Transversal Filter trained with NLMS

The following files are for the NLMS algorithm in DSP:

- **nlms.asm** Executable code
- **nlms_glb.asm** Global definitions and constants
- **nlms_sub.asm** Subroutines
- **nlms.cml** Linker file for memory locations

For your convenience, the nlms.asm file is shown below.

```assembly
LMS_START ; begin LMS processing of BMI data
LDI 1000h, ST ; clear the cache, just in case
OR 00100000000000b,ST ; enable the cache
LDP memi_addr ; page becomes internal SRAM
OR 000000010b,IOF ; make XF0 Output
OR 00100000b,IOF ; make XF1 Output
```

******************************************************************************************************************************************
* (c) 2003 University of Florida
* Computational Neuro Engineering Laboratory
* www.cnel.ufl.edu
* Applied Digital Hardware Research Laboratory
* www.add.ece.ufl.edu
* 
* Author: Scott A. Morrison
* E-Mail: scott@cnel.ufl.edu
* 
* Advisor: Dr. Jose Principe (CNEL)
* E-Mail: principe@cnel.ufl.edu
* 
* Advisor: Dr. Karl Gugel (ADD Lab)
* E-Mail: gugel@ece.ufl.edu
* 
* Date Created: 31 January 2003
* Date Modified: 17 March 2003
* 
******************************************************************************************************************************************

* filename = lms.asm
* Normalized Least Mean Squared with Weight Decay
* Purpose: To perform realtime NLMS processing on neural data.
* Inputs = 105 Dimensions1 neural data (1 dimension is for bias)
* Outputs = 3 dimensional hand position

******************************************************************************************************************************************
******************************************************************************
* Initialize address pointers to all Matrices and Vectors
*
START
XOR 01000000b, IOF ; TOGGLE X1
LDI @X_ADDR, AR3 ; reset x pointer
STI AR3, @X_PTR ; start at x(1)
LDI @Y_ADDR, AR3 ; reset y pointer
ADDI (3*(LAGS-1)), AR3 ; start at y(10)
STI AR3, @Y_PTR ;
LDI @E_ADDR, AR3 ; reset e pointer
ADDI (3*(LAGS-1)), AR3 ; start at e(10)
STI AR3, @E_PTR ;
LDI @D_ADDR, AR3 ; reset d pointer
ADDI (3*(LAGS-1)), AR3 ; start at d(10)
STI AR3, @D_PTR ;
******************************************************************************
* Prepare for LMS Loop
*
LDI (WIN-LAGS), AR0
STI AR0, @MYCOUNT ; Load main counter with 90 (=100-10)
******************************************************************************
* Calculate the Output Y(n)=W(n)'*X(n)
*
LOOP1
XOR 00000100b, IOF ; TOGGLE X0 for timing
LDI @X_PTR, AR3 ; Load current X pointer
LDI @Wx_ADDR, AR4 ; AR4 <= Address of Wx
LDI @Wy_ADDR, AR5 ; AR5 <= Address of Wy
LDI @Wz_ADDR, AR6 ; AR6 <= Address of Wz
LDF 0.0, R0 ; Clear R0
LDF 0.0, R1 ; Clear R1
LDF 0.0, R2 ; Clear R2
LDF 0.0, R3 ; Clear R3
LDF 0.0, R4 ; Clear R4
LDF 0.0, R5 ; Clear R5
STF R0, @POWER ; Zero the power cumulation
LDI (LAGS*CHAN-1), RC ; Get ready to repeat 10*104 times
RPTB HERE1 ; Repeat this block to calculate output
MPYF3 *AR4++, *AR3, R0 ; Wx*X parallel multiply and accumulate
|| ADDF3 R0, R2, R2
MPYF3 *AR5++, *AR3, R1 ; Wy*Y parallel multiply and accumulate
|| ADDF3 R1, R3, R3
MPYF3 *AR6++, *AR3, R4 ; Wz*Z non-parallel multiply and accumulate
ADDF R4, R5
MPYF3 *AR3, *AR3++, R7 ; accumulate the power of the input
ADDF @POWER, R7
HERE1 STF R7, @POWER
ADDF R0, R2 ; do last accumulate
ADDF  R1,R3
; do last accumulate
; last accumulate not necessary for Wz

* We went through the whole tap-input, now store results to Y(n)
LDI   @Y_PTR,AR0    ; load Y pointer
STF   R2,*AR0++    ; store Yx(n)
STF   R3,*AR0++    ; store Yy(n)
STF   R5,*AR0++    ; store Yz(n)
STI   AR0,@Y_PTR   ; store Y pointer back to memory for next time
; next, error calculation will use Y from R2,R3,R5

**************************************************************************
* Calculate the Error E(n)=D(n)-Y(n)
*
LDI   @E_PTR,AR0    ; AR0 --> e(n)
LDI   @D_PTR,AR6    ; AR6 --> d(n)
SUBF  R2,*AR6++,R7  ; ex(n) = dx(n)-yx(n)
STF   R7,*AR0++
SUBF  R3,*AR6++,R7  ; ey(n) = dy(n)-yy(n)
STF   R7,*AR0++
SUBF  R5,*AR6++,R7  ; ez(n) = dz(n)-yz(n)
STF   R7,*AR0++
STI   AR6,@D_PTR   ; store changed D pointer for next time
; don't store new E pointer, need it later
**************************************************************************
* Update the Weight Matrix
*
* Find 1/(1+q)
LDF   @POWER,R0    ; load the power of x(n)
ADDF  1.0,R0       ; R0=R0+1.0 (R0=q+1)
CALL  FPINV        ; R0=1/R0 (R0=1/(1+q))
STF   R0,@POWER    ; store back to memory
LDI   @X_PTR,AR3    ; Load current X pointer
LDI   @Wx_ADDR,AR4  ; AR4 --> Wx
LDI   @Wy_ADDR,AR5  ; AR5 --> Wy
LDI   @Wz_ADDR,AR6  ; AR6 --> Wz
LDI   @E_PTR,AR7    ; AR7 --> e(n)
LDF   *AR7++,R3     ; R3 <= ex(n)
LDF   *AR7++,R4     ; R4 <= ey(n)
LDF   *AR7++,R5     ; R5 <= ez(n)
STI   AR7,@E_PTR    ; save new E pointer to memory
LDF   @POWER,R6     ; R6 <= 1/(1+q)
MPYF  @ETA,R6      ; R6 <= ETA * 1/(1+q)
LDF   @DECAY,R7     ; R7 <= (1-DECAY)
LDI   (LAGS*CHAN-1),RC ; Get ready to repeat 10*104 times
RPTB  HERE2         ; Repeat this block to update weights
LDF   *AR3++,R2     ; R1 <= x(n)
MPYF 3 R2,R3,R0    ; R0 <= x(n)*ex(n)
MPYF  R6,R0       ; R0 <= ETA * 1/(1+q) * x(n)*ex(n)
MPYF 3 R7,*AR4,R1  ; R1 <= (1-DECAY) * Wx
ADDF  R1,R0       ; R0 <= (1-DECAY) * Wx + ETA * 1/(1+q) * x(n)*ex(n)
STF   R0,*AR4++    ; update weight in memory, increment pointer
MPYF 3 R2,R4,R0    ; R0 <= x(n)*ey(n)
MPYF R6,R0 ; R0 <= ETA * 1/(1+q) * x(n)*ey(n)
MPYF3 R7,*AR5,R1 ; R1 <= (1-DECAY) * Wy
ADDF R1,R0 ; R0 <= (1-DECAY) * Wy + ETA * 1/(1+q) * x(n)*ey(n)
STF R0,*AR5++ ; update weight in memory, increment pointer

MPYF3 R2,R5,R0 ; R0 <= x(n)*ez(n)
MPYF R6,R0 ; R0 <= ETA * 1/(1+q) * x(n)*ez(n)
MPYF3 R7,*AR6,R1 ; R1 <= (1-DECAY) * Wz
ADDF R1,R0 ; R0 <= (1-DECAY) * Wz + ETA * 1/(1+q) * x(n)*ez(n)
HERE2 STF R0,*AR6++ ; update weight in memory, increment pointer

LDI @X_PTR,AR0 ; load current X pointer
ADDI (CHAN),AR0 ; move pointer to next input time
STI AR0,@X_PTR ; store X pointer for next use

*************************************************************************
* Done with this iteration, loop back for next iteration
* LDI @MYCOUNT,AR0 ; retrieve counter from memory
SUBI 1,AR0
CMPI 0,AR0
BGED LOOP1
STI AR0,@MYCOUNT ; put counter back into memory
NOP
NOP
* Delayed Branch happens here!
BR START ; Loop forever for timing
RETSU ; return to DSP OS
LMS_END ; end of LMS routine

*************************************************************************
*************************************************************************
* Stored in Internal Memory
*************************************************************************
.data ; initialized data section
MSK .word 0FF7FFFFFH ; for fp_invert routine
FP_TABLE
   .word 0FF800000h ; table for C33FP <=> IEEEFP conversion
   .word 0FF000000h
   .word 07F000000h
   .word 080000000h
   .word 081000000h
   .word 07F800000h
   .word 000400000h
   .word 0007FFFFFH
   .word 07F7FFFFFH

* Global declarations for LMS data
.global X,X_END,Y,Y_END,E,E_END,D,D_END,X_PTR,MYCOUNT
WIN .set 200 ; Window size (number of 300ms time samples)
LAGS .set 10 ; Number of time lags for each computation
CHAN .set (104+1) ; Number of neuron channels (1 is for bias)
WINIT .set 0.001 ; Weight initialization
MYCOUNT .word 0 ; Main counter for LMS loop
Wx_ADDR .word Wx ; Constant pointer to top of Wx
Wy_ADDR .word Wy ; Constant pointer to top of Wy
Wz_ADDR .word Wz ; Constant pointer to top of Wz
X_ADDR .word X ; Constant pointer to top of X
Y_ADDR .word Y ; Constant pointer to top of Y
D_ADDR .word D ; Constant pointer to top of D
E_ADDR .word E ; Constant pointer to top of E

X_PTR .word X ; Changeable pointer to X
D_PTR .word D ; Changeable pointer to D
Y_PTR .word Y ; Changeable pointer to Y
E_PTR .word E ; Changeable pointer to E

POWER .float 0.0 ; Power of the input at each time

ETA .float 0.25 ; LMS step size
DECAY .float (1-0.005) ; Weight decay parameter

Y ; Output Vector, initialize to zeros
  .loop (WIN*3)
    .float 0.0
  .endloop
Y_END

D .include "d.asm" ; Desired Vector, monkey data
E ; Error Vector, initialize to zeros
  .loop (WIN*3)
    .float 0.0
  .endloop
E_END

.sect "weights"
.global  Wx,Wx_END,Wy,Wy_END,Wz,Wz_END

Wx ; Weight Matrix (10x104)
  .loop (LAGS*CHAN)
    .float 0.001
  .endloop
Wx_END

Wy
  .loop (LAGS*CHAN)
    .float 0.001
  .endloop
Wy_END

Wz
  .loop (LAGS*CHAN)
    .float 0.001
  .endloop
Wz_END

.sect "input"
X .include "x.asm" ; Input Matrix (100x104)
X_END

*************************************************************************
* End Data Section
*************************************************************************
.end ; end of file
RTRL in C

The following files are for RTRL on the PC and DSP:

- `rtrl_pc.c`  Executable code for PC
- `params.txt` Parameters for `rtrl_pc.c`
- `rtrl_dsp.c`  DSP code for Code Composer Studio
- `ready.asm` Assembly file for DSP initializations and pointers

For your convenience, `rtrl_dsp.c` is shown below.

```c
/* File:    "rtrl_dsp.c"
/*    Real-Time Recurrent Learning for an RMLP (DSP version)
/* Authors: Scott A. Morrison, scott@cnel.ufl.edu
/*    Deniz Erdogmus, deniz@cnel.ufl.edu
/*    Justin Sanchez, justin@cnel.ufl.edu
/*
/*    Computational NeuroEngineering Laboratory
/*    Department of Electrical and Computer Engineering
/*    The University of Florida
/*    Gainesville, FL, 32611, USA
/*
/* Created: 26 November 2002
/*
*****************************************************************************/

/* BE SURE THAT READY.ASM IS PLACED BEFORE THIS MAIN() SECTION */
/* CHECK THE CMD FILE TO SEE THIS IS TRUE */

#include <stdlib.h>
#include <math.h>

/* assign parameters */
#define TRAINING_START 0
#define TRAINING_END 300
#define TESTING_START 0
#define TESTING_END 300
#define TRAJ_LENGTH 30
#define TRAJ_PER_UPDATE 10
#define EPOCHS 50
#define N0 104
#define N1 5
#define N2 3
#define SLOPE 0.5f
#define DERIVATIVE_OFFSET 0.001f
#define ETA1_START -0.0001f
#define ETA2_START -0.00001f
#define ETAF_START -0.0001f
#define ADAPTIVE_STEP_SIZE 1
#define ETA_MIN 0.00000000001f
#define ETA_MAX 4.0f
#define ETA_INCREMENT 1.02f /* ETAnew = ETAold * ETA_INCREMENT */
#define ETA_DECREMENT 0.1f /* ETAnew = ETAold * ETA_DECREASE */
#define MOMENTUM 0.7f
#define USE_INITS 1
```
#define TRAINING_LENGTH 300
#define UPDATES_PER_EPOCH 1

/* Give access to C33 Registers */
extern cregister volatile unsigned int IE;  /* CPU int enable register */
extern cregister volatile unsigned int IF;  /* CPU interrupt flag register */
extern cregister volatile unsigned int IOF;  /* I/O Flags */
extern cregister volatile unsigned int ST;  /* Status Register */

/*************************** Temporary Variables ***************************/
unsigned int epoch = 0;
float d_power = 0.0f;  /* power of the desired signal, for normalized MSE */
float e_power = 0.0f;  /* power of the error signal, for normalized MSE */

extern float Wl_array[];  /* This is referencing Wl_array in asm file */
float *Wl = Wl_array;  /* Pointer to Wl */
extern float b1_array[];  /* This is referencing b1_array in asm file */
float *b1 = b1_array;  /* Pointer to b1 */
extern float W2_array[];  /* This is referencing W2_array in asm file */
float *W2 = W2_array;  /* Pointer to W2 */
extern float b2_array[];  /* This is referencing b2_array in asm file */
float *b2 = b2_array;  /* Pointer to b2 */
extern float Wf_array[];  /* This is referencing Wf_array in asm file */
float *Wf = Wf_array;  /* Pointer to Wf */

extern float G_W1_asm[];  /* gradient updates for above */
float *G_W1 = G_W1_asm;
extern float G_b1_asm[];
float *G_b1 = G_b1_asm;
extern float G_W2_asm[];
float *G_W2 = G_W2_asm;
extern float G_b2_asm[];
float *G_b2 = G_b2_asm;
extern float G_Wf_asm[];
float *G_Wf = G_Wf_asm;

extern float U_W1_asm[];  /* weight updates for above */
float *U_W1 = U_W1_asm;
extern float U_b1_asm[];
float *U_b1 = U_b1_asm;
extern float U_W2_asm[];
float *U_W2 = U_W2_asm;
extern float U_b2_asm[];
float *U_b2 = U_b2_asm;
extern float U_Wf_asm[];
float *U_Wf = U_Wf_asm;

extern float c_asm[];  /* intermediate update parameters */
float *c = c_asm;
extern float g_asm[];
float *g = g_asm;
extern float h_asm[];
float *h = h_asm;

extern float ylp_asm[];  /* previous Y1 */
float *ylp = ylp_asm;
extern float ylp_save_asm[];  /* previous Y1 */
float *ylp_save = ylp_save_asm;
extern float zl_asm[];  /* input to hidden layer */
float *zl = zl_asm;
extern float Y1_asm[];  /* output of hidden layer */
float *Y1 = Y1_asm;
extern float y2_asm[]; /* network output */
float *y2 = y2_asm;
extern float e_asm[]; /* network error */
float *e = e_asm;
extern float a_asm[]; /* save sech(z1)^2 */
float *a = a_asm;
extern float temp_asm[];
float *temp = temp_asm;
extern float chunkMSE_End[];
float *endMSE = chunkMSE_End;

far extern float input_array[]; /* pointers to data */
far extern float desired_array[];
far float *input = input_array;
far float *desired = desired_array;

float *G_W1,*G_b1,*G_W2,*G_b2,*G_Wf; /* pointers to gradients */
float *U_W1,*U_b1,*U_W2,*U_b2,*U_Wf; /* pointers to weight updates */
float *c,*g,*h; /* pointers to update parameters */
float *y1p,*z1,*Y1,*y2,*e,*a,*temp; /* pointers to network feedforward */

float *x_ptr = NULL;
float *d_ptr = NULL;
float MSE = 0.0f;
float MSEp = 0.0f;
float ETA1 = ETA1_START;
float ETA2 = ETA2_START;
float ETAF = ETAF_START;

/**************************** Function Prototypes *****************************/
void updateGradient();
void doTraining();

/**************************** Main program *****************************/
void main(void)
{
    unsigned int i = 0;
    unsigned int j = 0;
    unsigned int lines = 0;

    x_ptr = input; /* initialize the pointer to the input data */
    d_ptr = desired; /* initialize the pointer to the desired data */
    doTraining();
    /* we're done, so quit */
}

/**************************** End of Main program *****************************/

/**************************** Function Prototypes *****************************/
void updateGradient()
{
    unsigned int update = 0;
    unsigned int traj = 0;
    unsigned int i = 0;
    unsigned int T = 0;

    asm(" XOR 01000000b,IOF ; TOGGLE XF1");
    /* one epoch is a single pass through 300 samples */
}
for(epoch=0;epoch<EPOCHS;epoch++)
{
    asm(" XOR 00000100b,IOF ; TOGGLE XF0");
    /* reset the hidden state before starting epoch */
    /* resetting the state is the job of the PC */

    /* restore the previous state before continuing */
    for(i=0;i<N1;i++)
    {
        ylp[i] = ylp_save[i];
    }

    /* reset x_ptr */
    x_ptr = input;
    d_ptr = desired;

    /* reset for the error calculation */
    e_power = 0.0f;

    /* reset gradients and momentum terms */
    for (i=0;i<N0*N1;i++)
    {
        U_W1[i] = 0.0f;
        G_W1[i] = 0.0f;
    }

    for (i=0;i<N1;i++)
    {
        U_b1[i] = 0.0f;
        G_b1[i] = 0.0f;
    }

    for (i=0;i<N1*N1;i++)
    {
        U_Wf[i] = 0.0f;
        G_Wf[i] = 0.0f;
    }

    for (i=0;i<N1*N2;i++)
    {
        U_W2[i] = 0.0f;
        G_W2[i] = 0.0f;
    }

    for (i=0;i<N2;i++)
    {
        U_b2[i] = 0.0f;
        G_b2[i] = 0.0f;
    }

    for(traj=0;traj<TRAJ_PER_UPDATE;traj++)
    {
        /* reset c, g, and h */
        for(i=0;i<N1*N1;i++)
        {
            c[i] = 0.0f;
        }
        for(i=0;i<N1*N1*N0;i++)
        {
            g[i] = 0.0f;
        }
        for(i=0;i<N1*N1*N1;i++)
        {
h[i] = 0.0f;
for(T=0; T<TRAJ_LENGTH; T++)
{
    updateGradient();
    /* save the state */
    for(i=0; i<N1; i++)
    {
        ylp[i] = Y1[i];
    }
    /* move to next input and desired */
    x_ptr = x_ptr + N0;
    d_ptr = d_ptr + N2;
}
 /* end of loop over all trajectories within this block */
/* save previous MSE */
MSEp = *(endMSE);
/* write MSE to memory for PC reading */
*(endMSE) = e_power / (N2*TRAINING_LENGTH);
#if (ADAPTIVE_STEP_SIZE == 1)
    /* adjust the step size */
    if (epoch != 0)
    {
        if (*(endMSE) < MSEp)
        {
            /* increase the step size */
            ETA1 = ETA1 * ETA_INCREASE;
            ETA2 = ETA2 * ETA_INCREASE;
            ETAF = ETAF * ETA_INCREASE;
            if (ETA1 > ETA_MAX)
            {
                ETA1 = ETA_MAX;
            }
            if (ETA2 > ETA_MAX)
            {
                ETA2 = ETA_MAX;
            }
            if (ETAF > ETA_MAX)
            {
                ETAF = ETA_MAX;
            }
        }
        else
        {
            /* decrease the step size */
            ETA1 = ETA1 * ETA_DECREASE;
            ETA2 = ETA2 * ETA_DECREASE;
            ETAF = ETAF * ETA_DECREASE;
            /* make sure the step size isn't too small */
            if (ETA1 < ETA_MIN)
            {
                ETA1 = ETA_MIN;
            }
            if (ETA2 < ETA_MIN)
            {
                ETA2 = ETA_MIN;
            }
        }
    }
ETA2 = ETA_MIN;
}
if (ETAF < ETA_MIN)
{
    ETAF = ETA_MIN;
}
}
} /* end of adaptive step size block */
#endif /* end of if ADAPTIVE_STEP_SIZE */

/* update weights with the new step size */
for (i=0;i<N0*N1;i++)
{
    U_W1[i] = ETA1 * G_W1[i] + MOMENTUM * U_W1[i];
    W1[i] = W1[i] + U_W1[i];
}
for (i=0;i<N1;i++)
{
    U_b1[i] = ETA1 * G_b1[i] + MOMENTUM * U_b1[i];
    b1[i] = b1[i] + U_b1[i];
}
for (i=0;i<N1*N1;i++)
{
    U_Wf[i] = ETAF * G_Wf[i] + MOMENTUM * U_Wf[i];
    Wf[i] = Wf[i] + U_Wf[i];
}
for (i=0;i<N1*N2;i++)
{
    U_W2[i] = ETA2 * G_W2[i] + MOMENTUM * U_W2[i];
    W2[i] = W2[i] + U_W2[i];
}
for (i=0;i<N2;i++)
{
    U_b2[i] = ETA2 * G_b2[i] + MOMENTUM * U_b2[i];
    b2[i] = b2[i] + U_b2[i];
}
} /* end of loop over epochs */

/* save the state for next chunk of data, invoked by PC */
for(i=0;i<N1;i++)
{
    ylp_save[i] = ylp[i];
}
} /* end of doTraining() */

/***************************************************************************/
inline void updateGradient()
{
    /* temporary variables used in this function */
    int i = 0;
    int j = 0;
    int m = 0;
    int k = 0;
    int gamma = 0;
    int alpha = 0;
int beta = 0;

register float regFloat;

/* forward pass through the network */
for(j=0;j<N1;j++)
{
    regFloat = 0;
    for (i=0;i<N0;i++)
    {
        regFloat = regFloat + x_ptr[i]*W1[j*N0+i];
    }
    for (m=0;m<N1;m++)
    {
        regFloat = regFloat + y1p[m] * Wf[j*N1+m];
    }
    regFloat = regFloat + b1[j];
    Y1[j] = tanh(SLOPE*regFloat);
    z1[j] = regFloat;
    a[j] = SLOPE*(1-(Y1[j]*Y1[j])+DERIVATIVE_OFFSET);
}

for(k=0;k<N2;k++)
{
    /* y2[k]= 0; */
    regFloat = 0;
    for (j=0;j<N1;j++)
    {
        regFloat = regFloat + Y1[j]*W2[k*N1+j];
    }
    regFloat = regFloat + b2[k];
    e[k] = d_ptr[k] - regFloat;
    y2[k] = regFloat;
    e_power = e_power + (e[k] * e[k]); /* cumulate the error power */
}

/* The forward pass is done. Now backpropagate the updates. */

/* update c */
for(beta=0;beta<N1;beta++)
{
    for(j=0;j<N1;j++)
    {
        regFloat = 0.0f;
        for(i=0;i<N1;i++)
        {
            regFloat = regFloat + Wf[j*N1+i] * c[beta*N1+i];
        }
        if (j == beta)
        {
            c[beta*N1+j] = regFloat + 1;
        }
        else
        {
            }
c[β*N1+j] = regFloat;
}

c[β*N1+j] = c[β*N1+j] * a[j];
}

/* update g */
for(gamma=0;gamma<N0;gamma++)
{
    for(beta=0;beta<N1;beta++)
    {
        for(alpha=0;alpha<N1;alpha++)
        {
            regFloat = 0.0f;
            for (i=0;i<N1;i++)
            {
                regFloat = regFloat + Wf[alpha*N1+i] * g[gamma*N1*N1+beta*N1+i];
            }
            if (alpha == beta)
            {
                g[gamma*N1*N1+beta*N1+alpha] = regFloat + x_ptr[gamma];
            }
            else
            {
                g[gamma*N1*N1+beta*N1+alpha] = regFloat;
            }
        }
    }
    g[gamma*N1*N1+beta*N1+alpha] = g[gamma*N1*N1+beta*N1+alpha] * a[alpha];
}

/* update h */
for(gamma=0;gamma<N1;gamma++)
{
    for(beta=0;beta<N1;beta++)
    {
        for(alpha=0;alpha<N1;alpha++)
        {
            regFloat = 0.0f;
            for (i=0;i<N1;i++)
            {
                regFloat = regFloat + Wf[alpha*N1+i] * h[gamma*N1+beta*N1+i];
            }
            if (alpha == beta)
            {
                h[gamma*N1+beta*N1+alpha] = regFloat + ylp[gamma];
            }
            else
            {
                h[gamma*N1+beta*N1+alpha] = regFloat;
            }
        }
    }
    h[gamma*N1+beta*N1+alpha] = h[gamma*N1+beta*N1+alpha] * a[alpha];
}
/ * update the gradients */
for (k=0;k<N2;k++)
{
    G_b2[k] = G_b2[k] - e[k];
    for (j=0;j<N1;j++)
    {
        G_W2[k*N1+j] = G_W2[k*N1+j] - e[k]*Y1[j];
    }
}
for (j=0;j<N1;j++)
{
    regFloat = 0.0f;
    for (i=0;i<N2;i++)
    {
        regFloat = regFloat + e[i]*W2[i*N1+j];
    }
    for (i=0;i<N0;i++)
    {
        G_W1[j*N0+i] = G_W1[j*N0+i] - g[i*N1*N1+j*N1+j] * regFloat;
    }
    for (i=0;i<N1;i++)
    {
        G_Wf[j*N1+i] = G_Wf[j*N1+i] - h[i*N1+j*N1+j] * regFloat;
    }
}
} /* end of updateGradient() */
LIST OF REFERENCES


[38] Texas Instruments Incorporated. TMS320C3x user’s guide. Literature number SPRU031E. Dallas (TX) USA; 1997.
[39] Texas Instruments Incorporated. TMS320C3x/C4x assembly language tools user’s guide. Literature number SPRU035D. Dallas (TX) USA; 1998.

[40] Texas Instruments Incorporated. TMS320C3x/C4x optimizing C compiler user’s guide. Literature number SPRU034H. Dallas (TX) USA; 1998.
BIOGRAPHICAL SKETCH

Scott Adam Morrison was born in Miami, Florida, on the 23rd of February 1979. He graduated salutatorian from Miami Killian Senior High School in 1997 and attended the University of Florida in August of the same year. After receiving his Bachelor of Science in computer engineering in 2002, Scott joined the Computational NeuroEngineering Laboratory at UF for graduate work on the Brain-Machine Interface project.

Scott married Kate Clouser on the 19th of January 2003. After finishing his master’s thesis, Scott will graduate in May 2003 with a Master of Engineering degree. In July 2003, Scott and Kate will move to Dallas, Texas, where Scott will join a digital ASIC group at Texas Instruments Incorporated.