Nestor/9704001

Final Technical Report

Principal Investigator
Michael Glier
401-331-9640

Nestor, Inc.

1997

DARPA

Defense Advanced Research
Projects Agency
3701 North Fairfax Drive
Arlington, VA 22203-1714
ARPA Number 7017
Contract Number N00014-90-C-0010
January 22, 1990 - March 26, 1993

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency of the U.S. Government.

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

DTIC QUALITY INSPECTED 4

19970423 206

Nestor/9704001

Final Technical Report

Principal Investigator
Michael Glier
401-331-9640

Nestor, Inc.
One Richmond Square
Providence, RI 02906

April 1997

DARPA
Defense Advanced Research
Projects Agency
3701 North Fairfax Drive
Arlington, VA 22203-1714
ARPA Number 7017
Contract Number N00014-90-C-0010
January 22, 1990 - March 26, 1993

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency of the U.S. Government.

REPORT DOCUMENTATION PAGE	1. REPORT NO. Nestor/9704001	2.	3. Recipient's Accession No.
4. Title and Subtitle Final Technical Report			5. Report Date April 1997 6.
7. Author(s)			8. Performing Organization Rept. No. NESTOR/9704001
9. Performing Organization Name and Address Nestor Inc. One Richmond Square Providence, RI 02906			10. Project/Task/Work Unit No. 11. Contract(C) or Grant(G) No. (C) N00014-90-C-0010 (G)
12. Sponsoring Organization Name and Address Defense Advanced Research Projects Agency 3701 North Fairfax Drive Arlington, VA 22203-1714			13. Type of Report & Period Covered 14.
15. Supplementary Notes Michael Glier 401-331-9640 Clifford Lau 202-696-4961			
16. Abstract (Limit: 200 words)			
17. Document Analysis a. Descriptors b. Identifiers/Open-Ended Terms Digital Neural Network c. COSATI Field/Group			
18. Availability Statement TACTEC/Batelle Memorial Institute 505 King Ave., Columbus, OH 43201		19. Security Class (This Report) Unclassified 20. Security Class (This Page) Unclassified	21. No. of Pages 208 22. Price

(See ANSI-Z39.18) See Instructions on Reverse OPTIONAL FORM 272 (4-77)

☆ GPO : 1993 O - 381-526 (8393)

Table Of Contents

PROJECT SUMMARY **1**

PURPOSE 1
NEURAL NETWORK BRIEF 1
NI1000 CHIP BRIEF 1
INTRODUCTION TO APPENDICES 2

APPENDIX A, RADIAL BASIS FUNCTION (RBF) NEURAL NETWORKS **A-1**

APPENDIX B, THE RCE NEURAL NETWORK **B-1**

APPENDIX C, IMPLEMENTING NEURAL NETWORKS USING THE NI1000 ..... **C-1**

APPENDIX D, NI1000 TECHNICAL SPECIFICATION **D-1**

APPENDIX E, NI1000 RECOGNITION ACCELERATOR USER'S GUIDE **E-1**

Project Summary

Purpose

The Ni1000 Recognition Accelerator chip was developed by Nestor and Intel for the Defense Advanced Research Projects Agency to provide a processor with built-in neural network intelligence.

This document presents the project results.

Neural Network Brief

A neural network is an interconnected group of simple processing elements that is able to adapt to its environment, or learn, in a way that is similar to the way that the human mind learns. Artificial neural networks emulate the internal structure and behavior of human neural networks by using building blocks with similar behaviors and assembling them in a similar fashion.

Neural networks' ability to solve problems is a combination of the functions provided by processing elements, called neurons, and the interconnections among the neurons. Neural networks can solve very complex pattern classification problems, develop solutions independently and generalize based on limited data.

Ni1000 Chip Brief

The Ni1000 chip implements a type of neural network model called a Radial Basis Function neural network which has the ability to group similar objects together, based on their radial distance from known objects in feature space. It also implements the RCE neural network, a general-purpose, adaptive pattern classification engine.

The Ni1000 is a neural network VLSI chip with 100 inputs connected to each unit of a layer of 1000 prototype storing and recognizing units, each such prototype united connected to each unit of a 50-unit categorizing output layer. The processing time is approximately 1 μs per pass and a weight change time of 100 μs for retention longer than 10 years. Inputs and outputs are digital and TTL compatible.

Weight change calculations are executed on the chip. Silicon VLSI technology with polysilicon floating gates and thin oxides for tunnel elements are used to implement the electrically modifiable synapses in the network. Floating gate devices, programmed and erased by Fowler-Nordheim, tunneling through a thin dielectric were used. The Ni1000 is packaged in a conventional pin grid array packaged with 168 pins.

Introduction To Appendices

The appendices which follow provide information on specific topics.

Appendix A, Radial Basis Function (RBF) Neural Networks describes this particular type of neural work implemented by the Ni1000 chip. The RBF neural network is designed to group similar objects together, based on their radial distance from known objects in feature space. This appendix also provides general neural network concepts and information on additional neural network types.

Appendix B, The RCE Neural Network describes this particular neural network type, which is also implemented by the Ni1000 chip. The RCE neural network was designed as a general-purpose, adaptive pattern classification engine which can solve pattern recognition problems in which data classes are represented by disjoint class distributions, linearly and non-linearly separable class distributions, as well as non-separable classes whose class distributions overlap.

Appendix C, Implementing Neural Networks Using The Ni1000 provides structural information and available commands for the Ni1000 chip.

Appendix D, Ni1000 Technical Specification provides detailed architectural and component information on the Ni1000 chip.

Appendix E, Ni1000 Recognition Accelerator User's Guide provides comprehensive hardware information and programming instructions.

Appendix A
Radial Basis Function (RBF) Neural Networks

Radial Basis Function (RBF) Neural Networks

What is a Neural Network?

A *neural network* is an interconnected group of simple processing elements that is able to *adapt* to its environment, or *learn*. The ability to solve problems is a combination of the functions provided by the processing elements, called *neurons*, and the interconnections among the neurons. As a neural network learns, it modifies some, or all, of the following:

1. the number of neurons present in the network
2. the behavior, or response, of each neuron
3. the connections among neurons

Studies of *biological neural networks* led to the development of the first *artificial neural networks,* which attempted to emulate the internal structure and behavior of biological neural networks using mathematical algorithms and computers. The intent was to exhibit some of the higher level behaviors attributable to biological neural networks by using building blocks with similar primitive behaviors and assembling them in a similar fashion. Subsequently, other neural network models have been developed that emulate some of the higher level behaviors of biological neural networks, even though they may not look much like models of biological neural networks at the primitive level.

Neural networks can:

- solve very complex pattern classification problems with ease, such as:

  - *machine vision*
  - *process control*
  - *industrial parts inspection*
  - *character recognition*
  - *fingerprint recognition*
  - *voice recognition*
  - *and more . . .*
  - *fraud detection*
  - *financial trends analysis*
  - *medical image analysis*
  - *medical diagnostics*
  - *target recognition*
  - *guidance systems*

- develop solutions independently

  Through mechanisms of self organization, a neural network learns the solution to a problem based solely on examples of input data combined with simple feedback signals about the appropriateness of the network response. A network may also continue to learn and enhance its knowledge based on new input data.

  In other words, neural network algorithms are not solutions to applications, but a method by which the network can *synthesize* its own solution based on examples.

- generalize based on limited data

  The neural network is able to solve problems similar, but not identical, to those in the training data.

The Ni1000 implements a type of neural network model, or paradigm, called a Radial Basis Function (RBF) Neural Network. The term RBF refers to the network's ability to group similar objects together, based on their *radial* distance from known objects in *feature space*.

Features, feature vectors and feature space

In order to identify anything, some uniquely descriptive characteristics (features) of the object must be identified. In neural networks, a *feature* is a single measurable characteristic of a pattern, such as its length. Usually, a single feature is insufficient to distinguish among various patterns, so a set of features must be collected that is sufficient to distinguish among all of the types *(classes)* of patterns to be identified. A *feature vector* is an ordered list of feature values that describes a pattern well enough to identify, or *classify*, the pattern.

If every possible combination of feature values were plotted, it would map out a multidimensional *feature space*. The number of dimensions would be the number of features used. Any of the points in this space is specified by a unique set of feature values and it is possible to determine the "distance" between any two of the points. This distance provides a means for determining how similar the original patterns are.

Once a pattern is converted into a feature vector, it has been mapped to one of the points in the feature space. This feature vector is provided to the neural network to tell the it where the pattern fits into this feature space. When the network is learning, feature vectors are provided along with the correct identity, or *class*, of the object that they represent. This tells the neural network to which class the corresponding feature vector belongs. Some of the feature vectors are *committed* to neural memory, thereby becoming a *prototype*.

Once the network has learned everything it can from the available training data, it can then be asked to *classify* unknown patterns. Feature vectors describing the unknown patterns are presented to the network and it is asked to identify the class to which the pattern belongs. It finds the point in space corresponding to the feature vector and finds the nearest prototype(s) committed during learning.

The recognition performance of a neural network classifier is directly related to the quality of the feature set selected. A thorough understanding of the intended application is essential to select features that distinguish among different objects that must be recognized while permitting the network to properly recognize similar, but not identical, examples of the same object.

Because each application will require a unique set of features, no single set of features will suffice for all pattern recognition problems.

APPLICATION	*SAMPLE FEATURES*
Character Recognition	the number of strokes the changes in direction of the stroke the coordinates of the beginning, ending, and intersecting points of strokes the coordinates of points of maximum curvature

Process Control	temperature pressure flow rate spectral data
Fraud Detection	# of credit card purchases in a day, week, etc. date of purchase amount of purchase location of purchases
Animal Recognition	external covering (skin, fur, feathers) # of limbs offspring (born, hatched) where does it live? (land, air, water)

How do you teach a neural network?

In *15 Minutes to Your First Neural Network*, you will be walked through the development of a neural network that recognizes types of animals. It uses the features listed above for Animal Recognition, plus a few more. That example is used here as well to illustrate the concepts.

As with all training, the first step is to gather sample data. First, it is necessary to identify the *classes* or types of animals to be identified. It is important to ensure that the features selected are sufficient to distinguish any of these classes from any other. Then, create training examples that include the description of various types of animals by using the selected features, along with the correct answer. By presenting the set of feature values, along with the correct answer, the neural network will learn that this "point" in the feature space is an example (or *prototype)* of one of the classes. Later, when other patterns with slightly different feature values are presented, the neural network will be able to locate the prototype that most closely approximates the new pattern. This is similar to a child that *classifies* a zebra as a horse. If zebra is not one of the known classes and horse is, horse is probably the best approximation to the correct answer.

Neural networks can also return a *confidence* value that reflects just how similar the pattern was to one of the prototypes. This can be used to ensure that a snake isn't classified as a crocodile, just becase the crocodile was the longest thing in the training set. This confidence level can also be used to select the best answer when the neural network finds multiple prototypes that are similar to the new pattern.

In order to see how well the neural network has learned about animals, it should be testing using some animals that were not used in the training set. This helps to identify where additional training data may be necessary or where there are weaknesses in the selected feature set.

A neural network must use its best "judgement" when a feature vector does not point directly to a prototype. Responses are one of:

- identified, if one identification with adequate confidence occurred
- uncertain, if multiple identifications of adequate confidence occurred
- unidentified, if no identification of adequate confidence occurred

Selecting a Set of Features

When selecting features, it is usually not sufficient to select the obvious characteristics. Sometimes, these are extraneous, that is, other features already provide the same information and are more effective. Including extraneous features makes the feature space, and the neural network that maps that space, excessively large. Some features can be useless, since they may describe the object, but do nothing to discriminate one from another. These also needlessly increase the size of the feature space. Others can be confusing because they are sometimes, but not always true. A possible example for animals is color. White or brown may be found among several of the classes of animals to be identified, yet within one class, some may be white or brown and others may be black.

Some classification problems already have ready-to-use features. Process control problems are often like this. They use sensor data that is exactly the information required to identify normal and abnormal classes of observations. This can be fed directly, or with little modification, into the neural network. However, many problems are not so simple. For example, character recognition does not typically feed a page full of image pixels directly to the neural network. Instead, it does some image preprocessing first. This processing of raw data to produce the desired features is called *feature extraction.*

Feature Extraction

Feature extraction is the generation of a set of feature values which can adequately describe a pattern to the neural network. It is the process of converting raw data into an alternate representation that can be used more efficiently for identification.

In our animal classification problem, some mechanism must determine the type of covering (skin, fur, scales or feathers) and count limbs.

Training data

In addition to selecting descriptive and discriminating features and providing a means to extract them from the raw data, the selection of the training examples is crucial. First, it is necessary to provide examples of all of the kinds (classes) of patterns to be recognized. In our animal example, we would not expect the neural network to recognize dogs if we provided no examples of dogs. Because of the feature set that we chose, we would not need many examples of dogs, but there are those occasional dogs with a missing leg. Without an example of a dog with a missing leg, we are not certain what the neural network would call the dog with three legs, although it would likely call it a dog. But, what if the dog had only two legs?

Ni1000 Neural Network Paradigms

A neural network paradigm (or model) consists of three components: its structure, its learning algorithm and its classification algorithm. The Ni1000 provides one network structure, a Radial Basis Function using "city-block" distances in feature space, two built-in learning algorithms (plus an ability to do learning externally or program in custom algorithms) and two classification algorithms (or transfer functions).

The learning algorithms are:

1. Probabilistic Neural Network (PNN). This algorithm "memorizes" all of the input patterns, that is, it stores every training pattern as a prototype. Additionally, it keeps track of how many examples of each case occurred in the training set. This helps when the neural network is unsure of which way to go between possible choices. It uses this information to cause the network to be more likely to choose the class that occurs more often in the training set.
2. Restricted Coulomb Energy (RCE). This algorithm creates an "*influence field*" around each prototype that it stores. If another training example is presented that falls within the field and is of the same class, the latter example is not stored, since there is sufficient knowledge already present to properly classify that point. However, if a training example is presented that falls within the field and is of a different class, the field of the former prototype is reduced in size so that it does not contain the new pattern. Any time a pattern is presented that is not covered by the field of an existing prototype, it is committed as a new prototype. As in PNN, the patterns that RCE does not store are not ignored. They are counted and are used to bias decisions when there are multiple nearby prototypes.

The classification algorithms are:

1. Deterministic. If a feature vector falls within the *influence field* of an existing prototype, the feature vector is identified as belonging to the class of that prototype. It is possible to produce uncertain results, in which a feature vector falls within the influence field of more than one prototype.
2. Probabilistic. An exponentially decaying (Gaussian) function of the distance from the prototype to the feature vector's point in feature space is calculated. After weighting it (multiplying) by the count of training examples attributed to that prototype, a "raw" probability is produced. All of the contributions of all prototypes of a given class are summed to produce a *probability density* for that class. These can then be *normalized,* i.e. each is divided by the sum of all probability densities for all classes. This results in an estimated probability that this classification is correct, based on the training information provided.

The paradigms that result from the three valid combinations are:

1. PNN: PNN learning and Probabilistic classification.
2. RCE: RCE learning and Deterministic classification
3. PRCE: RCE learning and Probabilistic classification

Learning with Radial Basis Functions

These are all RBF (Radial Basis Function) types of neural networks, so named because they are created from radially symmetric clasification functions (influence fields and Gaussian functions) of the data inputs.

RBF networks have been applied to function approximation and pattern recognition problems. They have the capability of representing arbitrary functions and they converge quickly, resulting in rapid training. The reliability of the output generated by an RBF network may be affected by the features used for training and the volume of data used for training. Improvements in accuracy may occur by using a larger training set and/or better features.

Suggesting that one type of neural network paradigm is better than another is inappropriate, since the optimal type of paradigm to use may vary depending on the pattern recognition problem at hand and the training data available. Recognition memories for the same application could even be generated in different ways using each of the RBF paradigms mentioned above. The quality of these memories needs to be thoroughly tested in order to determine which method may be more suited to solve a specific pattern application problem. A brief description of each of the three paradigms is provided below in order to assist you in selecting an appropriate neural network paradigm to initially try with your application.

RCE

RCE is a three-layered, feed-forward, "*deterministic*" neural network paradigm of radius-limited prototypes.

The objective in pattern classification is to process input pattern vectors of specified classes (or categories) and to determine an appropriate class response. When an input pattern vector is presented to an RCE network, it appears as a pattern of activities on the input layer. The RCE network covers or "maps out" pattern class territories by covering the territories with a set of influence fields of internal layer prototypes.

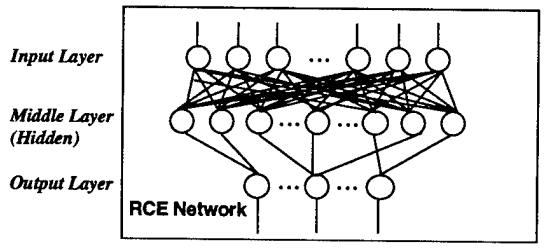


Figure 1: RCE Paradigm Diagram

The first layer of prototypes pass their input to a second hidden layer. Second-layer prototypes are radius-limited as well. Prototypes in the output layer are active if any of the prototypes to which they are connected are active. Input pattern vectors are presented to the network. Prototypes get committed for each new type of vector that is processed. Similar vectors may cause existing prototypes' radius values to shrink. Learning is accomplished rapidly, reaching convergence within in a few passes through the training data (4 or 5). RCE performs well when there are distinct classes with very unique characteristics.

Learning in the RCE network controls:

- the number of internal (hidden) layer prototypes
- the values of the connections between the internal and input layers
- the firing thresholds of internal layer prototypes
- the numbers of output layer prototypes
- the pattern of connectivity between the internal and output layers

To accomplish the above tasks, RCE is equipped with a *deterministic* type of internal prototype since its firing can cause an "identified" (or unambiguous)

response of the network for the corresponding pattern vector class. (It has seen an input vector, recognized it and matched it up with one identifying class.)

PRCE

PRCE is a three-layered, feed-forward, "*probabilistic*" neural network paradigm of radius-limited prototypes.

PRCE behaves much like RCE except that the hidden internal layer contains clusters of probabilistic prototypes. Applying user-specified variables, the PRCE paradigm is better able to make decisions about pattern vectors where there is confusion when trying to recognize the pattern vector (when there are two or more possible answers).

In the probabilistic RCE network, only *probabilistic* prototypes are committed in the internal layer of the network. Instead of one class firing, several classes may fire, causing some uncertainty about the response. The response class is determined based on the threshold which indicates the confidence requested in the response. This is an integer value supplied by the user. The higher this threshold, the higher the confidence that is required of the network prior to responding with a class. The lower the threshold, the less certain the network needs to be before responding with a class.

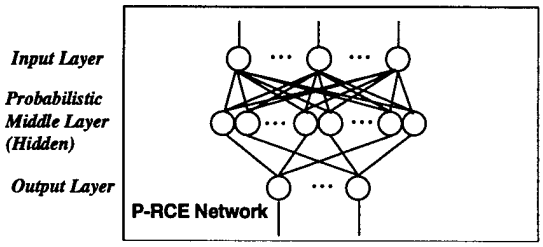


Figure 2: PRCE Paradigm Diagram

Learning in the PRCE network controls:

- the number of internal (hidden) layer prototypes
- the values of the connections between the internal and input layers
- the firing thresholds of internal layer prototypes
- the numbers of output layer prototypes
- the pattern of connectivity between the internal and output layers

To accomplish the above tasks, PRCE is equipped with a *probabilistic* type of internal prototype, and its task is to map out the confusion zones. A deterministic prototype becomes probabilistic if its influence field size shrinks below some user-specified threshold. When a probabilistic prototype fires, it fires the output prototype to which it is connected, but, weakly, in the sense that the output of the network is now officially "uncertain" between two or more possible classes. Even if only one output prototype is firing, if it is being stimulated only by probabilistic internal prototypes, the answer of the network remains uncertain. The output prototype's classification is offered as a possible, but not definite, classification of the input.

During learning, probabilistic prototypes block the commitment of deterministic prototypes in any space that they occupy. If an input pattern vector for class B falls only within the influence fields of one or more probabilistic prototypes for class A, the network will not commit a deterministic prototype for this pattern vector. It will, however, commit a probabilistic prototype for class B, centered at the input pattern vector site, with influence field size equal to a user-supplied threshold. The network allows the influence fields of probabilistic prototypes for one class to cover the center point of probabilistic prototypes for another class. This is another way in which they differ importantly from deterministic prototypes. Additionally, the influence field size of a probabilistic prototype is never adjusted; it remains at the user-supplied threshold. The commitment procedure will result in a layering of the confusion zone with probabilistic prototypes for A and a layering with probabilistic prototypes for B. In this way, any input pattern vector that falls within the confusion zone will fire at least one probabilistic prototype for A and one probabilistic prototype for B. This will cause the network to respond with the uncertain response, "A" or "B".

PNN

PNN is a four-layered, feed-forward, "*probabilistic*" neural network paradigm of radius-limited prototypes.

PNN is similar to PRCE in using probabilities to try and recognize pattern vectors. PNN differs in that it commits a prototype for every pattern vector that it sees during training, then it sums the inputs from the neurons committed which correspond to the category from which the training pattern was selected. In the figure below, these "summation" neurons are generated from two-input neurons in the previous layer. The prototypes committed during PNN training stay a fixed size and do not shrink. PNN training processes the data only once, which makes the training of a PNN neural network very fast, but because a prototype is required to be committed for every input vector, the network could also become quite large.

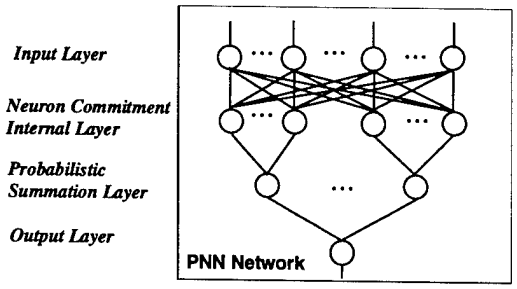


Figure 3: PNN Paradigm Diagram

In the PNN network, only *probabilistic* prototypes are committed.. Instead of one class firing, several classes may fire, causing some uncertainty about the response. The response class is determined based on the threshold which indicates the confidence requested in the response. This is an integer value supplied by the user. The higher this threshold, the higher the confidence that is required of the network prior to responding with a class. The lower the threshold, the less certain the network needs to be before responding with a class.

Appendix B
The RCE Neural Network

The RCE Neural Network*

by

Douglas L. Reilly, Ph.D.

Nestor, Inc.

I. Introduction

A. Background

The RCE neural network was designed as a general-purpose, adaptive pattern classification engine. Following a patent application submitted in 1980, a US patent was granted for the RCE network in 1982.[1] The first description of the network to appear in a technical journal was published in 1982, with a later elaboration appearing in 1987.[2,3]

As an adaptive pattern classification engine, the RCE network can solve pattern recognition problems in which data classes are represented by disjoint class distributions, linearly and non-linearly separable class distributions, as well as non-separable classes whose class distributions overlap. In this latter case, the RCE network outputs local probability density information that, along with known or assumed information on a priori class probabilities, can be used to compute an optimal pattern classification decision.

B. Network Description

The RCE network consists of three layers of "neuron cells", with a full set of connections (each represented by a connection weight) between the first and second layers, and a partial set of connections between the second and third layers. (See Figure 1.)

Each input layer cell represents a feature (a measurable characteristic) of an incoming pattern (an input signal) that the network assigns to some pattern class (category). The input signal is sometimes referred to as the pattern of activity (or activation pattern) of the input layer cells. The choice of input features is made based upon the nature and complexity of the pattern recognition problem.

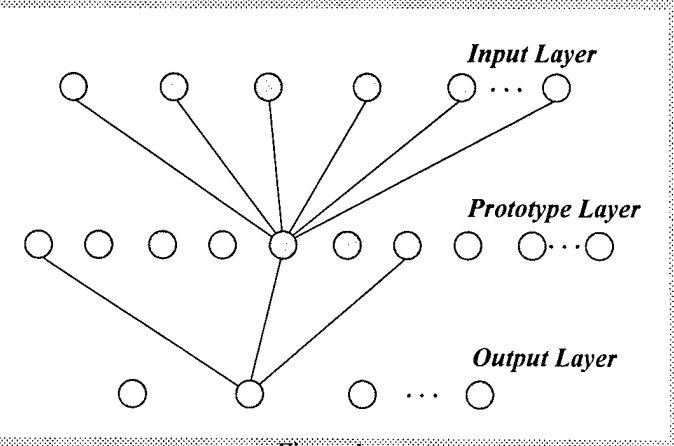


Figure 1

RCE Network

The middle-layer cells are called prototype cells. Each prototype cell contains information about an example of a learned pattern class that occurred in the training data. The connections between a prototype cell and the input layer cells store the feature values of the class exemplar associated with the prototype.

Each cell on the output layer corresponds to a different pattern class represented in the training data set. Prototype cells are class-specific. This class affiliation is represented structurally in the network by the

---

* To appear in CRC Press Industrial Electronics Handbook.

single connection that a prototype cell makes to one and only one output cell. (See Figure 1.) However, more than one prototype can be associated with the same pattern class. This means that an output cell can be connected to more than one prototype cell.

The knowledge about a class of patterns is stored in the network as a set of reference examples (prototypes) and the capability to generalize from these examples to new class instances. The RCE network applies a procedural, supervised training algorithm to grow the numbers of prototype and output cells, and to define values for their network connections, in order to perform pattern classification.

C. Relation to Other Neural Networks

RCE prototype cells use an exemplar-based function to compute their responses to a pattern of activity on the input layer. In most cases this function computes either the Euclidean or city-block distance between the signal on the input layer and the vector of weights (the prototype vector) associated with the cell. Because of this, the RCE network is related to the class of radial basis function networks (RBF's) introduced in 1988.[4]

The most commonly used training algorithm for neural networks is currently the back propagation of errors algorithm, first described in 1974 and, independently, in 1986.[5,6] The many variations of this algorithm all involve modifying network weights based upon a gradient descent approach to minimizing an error term. The error term is defined as a function of the difference between the desired and actual network responses to a pattern of activity on the network input layer. Whereas the back propagation of errors technique has the advantage of being able to train neural networks with arbitrary numbers of cell layers, it has the disadvantage of training very slowly, requiring many passes (epochs) through the training set before the network weights converge on a final set of values. Further, the training algorithm can occasionally result in the network becoming "stuck" in a condition that prevents further changes to the weights, without having arrived at an accurate solution to the pattern recognition problem.

By contrast, the RCE network uses a procedural training algorithm that avoids the long training times and problems of false convergence that can occur with back propagation. Because it does not employ the gradient descent approach to minimizing an error function, RCE training offers guaranteed convergence, completing its training usually in 3-4 training passes through the data. Unlike back propagation with its ability to train any neural network regardless of structure, the RCE training algorithm can only be applied to networks having three layers of cells. For pattern recognition applications, this is not a serious limitation since researchers have shown that any pattern classification problem can be solved by a neural network having at most three layers.[7] Additionally, systems have been constructed with multiple component RCE neural network modules, each of which learns to solve portions of a pattern recognition task and which, together, cooperate to provide an integrated solution to the overall classification problem.[8]

The RCE training algorithm grows the number of middle and output layer cells used by the network to solve the pattern recognition problem. RCE training differs from that employed in the related Probabilistic Neural Network (PNN), in that it allocates a new middle-layer cell only when the existing set of prototype cells is insufficient to correctly classify a pattern in the training set.[9] PNN allocates a middle-layer cell for each exemplar in the training set.

D. Advantages of RCE Network

The distinctive capability of the RCE network to automatically size itself during training solves a design issue for its users. By controlling the allocation of prototype and output layer cells, the RCE training algorithm eliminates the need to know in advance how many cells to specify in the middle layer of the network. This choice is a critical design parameter for users of networks trained with back propagation. Choosing a number of middle layer cells that is either too small or too large can prevent such networks from training to a good solution for a pattern classification problem.

During RCE training, the middle-layer prototype cells develop expertise in classifying input signals that occur within their neighborhood of feature space. Information about a pattern class is represented among a subset of these prototype cells. Because of this, and because of the ability of the network to commit such cells dynamically, it is possible to incrementally train a previously trained network on new data examples without having to re-present the entire training set to the network.

During the course of training on new data, the RCE network will produce incorrect network responses that will guide the developer in deciding which kinds of previously trained data needs to be re-presented. As an example, a network trained to recognize handwritten numbers will confuse 2's with examples of Greek α's when examples of Greek handwritten letters are first presented. In this case, only examples of 2's need be re-presented to the network while it is being trained on the new class α. This dynamic category learning can be important for applications where in-field training is required and re-presentation of an entire initial training set is not possible or practical.

The RCE network is a relatively simple network to understand, in terms of its training procedure and its mechanism for classification. The procedural aspect of the training function lends itself toward a straightforward description in terms of feature space diagrams. This, together with the relatively simple mathematics employed by the network, makes the RCE network intuitively easy to apply to a pattern recognition problem.

There are a number of variations of the RCE network that have been implemented for pattern classification problems. The following description characterizes the RCE network training algorithm and output response modes as they are executed in a commercially available chip (the Ni1000 Recognition Accelerator™) that implements RCE along with other radial basis function networks.

II. Training the RCE Network

A. Pattern Classification–Learning Territories in Feature Space

The clearest description of RCE network functions makes use of feature space diagrams. It is helpful to begin by introducing the term "feature space."

An input signal to the RCE network consists of a set of feature values, each value represented as the activation of a particular input cell. The set of features chosen to characterize an input signal for the network defines a feature space. The number of features in the set is referred to as the dimensionality (the number of axes) of the space.

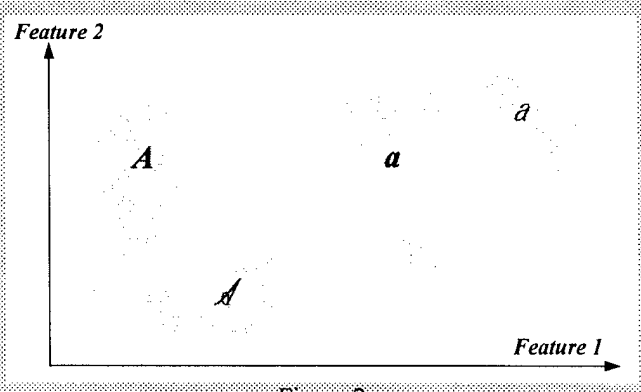


Figure 2
Disjoint regions for the class of patterns corresponding to the letter "A" in a hypothetical 2-D feature space.

The feature values describing a particular input signal locate the signal as a point in the feature space. The feature space itself is the set of all possible feature value combinations; i.e., it is the set of all possible points in the space. To measure the closeness or similarity between two input signals, a distance may be computed between their corresponding points in the feature space.

The correspondence between an input signal and a point in the feature space implies that a class of patterns is represented by a region or territory (i.e., a set of points) of the feature space. In general, the shape of the territory associated with a given class of patterns may be arbitrarily complex. A class of patterns may even consist of a collection of disconnected (disjoint) regions. (See Figure 2.)

The solution to a pattern recognition problem requires an accurate description of the relevant class territories in feature space. With such a description, the class of an input signal can be identified by determining if the signal is contained within any of the feature space regions associated with that class.

The challenge in solving a pattern recognition problem is to accurately characterize the shapes of class territories that may be arbitrarily complex. It is useful to distinguish between two kinds of problems. In the case of simple (or separable) class regions, each point in the feature space belongs to one and only one category of patterns. This means that there is no overlap between the territories of any classes, although their shapes may be arbitrarily complex and disjoint class regions are allowed. (See Figure 3a.)

Pattern classes whose regions overlap are said to have non-separable (or overlapping) class territories. (See Figure 3b.) Any point in their shared feature space regions is associated with more than one class. In such

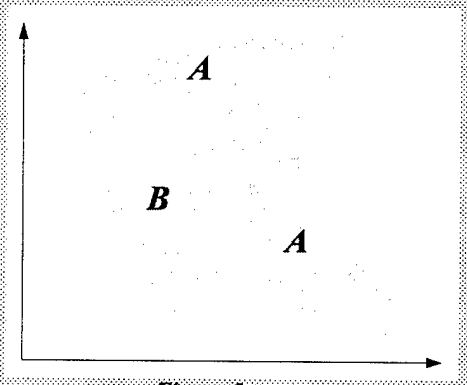


Figure 3a

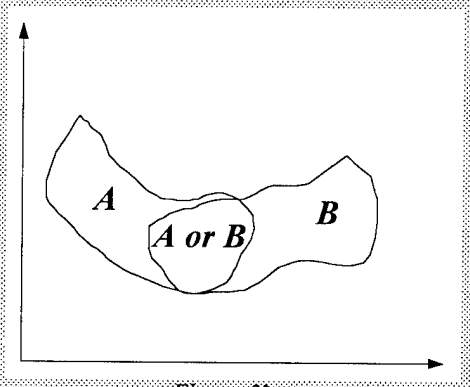


Figure 3b

In Figure 3a, pattern classes A and B are separable (A consists of two disjoint regions); in Figure 3b, pattern classes A and B are overlapping, sharing a region of points that could belong to either class A or class B.

cases, a probability of class membership must be estimated for a given point in the overlap regions.

B. Prototype Cell Responses During Training

When the RCE network trains on data, it learns the shapes of class territories in the feature space. These characterizations are developed by and stored in prototype cell parameters. An RCE prototype cell is characterized by five elements: its class, χ, its weight vector, $\underline{\omega}$, its cell threshold, λ, its pattern count, κ, and its smoothing factor, σ. During training, all but the smoothing factor play a role in prototype cell development.

The prototype cell weight vector, $\underline{\omega}$, represents the set of weighted connections between the prototype cell and each of the input layer cells. Because each prototype cell has one connection with each input cell, the prototype weight vector has the same dimensionality as the input signal. Just as the input signal defines a point in the feature space, so a prototype cell weight vector defines a point in the same feature space. (See Figure 4.)

In response to a signal on the input layer, each prototype cell computes a distance between the input signal and the prototype vector stored in its weights. When the "city-block" function is used for this distance, it is computed by the ith prototype cell as

$$d_i = \sum_{j=1}^{N_D} |\omega_{ij} - x_j| \qquad \text{(city block distance)} \qquad [1]$$

where ω_{ij} is the weight connecting the ith prototype cell to the jth input cell

x_j is the activity of the jth input cell (i.e., the jth feature value of the vector $\underline{x}$)

N_D is the number of input cells (i.e., the dimensionality of the feature space)

During training, a prototype cell will become active if the prototype-to-pattern distance, d, is less than the cell threshold, λ; if the distance d is greater than or equal to the cell threshold λ, then the prototype will not respond to the input signal. Referring to the output of the ith prototype cell as p_i,

$$p_i = 1 \quad \text{if} \quad d_i < \lambda_i \qquad \text{(prototype fires)} \qquad [2.a]$$

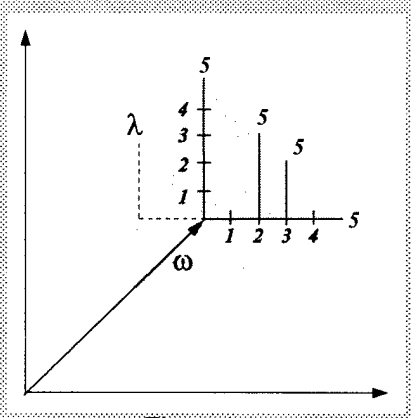


Figure 4a

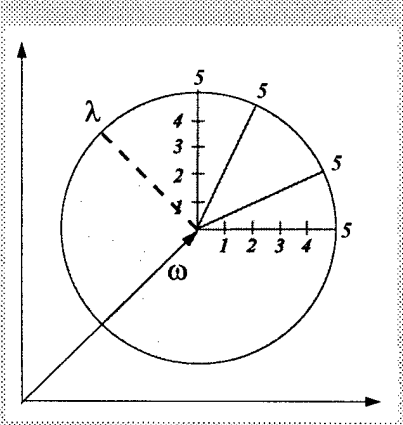


Figure 4b

Examples of differently shaped prototype influence fields, each of size λ=5, in a hypothetical 2-D feature space. The ω marks the location of the weight vector. Use of the city-block distance for the prototype-to-pattern comparison yields the diamond-shaped influence field of Figure 4a; use of the Euclidean distance function yields the circular influence field of Figure 4b.

$$p_i = 0 \quad \text{if} \quad d_i \geq \lambda_i \qquad \text{(prototype inactive)} \qquad [2.b]$$

The cell threshold, together with the city-block distance function, describes a "region of influence" around the prototype cell in the feature space. During training, a prototype cell will fire for any input signal whose corresponding feature space location lies within the prototype's influence field. In the two-dimensional feature space illustrated in Figure 4a, the city-block distance function creates an influence field that looks

like a diamond-shaped area centered on the point defined by the prototype weight vector.[1] As indicated earlier, it is also possible to choose a Euclidean distance function for prototypes.

$$d_i = \left[\sum_{j=1}^{N_D} (\omega_{ij} - x_j)^2 \right]^{1/2} \qquad \text{(Euclidean distance)} \qquad [3]$$

In this case, the influence field of a prototype in a two-dimensional feature space looks like a circular disk as shown in Figure 4b. Because it is easier to draw circles than diamonds, prototypes will be pictured with circular influence fields in the diagrams referred to in the following discussion.

C. The RCE Training Algorithm

Each prototype cell represents some local information (i.e., information in the small neighborhood of the feature space defined by its influence field) about the nature of the pattern class with which it is associated. During training, the RCE network will allocate prototype cells, positioning and sizing their corresponding influence fields so as to cover the feature space regions for each class of patterns present in the training data.

Before any training occurs, the RCE network can be pictured as consisting of a set of input cells and a set of unallocated prototype and output cells. By unallocated, we mean that they are simply not yet "wired into" the network.[2] The network is trained through a sequence of input signals, each presented with its correct classification. (A set of such patterns is called a labeled training set. A training algorithm that requires a labeled training set is called a supervised learning algorithm.)

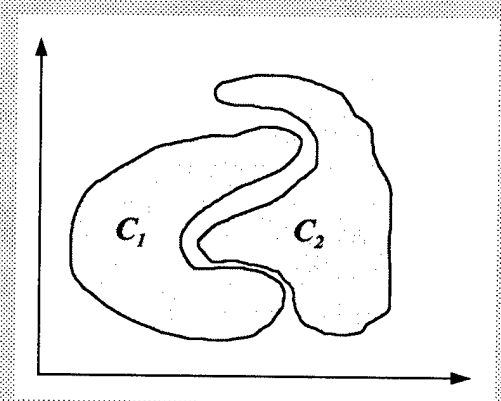


Figure 5
Hypothetical two-class recognition problem.

The training procedure makes use of three mechanisms: prototype cell commitment, prototype threshold modification and prototype pattern count modification. The process is illustrated for the pattern recognition problem shown in Figure 5, which portrays two non-linearly separable pattern classes, C_1 and C_2, in a hypothetical two dimensional feature space.

1. Prototype Cell Commitment

Let the first pattern presented to the network be an input signal, $\underline{x}_1$,belonging to class C_1. Presentation of this pattern causes a new prototype cell to become committed (i.e., wired up) in the network. The influence field of this new prototype will be centered on the pattern $\underline{x}_1$. (See Figure 6.) In the process of wiring up a prototype cell, several changes are made to the network.

First, the input signal is loaded into the prototype weight vector:

$$\underline{\omega}_1 \longleftarrow \underline{x}_1 \qquad [4]$$

---

[1] In n dimensions, the influence field is an n-dimensional tetrahedron.

[2] In the commercial chip that implements the RCE, there are a total of 1000 unallocated prototypes and 64 unallocated output cells available for training purposes.

This means that the influence field of the newly committed prototype will be centered on the pattern that caused the cell to be committed to the network. In effect, the prototype cell is "memorizing" a class exemplar from the training set.

Secondly, the prototype cell is assigned a cell threshold, λ. This assignment creates an influence field around the prototype. The prototype will use its influence field to determine how much it can generalize to respond to novel patterns that are similar to the memorized exemplar. In the case of the first prototype to be committed, the prototype is assigned the cell threshold λ_{max}, a user-specified parameter that defines the largest size that any prototype influence field can ever have:

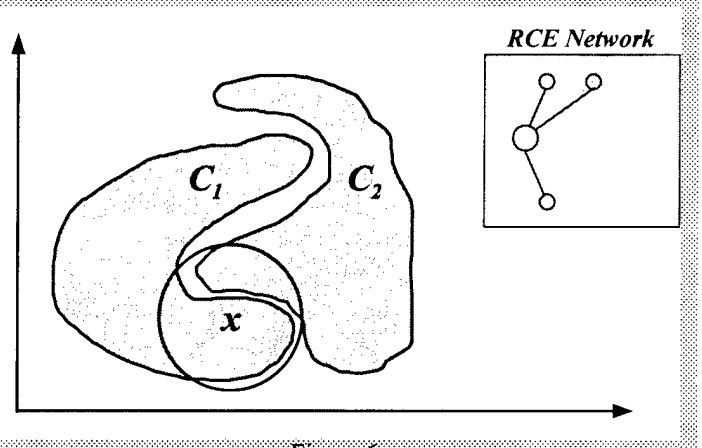


Figure 6

Prototype for C_1 class committed as a result of exemplar x. Pictured at upper right is diagram of RCE Network, with newly committed prototype and output cell for C_1.

$$\lambda_1 \longleftarrow \lambda_{max} \qquad [5]$$

Prototypes committed after this prototype will have their cell thresholds set either at λ_{max} or at some value less than λ_{max}, based upon their position with respect to other prototypes already present in the network.

Thirdly, a connection is made between the prototype cell and the output cell belonging to the class of the current input signal. This assigns a pattern class to the prototype:

$$\chi_1 \longleftarrow C_1 \qquad [6]$$

In this case, since no previous examples of this class (or any other) have been seen, a new output cell is committed to the network. Output cells are committed simply by establishing a connection to the newly committed prototype cell. The connection between the prototype cell and its associated output cell will carry a counter (the pattern counter, κ) that will store the number of times this prototype has correctly fired in response to a pattern belonging to its associated class. For a newly committed prototype, the pattern counter is set to one:

$$\kappa_1 = 1 \qquad [7]$$

When the next input signal is presented to the network, the prototype activation is computed according to [1] and [2]. If the input falls within the prototype's influence field, the prototype cell will fire; this, in turn, triggers the corresponding output cell to fire. If the input signal is an example of class C_1, the network output will correctly classify the pattern. In effect, the network uses the prototype to generalize to recognize this new instance of the pattern class.

As long as subsequent input signals belonging to this class fall within the influence field of the prototype representing this class, no additional prototype cells are committed and no changes occur to the influence field of the prototype. However, each time an input falls within the prototype's influence field and matches the prototype's class, the prototype's pattern counter is incremented in order to keep a count of the number of "correct-class" patterns that have occurred within the prototype's influence field.

If $\quad$ Class ($\underline{x}$) = Class ($\underline{\omega}_i$) AND $p_i = 1$,

then $\quad \kappa_i \longleftarrow \kappa_i + 1$ [8]

The first occurrence of an input signal that belongs to this class but falls outside the influence field of the existing class prototype causes a second prototype to be committed for the class. (See Figure 7.) The same commitment process occurs as described above: the input signal is loaded into the weight vector of the new prototype, the prototype cell threshold is set to λ_{max} and a connection is made between this new prototype and the output classification cell. The counter stored in this connection is initialized to 1.

As successive examples of this class are presented during training, each prototype cell determines its response by computing its distance to the input signal according to equation [1] and comparing that distance to the prototype cell threshold stored with each prototype. A new prototype is committed in the RCE network only when an input signal does not fall within the influence field of any existing prototype belonging to the input signal's class.

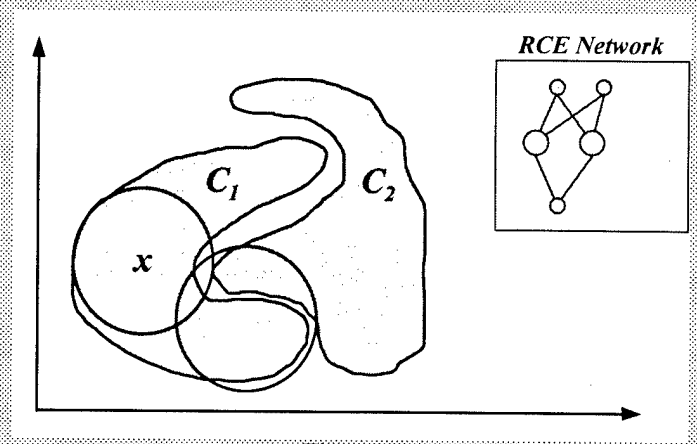


Figure 7

Second prototype for C_1 committed as a result of example pattern that is too dissimilar for initial prototype to classify. Upper right-hand picture shows additional prototype cell being committed and connected to output cell for class C_1.

Suppose an input signal belonging to a new pattern class, C_2, is presented to the network. Assume it falls outside the influence fields of any of the existing C_1 prototypes. This input will cause a new prototype cell to be committed; the input signal values will be loaded into the new prototype weight vector and the prototype will be assigned a cell threshold equal to λ_{max}. (See Figure 8.) Because this is the first example of a new class of patterns, a new output cell is committed as well, representing the class C_2. The counter connection between the new prototype cell and the C_2 output cell is initialized to 1.

As illustrated in Figure 8, the influence field of a newly committed prototype may overlap the influence fields of existing prototypes belonging to different pattern classes. During training, influence fields are only tentative hypotheses about the class membership of the feature space points they contain. As the next section will discuss, future training examples may cause prototype influence fields to be reduced in size. If the class affiliation of influence fields is still uncertain during training, what is certain is that the central point of a prototype influence field (as defined by the weight vector, $\underline{\omega}$) must have a non-zero probability of belonging to the class of the prototype. (There is at least one input training pattern located at that point in the feature space that belonged to that class; this is the pattern that gave rise to the prototype.) Thus, the training algorithm allows a newly committed prototype to overlap the influence fields of other prototypes (in the chance that further training may yet revise their current thresholds to yield smaller influence field

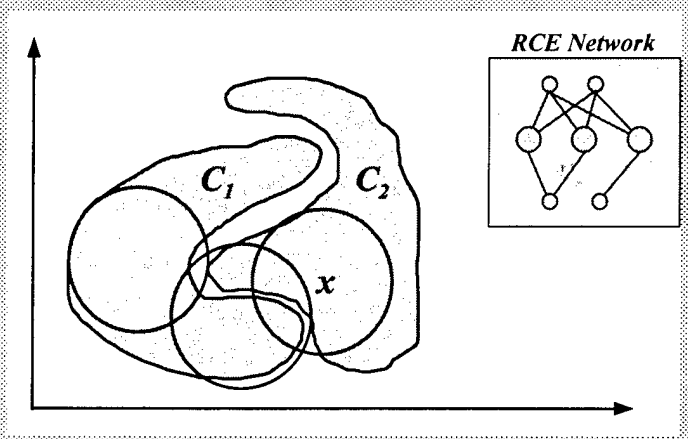


Figure 8

Example for new class C_2 causes new prototype cell (and output cell for C_2) to become committed in network. Note overlap between newly committed prototype and previous prototype for C_1.

sizes), but it will not allow the influence field of a newly committed prototype to be so large as to contain the central point of a prototype for an opposing class.[3]

Thus, the influence field size of a newly committed prototype is chosen to be the smaller of (1) the distance to the closest prototype of any other class (i.e., different from the class of the prototype to be committed) and (2) λ_{max}.

As we shall see in the next section, there is a value below which influence fields (and cell thresholds) are not reduced; this value is λ_{min}. The value of λ_{min} sets a lower bound on the size of newly committed influence fields. Thus, the full specification for influence field determination for newly committed prototypes is the following:

Initial threshold of newly committed prototype = [9]

the smaller of [λ_{max}, the larger of (λ_{min}, the distance to the closest opposing class prototype)]

2. Prototype Cell Threshold Modification

Now suppose that a new example of the first class C_1 is presented to the network, and that it falls within the influence field of a prototype for C_2. The C_2 prototype incorrectly fires, causing the output cell for C_2 to fire. (See Figure 9a.) The network's response is corrected during training by reducing the value of the threshold for the C_2 prototype to a point where the influence field of the C_2 prototype just excludes the input signal. (See Figure 9b.) (Remember that the size of the prototype influence field is controlled by the value of the cell threshold.)

Just as there is a user-specified parameter which determines the maximum size of a prototype threshold (and correspondingly, the maximum influence field size), so there is a user-specified minimum threshold, λ_{min}, beyond which prototype thresholds are not reduced. As we shall see in the section on RCE Responses, prototypes that have reached this minimum size participate differently than prototypes that are above this threshold value in generating the network response to a pattern.

Since prototypes cannot be reduced below this value of λ_{min}, the value of λ_{min} sets a lower bound on the size of the influence field of a newly committed prototype. The new threshold for the C_2 prototype is

$$\lambda_{modified} = \text{larger of } [\lambda_{min}\text{, distance between input signal and prototype for } C_2] \qquad [10]$$

If none of the influence fields for C_1 prototypes contain the input signal, then a new prototype is committed for C_1 based on the input signal. Otherwise, the reduction in the incorrect prototype cell's threshold is sufficient to correct the response of the network and correctly classify the input.

---

[3] A prototype that is to be committed with an influence field size of λ_{min} (defined as the smallest value an influence field can have) may contain the central point of prototypes that do not belong to its class.

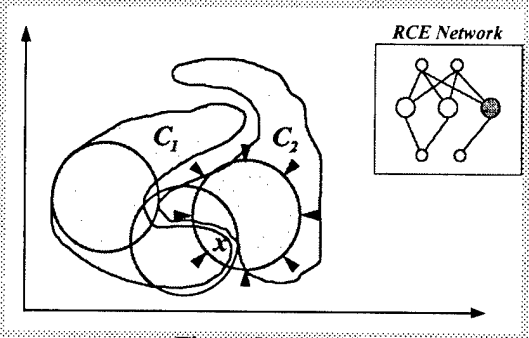


Figure 9a

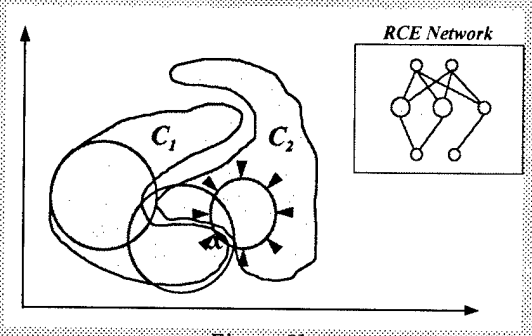


Figure 9b

Example of class C_1 causes prototype for C_2 to fire. The training algorithm identifies the incorrectly active prototype (shown as the darkened cell in the network diagram at upper right of Figure 9a) and reduces the cell threshold so that the influence field just excludes the input signal (Figure 9b.) Prototype cell with modified threshold is pictured with a smaller circle in the network prototype layer in the upper right-hand corner of Figure 9b.

3. Prototype Pattern Counts

In addition to prototype (and possibly output) cell commitment and prototype cell threshold modification, the only other mechanism involved in the RCE training procedure is the incrementing of prototype pattern counts. The pattern count of a prototype is incremented for every pattern that falls within the prototype influence field and that belongs to the same class as the prototype.

Prototype pattern counts are used by the network to approximate the local probability density values for a given class in a particular region of the feature space. This is important in those problems where the class territories are non-separable.

In RCE training, the last training epoch is one in which no new prototypes are committed and no prototypes have their cell thresholds modified. (Further training with the given training data set would cause no changes to the network parameters.) At the beginning of every training epoch, all prototype pattern counts are initialized to zero. The pattern counts that develop during the last training epoch are those that are finally stored with the prototypes.

As noted before, the reduction in the influence field size of a prototype can alter the subset of correct class training patterns that lie within its new influence field size. Obviously, the introduction of a new prototype during the course of a training pass can also result in a smaller pattern count for this prototype than would occur if the prototype existed at the beginning of the training epoch. Thus, the pattern counts that develop during the last training pass are the most accurate estimators of probability density values because they develop for a set of prototypes whose number and influence field sizes have remained unchanged during the training pass.

4. Guaranteed Rapid Convergence of RCE Training

Figure 10 shows that, for separable pattern class problems, this simple RCE training procedure will result in coverings of the pattern class regions that correctly approximate their shape, regardless of the complexity of the shape and regardless of the number of disjoint territories that may comprise the definition of a pattern class.

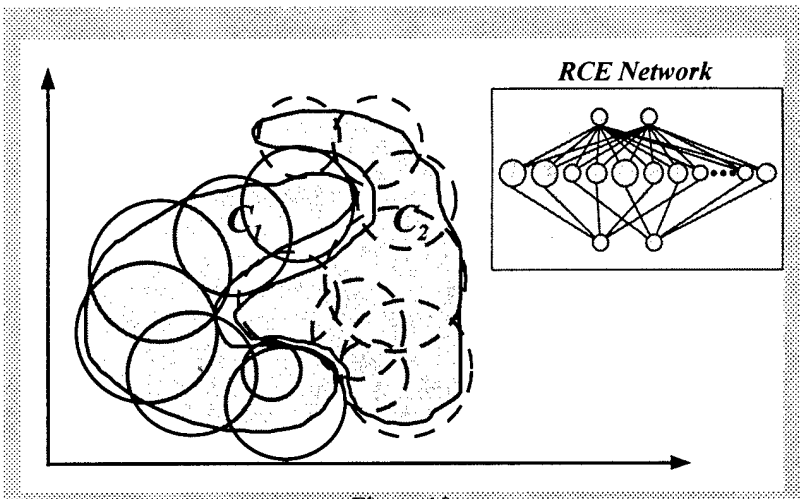


Figure 10

Fully trained network has committed sufficient prototype cells and modified their cell thresholds so that the prototypes for C_1 (shown as solid line circles) cover the class territory for C_1, while the prototypes for C_2 (shown as dashed circles) cover the class territory for C_2.

The RCE network requires only a small number of presentations of the training set before it converges to a final solution. More than one training pass is required because a reduction in the size of a prototype's influence field during training may result in its failure to identify patterns which, when initially presented, fell within its initially larger influence field. In such cases, these patterns may give rise to additional prototypes. Eventually, however, a training pass will occur in which no new prototypes are committed and no prototypes have their influence fields reduced. At this point, the RCE training has converged on a solution of the pattern recognition problem. This convergence is guaranteed to occur, and usually occurs in no more than 3-4 passes through the training set.[4]

III. RCE Network Responses

The RCE network can generate responses in either of two response modes. The first mode is geared toward providing a rapid identification of a pattern class that is separable from all other classes in the training set. However, if this mode does not provide a unique class identification, a second output mode can be invoked to provide an estimate of pattern class probabilities for the input signal.

A. Fast Response Mode

In this first mode of response, the network computes prototype cell activities for each prototype cell in the network by computing the pattern-to-prototype distances and comparing these with the threshold values stored with each prototype. In this mode of response, prototype cells use a modified version of the activation function used during network training.

$$\text{If} \quad (d_i < \lambda_i \quad \text{AND} \quad \lambda_i > \lambda_{min}) \quad \text{then} \quad p_i = 1 \qquad \text{(prototype fires)} \qquad [11]$$

As the condition [10] indicates, in order for a prototype to be active, it must not only contain the input signal within its influence field, it must also have an influence field larger than the minimum size.

Each output unit performs a simple OR function on the input signals arriving from the subset of prototype cells to which it is connected. Thus, the output cell functions as a detector to indicate if any of its associated prototype cells that are above minimum-influence-field size are responding to the input signal.[5]

---

[4] The number of training epochs required is sensitive to the ordering of class examples in the training set. Faster convergence occurs for a randomly ordered training set as opposed to a set in which all examples of one class are presented, followed by all examples of the next, etc.

A single responding cell on the output layer of the network indicates an unambiguous identification of the input signal with that pattern category. If multiple output cells are active, or if none are active, then a second mode of response can be invoked to determine the probabilities that the input signal belongs to the classes represented by output layer cells.

B. Output Probabilities Mode

In this mode, the response of an output cell is an approximation to $p(C|\underline{x})$, the conditional probability of class C, given input signal $\underline{x}$. To compute this response, each prototype cell uses a radially symmetric, decaying exponential function of the form

$$p_i = e^{-\sigma_i d_i} \qquad [12]$$

where σ_i, the prototype smoothing factor, controls the rate at which the term decays as a function of d_i, given by equation [1] as the distance between the input signal and the ith prototype. Each output cell then computes a weighted sum of the activations of the prototypes to which it is connected. In the case of the kth output cell, these are the prototypes associated with class C_k. In the activation sum, the activation of the ith prototype is weighted by the pattern count for that prototype, κ_i. Thus, the response of the output cell is given by

$$o_k = \sum_{P_i \in C_k} \kappa_i \, p_i \qquad [13]$$

The actual conditional probability $P(C_k|\underline{x})$ is computed by dividing o_k by N_f, a normalizing factor which is simply the sum of the activations of the output cells for all classes:

$$N_f = \sum_{k=1}^{N_c} o_k, \qquad [14]$$

where N_c is the number of output cells.

$$P(C_k \mid \underline{x}) = \frac{o_k}{N_f} \qquad [15]$$

C. RCE Network Responses on the Ni1000

To achieve very high operating speed targets and to satisfy the objective of a scalable pattern recognition architecture, certain design modifications were made to the implementation of the RCE network on the Ni1000 Recognition Accelerator chip.

For scalability, the Ni1000 implementation of the RCE's probability response mode requires that the final normalization of output cell responses (i.e., the computation of [14] and [15]) be done off-chip, by the host processor. This enables a pattern recognition task to be distributed among a number of Ni1000 processors, working in parallel. In such an application, each chip computes its output cell terms, o_k. Host logic computes the N_f term, based upon the sum of all output cell activities for all chips. In the computation of the class-specific output term, o_{class}, this logic may need to combine the output terms of different output

---

[5] When operating in this network response mode, the Ni1000 is designed to generate a list of classes represented among the "minimum-influence-field" prototypes that have been activated by the input signal. If no prototypes of any influence field size are active, this class list will be empty. In this case, host logic can produce a response of "Unidentified."

cells for different chips. (A given class can be represented by the kth output cell on one chip and the mth output cell on another.)

To enhance operating speed, the Ni1000 implementation of the RCE network uses a particular form of a decaying exponential activation function for the prototype cell layer that is more naturally supported in silicon. By implementing an exponential decay function in base 2 as opposed to base *e*, the Ni1000 avoids unnecessary and time-consuming computational overhead. Specifically, on the Ni1000, the expression [12] for prototype activation is replaced by the following:

$$p_i = 2^{-\sigma_i d_i} \qquad [16]$$

IV. Practical Guides to RCE Network Training and Use

Like any other neural network or statistical learning algorithm, the performance of the RCE network is dependent upon the nature of the problem, the effectiveness of the input signal representations (i.e., the feature set) and the choices made for values of the network internal parameters that govern training and output response generation.

A. Statistically Reliable Training Set

For pattern recognition systems to develop a good solution to a recognition problem, the training set must be chosen in a way that represents the problem. Training sets that are composed from unrealistic or biased sampling will not have the same statistics as real world data. These statistics determine, after all, the location of pattern class territories, and, in the case of overlapping classes, the relative probabilities for different classes in such territories. To the extent that the statistics (i.e., class distributions in feature space) of the training set do not accurately reflect the statistics of real-world data, the network performance on the training set will not be predictive of its performance on "live" data. In such cases, what the network learns from the training set will not allow it to perform well in the real world.

Nonetheless, in some cases, it is possible to convert the output probabilities of a network trained on one sample of data to those that should be produced when the network is applied to a second data sample whose statistics are different from that of the training set. This need arises in those problems in which some classes have extremely low probabilities of occurrence. In composing a training set, the more likely occurring pattern classes are undersampled in order to avoid creating training sets that are excessively large.

Suppose the true a priori probabilities of a set of classes are represented by $P(C_1), ..., P(C_N)$, while their a priori probabilities in the training set are given by $P'(C_1), ..., P'(C_N)$. If, in response to an input signal $\underline{x}$, the network produces a class probability $P(C_i|\underline{x})$ on the training set, the actual probability, in the context of the real world data, can be approximated by scaling $P(C_i|\underline{x})$ by the ratio $P(C_i)/P'(C_i)$. This is an approximation, and is useful only if the sampling of the training set has been random within each class of patterns.

B. Choice of Representation Features

The shape of the pattern class territory is very dependent on the selection of features chosen to characterize the input patterns. Omitting features from the representation that are critical in distinguishing one class of patterns from another will result in separable classes appearing as non-separable, overlapping territories. At the same time, the inclusion of features that have no relevance to a pattern recognition problem can result in class territories occupying larger volumes in the pattern space than they would otherwise, resulting in RCE networks with large numbers of prototype cells.

Even relevant features, if too "low-level," will result in a need for large numbers of training patterns because the class territories that arise may consist of a large number of disjoint regions scattered

throughout the pattern space. The RCE network is sensitive to the effects of too low-level a representation because its training does not generate new feature representations that can be used to re-engineer the feature space, rearranging pattern class territories. Within the feature space defined by the input signals, the RCE works to accomplish the best separation of input pattern classes and estimates of their class probabilities. A complex feature space class distribution that consists of numerous, unrelated, disjoint territories will result in a large number of prototype cells being committed by the RCE network.

In nearly all cases, an understanding of the problem domain creates the opportunity to engineer higher-level representations for presentation to the network. As an example, the best representations for a character recognition task are not pixel-based, but employ, instead, higher-level groupings of pixels that reflect some structures in the image (e.g., straight lines of different orientations, corners, intersections, etc.). Use of such higher-level representations can yield substantial benefits in terms of fewer prototypes committed, higher accuracies and better performance of the network in generalizing to correctly classify novel examples outside of the training set.

C. Choice of Values for RCE Network Parameters

The RCE parameters that most affect performance on a given problem are λ_{max}, λ_{min} and σ, the prototype smoothing factor.[6] In no case can choices be made for any of these values which would result in the RCE training failing to converge. The convergence guarantee is independent of the values chosen for λ_{max} and λ_{min}.[7] However, choices for these parameters can affect the numbers of prototypes committed during training and, to a lesser extent, recognition accuracy.

Smaller values for λ_{max} will result in the commitment of more prototypes, simply because the network will require more prototypes to cover pattern class territories. More specifically, if the size of λ_{max} is small relative to the average size of the class territories in the feature space, more prototypes will be required to solve the pattern recognition problem.

On the other hand, the effect of choosing larger values of λ_{max} will be a tendency by the system to generalize more aggressively when presented with novel input patterns. This effect will be most noticeable if the initial training set does not fully capture the statistics of the data that the network will process in the future. This effect will be particularly observable if the network is trained incrementally in the field through dynamic category addition, where, as often happens with in-field training, wholly new pattern classes are introduced at later times.

Similarly, choices for λ_{min} can have very noticeable effects on network performance. The larger the value for λ_{min}, relative to the average size of pattern class territories, the more likely that a separable pattern class problem will be treated as if it is non-separable. The smaller the value for λ_{min}, the more likely the network will commit a large number of prototypes for problems in which there are overlapping class territories that are large compared to the value chosen for λ_{min}. This is easy to see in the extreme case of choosing λ_{min} so small that only "point-value" influence fields are allowed. (The influence field is large enough to only contain the prototype weight vector.) In this case, training will cause a prototype to be committed for every distinct example of a pattern class contained in the overlap region.

Finally, although the value chosen for σ does not in any way affect the training of the RCE network (the training procedure is independent of σ), it can affect the probability responses generated by the network. As an example, choosing a value of $\sigma_i = 0$ for all prototypes (surely an extreme choice) would make the class probability estimate generated by the network independent of the actual value of the input signal. All

---

[6] The value of σ, the smoothing factor, is not used during RCE training.

[7] The values for λ_{max} and λ_{min} must be properly chosen; i.e., $\lambda_{max} \geq \lambda_{min} > 0$.

prototypes would have activations of value 1, and the output cell responses (computed as normalized probabilities) would simply be equal to the a priori probabilities of each class, as computed from the training data set.

In general, the further the input signal is from the prototype, the less that prototype should contribute to the estimate of the probability of the input signal's belonging to the given prototype's pattern class. As an example, choosing $\sigma_i = 1/\lambda_i$ ensures that the contribution of the ith prototype to an output cell's (unnormalized) probability response falls to a value of 1/e of its contribution for input signals at the edge of its influence field as compared to those signals that fall at its field center.

V. Applications of RCE to Pattern Recognition

As in other neural network systems for pattern recognition, the mechanics of RCE network training and classification are completely independent of the meaning of the input signals presented to the network. This makes the network applicable to a broad range of pattern recognition tasks. The network has been applied to a wide variety of pattern recognition problems, including character recognition, image recognition, and a range of decision-making tasks in the area of financial services. The following discussion highlights the approach to feature generation and network training taken in several of these applications.

A. Character Recognition

The network has been applied to the problem of recognizing unconstrained handwritten characters, described either by image-based information, as might be available from scanning devices, or by stroke-based information, as might be available from devices that capture handwriting information online. Different features are defined for these two different contexts.

In the case of image based character recognition, one set of possible feature values corresponds to the registration of feature templates positioned at different locations over an image box containing the pattern to be recognized.[10] As an example, if the pattern is represented by a grid, 256 x 256 pixels in dimension, templates can be defined that are 16 x 16 pixels, for line segments oriented at 0°, 30°, 45°, 60° and 90°. Additional templates can be defined for different corner combinations and intersection styles (T's, +'s, and X's). A given template is moved across the image to different sampled locations, and at each location, a function is computed which measures how well the template matches the pixel values in that area of the image. This produces a feature that measures the degree to which the given template is present at the particular image location. The set of all such feature values for each template at every sampled image location produces an input signal for the RCE network to use in classifying the image.

In the case of online character recognition, features can be defined that are based upon the sequence of points that occur as the pattern is being drawn. A stroke sequence of points can then be characterized by the magnitude of successive motions in the x and y directions, along with information on the rate of curvature change at various positions along the stroke. Such a representation will mean that very different feature input signals will be generated if a pattern is drawn using one sequence of strokes versus another. The RCE network will accommodate this by creating additional prototypes to learn these stroke variations.

B. Image Analysis Applications

One example of the application of the RCE network to an image analysis problem is vehicle detection on roadways.[11] Accurate, automatic vehicle detection systems can be used for a variety of traffic engineering applications, including queue length measurement and traffic disruption detection. A detection system must process a gray-scale image of a roadway view in order to determine the number of vehicles present in the scene.

The input features chosen for the problem convert the high-resolution gray-scale image provided by the video camera into a coarser representation for the neural network. An $a \times b$ pixel tile is defined whose

value is computed from the average of the pixel gray-scale values it contains. Converting the image from an $m \times n$ array of gray-scale pixels to an $p \times q$ array of tiles reduces the dimensionality of the data. (Tiles do not overlap.) The coarser representation of the image is still a low-level representation; it makes use of no features corresponding to structural primitives. Although this representation does not carry enough information to enable the network to solve the problem of vehicle identification (e.g., deciding the particular make of automobile in the scene), it preserves enough information for the comparatively less demanding task of vehicle detection. This image analysis application illustrates the point that the complexity of the pattern recognition problem influences the level of complexity required in the input feature set.

In a similar spirit, the Ni1000 has been applied to the problem of classifying a fingerprint image in terms of component orientation maps.[12] The maps, also known as ridge direction maps, are used in many fingerprint identification systems. The RCE network is trained to store in each of its prototype cells, a weight vector that corresponds to pre-generated templates for specific ridge directions. A fixed window is moved along the fingerprint image. To the center pixel of each window location, the network assigns the closest matching ridge classification. These local image classifications can then be provided as input to a second classification process to identify the fingerprint image.

C. Decision Making in Financial Services

Outside the realm of image data or general signal processing tasks, the RCE network has also been applied to pattern recognition problems that are at the heart of risk assessment problems in financial services. In particular, RCE networks have been trained to provide accept/decline decisions as made by underwriters on residential mortgage applications.[13] In this case, the features used to create input signals for the network are computed from information available on the loan application. Such applications contain information on the borrower (e.g., length of employment, salary, etc.), the borrower's credit history (number of trade lines open, number of foreclosures, number of times 30 days late, 60 days late, etc.) and various ratios that underwriters consider in making their decision (loan to value ratio, etc.). By training on samples of accepted and declined mortgages, the network can learn to emulate the quality of decision-making capabilities of the mortgage underwriters as reflected in the training data set.

Another application in the financial services area involves the use of the network to detect fraudulent activity in credit card usage.[14,15] Here the network is used to assign to each credit card transaction a score that reflects the likelihood of the transaction's being fraudulent. The input features that characterize a transaction are defined from characteristics of the transaction itself (amount of the transaction, location of the transaction, type of goods or service being purchased), the characteristics of recent purchase activity on the card (number of transactions made in the past several days, weeks and months, average dollar value of purchases in the past several days, weeks or months, etc.) as well as general information available about the cardholder's account (amount of available credit, how long the account has been open, elapsed time since a new credit card was mailed to the customer, etc.) All of these features, taken together, provide a picture of the current transaction in the context of the normal use of the card. By presenting the network with examples of both good and fraudulent transactions, each characterized by these features, the network is able to learn to identify a significant portion of fraudulent activity.

The above examples illustrate the broad applicability of the RCE network to pattern recognition problems. The principal requirements for applying the network to such problems are (1) access to a reliably labeled training data set of sufficient examples and (2) a means of characterizing data examples in terms of features that are relevant and appropriate for the problem domain.

VI. RCE Network on a Commercially Available Neural Network Chip

As is the case with other neural networks, there is a high degree of parallelism in the computations that the RCE network performs for both training and pattern classification. Recently, a special purpose neural network chip, the Ni1000 Recognition Accelerator, has been designed and developed to implement in truly parallel fashion many of the operations performed by the RCE network and other networks of similar structure.[8] Significantly, the Ni1000 has been designed so as to perform not only RCE recognition but also RCE training operations with on-chip logic.[16] This makes it ideally suited for applications that require rapid, real-time, in-field trainability.

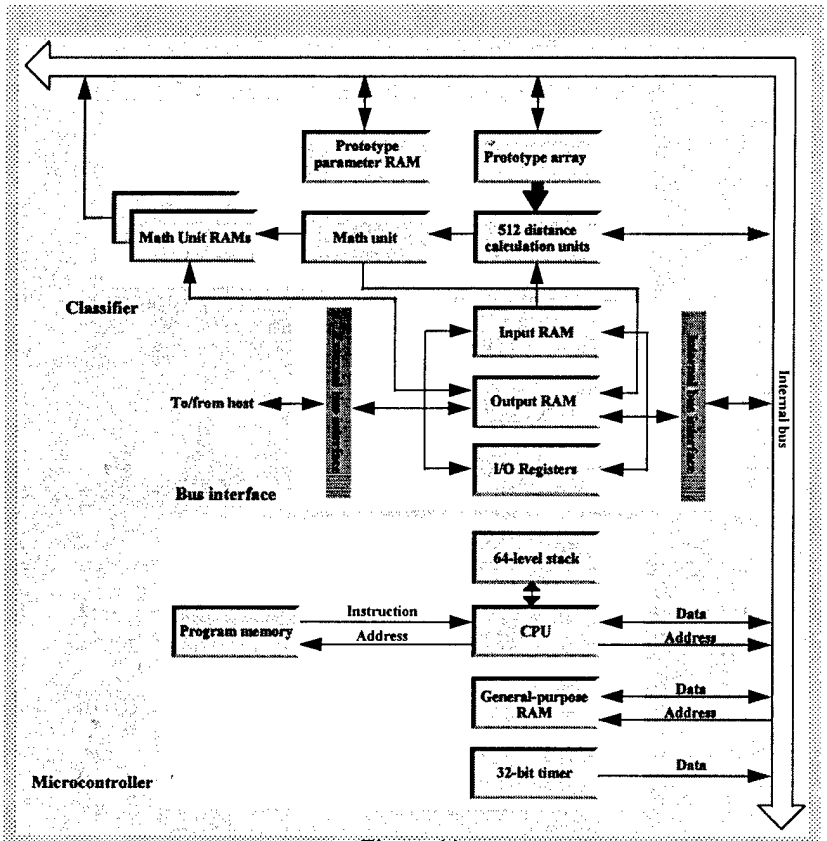


Figure 11
Block diagram of Ni1000 Recognition Accelerator Chip

The Ni1000 Recognition Accelerator supports classification of over 32,000 patterns per second, with real-time adaptation. The chip is compatible with commonly used radial basis function paradigms, including RCE and PNN networks. The Ni1000 is designed to accept input vectors with a maximum of 256 features, each with 32 levels of resolution, and produces up to 64 classes and/or probabilities. High-speed parallel processing units compute the city-block distance between an input vector and up to 1000 stored prototypes. A block diagram of the Ni1000 Recognition Accelerator appears in Figure 11. The on-chip, custom, 16-bit microcontroller has separate program and data memories. The 4K x 16-bit non-volatile FLASH EPROM memory can hold training algorithms, chip maintenance utilities and other software required by the application. A general purpose 256 x 16-bit RAM is also available to the microcontroller.

The microcontroller can enable an automatic classification mode in which a series of logic blocks, arranged as a pipeline, process data and output results to a host. The classification pipeline consists of input buffers, distance calculation units, a large FLASH prototype array that stores the results from the training process, a mathematical unit and its output memories, and an output buffer. At 33 MHz, the pipeline can classify over 32,000 input vectors per second, in which each input vector has up to 256 features with 5-bit resolution for each feature. The performance is made possible by the Ni1000 parallel architecture, which executes up to 16.5 billion operations per second. A typical Von Neumann machine would need to execute more than 65 billion instructions per second to approach the processing rate achieved by the Ni1000 Recognition Accelerator.

---

[8] The PNN network, as well as other radial basis function networks can also be implemented by the chip.

The Ni1000 makes it practical to apply the RCE network to numerous pattern classification tasks that have extremely high throughput requirements, or that require real-time or near real-time performance.

Acknowledgments

The author gratefully acknowledges the very careful editorial review and helpful suggestions made by Linda Mensinger Nunez, Michael Glier, Mark Laird and Christopher Bray in the preparation of this paper.

VII. References

[1] L. N. Cooper, C. Elbaum and D. L. Reilly, "Self-organizing general pattern class separator and identifier," U.S. Patent No. 4,326,259, Apr. 1982.

[2] D. L. Reilly, L. N. Cooper and C. Elbaum, "A neural model for category learning," *Biol. Cybern.*, vol. 45, 1982, pp. 35-41.

[3] C. L. Scofield, D. L. Reilly, C. Elbaum and L. N. Cooper, "Pattern class degeneracy in an unrestricted storage density memory," in Neural Information Processing Systems, Denver, CO, 1987, ed. D. Z. Anderson, American Institute of Physics, New York, NY, 1988, pp. 674-682.

[4] J. Moody and C. Darken, " Learning with localized receptive fields," in *Proc. of the 1988 Connectionist Models Summer School*, D.S. Touretzky, G.E. Hinton and T.J. Sejnowski, eds., Morgan Kaufman Publishers, San Mateo, CA, 1989, pp. 133-143.

[5] P. J. Werbos, "Beyond regression: new tools for prediction and analysis in the behavioral sciences," Ph.D. dissertation, Harvard University, 1974.

[6] D. E. Rumelhart, G. E. Hinton and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 332, 1986, pp. 533-536.

[7] B. Irie and S. Miyake, "Capabilities of three-layered perceptrons," in *Proc. IEEE Second Int. Conf. on Neural Networks*, San Diego, CA, July 1988, vol. I, pp. 641-648.

[8] D. L. Reilly and L. N. Cooper, "An overview of neural networks: early models to real world systems," in An Introduction to Neural and Electronic Networks, ed. S. F. Zornetzer, J. L. Davis and C. Lau, 1990, Academic Press, pp. 227-248.

[9] D. F. Specht, "Probabilistic neural networks for classification, mapping, or associative memory," in *Proc. IEEE Second Int. Conf. on Neural Networks*, San Diego, CA, July 1988, vol. I, pp. 525-532.

[10] C. L. Scofield, L. Kenton and J.-C. Chang, Multiple neural net architectures for character recognition, in *COMPCON Spring '91 Digest of Papers*, pg. 487-491.

[11] D. Bullock, J. Garrett, Jr., C. Hendrickson, "A neural network for image-based vehicle detection," in *Transportation Research - C*, vol. 1, no. 3, 1993, pp. 235-247.

[12] A. Shmurun, V. Bjorn, S. Tam and M. Holler, "Extraction of fingerprint orientation maps using a radial basis function recognition accelerator," in *Proceedings of WCCI*, Orlando, Fla., June, 1994, pp. 1186-1190.

[13] E. Collins, S. Ghosh and C. L. Scofield, "An application of a multiple neural network learning system to emulation of mortgage underwriting judgments," in Neural Networks in Finance and Investing, ed. R. T. Trippi and E. Turban, Probus Publishing Company, 1993, pp. 305-311.

[14] S. Ghosh and D. L. Reilly, "Credit card fraud detection with a neural network," in *Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, IEEE Computer Society, Wailea, Hawaii, January 1994 , pp. 621-630.

[15] D. L. Reilly, "Neural network fraud control in the bank card industry", to appear in Artificial Intelligence in the Capital Markets, ed. R. Freedman, R. Klein and J. Lederman, Probus Publishing Company, 1995.

[16] M. Holler, C. Park, J. Diamond, U. Santoni, S. C. The, M. Glier and C. L. Scofield, "A high performance adaptive classifier using radial basis functions," in *Proceedings of Government Microcircuit Applications Conference*, 1992, pp. 261-264.

Appendix C
Implementing Neural Networks Using The Ni1000

Implementing Neural Networks Using The Ni1000

Ni1000 Development System Release 2.0

Nestor, Inc.®

Ni1000™
RECOGNITION ACCELERATOR™

Copyright

Rev 2.0 October, 1995 - Nestor Document No PC-IN-Ni1000-1195

(Copyright Nestor Incorporated, 1995. All rights reserved. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language without the written permission of Nestor, Inc.

US GOVERNMENT RESTRICTED RIGHTS. This computer software and documentation are provided with RESTRICTED RIGHTS. Use, duplication or disclosure by the Government is subject to restrictions as set forth in the governing Rights in Technical Data and Computer Software clause - subdivision (b)(3)(B) of DAR 7-104.9(a) (May 1981) or subdivision (b)(3) of DOD FAR Supplemental 252.227-7013 (May 1981). Contractor/Manufacturer is Nestor, Inc. of One Richmond Square, Providence, RI 02906 USA.

The information in this document is subject to change without notice and does not represent a commitment on the part of Nestor, Inc.

Trademarks

The Nestor logo, Ni1000 and NestorACCESS are trademarks of Nestor, Incorporated in the United States and other countries. (Ni1000 Microcontroller and Ni1000 Emulator are not trademarked)

MS-DOS, Windows 3.1 , Visual C++ and Excel are the trademarks of Microsoft Corporation.

Every effort has been made to treat trademarks and service marks in accordance with the United States Trademark Commission's guidelines. Any omissions are unintentional and should not be regarded as affecting the validity of any trademark or service mark.

Communications

Nestor, Inc.
One Richmond Square
Providence, RI 02906

For customer service and technical support please call (401) 331-9640 Monday through Friday, 9am -5pm EST or EDT. Facsimile: (401) 331-7319. Compuserve: 76476,1072. Nestor Bulletin Board: (401) 331-2160. Also, see the Customer Support Appendix in the NestorACCESS User's Guide.

Nestor, Inc., founded in 1983, is the world leader in providing neural network-based systems for pattern recognition applications.

Printed in the United States of America
First Printing November, 1995

TABLE OF CONTENTS

1. ABOUT THIS BOOK 7

1.1 Structure Of This Book 7

1.2 Use Of Typography In This Book 7

1.3 Location Of Samples Code 7

2. NI1000 RECOGNITION ACCELERATOR OVERVIEW 9

3. NI1000 EXTERNAL INTERFACE DETAILS 13

3.1 IRAM 13
3.1.1 Vector Restrictions 13

3.2 ORAM 14
3.2.1 Probabilistic Results 14
3.2.2 Deterministic Results 14
3.2.3 Other Results 15

3.3 I/O REGISTERS 15
3.3.1 CMR 16
3.3.2 DIM 16
3.3.3 IDR 17
3.3.4 SSR 17
3.3.5 XIR 17
3.3.6 OP0 - OP5 17

4. NI1000 STANDARD MICROCODE 19

4.1 Microcode Communication Protocol 19

4.2 Ni1000 Standard Microcode Commands 20
4.2.1 READCONFIG 20
4.2.2 CLASSMODE 20
4.2.3 SETCLOCK 21
4.2.4 COLUMNREAD 21
4.2.5 COLUMNWRITE 21
4.2.6 PPRAMREAD 22
4.2.7 PPRAMWRITE 23
4.2.8 RAMREAD 23
4.2.9 RAMWRITE 23
4.2.10 LEARNBEGIN 24
4.2.11 LEARNVECTOR 24
4.2.12 LEARNEPOCH 25
4.2.13 LEARNEND 25

1. About This Book

This book describes the software implementation of Radial Basis Function neural networks using the Ni1000 Recognition Accelerator on either the ISA1000 or the PCI4000 board.

Before reading this book, you should have read and understood:

- *Radial Basis Function Neural Networks,* in the binder labeled Ni1000 Development System For Windows, and
- *The RCE Neural Network,* the document which immediately precedes this on in this binder.

1.1 Structure Of This Book

Chapter 2 gives an overview of the Ni1000 Recognition Accelerator.

Chapter 3 gives an abstract of information found in the *Ni1000 User's Guide,* focusing exclusively on the software interface presented.

Chapter 4 describes the interface between the software and the Ni1000 standard microcode. All commands issued by software eventually get carried out by the microprogram. The connection between these commands and the functions detailed in the *Ni1000 Libraries* document is described as well.

Chapter 5 illustrates common calling sequences used to accomplish various neural network functions on a single Ni100

Chapter 6 extends the concepts in chapter 5 to the multiple chip case.

Chapter 7 discusses some methods of post-training manipulation of network parameters which can be useful in performance tuning.

1.2 Use Of Typography In This Book

- Arial font is used to highlight references to software functions and for example pieces of software.
- *Italics* are used when new technical terms are introduced.

1.3 Location Of Samples Code

This book contains lots of sample code. In addition to these samples, working samples are provided in electronic format, and can be found in the \<install_path_root>\samples directory.

2. Ni1000 Recognition Accelerator Overview

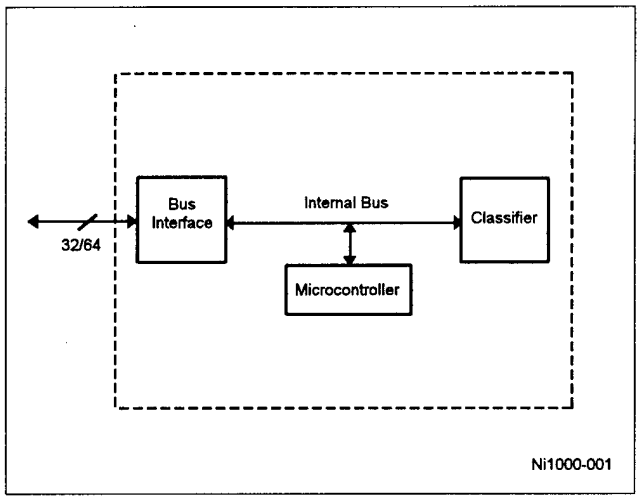


Figure 2-1. Block Diagram

The Ni1000 Accelerator supports classification of over 10,000 patterns per second, with real-time adaptation. The chip is compatible with commonly used Radial Basis Function (RBF) paradigms, including Restricted Coulomb Energy (RCE), Probabilistic RCE (PRCE), Probabilistic Neural Networks (PNN) and other algorithms. The flexible, on-chip microcontroller, with its 4K x 16-bit non-volatile microcode memory, also permits implementation of other custom algorithms.

The Accelerator accepts input vectors with a maximum of 222 features, each with 32 levels of resolution, and produces up to 64 class IDs and probabilities. High-speed parallel processing units compute the city-block distance between an input vector and up to 1000 stored prototypical examples. The Accelerator's high speed is suitable for computationally intensive applications like optical character recognition, fingerprint identification and industrial inspection.

Pattern recognition is the process of sorting input data into categories or classes that are significant to the user. Prototypical sets of values of differentiating traits (features) for each class must first be loaded into the chip's memory. The contents of the chip's memory can be developed manually or extracted from examples of data typical to the problem, using a learning algorithm. Feature sets are problem-specific and may consist partially or completely of stored data, such as historical records, or of direct sensor inputs. Once learning is complete, the system is ready to classify input data. The Ni1000 Recognition Accelerator supports incremental learning in the field, which may be necessary to further adapt the recognition system to its environment.

The Accelerator consists of two main parts: a classifier and a general-purpose 16-bit microcontroller. The classifier performs distance calculations between the input vector and a set of up to 1000 stored *prototypes* , using an array of 500 dedicated processors. Its outputs are firing class IDs and probability densities, with the latter calculated in six phases handled by a six-stage pipelined processor. The microcontroller implements on-chip learning algorithms and interacts with software running on the host.

Both the input and output stages of the classifier are equipped with alternating double buffers, so that the classifier does not stall during I/O. The host interface can be selected for 32- or 64-bit data bus width, and it supports a single-transfer-per-clock burst mode.

Figure 2-2 is a block diagram of the internal hardware architecture. The upper part of Figure 2-2 shows the classifier, the bottom part shows the microcontroller, and the middle part shows the interface to the host.

In an application environment, the classifier receives data from the host system through the bus interface, processes it, and sends the classification results back through the bus interface to the host. The classifier exploits both array and pipeline parallelism to perform over 10,000 classifications per second. The parallel hardware of the Distance Calculation Units and their tight coupling to the Prototype Array (PA) are responsible for much of this processing power. The Prototype Array holds 1024 (raw prototypes) x 256 (features) x 5 (bits per feature), for a total of 1.3 million non-volatile Flash storage bits. Note that the 1024x256 physical array provides 1000x222 storage locations usable for classification. Additional prototype storage is possible using multiple Ni1000 Accelerators and a higher effective input feature resolution is possible using two or more Ni1000 features to represent each feature.

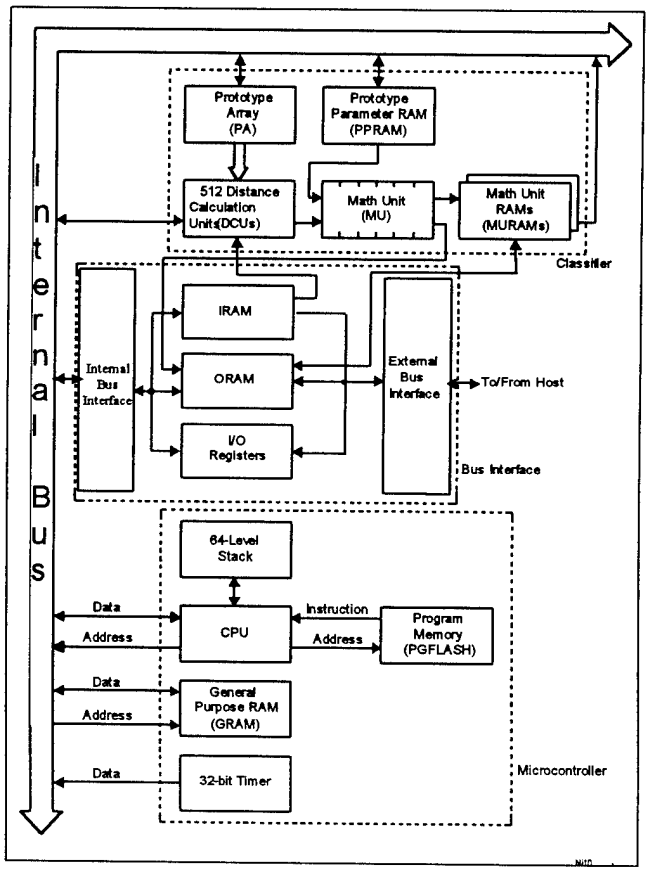


Figure 2-2. Internal Architecture

Each of the 512 parallel Distance Calculation Units calculates the city-block distance by summing the differences produced by their absolute value subtractors. The subtraction is performed on each feature of the input vector and the corresponding feature of one of the prototype vectors stored in the Prototype Array. The DCUs are multiplexed twice in time to achieve a sustainable processing rate of over 12 billion operations per second and a bandwidth of over 30 Gbps.

The classifier's Math Unit (MU) calculates probability densities and results classes concurrently. The MU uses a six-stage pipeline with a resolution of 16-bits for floating-point computations (10-bit mantissa and

6-bit exponent). It places results into one of two static RAMs. This double-buffering scheme allows the Math Unit to continue processing a second vector without interrupting the classification pipeline. The Prototype Parameter RAMs (PPRAMs) hold parameters like the radius (*r*), smoothing factor (*k*), and Count (*C*).

The bottom part of Figure 2-2 shows the microcontroller. It is a fully custom, 16-bit, Harvard-architecture microcontroller that supervises learning, performs chip maintenance tasks, and maintains communication with the host. It can also exchange interrupts with the host. The 4k x16-bit PGFLASH Flash memory stores the microcontroller programs. All memory devices are memory-mapped to the microcontroller's address space, with the exception of the microcontroller's program memory (PGFLASH). Other facilities available to the microcontroller include 256 words of general-purpose static RAM (GRAM) and a free-running 32-bit timer. Classification must stop while the microcontroller accesses these memories.

The microcontroller can enable an automatic classification mode in which a series of logic blocks, arranged as a pipeline, process data and output the results to the host. The classification pipeline consists of input buffers, distance calculation units, a large FLASH prototype array that stores the results from the learning process, a mathematical unit and its output memories, and output buffer. At 25MHz, the pipeline can classify over 10,000 input vectors per second, in which each input vector has up to 222 5-bit features. The performance is made possible by the Ni1000 parallel architecture, which can execute over 12 billion operations per second. A typical Von Neumann machine would need to execute more than 40 billion instructions per second to approach the processing rate achieved by the Ni1000 Recognition Accelerator.

The middle part of Figure 2-2 shows the interface to the host, which consists of input buffers (IRAM), an output buffer (ORAM), and sixteen I/O control registers. The external data bus can be either 32 or 64 bits wide and will perform single-clock burst transfers. The input stage buffers two full-sized vectors. The outputs can be either in IEEE standard 32-bit floating-point format or the internal 16-bit floating-point format. Both the host and the Accelerator's on-chip microcontroller can access the sixteen 16-bit I/O control registers. The registers contain various control parameters for the Accelerator and provide a general channel for communication between the microcontroller and the host.

3. Ni1000 External Interface Details

The Ni1000 operates in two basic modes, *training* and *classification*. During training, *feature vectors* are taken into the chip and a neural network is created. The training is accomplished under the supervision of the Ni1000's internal microprogram. During training, some of the feature vectors are saved as *prototypes*, and various network parameters are adjusted for later use during classification. During classification, feature vectors are taken into the chip, compared against the saved prototypes, and *probabilistic* and/or *deterministic classification* results are produced. The Ni1000 microprogram is not involved in classification.

The Ni1000's interfaces to the outside world to accomplish all of these functions are the IRAM, which accepts feature vectors as input, the ORAM, where the probabilistic and/or deterministic results are deposited as output, and the set of 16 I/O registers, which provide a way to pass information back and forth between software and the Ni1000's microprogram. A short, somewhat more detailed explanation of these pieces will help you better understand the use of the parameters in the various Ni1000 Libraries.

3.1 IRAM

The IRAM accepts up to 256 5-bit features, 4 features at a time on the 32-bit bus. The 5-bit features must appear in the 5 most significant bits of each byte. This is illustrated in Figure 3-1.

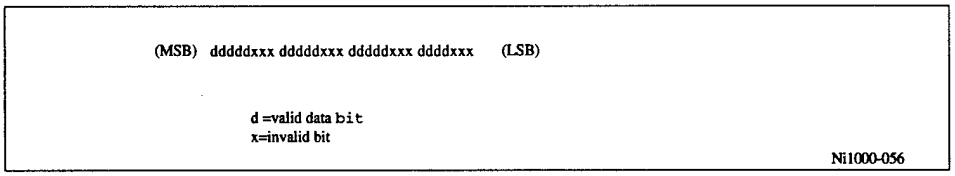


Figure 3-1. Data Alignment on 32-Bit External Bus

3.1.1 Vector Restrictions

The following restrictions are placed on feature vectors to insure reliable and accurate operation. These restrictions, taken as a set, reduce the usable number of features from 256 to 222.

3.1.1.1 Vector Padding

The chip does not calculate results reliably unless certain features are always 0. The Ni1000 library functions pad each input vector appropriately, using up 32 of the available 256 features.

3.1.1.2 Two Features Unusable

The last two features may not be used. No padded feature vector may be longer than 254 features.

3.2 ORAM

ORAM holds the results of a classification until read out of the Ni1000. For each vector, the chip calculates both probabilistic and deterministic results for up to 64 classes. The user has the option of reading one or the other or both.

3.2.1 Probabilistic Results

Probabilistic classification produces the *probability density* for each class. Section 5.1.5 of the *Ni1000 User's Guide* describes the possible formats of the output data. Although the Ni1000 provides two format options, the functions in the Ni1000 libraries always return probabilistic results in 32-bit IEEE standard floating point format. There will always be one result for each possible class for each vector, even if the probability density is 0. The number of results is $(n+1)$, where n is the highest class number in use. (See DIM, below)

3.2.2 Deterministic Results

Deterministic classification produces a *class firing list*. Unlike the probabilistic results, the number of results produced depends on the number of classes that *fire*, not the number of classes in the problem. The deterministic data result format is shown in Figure 3-2. A list entry is 1 byte long, and 4 of them may be read at a time.

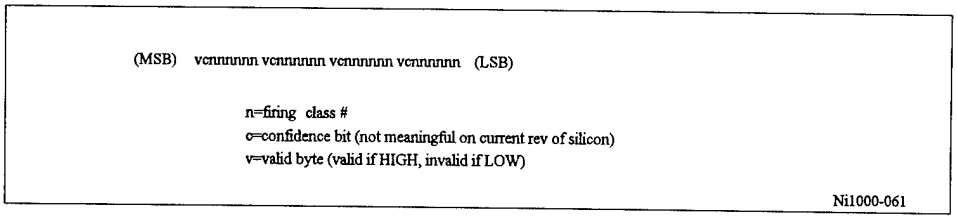


Figure 3-2. Format of Deterministic Results

3.2.2.1 Confidence Indicator Implementation

As shown in Figure 3-2, each entry in the class list has a confidence indicator. The intent of this bit is to indicate when the prototype(s) in the neural memory which caused the corresponding class number to fire is/are not valid for deterministic classification, which classifies a vector by answering the question, "Does this input feature vector fall inside a prototype's radius?" This can occur when, during training, the Ni1000 microcode wants to shrink the radius of a prototype but is prevented from doing so by the minimum radius parameter. As a result, the region within this prototype's radius does include a known conflict. When this occurs, the confidence indicator (or *probabilistic bit*) is set to indicate that probabilistic classification should be used to resolve such possible conflicts. A prototype with the probabilistic bit set is called a *probabilistic prototype*.

The *probabilistic* bits are collected from the on-chip network in the order that prototypes are processed. Thus, for the bits to be valid in the ORAM, it is necessary to ensure that all *deterministic prototypes* are processed before any probabilistic prototypes for a given class. This requires external software intervention after on-chip training has been completed. Complete details are available from Nestor.

3.2.3 Other Results

The contents of a PA column can be accessed and read out of the chip through the ORAM. The features are returned one feature per byte, with the 5-bit feature in the 5 most significant bits of each byte. This is identical to the IRAM load format shown in Figure 3-1.

3.3 I/O REGISTERS

A set of 16 I/O registers are available for sending commands, passing parameters, and reading status. Table 3-1 shows the full register set. Each register is 16 bits wide. The interesting subset (from an applications point of view) of these registers is discussed below.

Address (Hex)	Name	Description
0000	CMR	Chip Mode Register.
0008	DIM	Vector Dimension Register.
0010	IDR	Chip ID Register.
0018	SSR	Software Status Register.
0020	HS1	Hardware Status Register 1.
0028	HS2	Hardware Status Register 2.
0030	XIR	External Interrupt Register.
0038	IIR	Internal Interrupt Register.
0040	CRA	Control Register A.
0048	CRB	Control Register B.
0050	OP0	General-purpose operand register 0.
0058	OP1	General-purpose operand register 1.
0060	OP2	General-purpose operand register 2.
0068	OP3	General-purpose operand register 3.
0070	OP4	General-purpose operand register 4.
0078	OP5	General-purpose operand register 5.

Table 3-1. I/O Register Map

3.3.1 CMR

The Chip Mode Register (CMR) format is shown in Figure 3-3. At power up and/or chip reset, bit 15 is set to 1 automatically. This prevents execution of the Ni1000's microprogram. A 0 must be written to this register as part of the initialization sequence to allow the microprogram to execute. After initialization, this register is used to send commands to the microprogram.

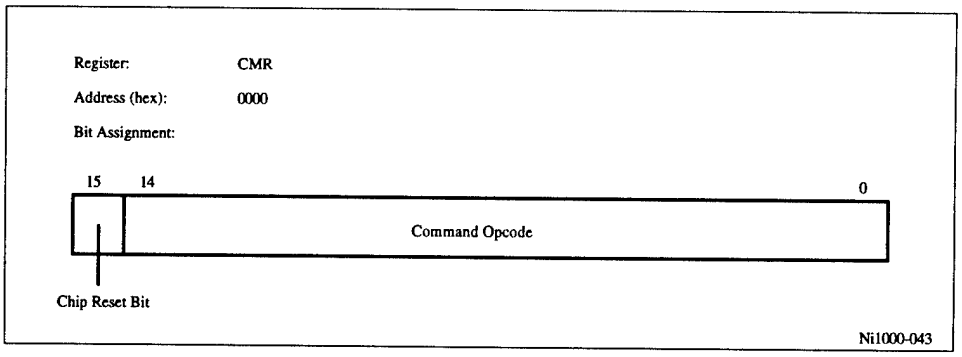


Figure 3-3. The CMR Register

3.3.2 DIM

The DIMension register tells the Ni1000 how many features (or dimensions) are to be expected during certain operations, and is shown in Figure 3-4.

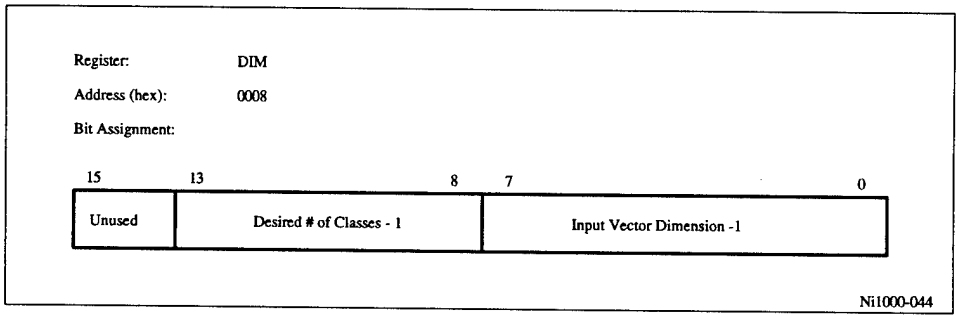


Figure 3-4. The DIM Register

Bits[7:0] indicate the number of features in the input vector minus 1. THIS SHOULD BE THE PADDED. IT DOES NOT INCLUDE EXTRA 0's PROVIDED TO FILL OUT 32-BIT WORDS. The IRAM recognizes the end of one vector and the beginning of another by counting the number of features written until the total equals this number.

Bits[13:8] generally indicate the highest class number minus 1 on the chip after training, which can be read to determine how much space must be allowed for classification results. These bits are occasionally used in other ways, and these will be discussed when the situation arises.

3.3.3 IDR

The ID Register (IDR) contains a constant which is used to uniquely identify the version of the Ni1000 chip, as shown in Figure 3-5. This document describes Ni1000s with ID 315B (hex).

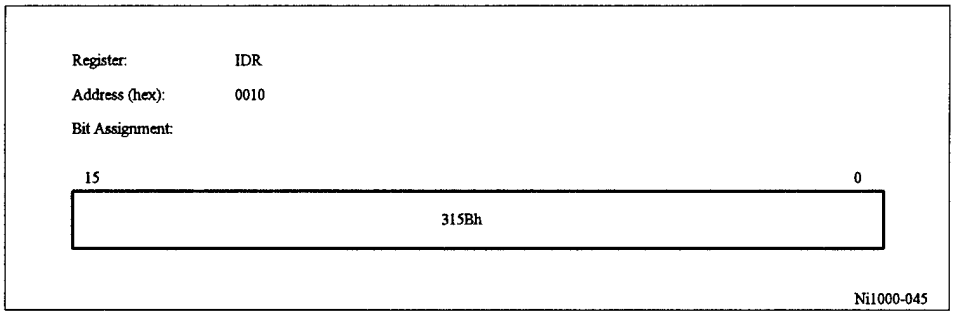


Figure 3-5. The IDR Register

3.3.4 SSR

The Software Status Register (SSR) is a 16-bit general purpose register. None of the bits has a predefined function in the Ni1000 hardware, but certain bits have been defined for use with the Ni1000 Standard Microcode. That discussion will be deferred until Chapter 4.

3.3.5 XIR

The eXternal Interrupt Register (XIR) is a 16-bit general purpose register. None of the bits has a predefined function in the Ni1000 hardware, but certain bits have been defined for use with the Ni1000 Standard Microcode. That discussion will be deferred until Chapter 4.

3.3.6 OP0 - OP5

The 6 OPerand registers (OP#) are 16-bit general purpose registers. None of the registers has a predefined function in the Ni1000 hardware, but certain registers have been defined for use with the Ni1000 Standard Microcode. That discussion will be deferred until Chapter 4.

4. Ni1000 Standard Microcode

Each Ni1000 is shipped with the Ni1000 Standard Microcode already loaded in the non-volatile control store. This microprogram handles on-chip maintenance, supervises training, etc. An application communicates with the microcode by sending it commands. The protocol for issuing a command and retrieving the results is handled entirely by the Ni1000 library routines, so this section will focus on the application's view of the commands implemented by this microcode. Each of the key commands (again, from the application programmer's viewpoint) are given in Table 4-1, and described briefly in section 4.2.

Opcode	Ni1000 Command Name	Command Function
00	READCONFIG	Read configuration information
01	CLASSMODE	Setup for classification
02	SETCLOCK	Tell microprogram the clock frequency
04	COLUMNREAD	Read the contents of a PA column
05	COLUMNWRITE	Write a PA column with specified data
08	PPRAMREAD	Read the network parameters of a prototype
09	PPRAMWRITE	Write the network parameters of a prototype
0A	RAMREAD	Read a location in the Ni1000 memory map
0B	RAMWRITE	Write a location in the Ni1000 memory map
0C	LEARNBEGIN	Setup for training
0D	LEARNVECTOR	Present a training vector
0E	LEARNEPOCH	Indicate the end of one pass through the training data
0F	LEARNEND	Indicate the end of training

Table 4-1. "Key" Ni1000 Standard Microcode Commands

DISCLAIMER: The entire contents of this chapter are specific to the Ni1000 Standard Microcode. If you customize the microcode, this information will no longer be accurate. Information and training on creating customized microcode is available from Nestor.

4.1 Microcode Communication Protocol

The Ni1000 command is sent to the microprogram by writing the opcode into the CMR register. Any operands are written into the OP# registers, or, if a feature vector is required, into IRAM, prior to writing the command into CMR. Data may be returned in the OP# registers as well, or, in some cases, in ORAM. On completion of a command, the microprogram writes the XIR register with status as shown in Figure 4-1. A completed command echoes the opcode in XIR. In general, applications programmers will not receive error codes directly from the microprogram. Any such errors are handled by the Ni1000 library. Details on the error codes, if they are encountered, are found in the *Ni1000 User's Guide*, section 7.4.1. More information, if required, can be obtained from Nestor.

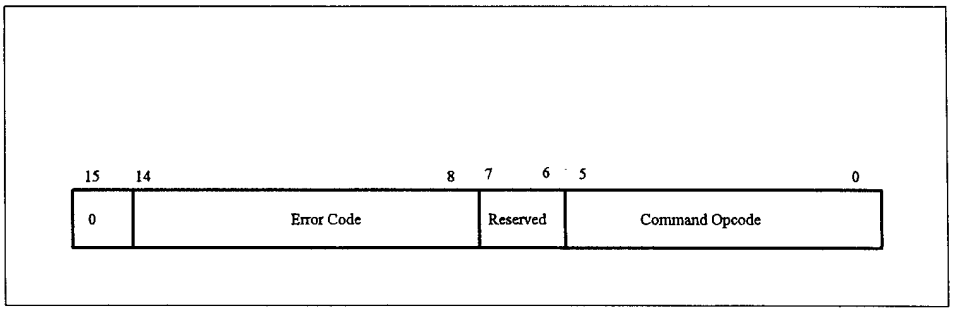


Figure 4-1. XIR Format On Command Completion

4.2 Ni1000 Standard Microcode Commands

4.2.1 READCONFIG

CMR: 00
Inputs: none
Outputs: XIR = 0 or error code
DIM[13:8] = (# returned bytes/2) -1
ORAM = configuration data

This command reads current configuration data out of the chip. A total of 36 bytes are currently defined, but the actual number of bytes is returned in DIM to allow for future expansion. The information returned is given in Table 4-2.

Word #	Information Returned
0	Microcode ID
1	Microcode Revision Number
2	Currently undefined
3	Currently undefined
4	Paradigm used to train network (0=no network, 1=RCE/PRCE, 2=PNN)
5	Highest class number in the network
6	Number of prototypes in the network
7	Number of features in the network (padded number)
8	Maximum radius used during training of the network
9	Minimum radius used during training of the network
10	Sector number of the network (currently always 0)
11	Smoothing factor exponent offset
12-17	Currently undefined

Table 4-2. READCONFIG Returned Information

4.2.2 CLASSMODE

CMR: 01
Inputs: DIM[7:0] = Number of features (padded) in feature vectors
OP0 = 1
Outputs: XIR = 1 or error code
DIM[13:8] = highest class number - 1

This command must be issued before starting a series of classifications. It places the Ni1000 into classification mode, which takes control from the microprogram and passes it to an on-chip hardware controller to maximize performance. The microcode has no function during actual classifications.

The CLASSMODE command is issued by the ClassifyBegin function.

4.2.3 SETCLOCK

CMR: 02
Inputs: OP0 = Clock period in nanoseconds
Outputs: XIR = 2 or error code

The Ni1000 microprogram programs the non-volatile neural network memory during training. The technology requires programming pulses of a certain duration to work properly, and this command informs the microprogram of the external clock frequency so that it can time these pulses correctly. The command should be issued once during chip initialization.

The SETCLOCK command is issued by the InitializeChip function.

4.2.4 COLUMNREAD

CMR: 04
Inputs: OP0 = column #
OP1 = first row #
DIM[13:8] = (# of rows to read-1) / 2
OP2 = 0
Outputs: XIR = 4 or error code
ORAM = PA column data

This command reads the specified number of features from an arbitrary location (column, first row) into the ORAM. This command is useful for saving a trained memory to a file. The terms *column* and *row* refer to a physical location in the Prototype Array (PA). Column number and prototype number may differ at times.

The COLUMNREAD command is issued by the ReadColumn function.

4.2.5 COLUMNWRITE

CMR: 05
Inputs: OP0 = column #
OP1 = first row #
DIM[7:0] = number of features to write - 1
OP2 = 0
IRAM = input vector
Outputs: XIR = 5 or error code

This command writes the specified number of features starting at an arbitrary location (column, first row). The data to be written is placed in IRAM. This command is useful for restoring a memory from a file.

The COLUMNWRITE command is issued by the WriteColumn function.

4.2.6 PPRAMREAD

CMR: 08
Inputs: OP0 = column #
Outputs: XIR = 8 or error code
OP0 = PPRAM1 data
OP1 = PPRAM2 data
OP2 = PPRAM3 data

This command reads the neural network parameters that were computed during training for the specified column. The parameters are broken up into three 16-bit words and stored in three simultaneously accessed banks of RAM. The format of the PPRAM words is shown in Figure 4-2. This command is useful for extracting information when saving a trained memory to a file.

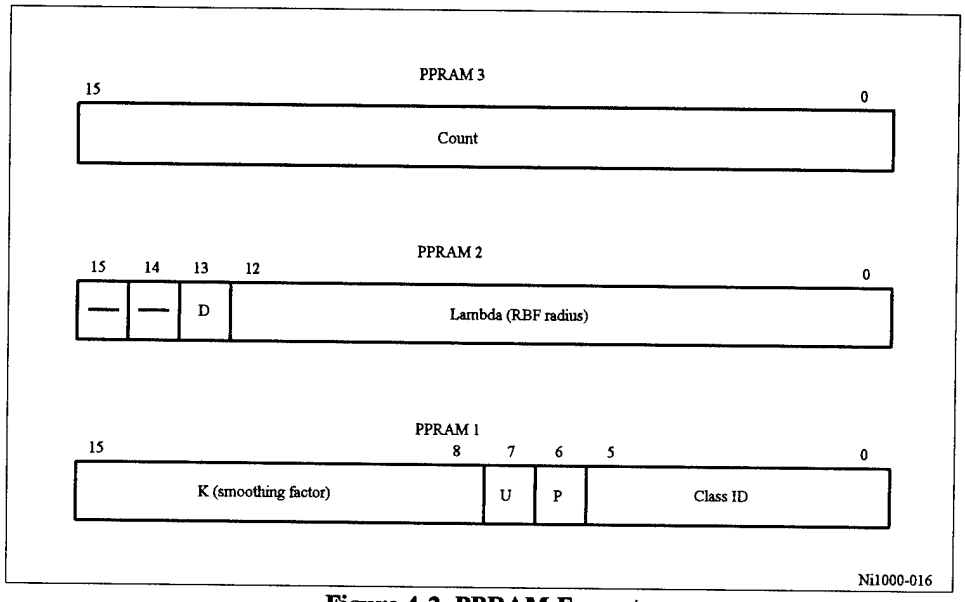


Figure 4-2. PPRAM Format

The fields of the PPRAM words are:

- Count - The number of training vectors that fell within this prototype's radius during the last epoch of training; used as a factor during classification when calculation probability density
- Disable Flag (D) - Set to disable this prototype
- Radius - The final radius produced by RCE/PRCE training
- Smoothing Factor - The decay constant which governs how quickly probability density decreases as the distance from the prototype increases; Used during probabilistic classification
- "Used" Flag - Set when there is a valid prototype
- Probabilistic Flag (P) - Set to indicate that the RCE/PRCE training algorithm wanted to shrink the radius further, but was prevented from doing so by the minimum radius training parameter; Used during classification to indicate that the prototype should only be used for probabilistic classification
- Class - The class designation for the prototype

The PPRAMREAD command is issued by the ReadProtoParamsEntry function.

4.2.7 PPRAMWRITE

CMR: 09
Inputs: OP0 = PPRAM1 data
OP1 = PPRAM2 data
OP2 = PPRAM3 data
OP3 = column #
Outputs: XIR = 9 or error code

This command writes the specified data into the neural network parameters for the specified column. Refer to Figure 4-2 for the PPRAM format. This command is useful for restoring a memory from a file.

The PPRAMWRITE command is issued by the WriteProtoParamsEntry function.

4.2.8 RAMREAD

CMR: 0A
Inputs: OP0 = address
Outputs: XIR = 0A or error code
OP5 = data

This command provides a way to read the value of a specific location in the Ni1000 memory map. While most of the mapped memory is accessible, the application programmer will want to confine himself to certain locations in the microprogram's scratch RAM (GRAM), which runs from location 1000 (hex) to location 1fff (hex). The entire memory map can be found in the *Ni1000 User's Guide*, Table 5-6.

The RAMREAD command is issued by the ReadMappedMemoryWord function.

4.2.9 RAMWRITE

CMR: 0B
Inputs: OP0 = address
OP1 = data
Outputs: XIR = 0B or error code

This command provides a way to write the value of a specific location in the Ni1000 memory map. While most of the mapped memory is accessible, the application programmer will want to confine himself to certain locations in the microprogram's scratch RAM (GRAM), which runs from location 1000 (hex) to location 1fff (hex). The entire memory map can be found in the *Ni1000 User's Guide*, Table 5-6. Note: The Ni1000 Standard Microcode blocks direct access to some locations.

The RAMWRITE command is issued by the WriteMappedMemoryWord function.

4.2.10 LEARNBEGIN

CMR: 0C
Inputs: OP0 = Training paradigm (1 = RCE/PRCE, 2 = PNN)
OP1 = Smoothing factor
OP2 = Minimum Radius
OP3 = Maximum Radius
DIM[7:0] = Number of features (padded) -1
Outputs: XIR = 0C or error code

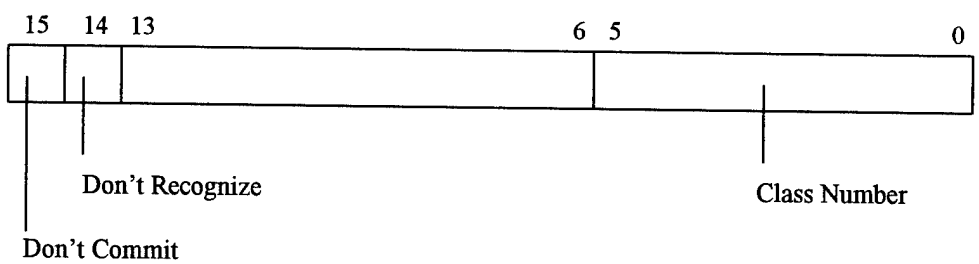
This command tells the microcode that training is about to begin. The previous network will be overwritten, and the training parameters given are saved away for use when the vectors arrive.

The LEARNBEGIN command is issued by the LearnBegin function.

4.2.11 LEARNVECTOR

CMR: 0D
Inputs: OP0 = class #
IRAM = input feature vector
Outputs: XIR = 0D or error code
SSR = various indicators, see discussion
OP4 = local minimum distance

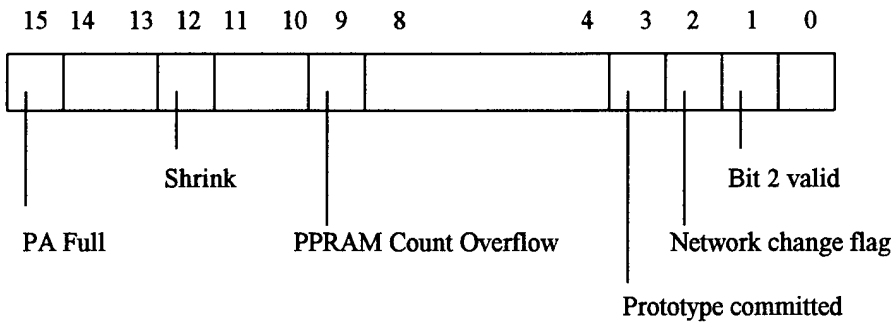
This command performs training with the given vector according to the paradigm and parameters set during the LEARNBEGIN command. The class number parameter is shown in Figure 5-3. The Don't Commit and Don't Shrink bits are only used during multiple chip training (see chapter 6), and should be 0 otherwise.



bit 15 = Don't Commit, in multiple chip training, instructs the microcode not to commit the input vector as a prototype, even if it wants to
bit 14 = Don't Recognize, in multiple chip training, makes a hole in the feature space corresponding to the feature vector
bit 5:0 = Class number of the feature vector

Figure 4-3. Class Number format for LEARNVECTOR

The SSR register bits of interest are shown in Figure 4-4.



bit 1 = 1 when bit 2 is valid
bit 2 = Network Change Flag, set to a 1 when the network has been changed during the current epoch
bit 3 = Set by LEARNVECTOR when it commits a prototype. If Don't Commit is set, this bit gets set if the microprogram *wanted* to commit a prototype.
bit 9 = Set during LEARNVECTOR if the PPRAM count field overflows
bit 12 = Shrink bit, set by LEARNVECTOR if a previously existing prototype was shrunk to accommodate a new prototype
bit 15= PA full, set by LEARNVECTOR if the PA is full, meaning it can't commit any more prototypes

Figure 4-4. SSR bits of interest during LEARNVECTOR

The local minimum distance is required for multiple chip training.

The LEARNVECTOR command is issued by the LearnVector function.

4.2.12 LEARNEPOCH

CMR: 0E
Inputs: none
Outputs: XIR = 0E or error code

This command indicates to the microprogram that the current pass through the training data (an *epoch*) has been completed, and that another pass is necessary. The microprogram resets the Network Change Flag and all the counts in PPRAM, so it must not be used at the end of the last epoch; use LEARNEND instead.

The LEARNEPOCH command is issued by the LearnEpoch function.

4.2.13 LEARNEND

CMR: 0F
Inputs: none
Outputs: XIR = 0F or error code

This command indicates to the microprogram that training is over. The microprogram exits training mode, saving all of the accumulated parameters.

The LEARNEND command is issued by the LearnEnd function.

Appendix D
Ni1000 Technical Specification

Ni1000 Technical Specification

NI1000 RECOGNITION ACCELERATOR

High-Speed Classification Engine for Pattern Recognition

- **Accurate Recognition**
  - Up to 222, 5-bit Inputs/Vector
  - Produces up to 64 Class Probability Estimations Using Weighted Sums of up to 1000 Radial-Basis Functions
  - Estimates Class Probabilities Even When Class Distributions Overlap
  - Parzen Windows Technique
  - Emulates Local Receptive Field Neural Network
- **Fast Recognition**
  - 200x Acceleration Over Typical Host PC
  - Scalable Acceleration with Multiple Ni1000 Chips
- **Learning**
  - RCE & PNN Learning On-Chip
  - Fast Learning: 1-5 Epochs
  - Incremental Learning Capability
  - Downloading Support Facilitates Off-Chip Learning and other Learning Paradigms
- **Memory**
  - 1000 Prototypes Representing up to 64 Classes Stored On-Chip
  - 222, 5-bit Features per Prototype
  - Non-Volatile Flash Prototype Storage Array
  - Scalable Prototype Storage with Multiple Ni1000 Chips
- **Sample Applications**
  - Hand- or Machine-Printed Character Recognition
  - Machine Vision
  - Medical Imaging
  - Fingerprint Recognition
  - Speech Recognition
  - Industrial Inspection
- **Pipelined Parallel Processing**
  - 512 Distance Calculation Units
  - Three Stage Classification Pipeline
  - Double-Buffered I/O RAMs
  - 16-Bit Floating-Point Math Unit
- **Microcontroller**
  - 16-Bit Microcontroller
  - 4K x 16-bit Flash Program Memory
  - RCE & PNN Learning Code Supplied
  - May be Programmed for Other Radial Basis Function Paradigms
- **Multichip Support**
  - Bus-Oriented Data Interface
- **x86-Compatible Interfacing**
  - 32- or 64-bit Data Bus
  - Maximum Data Transfer Rate of Over 200MB/sec
  - Digital CMOS/TTL Compatible
- **State-Of-The-Art Technology**
  - 0.8μ CHMOS-IV

Version	Clock Speed	Processing Time	Connections Per Second	Minimum* Classification Rate	Typical** Classification Rate
Ni1000-25	25MHz	96 μsec	2.7 Billion	> 10,000 patterns/sec	17,000 patterns/sec
Ni1000-10	10 MHz	240 usec	1.1 Billion	> 4,000 patterns/sec	7,000 patterns/sec

Rates assume unpipelined processing; continuously pipelined operation doubles performance.
* - 1000 prototypes, 222 features and 64 classes with 32-bit interface; deterministic classification.
** - 700 prototypes, 100 features and 32 classes. Other attributes unchanged.

Contents

1. GENERAL DESCRIPTION 1

2. PINOUT 6

2.1 PIN LIST 6
2.2 PIN CONFIGURATION 6

3. SIGNAL DESCRIPTIONS 8

4. ARCHITECTURE 11

4.2 BUS INTERFACE 14
4.3 THE CLASSIFIER 33
4.4 MICROCONTROLLER 50
4.5 SYSTEM-LEVEL ARCHITECTURE 60
4.6 CLASSIFICATION TIMING 61
4.7 COMPUTATIONAL PRECISION 62
4.8 SOFTWARE MODES AND CONFIGURATIONS 64

5. BUS OPERATIONS 67

5.1 HARDWARE MODES 69
5.2 BUS CYCLES 73

6. ELECTRICAL CHARACTERISTICS 81

6.1 ABSOLUTE MAXIMUM RATINGS 81
6.2 C.D. CHARACTERISTICS 81
6.3 A.C. CHARACTERISTICS 83

7. MECHANICAL AND THERMAL CHARACTERISTICS 87

8. GLOSSARY 88

9. BIBLIOGRAPHY 92

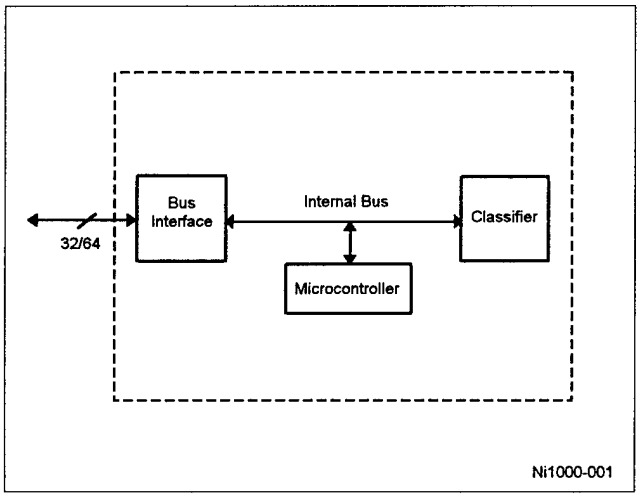


Figure 1-1. Block Diagram

1. General Description

The Ni1000 Accelerator supports classification of over 10,000 patterns per second, with real-time adaptation. The chip is compatible with commonly used Radial Basis Function (RBF) paradigms, including Restricted Coulomb Energy (RCE), Probabilistic RCE (PRCE), Probabilistic Neural Networks (PNN) and other algorithms. The flexible, on-chip microcontroller, with its 4K x 16-bit non-volatile microcode memory, also permits implementation of custom algorithms.

The Accelerator accepts input vectors with a maximum of 222 features, each with 32 levels of resolution, and produces up to 64 class IDs and/or probabilities. High-speed parallel processing units compute the city-block distance between an input vector and up to 1000 stored prototypical examples. The Accelerator's high speed is suitable for computationally intensive applications like optical character recognition, fingerprint identification and industrial inspection.

Pattern recognition is the process of sorting input data into categories or classes that are significant to the user. Prototypical sets of values of differentiating traits (features) for each class must first be loaded into the chip's memory. The contents of the chip's memory can be developed manually or extracted from examples of data typical to the problem, using a learning algorithm. Feature sets are problem-specific and may consist partially or completely of stored data, such as historical records, or of direct sensor inputs. Once learning is complete, the system is ready to classify input data. The Ni1000 Recognition Accelerator supports incremental learning in the field, which may be necessary to further adapt the recognition system to its environment.

The Accelerator consists of two main parts: a classifier and a general-purpose 16-bit microcontroller. The classifier performs distance calculations between the input vector and a set of up to 1000 stored *prototypes* , using an array of 500 dedicated processors. Its outputs

are firing class IDs and probability densities, with the latter calculated in six phases handled by a six-stage pipelined processor. The microcontroller implements on-chip learning algorithms and interacts with software running on the host.

Both the input and output stages of the classifier are equipped with alternating double buffers, so that the classifier does not stall during I/O. The host interface can be selected for 32- or 64-bit data bus width, and it supports a single-transfer-per-clock burst mode.

Figure 1-2 is a block diagram of the internal hardware architecture. The upper part of Figure 1-2 shows the classifier, the bottom part shows the microcontroller, and the middle part shows the interface to the host.

In an application environment, the classifier receives data from the host system through the bus interface, processes it, and sends the classification results back through the bus interface to the host. The classifier exploits both array and pipeline parallelism to perform over 10,000 classifications per second. The parallel hardware of the Distance Calculation Units and their tight coupling to the Prototype Array (PA) are responsible for much of this processing power. The Prototype Array holds 1024 (raw prototypes) x 256 (feature values) x 5 (bits per feature), for a total of 1.3 million non-volatile Flash storage bits. Note that the 1024 x 256 physical array provides 1000 x 222 storage locations usable for classification. Additional prototype storage is possible using multiple Ni1000 Accelerators and a higher effective input feature resolution is possible using two or more Ni1000 features to represent each feature.

Each of the 512 parallel Distance Calculation Units calculates a city-block distance (see Chapter 2) by summing the differences produced by its absolute value subtractor. The subtraction is performed on each feature of the input vector and the corresponding feature of one of the prototype vectors stored in the Prototype Array. The DCUs are multiplexed twice in time to achieve a sustainable processing rate of over 12 billion operations per second and a bandwidth of over 30 Gbps.

The classifier's Math Unit (MU) calculates probability densities and results classes concurrently. The actual calculations performed are discussed in section 4.3.5.

The MU uses a six-stage pipeline with a resolution of 16-bits for floating-point computations (10-bit mantissa and 6-bit exponent). It places results into one of two static RAMs. This double-buffering scheme allows the Math Unit to continue processing a second vector without interrupting the classification pipeline. The Prototype Parameter RAMs (PPRAMs) hold parameters like the radius(r), smoothing factor (k), and Count(C), described in Section 4.3.3.

The bottom part of Figure 1-2 shows the microcontroller. It is a fully custom, 16-bit, Harvard-architecture microcontroller that supervises learning, performs chip maintenance tasks, and maintains communication with the host. It can also exchange interrupts with the host. The 4k x 16-bit PGFLASH Flash memory stores the microcontroller programs. All memory devices are memory-mapped to the microcontroller's address space, with the exception of the the microcontroller's program memory (PGFLASH). Other facilities available to the microcontroller include 256 words of general-purpose static RAM (GRAM) and a free-running 32-bit timer. Classification must stop while the microcontroller accesses these memories.

The microcontroller can enable an automatic classification mode in which a series of logic blocks, arranged as a pipeline, process data and output the results to the host. The classification pipeline consists of input buffers, distance calculation units, a large FLASH

prototype array that stores the results from the learning process, a mathematical unit and its output memories, and an output buffer. At 25MHz, the pipeline can classify over 10,000 input vectors per second, in which each input vector has up to 222 5-bit features. The performance is made possible by the Ni1000 parallel architecture, which can execute over 12 billion operations per second at 25 MHz. A typical Von Neumann machine would need to execute more than 40 billion instructions per second to approach the processing rate achieved by the Ni1000 Recognition Accelerator.

The middle part of Figure 1-2 shows the interface to the host, which consists of input buffers (IRAM), an output buffer (ORAM), and sixteen I/O control registers. The external data bus can be either 32 or 64 bits wide and will perform single-clock burst transfers. The input stage buffers two full-sized vectors. The outputs can be either in IEEE standard 32-bit floating-point format or the internal 16-bit floating-point format. Both the host and the Accelerator's on-chip microcontroller can access the sixteen 16-bit I/O control registers. The registers contain various control parameters for the Accelerator and provide a general channel for communication between the microcontroller and the host.

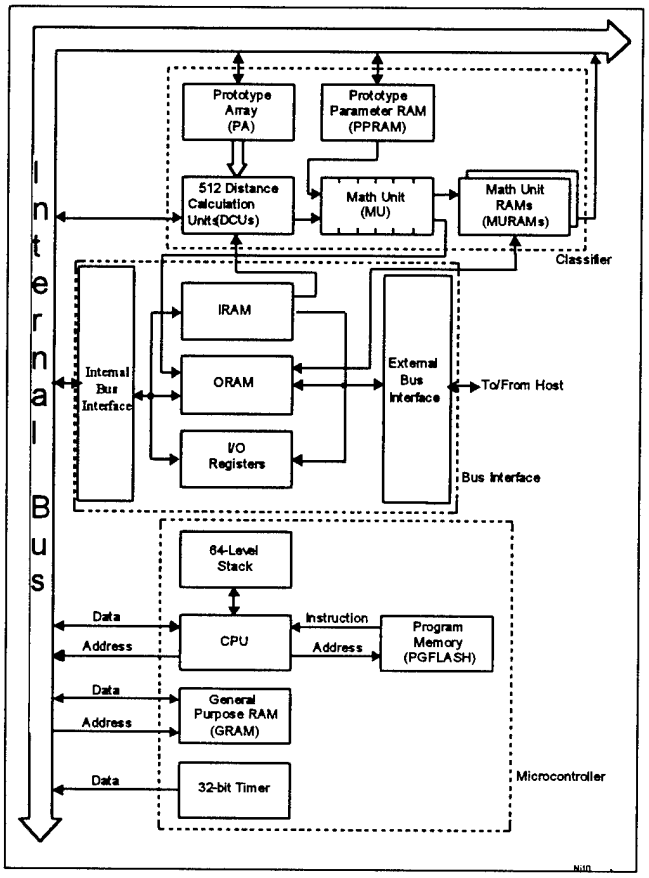


Figure 1-2. Internal Architecture

In most applications, the Accelerator will reside on a bus that is shared with a host CPU and perhaps other Ni1000 Accelerators, as shown in Figure 1-3. The Accelerator is a slave device on the host bus; it will not initiate data transfers on the bus. Both the host and the on-chip microcontroller have the ability to interrupt each other.

Figure 1-3 shows a local host CPU on an add-in card for personal computers and workstations. The CPU manages the flow of data to/from the Accelerator(s). The CPU may also have other functions, such as preprocessing data, interpreting classification results, or coordinating the operation of multiple chips. Other implementations may rely on the CPU of the system board for these functions.

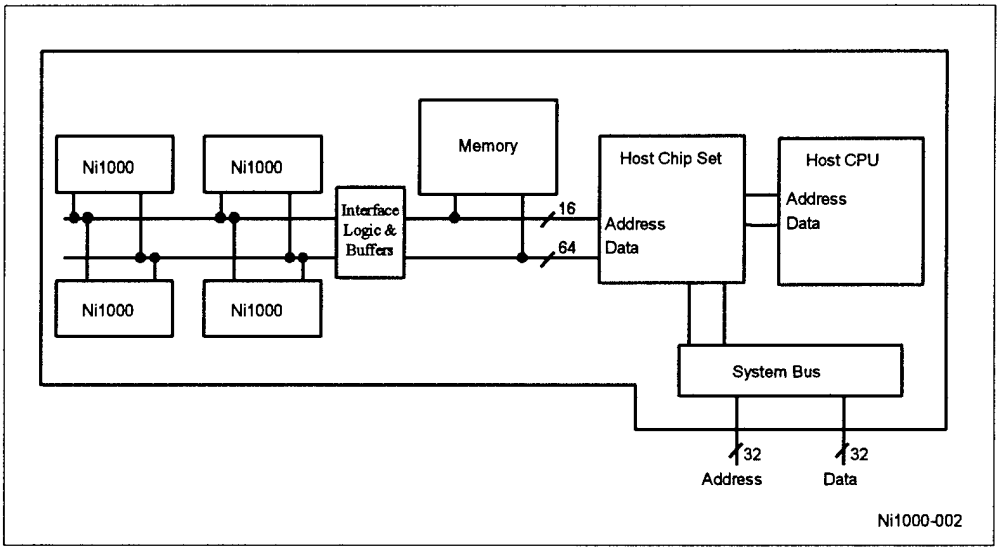


Figure 1-3. Multichip Add-In Board

2. Pinout

2.1 Pin List

Table 2-1. Pin List (Sorted By Pin Number)

Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal
A1	A3	C9	TEST#	J15	D15	Q10	D16
A2	A4	C10	64/32#	J16	Vcc	Q11	CLK
A3	A7	C11	MC#	J17	Vss	Q12	RDY#
A4	A9	C12	Vcc	K1	Vss	Q13	D43
A5	A11	C13	IACK#	K2	Vcc	Q14	Vss
A6	A13	C14	Vss	K3	D61	Q15	Vcc
A7	Vss	C15	Vcc	K15	D36	Q16	D9
A8	Vss	C16	D1	K16	Vcc	Q17	D41
A9	Vss	C17	D35	K17	Vss	R1	D24
A10	Vss	D1	A0	L1	D28	R2	D55
A11	SCRH#	D2	Vcc	L2	D60	R3	D22
A12	RESET#	D3	Vss	L3	D29	R4	Vcc
A13	MCINT#	D15	Vss	L15	D37	R5	Vss
A14	ERROR#	D16	Vcc	L16	Vcc	R6	Vcc
A15	D46	D17	D3	L17	Vss	R7	D50
A16	D47	E1	N/C	M1	Vcc	R8	Vcc
A17	D33	E2	Vss	M2	D27	R9	Vcc
B1	A2	E3	VCX	M3	D59	R10	Vcc
B2	A5	E15	BLAST#	M15	D7	R11	Vcc
B3	A8	E16	Vss	M16	D38	R12	BRDY#
B4	Vcc	E17	D4	M17	D6	R13	Vss
B5	Vss	F1	D63	N1	D26	R14	Vcc
B6	A15	F2	Vss	N2	Vss	R15	D12
B7	Vcc	F3	CMON#	N3	D58	R16	D11
B8	Vcc	F15	W/R#	N15	D40	R17	D42
B9	Vcc	F16	D2	N16	Vss	S1	D23
B10	Vcc	F17	Vcc	N17	D39	S2	D54
B11	MULTCHIP#	G1	Vss	P1	D57	S3	D53
B12	Vcc	G2	Vcc	P2	Vcc	S4	D52
B13	Vss	G3	D31	P3	Vss	S5	D51
B14	Vcc	G15	ADS#	P15	Vss	S6	D18
B15	D14	G16	D0	P16	Vcc	S7	D49
B16	D32	G17	D5	P17	D8	S8	Vss
B17	D34	H1	Vss	Q1	D25	S9	Vss
C1	A1	H2	Vcc	Q2	D56	S10	Vss
C2	A6	H3	D62	Q3	Vcc	S11	Vss
C3	Vcc	H15	CS#	Q4	Vss	S12	BERR#
C4	Vss	H16	Vcc	Q5	D21	S13	SRQ#
C5	A10	H17	Vss	Q6	D20	S14	D45
C6	A14	J1	Vss	Q7	D19	S15	D13
C7	A12	J2	Vcc	Q8	D17	S16	D44
C8	Vpp	J3	D30	Q9	D48	S17	D10

2.2 Pin Configuration

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	
S	D23	D54	D53	D52	D51	D18	D49	Vss	Vss	Vss	Vss	BERR#	SRQ#	D45	D13	D44	D10	S
R	D24	D55	D22	Vcc	Vss	Vcc	D50	Vcc	Vcc	Vcc	Vcc	BRDY#	Vss	Vcc	D12	D11	D42	R
Q	D25	D56	Vcc	Vss	D21	D20	D19	D17	D48	D16	CLK	RDY#	D43	Vss	Vcc	D9	D41	Q
P	D57	Vcc	Vss												Vss	Vcc	D8	P
N	D26	Vss	D58												D40	Vss	D39	N
M	Vcc	D27	D59												D7	D38	D6	M
L	D28	D60	D29												D37	Vcc	Vss	L
K	Vss	Vcc	D61												D36	Vcc	Vss	K
J	Vss	Vcc	D30					168-pin PGA Bottom (Pin-Side) View							D15	Vcc	Vss	J
H	Vss	Vcc	D62												CS#	Vcc	Vss	H
G	Vss	Vcc	D31												ADS#	D0	D5	G
F	D63	Vss	CMON#												W/R#	D2	Vcc	F
E	N/C	Vss	Vcx												BLAST#	Vss	D4	E
D	A0	Vcc	Vss												Vss	Vcc	D3	D
C	A1	A6	Vcc	Vss	A10	A14	A12	Vpp	TEST#	64/32#	MC#	Vcc	IACK#	Vss	Vcc	D1	D35	C
B	A2	A5	A8	Vcc	Vss	A15	Vcc	Vcc	Vcc	Vcc	MULT-CHIP#	Vcc	Vss	Vcc	D14	D32	D34	B
A	A3	A4	A7	A9	A11	A13	Vss	Vss	Vss	Vss	SCRH#	RESET#	MCINT#	ERROR#	D46	D47	D33	A
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	

Figure 2-1. Pin Configuration (Bottom View)

3. Signal Descriptions

Name	Type	Description
Clock, Address, and Data (Synchronous)		
CLK	I	**Clock.** This clock must be shared with or divided down from the bus clock, so that all bus transactions are synchronous with it.
A[0:15]	I	**Address**. Driven by the host to access the Accelerator's microcontroller program memory (PGFLASH), prototype-array memory (PA), prototype-parameter memory (PPRAM), and control and status registers. Detailed memory and register address maps are given in the Ni1000 Recognition Accelerator User's Guide.
D[0:63]	I/O	**Data**. As inputs, the host writes pattern vectors for classification, control information, and microcontroller programs on this bus. The inputs include 5-bit input vector components; 16-bit data, register contents, and microcontroller instructions; or 64-bit multiple-input vectors. As outputs, the host reads pattern classifications or probabilities, status information, and microcontroller-program verification. The outputs include 8-bit classes; 16-bit data, register contents, and microcontroller instructions; 32-bit IEEE standard floating point values; or 64-bit groups of classes or probabilities.
Bus-Cycle Definition and Control (Synchronous)		
ADS#	I	**Address Strobe**. When asserted by the host on a rising edge of CLK, this signal causes the Accelerator to sample CS# and the address on A[0:15], thereby initiating a bus cycle.
CS#	I	**Chip Select**. Asserted by the host to indicate that the Accelerator is being addressed. The signal must be held asserted throughout the bus cycle. The signal is used to select one of potentially multiple Ni1000 Accelerators.
BLAST#	I	**Burst Last**. When asserted by the host, this signal indicates the last data transfer in the current cycle, whether burst or non-burst. For burst cycles, the host holds BLAST# negated until the last data transfer of the cycle, during which it asserts BLAST#. For non-burst cycles, the host asserts BLAST# during the first (and only) data transfer. The signal is compatible with the x86 BLAST# architecture.
W/R#	I	**Write or Read**. Driven by the host on the same rising clock edge as ADS#, CS#, and BLAST#, to indicate that the current bus cycle is a write (high) or read (low).
RDY#	O	**Non-Burst Ready**. When asserted by the Accelerator, this signal indicates that the data on D[0:63] is valid (for output) or accepted (for input) and that it is the last data transfer in the current bus cycle. The signal terminates the bus cycle. For a burst cycle, RDY# is only asserted on the last transfer of the burst.

BRDY#	O	**Burst Ready**. When asserted by the Accelerator, this signal indicates that the data on D[0:63] is valid (for output) or accepted (for input) and that more data may be transferred in the current bus cycle. The signal does not terminate the current bus cycle and is not asserted on the last transfer of a burst; instead, RDY# is asserted.
BERR#	O	**Bus Error**. When asserted by the Accelerator, this signal indicates that bus is in an illegal state. For example, the host may attempt to write to the input buffer when the Accelerator is not in an appropriate mode or when the buffer is full, or the host may attempt to access the output buffer before data is available. The signal also terminates the current bus cycle. This signal is open collector.
64/32#	I	**64-Bit or 32-Bit Data Bus**. Driven by the host to select 64-bit (high) or 32-bit (low) operation on the D[0:63] bus.
MC#	I	**Microcontroller**. Asserted by the host on the same rising clock edge as ADS# and CS# to read or write the Accelerator's microcontroller-program memory (PGFLASH).
MULTCHIP#	I	**Multi-Chip Operation.** Asserted by the host or tied to ground when multiple Ni1000 Accelerator chips are to operate in parallel on the same address and data bus. A learning algorithm may sample this to alter behavior for multi-chip learning.

Interrupt Control (Asynchronous)		
SRQ#	O	**Service Request.** Asserted by the Accelerator's microcontroller to indicate that valid output is available on the data bus, an error has occurred, or asserted when the microcontroller writes to the XIR register, indicating that some action by the host is needed. It is held asserted until the host asserts the IACK# signal.
IACK#	I	**Interrupt Acknowledge.** Asserted by the host to acknowledge that it sampled the Accelerator's assertion of SRQ#.
MCINT#	I	**Microcontroller Interrupt.** Asserted by the host to force the Accelerator's microcontroller to jump to a specified program address.
ERROR#	I/O	**Error.** As an input, asserted by the host to interrupt to the microcontroller. On receiving an error, the Accelerator will determine the reason for the interrupt by reading the IIR register. As an output, asserted by the Accelerator to indicate that an internal error, such as data underflow or overflow in an I/O buffer. On receiving an error, the host should read the status register, XIR, to determine the nature of the error. This signal is open collector.
RESET#	I	**Reset.** Asserted by the host to halt and reinitialize the Accelerator. The Ni1000 remains in the reset state until, the host writes a 0 to bit 15 of CMR, whereupon the microcontroller begins executing instructions in NORMAL mode from location 1 in the PGFLASH (address F001h). The host can also reset the Accelerator by writing a 1 to bit 15 of the CMR register.
System and Power		
V_{cx}	P	**+5 Volt Memory Supply.** Used during normal operation by the prototype array (PA) and the microcontroller's program flash memory (PGFLASH).
V_{pp}	P	**+12 Volt Programming Supply.** Used during programming by the prototype array (PA) and the microcontroller's program flash memory (PGFLASH).
V_{cc}	P	**+5 Volt Supply.**
V_{ss}	P	**Ground.**

Type: I = Input, O = Output, P = Power or Ground.

4. Architecture

The internal hardware architecture was shown in Figure 1-2. In the following section, the three parts that constitute the Accelerator (the bus interface, the classifier and the microcontroller) are described after showing how the memory space of the Accelerator is divided into logic blocks.

4.1.1 Internal Address Map

Table 4-1 summarizes the memory and registers used in the Ni1000 Accelerator. The third column indicates whether the host can write and/or read the location. The fourth column indicates the same information for the microcontroller (MC). A dash (-) indicates that the location is inaccessible.

Table 4-1. Memory and Register Address Map

Address (Hex)	Name	Host W/R	MC W/R	Description
I/O Registers				
0000	CMR	W/R	R	Chip Mode Register.
0008	DIM	W/R	W/R	Vector Dimension Register.
0010	IDR	R	R	Chip ID Register.
0018	SSR	W/R	W/R	Software Status Register.
0020	HS1	R	R	Hardware Status Register 1.
0028	HS2	R	R	Hardware Status Register 2.
0030	XIR	R	W/R	External Interrupt Register.
0038	IIR	W/R	R	Internal Interrupt Register.
0040	CRA	W/R	W/R	Control Register A.
0048	CRB	W/R	W/R	Control Register B.
0050	OP0	W/R	W/R	General-purpose operand register 0.
0058	OP1	W/R	W/R	General-purpose operand register 1.
0060	OP2	W/R	W/R	General-purpose operand register 2.
0068	OP3	W/R	W/R	General-purpose operand register 3.
0070	OP4	W/R	W/R	General-purpose operand register 4.
0078	OP5	W/R	W/R	General-purpose operand register 5.

Table 4-1. Memory and Register Address Map (continued)

GRAM				
1000-10FF	GRAM	-	W/R	Microcontroller general purpose memory, 256x16.
TIMER				
1C00-1C01	TIMER	-	R	Clock count. Low word in 1C00, high word in 1C01. Cleared upon reset.
IRAM				
2000	IRAM_HW	W	-	IRAM host writable address.
2000-20FF	IR1_MCR	-	R	IRAM1 microcontroller readable addresses.
		-	W	IRAM1 microcontroller byte-oriented pre-write latches. Data occupy the high 5 bits of each byte.
2100-21FF	IR2_MCR	-	R	IRAM2 microcontroller readable addresses.
		-	W	IRAM2 microcontroller byte-oriented pre-write latches. Data occupy the high 5 bits of each byte.
2400-24FF	IR1_MCW	-	W	IRAM1 microcontroller writable addresses.
2500-25FF	IR2_MCW	-	W	IRAM2 microcontroller writable addresses.
ORAM				
2800	ORAM_HR	R	-	ORAM host readable address.
2800-283F	OR_MCR	-	R	ORAM microcontroller readable addresses.
2C00-2C3F	OR_MCW	-	W	ORAM microcontroller writable addresses.
PADCU Registers				
3001	CSA	-	W/R	PADCU Control and Status register.
3002	MODE	-	W/R	PADCU Mode register.
3004	DCU_DIM	-	W	PADCU Dimension register. It contains the dimension of the input vector minus 1. The value is from 0 to 255, inclusive.
3008	NCA	-	W	PADCU register that contains the prototype array index of the last prototype in use by the active network. (see description)
3010	NCB	-	W	PADCU register that contains the MU clock count.
3020	AUX	-	W/R	PADCU auxiliary register.

Table 4-1. Memory and Register Address Map (continued)

3040	CSB	-	W/R	PADCU control and status register.
3200	ARR	-	W/R	PADCU Address Relocation register. Contains the starting position (lowest number row and column) of the PA block in use. Value for dimension must be multiple of 32, and column boundary must be multiple of 128.
PPRAMs and Registers				
4081	PPRAMCR3	-	W/R	PPRAM3 control register.
4101	PPRAMCR2	-	W/R	PPRAM2 control register
4201	PPRAMCR1	-	W/R	PPRAM1 control register
4381	PPRAM_CR	-	W	PPRAM global control register. It is used to write all three PPRAMs.
4400-47FF	PPRAM1	-	W/R	For each prototype vector, this RAM holds a 6-bit class ID, a 1-bit probabilistic flag, a 1-bit Used flag, and an 8-bit smoothing factor (mantissa plus exponent).
4800-4BFF	PPRAM2	-	W/R	For each prototype vector, this RAM holds a 13-bit threshold radius and configuration flags.
5000-53FF	PPRAM3	-	W/R	For each prototype vector, this RAM holds a 16-bit count of the number of times that vector fired in the last epoch of the training process.
MURAMs and Registers				
6080	MURAM1	-	R	Firing class count for MURAM1.
60C0	MURAM2	-	R	Firing class count for MURAM2.
6100	MURAM_CR	-	W/R	MU mode-control register.
6200-623F	Flag MURAM	-	R	64 flags, used to indicate the firing classes for the current MURAM. Only the LSB is used.
6400-643F	Firing Class List MURAM1	-	W/R	One of the two alternating 64x8-bit buffers, reserved for the class IDs of firing classes.
6440-647F	Firing Class List MURAM2	-	W/R	One of the two alternating 64x8-bit buffers, reserved for the class IDs of firing classes.
6800-683F	Probability MURAM1	-	W/R	One of the two 64x16-bit buffers, used to accumulate the probability densities of the input vector for each of the 64 classes.
6840-687F	Probability MURAM2	-	W/R	One of the two 64x16-bit buffers, used to accumulate the probability densities of the input vector for each of the 64 classes.

Table 4-1. Memory and Register Address Map (continued)

PGFLASH Registers				
7700	PGF_DR	W/R	-	PGFLASH data register.
7701	PGF_CR1	W/R	-	PGFLASH control register 1.
7702	PGF_CR2	W/R	-	PGFLASH control register 2.
7703	PGF_SR	R	-	PGFLASH status register.
7704	PGF_ADR	W/R	-	PGFLASH address register.
Prototype Array				
B000-B3FF	PNUM DCU Used Flags DCU Distances	-	W/R	*PNUM*—Prototype number, which selects the prototype array column. The number must be from 0 to 1023, inclusive. *DCU Used Flags*—1-bit (bit 13) flag for each of the 1024 prototype vectors. DCU hardware operates on a prototype only when the corresponding flag is set. *DCU Distances*—1024 City Block distances between an input vector and each of the prototypes. Each distance is 13-bits and aligned low. PADCU registers, MODE and AUX, are used for selection.
B800-B8FF	PDIM	-	W/R	Prototype dimension, which selects the prototype array row. The number must be from 0 to 255, inclusive.
PGFLASH				
F000-FFFF	PGFLASH	W/R	R	Microcontroller program memory, 4Kx16-bit. The microcontroller can only fetch instructions through PDbus. The host can access PGFLASH only in the PG hardware-controlled access mode.

4.2 Bus Interface

The bus interface is used for communication between the host and either the microcontroller or the classifier. It is used by the host to program the flash memory, write vectors to the input buffer, read classification results from the output buffer, and access a set of registers used to interact with the microcontroller.

The bus interface has an input and an output side, as shown in Figure 4-1 The external bus interface handles bus cycles to the host, and the internal bus interface handles bus cycles from the microcontroller. These interfaces are used to access these resources:

- I/O Registers—a set of sixteen 16-bit registers used for communication between the host and the microcontroller.
- Input RAM (IRAM)—a double buffer consisting of two 256 x 5 memories. Each memory can store one input vector. The IRAM is also integrated into the architecture of the classifier's pipeline.
- Output RAM (ORAM)—a buffer that receives the contents of the current MURAM, and optionally reformats the probability values from the internal 16-bit floating-point format into the standard IEEE 32-bit floating-point format. The ORAM is also integrated into the architecture of the classifier's pipeline.

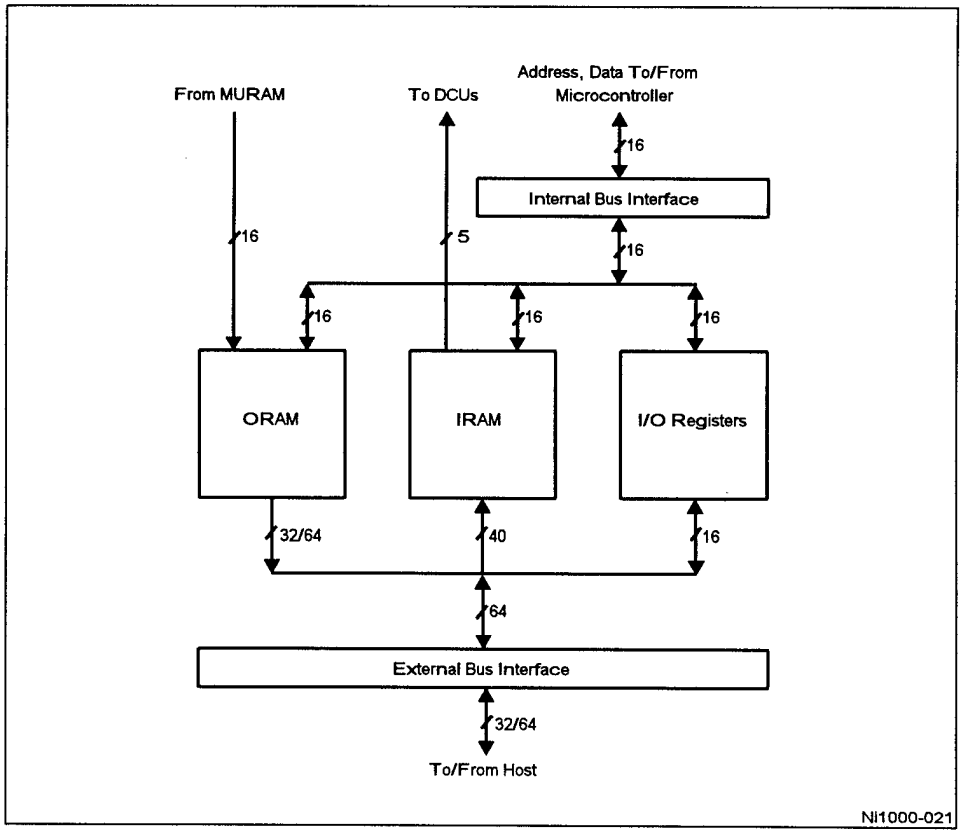


Figure 4-1. Bus Interface

Both 32- and 64-bit data bus widths are available, as selected by the 64/32# signal. This signal is not allowed to change dynamically. The chip must be reset following a change.

The Ni1000 Accelerator may appear to external hardware as a block of memory. However, the host can only access IRAM and ORAM through addresses that act like I/O ports, in which the same address is accessed over and over, until all data have been transferred. Register bits indicate when the buffer memories are about to overflow or underflow, and a bit in the CRA register can be programmed to cause assertion of the service request SRQ# output to the host when the ORAM is full (see Chapter 5 for a description of the CRA register).

The IRAM, ORAM and virtually all other memories in the Accelerator are mapped into the microcontroller's address space and can be accessed by the microcontroller when the classifier is not running.

The main signals of the bus interface are:

- CLK—clock input.
- A[0:15]—address bus, input from host.
- D[0:63]—64-bit bidirectional data bus.
- ADS#—address/data strobe input from host.
- W/R#—read/write input from host.
- RDY#—bus cycle termination output to host.
- BRDY#—bus cycle termination output with burst-mode request.
- BLAST#—input from host indicating the last data transfer of a cycle.

The last two signals, BRDY# and BLAST#, are used for burst cycles, in which one 32-bit or 64-bit word is transferred per clock period. Burst cycles begin like non-burst cycles, which take a minimum of two clock periods each (assuming the external logic can return RDY# in the second clock), with the assertion of ADS#. However, the assertion of BRDY# by the chip allows the host to enter a bus mode in which each additional data transfer requires only one additional cycle. The chip can exit burst mode by asserting RDY# instead of BRDY#, and the host can exit burst mode by asserting BLAST# to indicate that the current clock period is the last clock period of a burst. Host support for burst mode is optional. See the "Signal Descriptions" chapter for a detailed description of the bus signals, and the "Bus Operations" chapter for the timing diagrams of bus cycles.

4.2.1 I/O Registers

The 16-bit I/O registers occupy addresses 0000h through 0078h in the memory map. They can be read by both the host and the microcontroller, but not all bits in all registers can be written. Their functions include:

- Control of the operating mode of the IRAM and ORAM.
- Access to hardware status signals.
- Communication between the host and the microcontroller, including host commands to the microcontroller, or input/output parameters between the host and the microcontroller.

4.2.1.1 CMR (Chip Mode Register)

This 16-bit register is generally used to transfer commands from the host to the microcontroller. The microcontroller should not write to this register. A write by the host sets

IIR[15], which causes an internal interrupt to the microcontroller if the Interrupt Enable flag (HS1[6]) is set. When this happens, the interrupt request flag, IR, is set and visible to the microcontroller. Figure 4-2 shows the register, followed by its bit assignments.

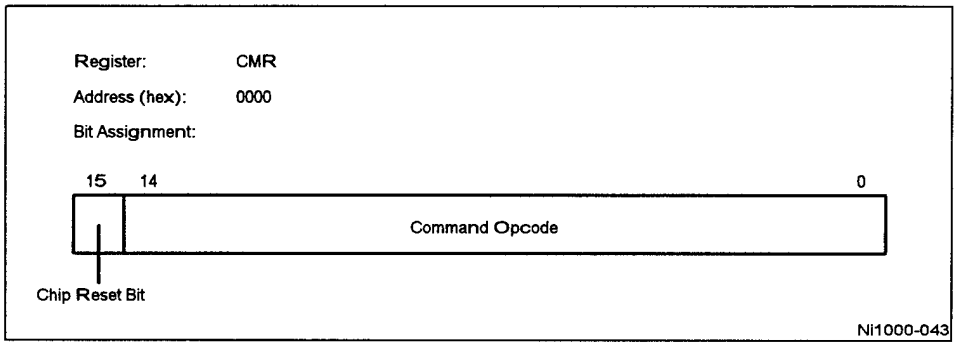


Figure 4-2. The CMR Register

bits [0:14] *Command Opcode*
(read and write by host or microcontroller)
User command opcode. The opcode is interpreted by the microcontroller software.

bit 15 *Chip Reset*
(read and write by host or microcontroller, initialized to 1 upon chip reset.)
1 = Accelerator is reset. This bit is also set when the RESET# or MC# pin is asserted.
0 = Accelerator is not reset. The value can be written only by the host.

4.2.1.2 DIM (Vector Dimension Register)

This 16-bit register contains the highest input vector dimension minus 1 (0-255) and the desired number of classes minus 1 (0-63) in PRCE output. Both the host and the microcontroller can read and write to this register with no immediate side-effects. However, the value in the register must be stable before and throughout the classification process. Figure 4-3 shows the register, followed by its bit assignments.

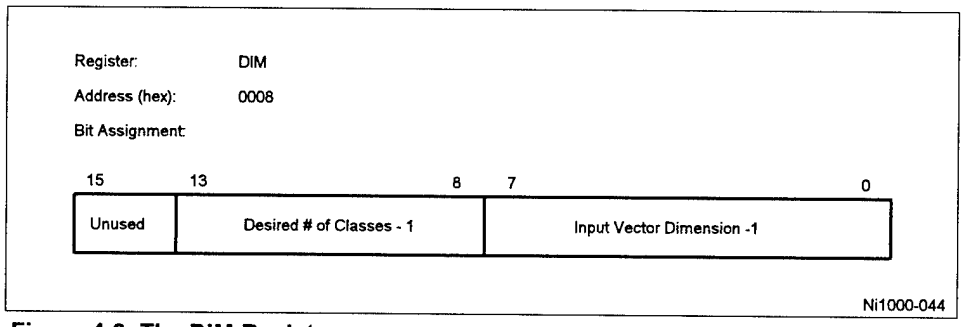


Figure 4-3. The DIM Register

bits [0:7] *Input Dimension*
(read and write by host or microcontroller)
The number of Ni1000 input dimensions of input vectors minus 1.

bits [8:13] *Output Classes*
(read and write by host or microcontroller)
The desired number of classes for PRCE output minus 1.

bits [14:15] *Reserved.*

4.2.1.3 IDR (Chip ID Register)

This 16-bit register is read-only by both the host and the microcontroller and hard-coded with the value 315Bh. It is the chip identification. Figure 4-4 shows the register.

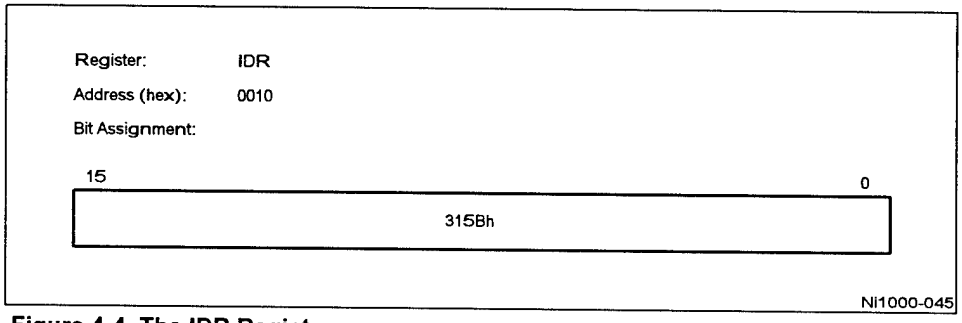


Figure 4-4. The IDR Register

4.2.1.4 SSR (Software Status Register)

This 16-bit register is used to show the status of the microcontroller's software to the host. It can be written only by the microcontroller. It has no effect on the Accelerator's hardware. Figure 4-5 shows the register. Bit-assignments must be defined in the host program. The Ni1000 Recognition Accelerator User's Guide explains how it is used in the standard microcontroller program that is shipped with the chip.

Register: SSR

Address (hex): 0018

Bit Assignment:

15	0
Not pre-defined. Meanings determined by active microcode.	

Ni1000-046

Figure 4-5. The SSR Register

4.2.1.5 HS1 (Hardware Status Register 1)

This 16-bit register is used to store the states of the microcontroller's flags, which are sampled at each clock cycle. It is intended to be read by the host. Reading of this register by the microcontroller is less efficient than using the built-in microcontroller flag-testing instructions. *CSW* is the microcontroller flag register. See the "Microcontroller" Section for its bit assignments. Figure 4-6 shows the HS1 register, followed by its bit assignments, which are identical to that of CSW. Microcontroller flags are not the only flags that are used by the Accelerator. The Prototype Parameter RAMs (PPRAMs) have *Used* and *Disabled* flags, and the Distance Calculation Units (DCUs) also have *Used* flags.

Register: HS1

Address (hex): 0020

Bit Assignment:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved		DC	M2	M1	FW	MC	GE	SE	IE	TR	O	P	N	Z	C

NI1000-047

Figure 4-6. The HS1 Register

All bits are read-only by both the host and the microcontroller. 1 = set, 0 = clear.

bit 0	*Carry*
bit 1	Zero
bit 2	Negative
bit 3	Positive
bit 4	Overflow
bit 5	Interrupt Request
bit 6	Interrupt Enable
bit 7	Stack Error
bit 8	General Error
bit 9	Multi-Class Firing
bit 10	FLASH-Write
bit 11	MURAM1 Ready
bit 12	MURAM2 Ready
bit 13	PADCU Busy
bit 14	Reserved
bit 15	Reserved (always cleared to 0)

4.2.1.6 HS2 (Hardware Status Register 2)

This 16-bit register is used to indicate the status of the hardware units other than the microcontroller. Bits 10 through 15 are particularly important, since they indicate the mode of the Accelerator and the full or empty status of IRAM and ORAM. They should be checked by the host before loading input vectors for classification, and before reading classification results. Figure 4-7 shows the register, followed by its bit assignments.

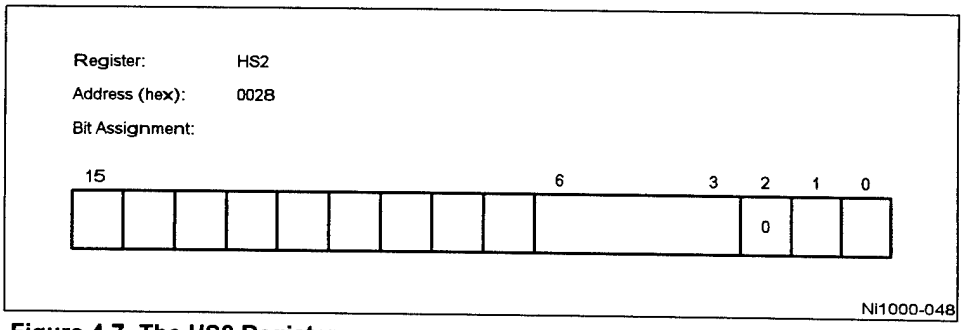


Figure 4-7. The HS2 Register

bit 0 *PADCU Busy*
(read-only by host)
1 = PADCU is busy.
0 = PADCU is not busy.

bit 1 *MU Busy*
(read-only by both the host and the microcontroller)
1 = MU is busy.
0 = MU is not busy.

bit 2 *Complement of SRQ# Output*

bits [3:6] *Last I/O Register Written*
(read-only by both the host and the microcontroller)
Contain bits [3:6] of the address of the last I/O register written, either by the microcontroller or the host. Thus, either the host or microcontroller can determine the address of the last I/O register written by reading HS2.

bit 7 *64/32# Status*
(read-only by both the host and the microcontroller)
1 = Host data bus is 64-bit.
0 = Host data bus is 32-bit.

bit 8 *MULTICHIP# Status*
(read-only by both the host and the microcontroller)
The MULTICHIP# bit reflects the inverse of the state of the MULTCHIP# pin. Although originally intended to be used for multi-chip training, it is not used in the standard microcode.

bit 9 *Multiple Firing Classes*
(read-only by both the host and the microcontroller)
1 = ORAM contains multiple firing classes.
0 = ORAM does not contain multiple firing classes.

bit 10 *ORAM Fully Read*
(read-only by both the host and the microcontroller)
1 = ORAM has been fully read by the host.
0 = ORAM has not been fully read by the host.
This bit is necessary because multiple data transfers may be needed to read all of the data in ORAM.

bit 11 *IRAM Fully Written*
(read-only by both the host and the microcontroller)
1 = IRAM has been fully written by the host.
0 = IRAM has not been fully written by the host.
This bit is necessary because multiple data transfers may be needed to write all of the data into IRAM.

bit 12 *ORAM Mode*
(read-only by both the host and the microcontroller)
1 = ORAM is in *Classify* mode. The host can read ORAM if it is not empty.
0 = ORAM is in *Microcontroller* mode. The microcontroller can access (read and write) ORAM. Any read attempt by the host is illegal and causes the Accelerator to assert BERR#.

bit 13 *IRAM Mode*

(read-only by both the host and the microcontroller)

1 = IRAM is in *Classify* mode. The host can write to IRAM if it is not full.

0 = IRAM is in *Microcontroller* mode. The microcontroller can access (read and write) IRAM. Any write attempt by the host is illegal and causes the Accelerator to assert BERR#.

bit 14 *ORAM Full*

(read-only by both the host and the microcontroller)

1 = ORAM is full. The host can read ORAM if HS2[12] is 1.

0 = ORAM is not full. Any read attempt by the host is illegal and causes the Accelerator to assert BERR#.

bit 15 *IRAM Full*

(read-only by both the host and the microcontroller)

1 = IRAM is not full. The host can write another vector to IRAM if HS2[13] is 1.

0 = IRAM is full. Any write attempt by the host is illegal and causes the Accelerator to assert BERR#.

4.2.1.7 XIR (External Interrupt Register)

This 16-bit register is used to identify the reason for a service request (assertion of SRQ#) from the microcontroller to the host. It can be written only by the microcontroller. A value FFFFh read by the host indicates that ORAM is full. When the microcontroller writes to XIR, the SRQ# pin is also asserted, and stays that way until the host asserts the service acknowledge pin, IACK#.

Figure 4-8 shows the register. There are no specific bit assignments for the XIR register. The host and the microcontroller should follow the same convention to code or decode its contents. For how it is used in the standard microcontroller program that is shipped with the chip, see the *Ni1000 Recognition Accelerator User's Guide*.

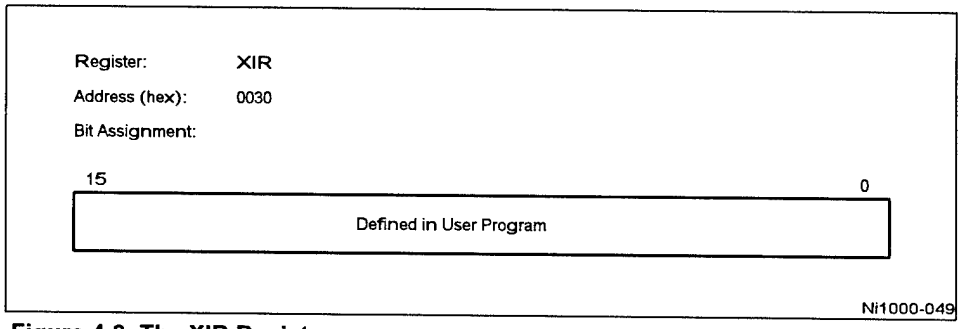


Figure 4-8. The XIR Register

4.2.1.8 IIR (Internal Interrupt Register)

This 16-bit register is used to identify the reason for an interrupt request from the host to the microcontroller. It can only be written by the host. However, the microcontroller responds to this interrupt only when its interrupt-enable (IE) flag is set. The contents of IIR may be changed by on-chip hardware conditions, such as the loading of the CMR register, or the assertion of the MCINT# pin by the host. Each bit position represents a different hardware condition. The value in IIR is only an interrupt identifier, not an interrupt vector. Figure 4-9 shows the register, followed by its bit assignments.

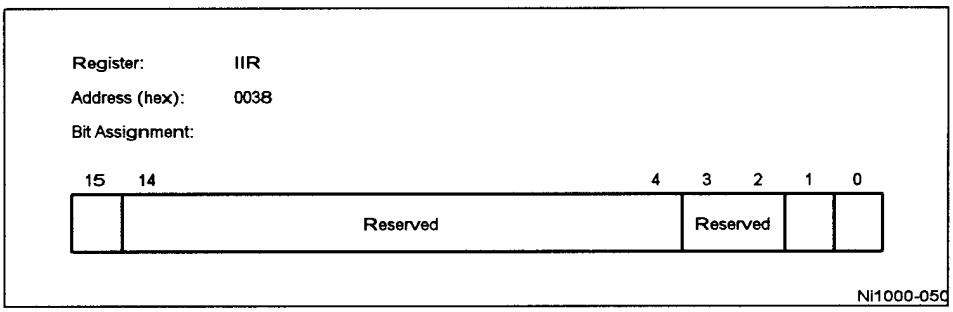


Figure 4-9. The IIR Register

bit 0 *MCINT# Status*
(read and write only by host)
1 = The MCINT# pin is asserted by the host.
0 = The MCINT# pin is deasserted.

bit 1 *ERROR# Status*
(read and write only by host)
1 = The ERROR# pin is asserted by the host.
0 = The ERROR# pin is not asserted.

bits [2:3] *Reserved.*

bits [4:14] *Reserved*
(Always reads zero.)

bit 15 *CMR Written*
(read-only by host)
1 = The CMR register has been written by the host.
0 = The CMR register has not been written by the host.

4.2.1.9 CRA (Control Register A)

This 16-bit register is used by the host to monitor and control the behavior of the IRAM and ORAM when the Accelerator is operating in classify mode. However, it can be written by both the host and the microcontroller. Figure 4-10 shows the register, followed by its bit assignments.

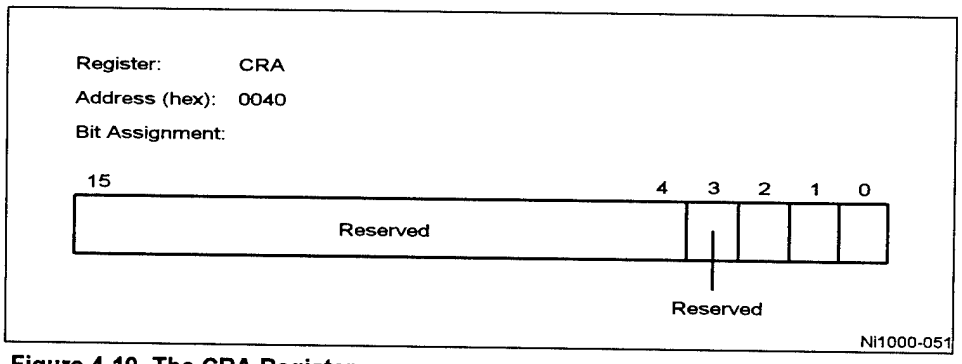


Figure 4-10. The CRA Register

bit 0 *ORAM Mode*
(read and write by both the host and the microcontroller)
(Any change in the value of this bit resets, then re-enables ORAM; HS2[14] is cleared to 0 to mark ORAM empty.)
1 = ORAM is in *Classify* mode. Results from MURAM are loaded into ORAM whenever ORAM is empty.
0 = ORAM is in *Microcontroller* mode.

bit 1 *RCE/PRCE*
(read and write by both the host and the microcontroller)
(Any change in the value of this bit resets, then re-enables ORAM; HS2[14] is cleared to 0 to mark ORAM empty.)
1 = Results from MURAM are PRCE results, probability densities.
0 = Results from MURAM are RCE results, firing class IDs.

bit 2 *ORAM Service Request*
(read and write by both the host and the microcontroller)
(Initialized to 1 upon chip reset.)
1 = SRQ# will not be asserted when ORAM becomes full.
0 = SRQ# will be asserted when ORAM becomes full, as indicated by HS2[14] = 1; FFFFh is loaded into the XIR register.

bits [3:15] *Reserved*

4.2.1.10 CRB (Control Register B)

This 16-bit register is used to control the software modes of the ORAM and the IRAM. It is written by the microcontroller. Figure 4-11 shows the register, followed by its bit assignments.

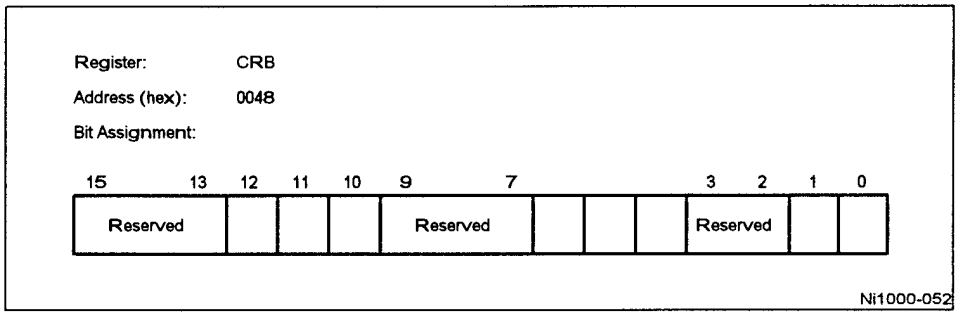


Figure 4-11. The CRB Register

bit 0 *IRAM Reset*
(read and write by both the host and the microcontroller)
(Initialized to 1 upon chip reset.)
1 = puts IRAM into the reset state.
0 = releases IRAM from the reset state.

bit 1 *ORAM Reset*
(read and write by both the host and the microcontroller)
(Initialized to 1 upon chip reset.)
1 = puts ORAM into the reset state.
0 = releases ORAM from the reset state.

bits [2:3] *Reserved*

bit 4 *Floating-Point Conversion*
(read and write by both the host and the microcontroller)
(Initialized to 1 upon chip reset.)
1 = ORAM converts the PRCE probability densities obtained from MU into IEEE-754 32-bit format.
0 = PRCE data read out of the ORAM are in the MU internal 16-bit format.

bit 5 *ORAM Mode*
(read and write by both the host and the microcontroller)
(Initialized to 1 upon chip reset.)
1 = ORAM is in *Classify* mode.
0 = ORAM is in *Microcontroller* mode.

bit 6 *IRAM Mode*
(read and write by both the host and the microcontroller)
(Initialized to 1 upon chip reset.)
1 = IRAM is in *Classify* mode.
0 = IRAM is in *Microcontroller* mode.

bits [7:9] *Reserved.*

bit 10 *IRAM1 Full*
(read and write by both the host and the microcontroller)
1 = IRAM1 is full.
0 = IRAM1 is not full.

bit 11 *IRAM2 Full*
(read and write by both the host and the microcontroller)
1 = IRAM2 is full.
0 = IRAM2 is not full.

bit 12 *ORAM Full*
(read and write by both the host and the microcontroller)
1 = ORAM is full.
0 = ORAM is not full.

bits [13:15] *Reserved*

4.2.1.11 OP[0:5] (General Operand Registers)

These six 16-bit registers are storage locations used to pass parameters between the host and microcontroller. They have no side-effects on hardware when written. Conventions for using these registers must be established and followed by both the host and the microcontroller. For how they are used in the standard microcontroller program that is shipped with the chip, see the *Ni1000 Recognition Accelerator User's Guide*.

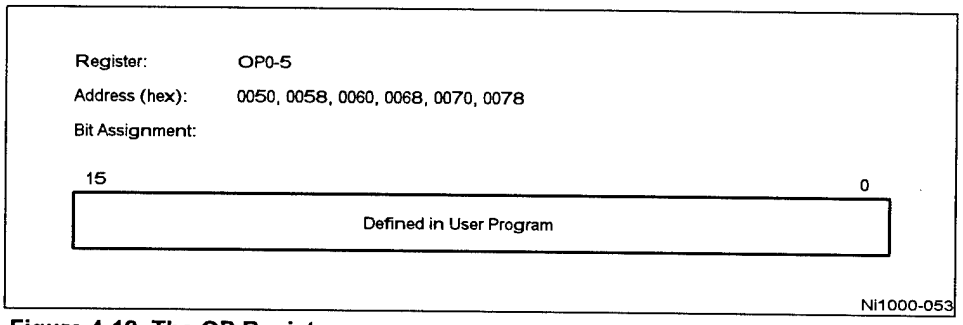


Figure 4-12. The OP Registers

4.2.2 Input RAM (IRAM)

The IRAM is shown in Figure 4-13 It is a double buffer consisting of two 32 x 40 banks. Each bank can store one 222-feature input vector (padded to 256), with 8 five-bit features packed into each 40-bit word. When the classifier is running, these banks are inaccessible to the microcontroller, however the host can load data into the IRAM. When the classifier is not running, the microcontroller can directly access the IRAM. The IRAM is mapped into the microcontroller's address space.

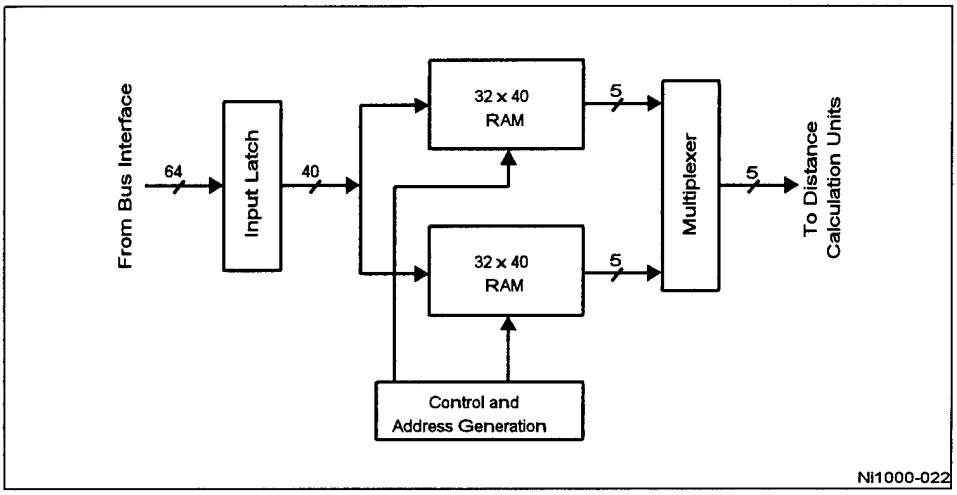


Figure 4-13. Input RAM (IRAM)

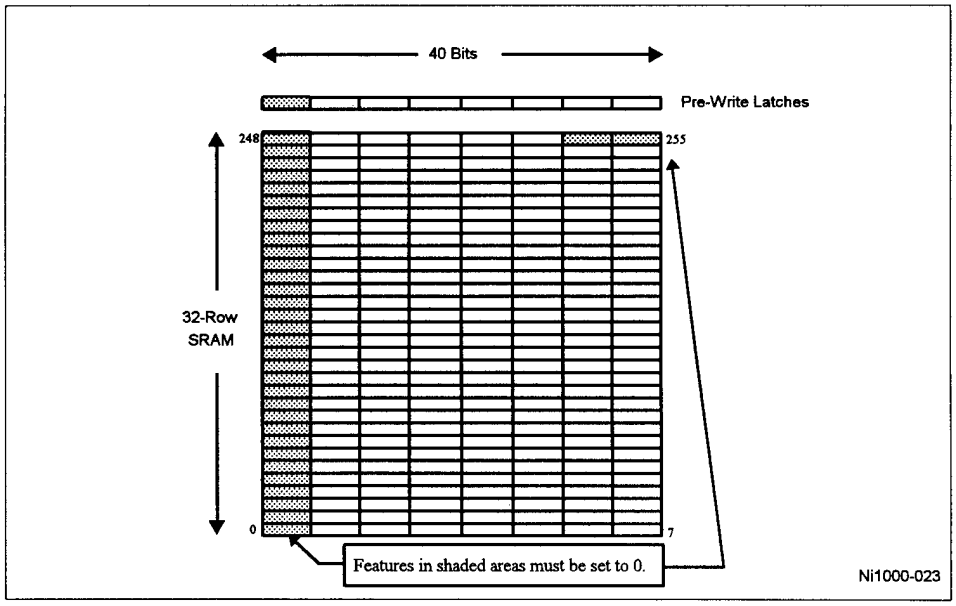


Figure 4-14. IRAM Pre-Write Latch

Each bank of the IRAM has its own set of latches and access addresses, as shown in Table 4-2.

Microcontroller writes to the IRAM require loading of a pre-write latch, illustrated in Figure 4-14. The latch is addressable in the microcontroller's address space. Note that unless the

write latch is entirely filled with valid data, memory locations adjacent to the destination of the write may get overwritten with spurious data. Writing to a location in the IRAM loads the entire row with the contents of the latch.

Table 4-2. IRAM Access Addresses in Microcontroller Mode

Location	Starting Address	Ending Address	# of Addresses	Resolution of Data
IRAM1 Readable Addresses	2000h	20FFh	256	5 bit
IRAM2 Readable Addresses	2100h	21FFh	256	5 bit
IRAM1 Latchable (Pre-Write) Addresses	2000h	20FFh	8 (3 least-significant address bits specify position)	5 bit (5 MSBs of the *low* byte on the DBUS)
IRAM2 Latchable (Pre-Write) Addresses	2100h	21FFh	8 (3 least-significant address bits specify position)	5 bit (5 MSBs of the *low* byte on the DBUS)
IRAM1 Writable Addresses	2400h	24FFh	32 (higher address bits specify RAM row to be written)	40 bits (written simultaneously)
IRAM2 Writable Addresses	2500h	25FFh	32 (higher address bits specify RAM *row* to be written)	40 bits (written simultaneously)

The features of the input vector, as received from the bus interface, are five-bit quantities aligned to the five most-significant bits of each byte on the bus interface. The three least significant bits of each byte are ignored.

The autosequencing logic for loading the IRAM works differently for 32-bit and 64-bit external data bus widths. The vector loaded into the IRAM, as visible to the microcontroller, has a different organization depending on which bus width is selected. Figure 4-15 and 4-16 show the data alignments.

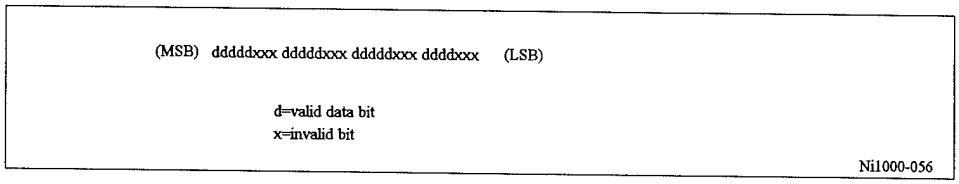


Figure 4-15. Data Alignment on 32-Bit External Bus

```
(MSB)  dddddxxx dddddxxx dddddxxx dddddxxx
       dddddxxx dddddxxx dddddxxx dddddxxx        (LSB)

       d=valid data bit
       x=invalid bit
                                                  Ni1000-057
```

Figure 4-16. Data Alignment on 64-Bit External Bus

When the microcontroller accesses IRAM (read or write), data appear in the most significant 5 bits of the least significant byte, as shown in Figure 4-17.

```
(MSB)  xxxxxxxx dddddxxx   (LSB)

       d=valid data bit
       x=invalid bit
                                                  Ni1000-058
```

Figure 4-17. Internal Bus Data Alignment

4.2.3 Output RAM (ORAM)

The ORAM is shown in Figure 4-18. It is a small buffer, with a pre-write latch between it and the MURAMs. After the Math Unit completes processing of the contents in one bank of the Math Unit RAM (MURAM), the contents of that buffer are transferred to the ORAM while a new input vector is being read into the other MURAM buffer.

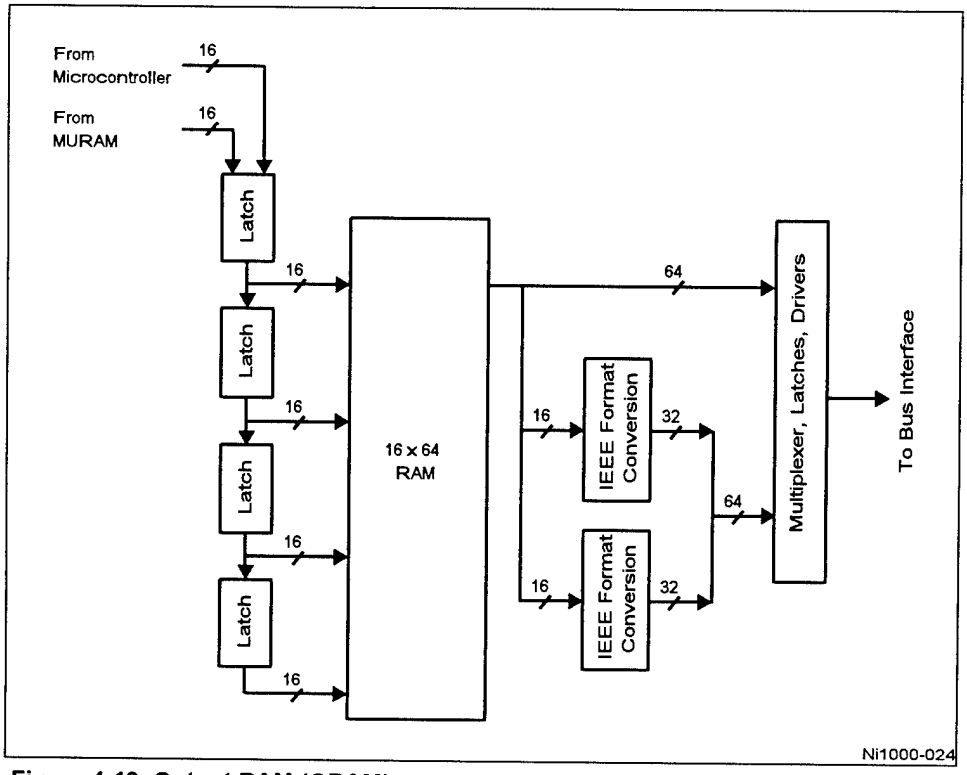


Figure 4-18. Output RAM (ORAM)

The ORAM is 16 x 64-bits in size, so it can hold all of the data generated by classifying an input vector.

When the classifier is running, the ORAM is accessible to the host. The number of valid reads that can be made from the ORAM depends on the mode. If the class-list MURAM is being uploaded, the number of entries in the ORAM will be equal to the number of firing classes. If the probability MURAM is being uploaded, the number of entries will be specified by a byte in the DIM register.

The microcontroller may directly access the ORAM. On reads, the ORAM is mapped into the microcontroller's address space, as shown in Table 4-3.

Table 4-3. ORAM Access Addresses in Microcontroller Mode

Location	Starting Address	Ending Address	Number of Addresses	Resolution of Data
ORAM Readable Addresses	2800h	283Fh	64 (One of 4 words in 16 rows)	16 bit
ORAM Latchable (Pre-Write) Addresses	2800h	283Fh	1 (the first block in the shift register)	8 or 16 bits (selected with RCE#)
ORAM Writable Addresses	2C00h	2C3Fh	16 (possible rows in the RAM)	64 bits (written simultaneously)

Microcontroller writing to the ORAM requires loading a pre-write latch, illustrated in Figure 4-19. The write latch is also addressable in the microcontroller's address space. Like the IRAM, there is a special range of addresses for referencing the destination of a write. Unlike the IRAM, the ORAM write latch is a four-word shift register. Writing less than four words of data to the write latch before invoking a write operation may result in data appearing in the wrong position within a word.

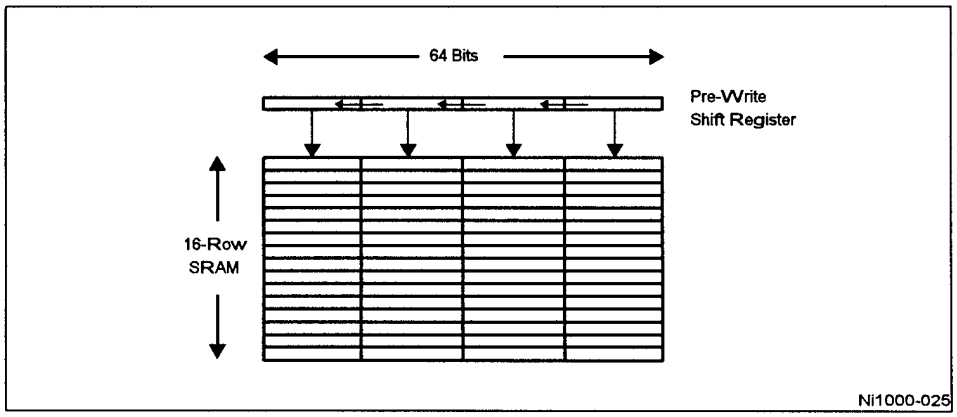


Figure 4-19. ORAM Pre-Write Latch

When the host reads the classification results from ORAM, the output is one of those summarized in Table 4-4.

Table 4-4. ORAM Output Possibilities

CRA[1]	CRB[4]	64/32#	Output (per bus cycle)	Maximum Possible Number of Bus Cycles
0	0	0	4 *Classes*	16
0	0	1	8 *Classes*	8
0	1	0	4 *Classes*	16
0	1	1	8 *Classes*	8
1	0	0	2 Unformatted *Probability Densities*	32
1	0	1	4 Unformatted *Probability Densities*	16
1	1	0	1 Formatted *Probability Density*	64
1	1	1	2 Formatted *Probability Densities*	32

Class information is given in the format shown in Figure 4-20. The number of bytes available when *class* information is requested is determined by the number of firing classes calculated by the math unit (MU). Regardless of the value of CRA[1], if the number of firing classes is greater than one, HS2[9] will be set to 1. The class number in the least significant 6 bits of each byte is valid only if the corresponding valid bit is set to indicate that this byte of information is valid.

(MSB) vpnnnnnn vpnnnnnn vpnnnnnn vpnnnnnn (LSB)

n=firing class #
p=probabilistic bit
v=valid byte (valid if HIGH, invalid if LOW)

Ni1000-061

Figure 4-20. Format of RCE Classification Results

The probabilistic bit indicates whether the classification for that class number is a reliable deterministic classification (p = 0) or whether probabilistic classification should be used (p = 1). Note that the probabilistic bit only reliably indicates deterministic if prototypes are arranged so that the DCUs will process all probabilistic prototypes before any deterministic prototypes for a given class. This typically requires reordering of the prototypes after on-chip learning. Reordering may also be necessary if the prototype array is loaded from external data and a disabled column exists in the prototype array such that the prototype array must be shifted to avoid the disabled column.

The order in which prototypes are processed is:

1. From the highest numbered column in use that is less than 512 down to 0
2. If necessary, from the highest numbered column in use above 512 down to 512

Probability densities appear in one of the two formats, MU Internal or IEEE Standard. See the "Computational Precision" section for details. The number of probability densities output is determined by the *number of desired classes*, which is specified in the high byte of the DIM register (bits 8 through 15).

4.3 The Classifier

The classifier consists of the pipeline shown in Figure 4-21. While data is being loaded into the double buffer at the input of the pipeline or being read from the output of the pipeline, the classifier can be comparing a previously loaded input vector against the prototype vectors in the prototype array.

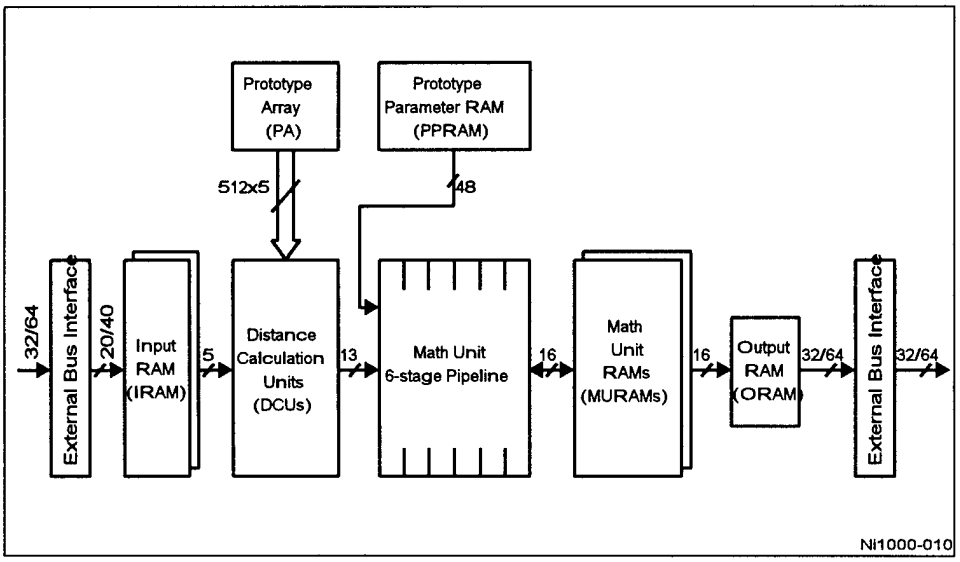


Figure 4-21. Classifier

The classifier consists of the following units:

- *Input RAM (IRAM)*—a double buffer consisting of two 256 x 5 memories. Each memory can store one input vector. The IRAM is part of the bus interface unit, which is described in Section 3.2.
- *Prototype Array (PA)*—a flash memory that holds the prototype vectors allocated during learning, i.e. the coordinates of the Radial Basis Function (RBF) centers.
- *Distance Calculation Units (DCUs)*—a 512-processor array that performs the distance calculations between an input vector and each prototype vector in the PA.
- *Prototype Parameter RAMs (PPRAMs)*—a memory that holds all of the data that defines a radial basis function except its prototype vector (which is stored in the PA). This data includes the RBF radius, number of vectors it recognized during the last pass through the

training set, etc. Unlike the PA, the PPRAM is not flash memory; it is static RAM. Typically, it is loaded during power-on initialization from off-chip or from a reserved section of the prototype array (PA).

- *Math Unit (MU)*—a six-stage pipelined processor that implements the exponential for calculating probability densities. It also applies the threshold function, to decide whether an input vector falls within a prototype's influence field.
- *Math Unit RAMs (MURAMs)*—a set of memories that receives the class IDs that are classified as similar to the input vector. It also holds the accumulated probability density for each class.
- *Output RAM (ORAM)*—a buffer that receives the classification results for a vector and optionally reformats the probability values from the internal 16-bit floating-point format into a format compatible with the standard IEEE 32-bit floating-point format. The ORAM is also part of the bus interface unit, which is described in the previous section.

4.3.1 Distance Calculation Units (DCUs)

Figure 4-22 shows an individual distance calculation unit (DCU) from the 512-unit array. The DCUs are statically associated with PA columns. Due to redundant array elements, only 500 DCUs are used for classification at any time. A DCU computes the absolute value of the difference between a feature (dimension) of the input vector and the corresponding feature of a prototype. The DCU then accumulates that value into a running tally of the distance between the input vector and the prototype. Such distance is calculated between the input vector and each valid (i.e. not disabled) prototype.

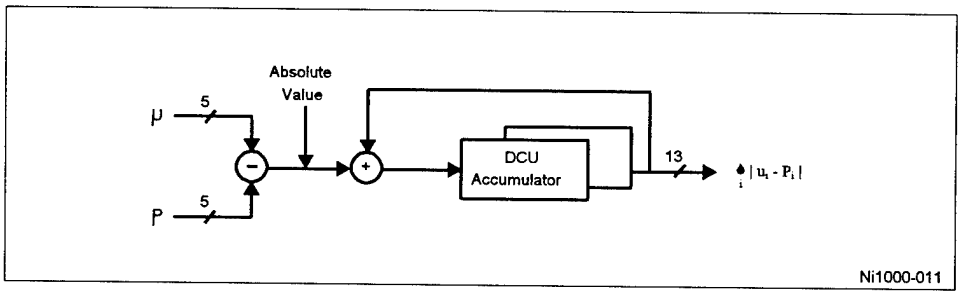


Figure 4-22. Distance Calculation Unit (DCU)

The DCU accumulates a sum of the absolute differences of each feature in the input vector and the corresponding feature in a prototype, called *city-block distances*. The following equation expresses the city-block distance, *d,* between an input vector *U* with *i* dimensions and a prototype vector *P*.

$$d = | u_0 - p_0 | + | u_1 - p_1 | + \ldots | u_i - p_i |$$

The DCU has two accumulators and is used in a two-phase mode, in which half of the prototypes in the PA are processed during one phase, and the other half in the following phase. When there are 500 prototypes or less, a single phase is used that does not require the second accumulator. When there are more than 500 prototypes, two-phase processing is

used, with the other accumulator being used during the second phase. Section 3.6 describes the timing of the DCUs and the classification pipeline.

At the end of each classification pass through the prototype array, the values in the accumulators represent the city-block distance between the input vector and each valid prototype vector stored in the PA. This is used by the MU pipeline for evaluating the probability and threshold functions.

4.3.2 Prototype Array (PA)

The Prototype Array (PA) is a non-volatile flash memory. PA occupies addresses B000h through B8FFh in the memory. It can store up to 1000 222-feature prototype vectors with 5-bit resolution per feature. When the prototypes have 32 dimensions or less, the PA can store as many as 8000 such vectors by reorganizing the array using the *ARR*.

The PA is organized as two 256 (row) x 512 (column) arrays. A row corresponds to a feature and a column corresponds to a prototype. Each array has 512 individually erasable blocks, each containing two columns. For example, column number 0 and column 512 are in the same block as are columns 1 and 513, etc. Data for either column (prototype) in a block can be written individually into the PA, but erasing operates on a block, potentially erasing two prototypes if both columns are used. It is the designer's responsibility to save the vector not intended for erasing. The standard microcontroller software saves and restores these columns when doing a COLUMNERASE command. When doing a COLUMNWRITE command, the host software must backup the "other" column . See Chapter 7of the *Ni1000 Recogntion Accelerator User's Guide*, for more information.

Figure 4-23 shows a conceptual view of the PA flash memory being accessed by the DCUs. An input vector is presented as a stream of 5-bit integers. These are the individual features of the input vector. The 512 DCUs operate one feature of the input vector against the corresponding feature in up to 512 prototype vectors simultaneously. After the last prototype has been processed, the DCUs pass their accumulated city-block distances to the next stage of processing, the math unit (MU) pipeline.

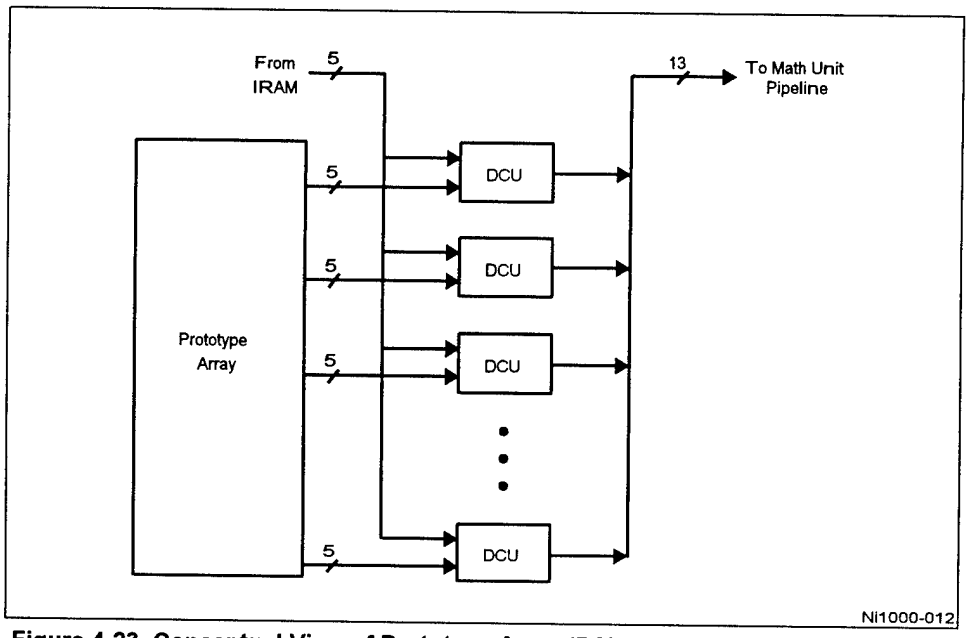


Figure 4-23. Conceptual View of Prototype Array (PA) and DCUs

PA flash memory can only be programmed by the microcontroller. Programming PA requires a 12V voltage applied to the Vpp.

Two address ranges in the microcontroller's address space are used to read the PA. An address in the range from B000h to B3FFh is used to specify one of the 1024 prototype vectors to read. Another range of 256 addresses from B800h to B8FFh specifies which features of the column are to be read. Reading is a two-step process in which a first read specifies either the column or the feature, and a second read specifies the remaining quantity. Either the column or the feature can be specified first. Valid data is returned on the second read. The upper six bits of the data are undefined. The lower ten bits are the value of one 5-bit feature in both true and complement form. Figure 4-24 shows the alignment of a 5-bit feature, p[0:4], and Figure 4-25 shows the architecture of the PA during programming.

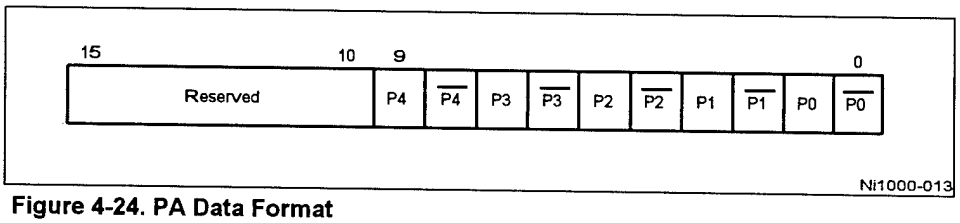


Figure 4-24. PA Data Format

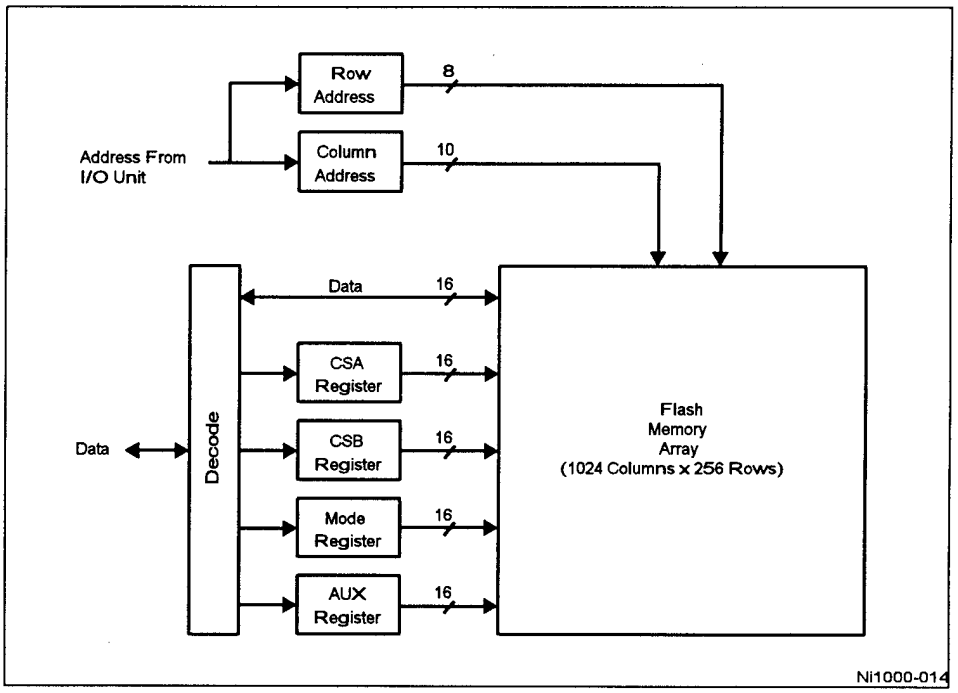


Figure 4-25. PA During Programming

The last column of the PA (storage for prototypes number 511 and 1023) is used as *a Bad Column Table(BCT).* If the column is faulty, the next column (for prototype number 510 and 1022) is used. The BCT stores the die's serial number, sorted date, and faulty column locations in the PA flash memory. The BCT is organized as 256 10-bit numbers. The BCT specifies locations of faulty blocks (containing two columns), rather than columns. The whole block must not be used if either column is faulty. This avoids potential problems in PA programming and erasing.

Use of the BCT varies, depending on the application software. For example, when new prototypes are committed and programmed into the PA during learning, the host or microcontroller must keep track of the locations of unused good columns. Otherwise, it may take a long time or even damage the chip to locate the bad columns. The standard microcontroller software that is shipped with the chip tracks bad columns flagged in the BCT and ensures that these columns are not accessed.

4.3.2.1 Control and Status Registers (CSA and CSB)

CSA and CSB are the 16-bit control and status registers of the PA and the DCU. Their contents are the status of the hardware finite state machine (FSM). They are initialized to 0000h upon power-up and chip reset (by either the host asserting the RESET# pin or writing a 0 to CMR[15]). The microcontroller must write to these registers to provide a valid initial state

or to change the mode of operation. The register settings for the three modes are shown in Figure 4-26 and Figure 4-27.

4.3.2.2 Hardware Setting Registers (MODE and AUX)

The MODE and AUX registers are 16-bit hardware-mode-setting registers. They are used for reading, erasing and programming the flash EPROM cells of the PA. They are initialized to 0000h. Only the values given in Figure 4-28 and Figure 4-29 are valid. Do not set them to any other values.

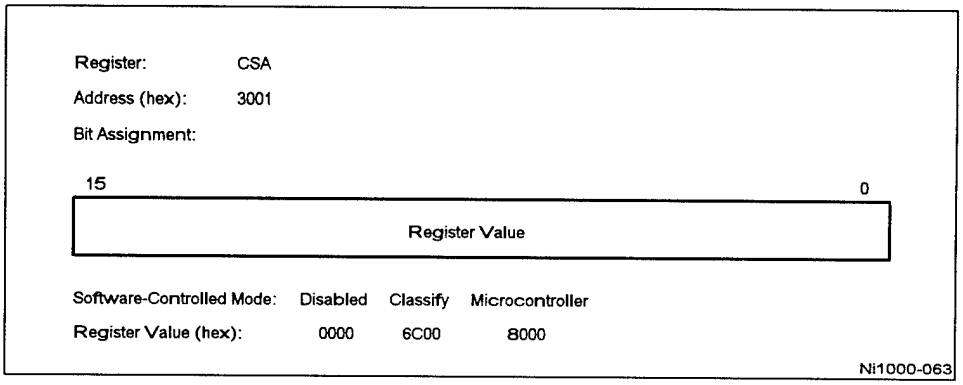


Figure 4-26. The CSA Register

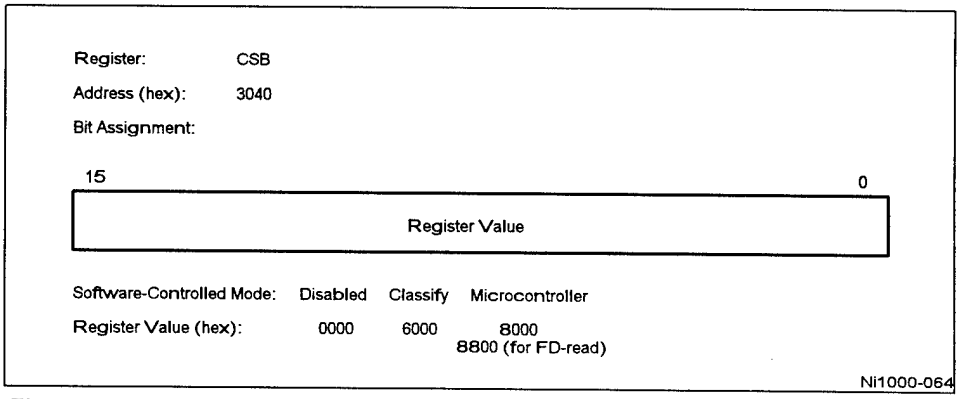


Figure 4-27. The CSB Register

4.3.2.3 Address Relocation Register (ARR)

The PA can store as many as 1000 prototypes of 222 features (padded to 256). When there are fewer than 1000 vectors or fewer than 222 dimensions in each vector, the PA can be segmented into blocks. Each block may store the prototype vectors for a particular application

problem. The 16-bit *ARR* register specifies the starting position in the PA of the block in use. Figure 4-30 shows this register followed by its bit assignments. The column offset gives the starting column number, modulo 128, and the row offset gives the starting row number, modulo 32. This representation results in a total of 64 possible blocks.

Register: MODE

Address (hex): 3002

Bit Assignment:

15	0
Register Value	

Software-Controlled Mode:	Disabled	Classify	Microcontroller
Register Value (hex):	0000	0100	0040 (set flags) 0020 (clear flags) 0080 (read flags)

Ni1000-065

Figure 4-28. The MODE Register

Register: AUX

Address (hex): 3020

Bit Assignment:

15	0
Register Value	

Software-Controlled Mode:	Disabled	Classify	Microcontroller
Register Value (hex):	0000	0000	0000

Ni1000-066

Figure 4-29. The AUX Register

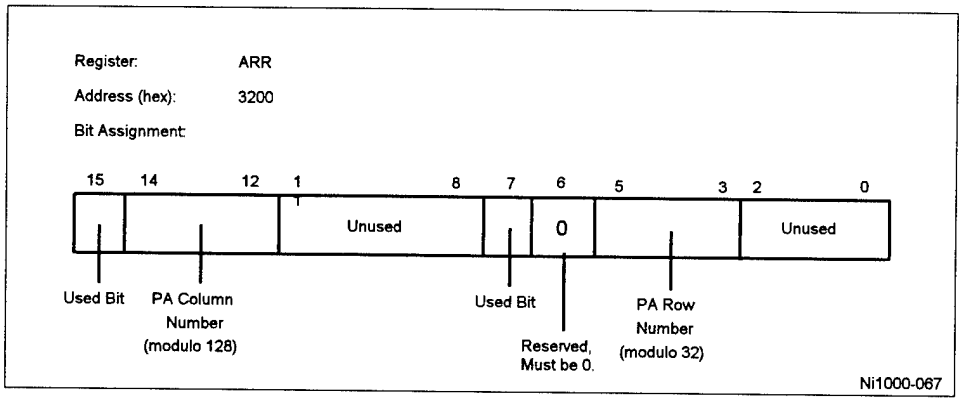


Figure 4-30. The ARR Register

bit(s)	*Function/Value*	**Description**
[0:2]	*Reserved*	
[5:3]	*Row Offset*	Starting row number, modulo 32.
[6]	*Reserved, must be 0.*	Forces high half of 512x512 memory if 1. Used for testing.
7	*Row Relocation Used Bit* 1 0	 PA row relocation is used. Row offset is given by bits [4:6]. PA row relocation is not used.
[8:11]	*Reserved*	
[12:14]	*Column Offset*	Starting column number in PA, modulo of 128.
15	*Column Relocation Used Bit* 1 0	 PA column relocation is used. Column offset is given by bits [12:14]. PA column relocation is not used.

Two additional registers specify the size of the block. *DCU_DIM* contains the dimensions of the prototype vectors in the block. *NCA* contains the index of the last prototype vectors in the region. This index is from 0 to 127, inclusive, when using address relocation in the low half of the prototype array. It is necessary to add 512 to the index when using address relocation in the upper half of the array. Figure 4-31 provides a graphical explanation. The starting position of a region (given by the ARR register) must coincide with the cross-points of the vertical and horizontal lines. The size of a block, however, is limited. When row relocation is enabled, a maximum of 32 features (28 usable) can be used and the block cannot cross the 511/512 boundary. When column relocation is enabled, a maximum of 128 columns are available. This number is reduced by bad columns or if the reserved columns fall within the block.

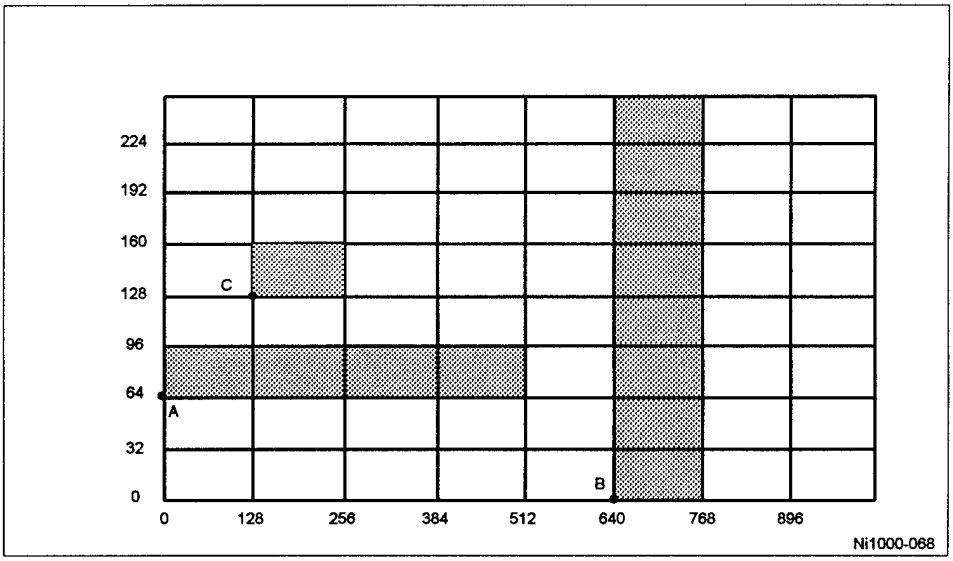


Figure 4-31. Prototype Array Segmentation

Point A is the starting point for a network with row relocation enabled. It is limited to 32 dimensions, but may use up to 512 columns. Point B is the starting point for a network with column relocation. It is limited to 128 columns, but may use all 256 Ni1000 dimensions. Point C is the starting point for a network using both row and column relocation. It is limited by both the 128 prototype limit and the 32 dimension limit.

The column and row offsets given by the ARR register are relative to location B000h, B800h. When the *Column* and *Row Relocation Used Bits* of ARR are 0, PA is addressed as described at the beginning of this section. When the bits are set, addressing is relative to the position in PA specified by the *Column* and/or *Row Offset* bits of ARR.

4.3.2.4 Other Registers (DCU_DIM, NCA and NCB)

The DCU_DIM and NCA registers contain information about the window of columns in use in the PA. These registers are shown in Figure 4-32 and 4-33, respectively. NCA is the index of the last prototype in the active network. As noted in the ARR description, above, this number is affected by column relocation. If column relocation is enabled, the number is in the range of 0 to 127, inclusive (index within the relocated network) if the network is in the low half of the prototype array. If the relocated network is in the high half of the prototype array, a 512 offset must be added to the index within the network. The NCB register, shown in Figure 4-34, contains the clock count in the math unit. This value should be 8*h*.

4.3.3 Prototype Parameter RAMs (PPRAMs)

As each 13-bit city-block distance enters the Math Unit, it is accompanied by 48 bits of parameters for its prototype, which come from the Prototype Parameter RAMs (PPRAMs).

These parameters include several fields, such as the threshold radius. The PPRAMs can only be written by the microcontroller, and appear as three 16-bit banks in its address space. The standard microcode shipped with the Ni1000 provides microcode operations that perrmit external software to indirectly load PPRAM. This is important for externally implemented learning algorithms and for restoring memories from a file.

Figure 4-35 shows the PPRAMs. The PPRAMs occupy addresses 4400h through 53FFh. The fields of the 48-bit word passed to the MU pipeline, broken down by bank, are shown in Figure 4-36.

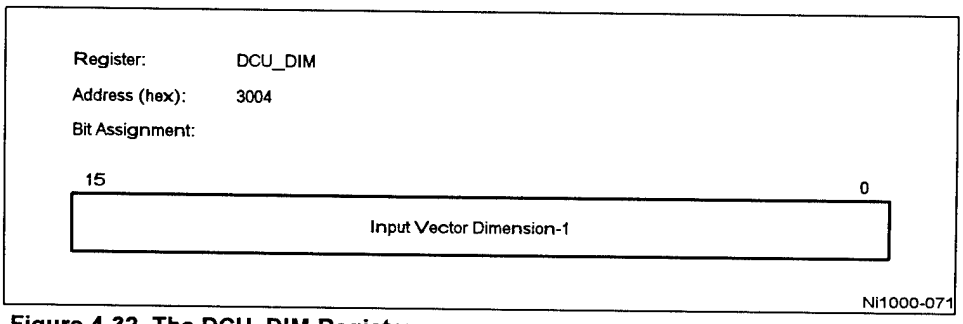


Figure 4-32. The DCU_DIM Register

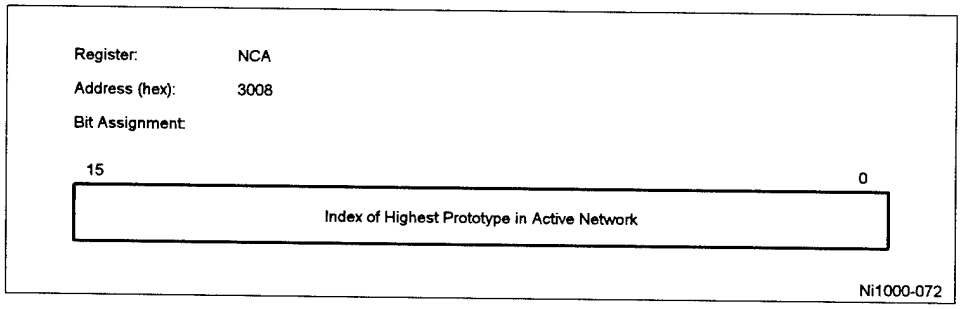


Figure 4-33. The NCA Register

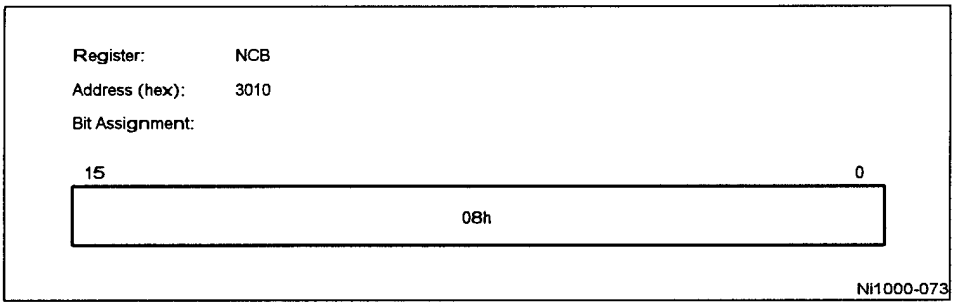


Figure 4-34. The NCB Register

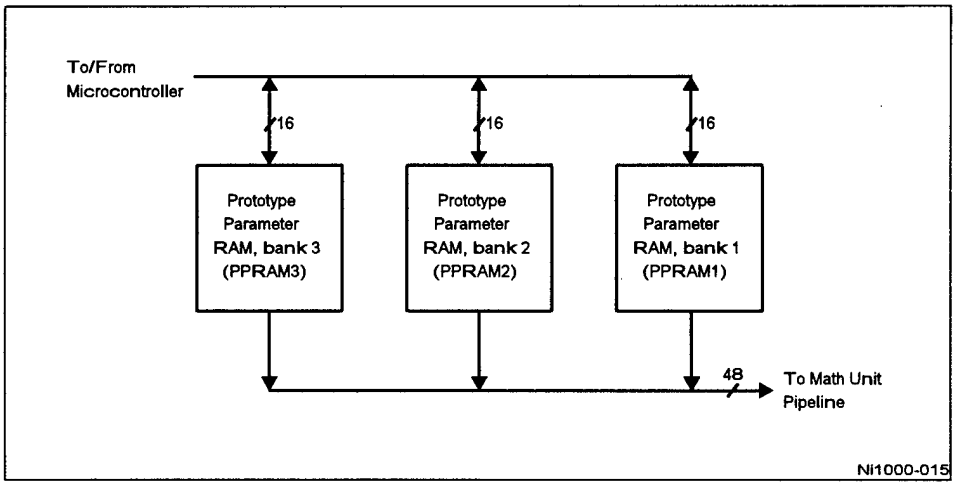


Figure 4-35. Prototype Parameter RAMs (PPRAMs)

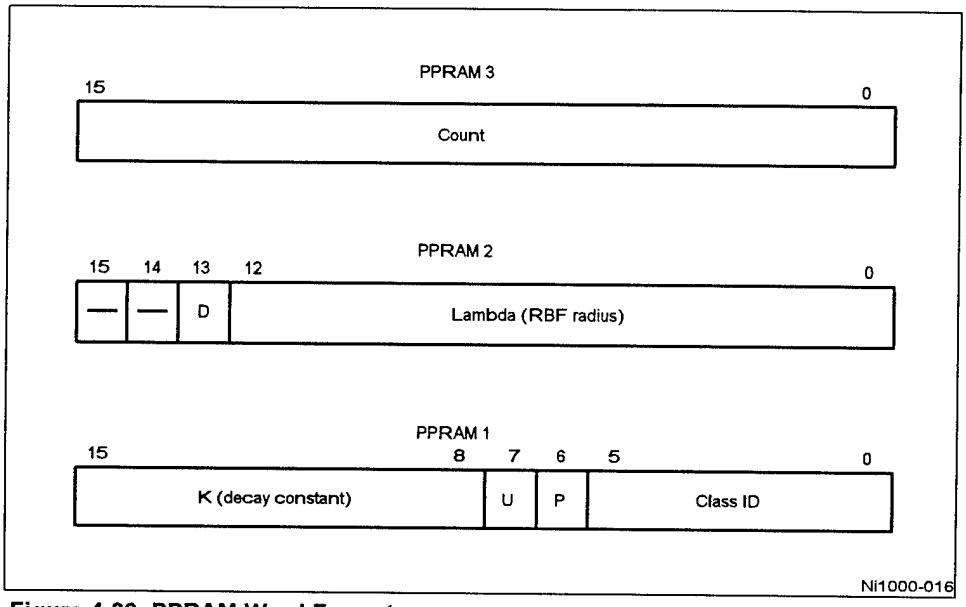


Figure 4-36. PPRAM Word Format

The fields of a PPRAM word are:

- *Count—C[0:15]*—the number of training vectors that fall within this prototype's influence field during the final learning epoch; used as a factor during classification when calculating probability density.
- *Disable Flag—D*—set to disable this prototype.
- *Radius—R[1:12]*—the RBF threshold radius.
- *Smoothing Factor Mantissa—K_m[0:3]*—unsigned mantissa of the smoothing factor of the exponential function.
- *Smoothing Factor Exponent—K_e[4:7]*—signed exponent of the smoothing factor of the exponential function.
- *"Used" Flag—U*—set when the PPRAM word is loaded with a valid prototype.
- *Probabilistic—P*—indicates that the RBF threshold radius for this prototype is the minimum radius. This bit is passed through to indicate that only probabilistic, not deterministic, classification is possible with this prototype. This is only valid when all deterministic prototypes for a given class are located at higher numbered protoypes than any probabilistic prototypes for the same class.
- *Class—S[0:5]*—the class ID of the prototype.

The 4-bit signed exponent of the exponential function's smoothing factor is added to a built-in bias of negative 13 (i.e., 13 is subtracted from the stored value). For example, an exponent of 0 is really an exponent of -13. The value entered must be less than 4. Since a value of -7 to +3 can be entered into this field, the effective exponent is -20 to -10. The 8-bit floating-point value for the smoothing factor has the following characteristics:

- The mantissa has only explicit bits, no implicit leading 1 as in the IEEE floating-point 32-bit format.

- The mantissa's binary point is to the right of the value (bbbb.).
- The exponent binary point is to the right, but the binary point is found by moving the binary point left 13 places from the location indicated by the exponent. Thus, if all 4 bits of the exponent are zero, the mantissa is multiplied by 2^{-13}.
- Zero is represented by a zero mantissa, regardless of the exponent.
- The resulting number is always non-negative.
- The smallest non-zero value is represented by 1111 0001, or $2^{(-7-13)} * 1 = 2^{-20}$.
- The largest value is represented by 0011 1111, or $2^{(3-13)} * 15 = 2^{-10} * (2^4 - 1) = 2^{-6} - 2^{-10}$.
- This floating-point format is only used for the smoothing factor for PRCE and PNN calculations.

Note that the effective range of the smoothing factor exponent is affected by the global offset stored in MURAM_CR[2:5].

Each PPRAM entry corresponds to one prototype vector stored in the Prototype Array. For example, the parameters for prototype number 200h are stored in PPRAM at address 4400h+200h, 4800h+200h, and 5000h+200h.

The standard microcontroller code that is shipped with the chip will copy the PPRAM *Used* flags to the PADCU. As a result, all prototypes that have their *Used* flag set to 1 will be processed by the classifier. To avoid the possibility of spurious data being processed, all locations in the PPRAMs should be written when they are loaded with new prototypes, and unused prototypes should have their *Used* flags cleared to 0.

Figure 4-37 shows the PPRAM registers.

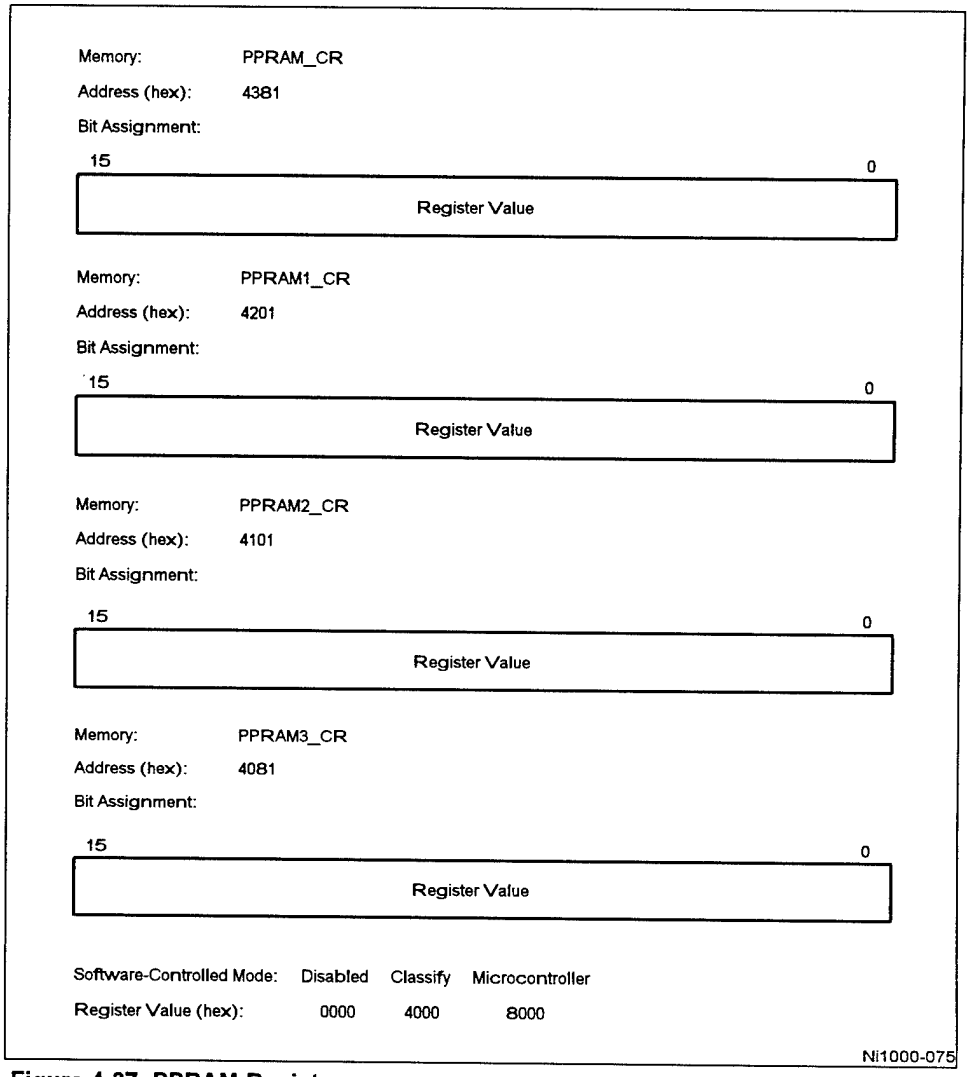


Figure 4-37. PPRAM Registers

4.3.4 Math Unit (MU)

The inputs to the first stage of the Math Unit (MU) pipeline are the fields described above for the PPRAM, accompanied by D, the 13-bit city-block distance calculated between the input vector and the prototype by the DCUs. The MU pipeline has two functions: it determines whether an input vector falls within a prototype's field of influence, and it calculates the

prototype's probability density at the point in feature space described by the input vector. The MU transfer function is described in the next section.

The MU pipeline and the next functional block of the classifier, the math unit RAMs (MURAMs) are closely tied together. Several stages of the pipeline access data in the MURAMs.

In the final stage of the pipeline, the scaled value for the probability density function of a single RBF is added to the accumulated value for all prototypes of the same class. This floating-point sum is written to the probability MURAM after the sixth stage of the MU pipeline, addressed by the sixth-stage class ID.

Once the last probability calculation has been performed for an input vector, the double-buffered MURAMs reverse roles, so that the classification results for the previous vector can be uploaded to the host through ORAM while the next vector is processed. Both the class list and probability densities are computed simultaneously, so either or both can be uploaded to the host without re-running the classification.

4.3.5 Math Unit RAMs (MURAMs)

The output of the MU pipeline is loaded into the MURAMs. The architecture of the MURAMs, shown in Figure 4-38, is intimately tied to the pipeline. The MURAM memories include:

- *Flag MURAM*—a 1 x 64 memory used as a table of classes which have already recognized the input vector being presented. Each entry corresponds to one of the 64 possible classes. It occupies addresses 6200h through 623Fh.
- *Class List MURAMs*—an 8 x 64 x 2 double buffer holding a list of the class IDs of recognized classes. A new byte is allocated every time a new class is encountered. (Unlike the other MURAM memories, the memories in this buffer are not indexed by class ID; they are addressed by counters, so they grow up from address zero.) They occupy addresses 6400h through 647Fh.
- *Probability MURAMs*—a 16 x 64 x 2 double buffer which accumulates the probability value of the input vector for each class. As with the flag MURAM, each MURAM address corresponds to one of the 64 classes. They occupy addresses 6800h through 687Fh.

The MU pipeline sends its data to both the class-list and probability sections of the MURAM bank currently waiting to receive input while the other bank is available to unload data into the ORAM, discussed later. After the input vectors have been processed, the MURAMs change roles.

The flag MURAM is not accessible to the ORAM, so it is reused every cycle. It is indexed by the class ID. When a class is recognized, the bit addressed by the class ID is set. If the bit previously was clear, that indicates the class had not yet been recognized during the processing of the input vector. This causes the class counter to be incremented and allocates a word in the class-list MURAM. The counter keeps a running tally of the number of classes, which is used to address the class-list MURAM when a new word is allocated.

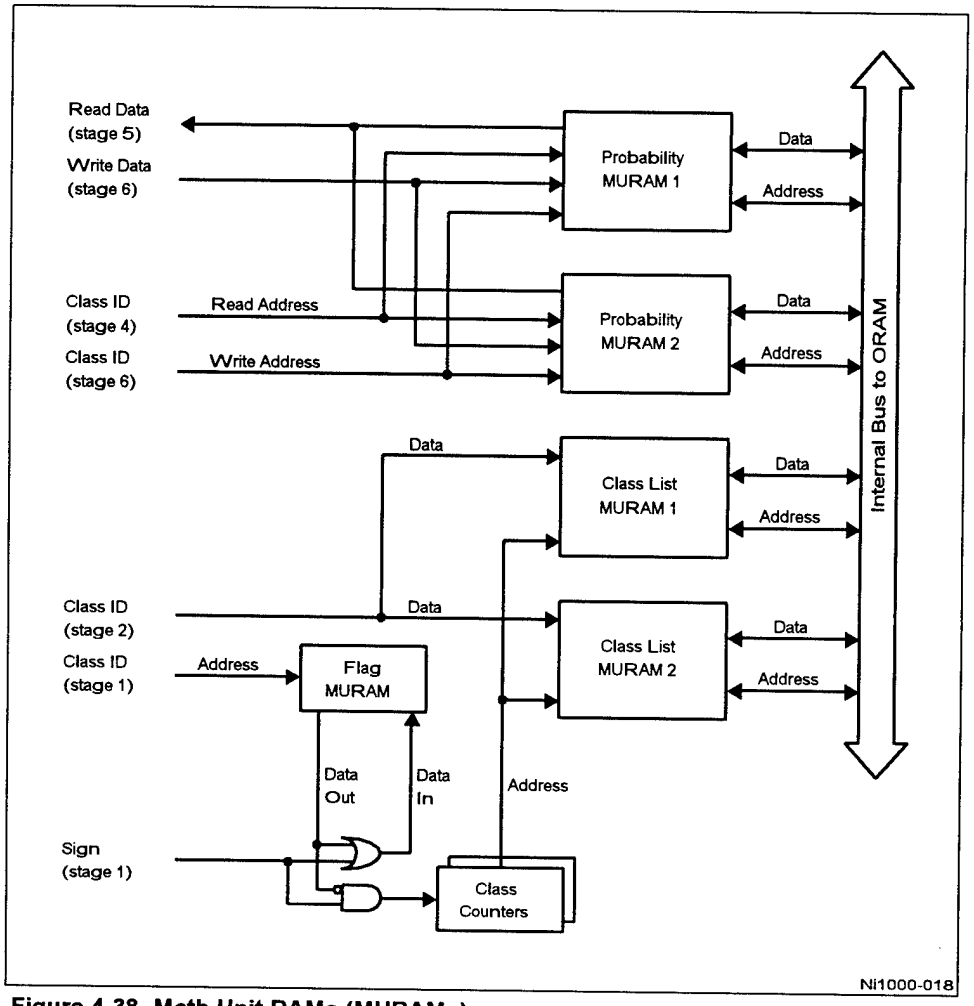


Figure 4-38. Math Unit RAMs (MURAMs)

The class-list MURAMs are 8 bits wide, consisting of a six-bit class ID, a seventh bit to indicate that a prototype with a probabilistic bit set is recognizing the input vector (i.e. a deterministic classification cannot be made with high confidence), and an eighth bit to indicate validity. Figure 4-39 shows the format of a byte in the class-list MURAMs.

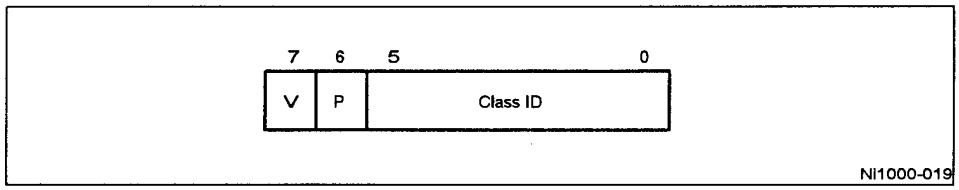


Figure 4-39. Class-List MURAM Word

The fields of a class-list MURAM word are:

- *Class ID S[0:5]*—Class ID of a class that includes the input vector.
- *Probabilistic*—if set, indicates that the first prototype that produced this class firing is known to include at least one training example within its influence field because the minimum radius prevented it from shirinking further. This indicates lack of confidence in the correctness of a deterministic classification, recommended the use of probabilistic classification instead. The *Probabilistic* bit is only meaningful if all probabilistic prototypes of a given class are processed by the DCUs before any deterministic prototypes. Prototypes are processed in the following order:
  1. From the highest column in use below 512 down to 0.
  2. Then, from the highest column in use above 512 down to 512.
- *Valid*—this word has been written since reset initialization.

The class-list MURAMs are addressed by a counter. The counter begins at zero and increments as new classes are encountered.

The probability MURAMs consist of a 16-bit floating-point accumulator in the internal format of the Ni1000 Accelerator. The internal format is provided to the outside world in its internal format or translated into an IEEE-compatible format as it passes out the ORAM. Figure 4-40 shows the internal format of a word in the probability MURAMs.

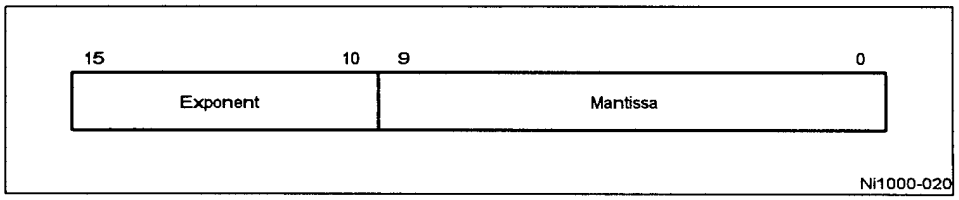


Figure 4-40. Probability MURAM Word

The fields of a probability MURAM word are:

- *Exponent*—six-bit 2's-complement exponent.
- *Mantissa*—10-bit fractional mantissa (i.e. 0 <= mantissa < 1).

Once the last probability calculation has been performed for the current input vector, the pairs of class-list and probability MURAMs are swapped. This allows the processing of a new vector to begin immediately, while the old vector is uploaded to the host. Both the class list and probability densities are computed, so both are available.

The Math Unit transfer function is:

$$2^{-K_m \cdot d \cdot 2^{(K_e + \overline{K}_{off} - 13)}}$$

$$0 \le K_m \le 15$$

where,

K_m = unsigned mantissa of the smoothing factor, K, obtained from PPRAM,
K_e = signed exponent of the smoothing factor, K, obtained from PPRAM,
K_{off} = global offset, stored in MURAM_CR[2:5], which is added to the exponent for <u>every</u> prototype
d = the calculated city-block distance

Register: MURAM_CR
Address (hex): 6100
Bit Assignment:

15	5	2	1	0
	Math Unit Transfer Function Offset			

Software-Controlled Mode:	Classify	Microcontroller
Register Value (hex):	0000-0022	0000
Bit 1	1	0
Bits [2:5]	0000-1000	0000

NI1000-078

Figure 4-41. The MURAM_CR Register

4.4 Microcontroller

The 16-bit, custom microcontroller (MC) has a Harvard architecture (i.e. physically separate instruction and data memories). It is supported with 4K words of flash program memory, 256-words of general-purpose data RAM (GRAM), and a 32-bit timer.

4.4.1 Microcontroller Datapath

Figure 4-42 shows the architecture of the microcontroller datapath. It has four general-purpose registers and a simple instruction set. Instructions consist of one or two 16-bit words.

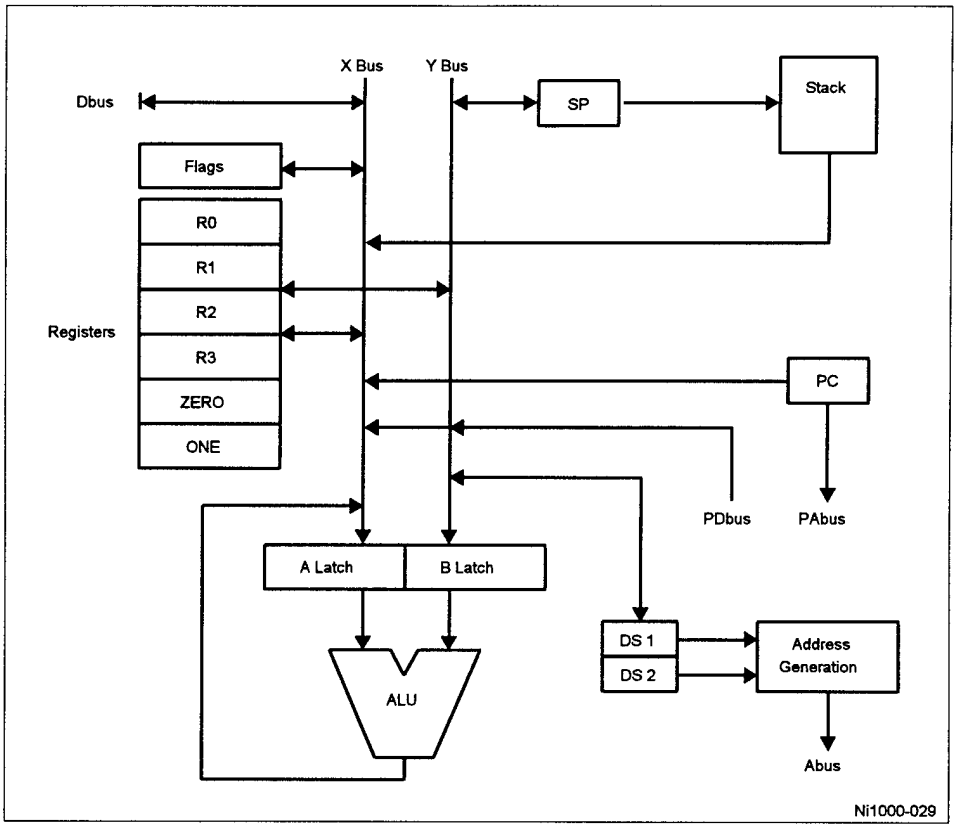


Figure 4-42. Microcontroller Datapath

There are four important microcontroller buses, shown in Figure 4-43:

- *PABUS*—program memory address bus.
- *PDBUS*—program memory data bus.
- *ABUS*—general-purpose address bus.
- *DBUS*—general-purpose data bus.

The PABUS and PDBUS are used to access the microcontroller's flash memory. The ABUS and DBUS access the microcontroller's 256-word RAM, 32-bit timer, and almost all of the registers and memories within the classifier. The 16-bit external data interface to the Bus Control Unit is for PG access modes, since the PGFLASH is 16 bits wide.

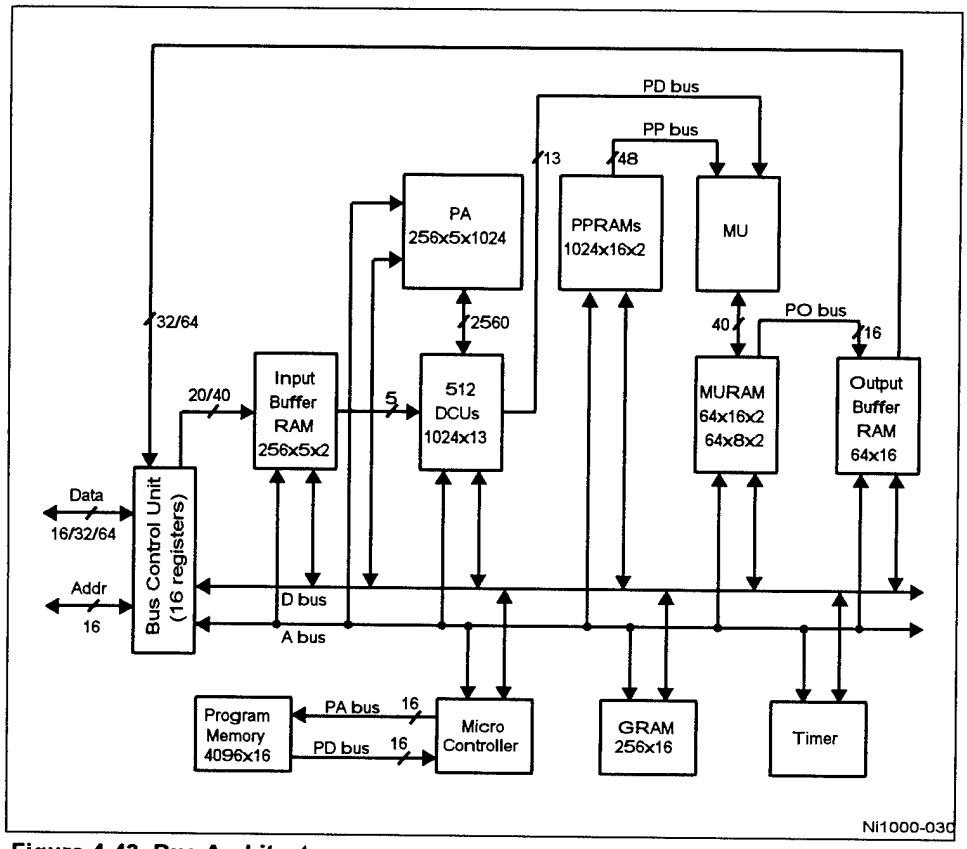


Figure 4-43. Bus Architecture

4.4.2 Microcontroller Registers and Flags

There are 11 microcontroller registers, listed in Table 4-5.

The data segment registers are used for address generation. The stack is 64 locations deep. SP is unsigned, starting with value 0, and points to the location after the current one. The first stack address is 0. Thus only 63 stack locations are available.

The CSW register, shown in Table 4-5, is broken into the flags shown in Table 4-6. If the stack overflows or underflows, the SE and IR flags will be set. If the IE flag is also set, an interrupt will occur. The interrupt will jump to the routine whose address is stored in PPGRAM[0]. All conditional jump instructions test the flags for the corresponding conditions (encoded with a four-bit binary field in the instruction word).

Table 4-5. Microcontroller Registers

Register	Size	Type	Code (hex)	Description
R0	16 bits	W/R	0	General purpose register 0.
R1	16 bits	W/R	1	General purpose register 1.
R2	16 bits	W/R	2	General purpose register 2.
R3	16 bits	W/R	3	General purpose register 3.
ZERO	16 bits	R	6	Always reads 0.
ONE	16 bits	R	7	Always reads 1.
DS1	16 bits	W/R	8	Data segment register 1.
DS2	16 bits	W/R	9	Data segment register 2.
SP	16 bits	W/R	A	Stack Pointer.
PC	12 bits	W/R	-	Program Counter.
CSW	15 bits	W/R	-	Control Status Word.

Table 4-6. Microcontroller Flags

CSW Bit Number	Flag Name	Abbreviation	Code ($flg)
0	Carry	C	0000
1	Zero	Z	0001
2	Negative	N	0010
3	Positive	P	0011
4	Overflow	O	0100
5	Interrupt Request	IR	0101
6	Interrupt Enable	IE	0110
7	Stack Error	SE	0111
8	General Error	GE	1000
9	Multi-Class Firing	MC	1001
10	Flash-Write	FW	1010
11	MURAM1 Ready	M1	1011
12	MURAM2 Ready	M2	1100
13	PADCU Busy	DC	1101

Bits in one of the I/O registers (HS1), correspond to the above listed flags. The register is read by the host to access the condition flags of the microcontroller. In addition to the conditional jump instructions, three other instructions contain a flag-specifier field. Two instructions, SFLG and CFLG, allow any flag to be set or cleared. One instruction, WAIT, suspends execution until a specified flag is set.

4.4.3 Microcontroller Instruction Addressing Modes

The read and write instructions support six addressing modes:

- *Indirect Off DS with 8-bit Offset*—the address is the sum of an 8-bit field in the instruction and either the DS1 or DS2 registers.
- *Indirect*—the address is in R0, R1, R2, or R3.
- *Indirect with Register Offset*—the address is the sum of R0, R1, R2, or R3 and DS1 or DS2.
- *Indirect Autoincrement with Register Offset*—the address is the sum of R0, R1, R2, or R3 and DS1 or DS2. The general register is incremented after the operation.
- *Indirect Autodecrement with Register Offset*—the address is the sum of R0, R1, R2, or R3 and DS1 or DS2. The general register is decremented before the operation.

The jump instructions have the following addressing modes:

- *Immediate*—the address is coded into the instruction.
- *Register*—the value stored in the named register is placed in PC.
- *Relative*—the value, either the named address offset or the value stored at the named register, is added to PC.

4.4.4 Microcontroller Instruction Set

The instruction set includes the following basic arithmetic and logical instructions:

- *Double-Operand Arithmetic Instructions*—add (ADD), add with carry (ADC), subtract (SUB), and compare (CMP).
- *Single-Operand Arithmetic Instructions*—increment (INC) and decrement (DEC).
- *Double-Operand Logical Instructions*—and (AND), or (OR), and exclusive-or (XOR).
- *Single-Operand Logical Instructions*—complement (NOT), shift left (SHL), shift right (SHR), rotate left (ROTL), and rotate right (ROTL).

Arithmetic and logical instructions only operate on 16-bit register operands. There are no operations on memory operands, other than reading data into, or writing from, a register.

The six data movement instructions are: move (MOV), load (LD), read (RD), write (WR), push (PUSH), and pop (POP). The move instruction only transfers data from one register to another. The load instruction puts a 16-bit immediate operand following the instruction word into a register. The read and write instructions transfer a word between a memory location and a register. There are many flavors of the latter two instructions, to support a variety of addressing modes. The push and pop instructions transfer data between a register and the top of the stack. The stack is a separate 16-bit wide memory space consisting of 64 words.

Eight conditional jumps utilize the flags described above. The address of the target of the jump can be calculated using a register or a 16-bit immediate operand following the instruction. The address can be absolute or PC-relative; *i.e.* the address can replace the contents of the program counter, or it can be added to it. Both positive and negative polarities are supported for each condition. All combinations of these three options are supported.

Ten conditional jumps first test common microcontroller conditions then jump to an address specified with a PC-relative 8-bit address embedded in the instruction. Four unconditional jumps result from combinations of where the address comes from (i.e. register or immediate) and how it is applied to the PC (i.e. absolute or relative).

Five "unconditional jump to subroutine" (JS) instructions are provided. They push the program counter on the stack, then jump to an absolute address specified by a register or 16-bit immediate, or a PC-relative address specified by a register, 16-bit immediate, or 8-bit field embedded in the instruction word.

In the following sections, microcontroller instructions are grouped according to their functions. The mnemonics and English descriptions are given. See the *Ni1000 Recognition Accelerator User's Guide* for more details on the instructions.

4.4.4.1 Subroutine Calls

JS	Jump to Subroutine
JSR	Jump to Subroutine Register
JSRR	Jump to Subroutine Register Relative
JSI	Jump to Subroutine Immediate
JSIR	Jump to Subroutine Immediate Relative
RETS	Return from Subroutine

4.4.4.2 Stack Operations

PUSH	Push
POP	Pop

4.4.4.3 Conditional Jumps

Flag Condition	Short Immediate Relative	Register	Register Relative	Long Immediate	Long Immediate Relative
Unconditional	JMP	JMPR	JMPRR	JMPI	JMPIR
Carry	JC	JCR	JCRR	JCI	JCIR
Zero	JZ	JZR	JZRR	JZI	JZIR
Negative	JN	JNR	JNRR	JNI	JNRI
Positive	JP	JPR	JPRR	JPI	JPIR
Overflow	JO	JOR	JORR	JOI	JOIR
Interrupt Request	JIR	JIRR	JIRRR	JIRI	JIRIR
Interrupt Enable	JIE	JIER	JIERR	JIEI	JIEIR
Stack Error	JSE	JSER	JSERR	JSEI	JSEIR
General Error	JGE	JGER	JGERR	JGEI	JGEIR
Multi-Class Firing	JMC	JMCR	JMCRR	JMCI	JMCIR
Flash Write	JFW	JFWR	JFWRR	JFWI	JFWIR
MURAM 1 Ready	JM1	JM1R	JM1RR	JM1I	JM1IR
MURAM 2 Ready	JM2	JM2R	JM2RR	JM2I	JM2IR
PADCUs Busy	JDC	JDCR	JDCRR	JDCI	JDCIR
No Carry	JNC	JNCR	JNCRR	JNCI	JNCIR
No Zero	JNZ	JNZR	JNZRR	JNZI	JNZIR
No Negative	JNN	JNNR	JNNRR	JNNI	JNNIR
No Positive	JNP	JNPR	JNPRR	JNPI	JNPIR
No Overflow	JNO	JNOR	JNORR	JNOI	JNOIR
No Interrupt Request	JNIR	JNIRR	JNIRRR	JNIRI	JNIRIR
No Interrupt Enable	JNIE	JNIER	JNIERR	JNIEI	JNIEIR
No Stack Error	JNSE	JNSER	JNSERR	JNSEI	JNSEIR
No General Error	JNGE	JNGER	JNGERR	JNGEI	JNGEIR
No Multi-Class Firing	JNMC	JNMCR	JNMCRR	JNMCI	JNMCIR
No Flash Write	JNFW	JNFWR	JNFWRR	JNFWI	JNFWIR
No MURAM 1 Ready	JM1	JM1R	JM1RR	JM1I	JM1IR
No MURAM 2 Ready	JM2	JM2R	JM2RR	JM2I	JM2IR
No PADCUs Busy	JDC	JDCR	JDCRR	JDCI	JDCIR

4.4.4.4 Flag Operations

RDFLG	Read Flags
WDFLG	Write Flags
SFLGC	Set Carry Flag
SFLGZ	Set Zero Flag
SFLGN	Set Negative Flag
SFLGP	Set Positive Flag
SFLGO	Set Overflow Flag
SFLGIR	Set Interrupt Request Flag
SFLGIE	Set Interrupt Enable Flag
SFLGSE	Set Stack Error Flag
SFLGGE	Set General Error Flag
SFLGMC	Set Multi-Class Firing Flag
SFLGFW	Set Flash Write Flag
SFLGM1	Set MURAM 1 Ready Flag
SFLGM2	Set MURAM 2 Ready Flag
SFLGDC	Set PADCUs Busy Flag
CFLGC	Clear Carry Flag
CFLGZ	Clear Zero Flag
CFLGN	Clear Negative Flag
CFLGP	Clear Positive Flag
CFLGO	Clear Overflow Flag
CFLGIR	Clear Interrupt Request Flag
CFLGIE	Clear Interrupt Enable Flag
CFLGSE	Clear Stack Error Flag
CFLGGE	Clear General Error Flag
CFLGMC	Clear Multi-Class Firing Flag
CFLGFW	Clear Flash Write Flag
CFLGM1	Clear MURAM 1 Ready Flag
CFLGM2	Clear MURAM 2 Ready Flag
CFLGDC	Clear PADCUs Busy Flag
WAIT	Equivalent to WAITIR
WAITC	Wait For Carry Flag
WAITZ	Wait For Zero Flag
WAITN	Wait For Negative Flag
WAITP	Wait For Positive Flag
WAITO	Wait For Overflow Flag
WAITIR	Wait For Interrupt Request Flag
WAITIE	Wait For Interrupt Enable Flag
WAITSE	Wait For Stack Error Flag
WAITGE	Wait For General Error Flag
WAITMC	Wait For Multi-Class Firing Flag
WAITFW	Wait For Flash Write Flag
WAITM1	Wait For MURAM 1 Ready Flag
WAITM2	Wait For MURAM 2 Ready Flag
WAITDC	Wait For PADCUs Busy Flag

4.4.4.5 Data Transfer Operations.

MOV	Move Register to Register
LDI	Load Immediate
RDI	Read Immediate
RDR	Read Indirect Register
RD1	Read Indexed Base Register using DS1 as base, 8 bit immediate as offset.
RD2	Read Indexed Base Register using DS2 as base, 8 bit immediate as offset.
RD1R	Read Indexed Base Register using DS1 as base, named register for offset.
RD2R	Read Indexed Base Register using DS2 as base, named register for offset.
RD1RI	Read Indexed Base Register using DS1 as base, named register for offset.
RD2RI	Read Indexed Base Register using DS2 as base, named register for offset.
RD1RD	Read Indexed Base Register using DS1 as base, named register for offset.
RD2RD	Read Indexed Base Register using DS2 as base, named register for offset.
WRI	Write Immediate
WRRI	Write Indirect Register Immediate
WRR	Write Indirect Register
WR1	Write Indexed Base Register using DS1 as base, 8-bit immediate as offset.
WR2	Write Indexed Base Register using DS2 as base, 8-bit immediate as offset.
WR1R	Write Indexed Base Register using DS1 as base, named register for offset.
WR2R	Write Indexed Base Register using DS2 as base, named register for offset.
WR1RI	Write Indexed Base Register using DS1 as base, named register for offset.
WR2RI	Write Indexed Base Register using DS2 as base, named register for offset.
WR1RD	Write Indexed Base Register using DS1 as base, named register for offset.
WR2RD	Write Indexed Base Register using DS2 as base, named register for offset.

4.4.4.6 Math and Logical Operations.

ADD	Add
ADC	Add with Carry
SUB	Subtract
CMP	Compare
INC	Increment
DEC	Decrement
NOT	Logical Negation
OR	Logical Or
XOR	Logical Exclusive Or
SHL	Shift Register Left
SHR	Shift Register Right
ROTL	Rotate Register Left
ROTR	Rotate Register Right
NOOP	No Operation

4.4.5 Microcontroller Interrupt Handling

The Ni1000 on-chip microcontroller has a single interrupt vector at address F000h, which is the beginning of the microcontroller program memory PGFLASH. The program counter PC is initialized to 1 instead of 0 for this reason.

The microcontroller interrupt request (IR) flag, which is CSW[5], is set when one of the following occurs:

- The host writes to CMR, with CMR[15]=0
- The host writes to IIR
- The host asserts the MCINT# pin
- The host asserts the ERROR# pin
- The microcontroller clears the General Error flag (CSW[8]) after setting it.

Note that the value written into CMR or IIR does not matter; any write to these registers causes IR to be set (except, of course, if CMR[15] is set, which resets the chip). For host interrupts of the micrcontroller, IIR[2:3] can be used to communicate the type of interrupt to the interrupt handler.

There is no acknowledge pin for the microcontroller interrupts. The host can determine that the microcontroller has acknowledged the interrupt request in several ways; see the example given at the end of this section.

When the interrupt is requested, the IR flag is set immediately. IR is cleared when the microcontroller software acknowledges the interrupt. If multiple interrupts occur before the microcontroller software acknowledges the first one, only one interrupt will occur.

Interrupt request is serviced when the interrupt enable (IE) flag is set. Service is provided in the following steps:

- Clear IE flag.
- Read interrupt service routine (ISR) entry from the first location of PGFLASH, at address F000h.
- Jump to subroutine ISR.

Clearing the IR flag and setting the IE flag should be done by software.

A service request by the microcontroller to the host is indicated by the SRQ# pin. It is asserted when the microcontroller writes to the XIR register. SRQ# is deasserted when the host asserts the IACK# pin. An outstanding service request is indicated by HS2[2].

4.4.6 Reset Initialization

When the microcontroller is reset, the PC is initialized to address 0001h in the PGFLASH to begin execution. The microcontroller can be reset by setting the MSB of the chip mode register (CMR). Setting this bit puts the entire chip into reset mode; clearing the bit clears the reset condition. The bit is set automatically when the RESET# signal is asserted.

There are separate reset bits for the IRAM and ORAM. The IRAM and ORAM are unavailable to the microcontroller or the classifier while in reset mode. The reset bits must be cleared to enable operation of the classifier. Before they are cleared, the following things must be initialized:

- *32/64-Bit Bus Width*—the width of the external data bus, as defined by the level on the 64/32# signal.

- *Output Mode*—whether class IDs or probability densities are the output, controlled by a bit in the CRA register.
- *Vector Length*—the number of features in an input vector, loaded into the DIM register. For probabilistic classification, the number of desired classes to upload must also be initialized in this register.
- *Floating-Point Format*—for probabilistic classification, whether the native 16-bit floating-point format or the IEEE-compatible 32-bit format is used for output, controlled by a bit in the CRB register.

4.4.7 Errors

Three types of errors can occur:

- *External Error*—an attempt by the host to write to a full IRAM or read from an empty ORAM, or the assertion by external logic of the SRQ# signal. The BERR# signal is asserted.
- *Internal Stack Error*—microcontroller has overrun or underrun the stack space. The stack is a physically separate 16 x 64 RAM. Popping an empty stack or pushing onto a full stack causes an error that asserts the stack-error flag.
- *Internal General Error*—microcontroller has overrun or underrun a buffer, which asserts the general-error flag.

4.5 System-Level Architecture

Because the Accelerator is addressed like memory, it will always be a bus slave. There are several system design options available for the Accelerator:

- *Processor Bus*—placement directly on the bus with the processor.
- *Local Bus*—interface through a local bus standard, such as PCI or VL-Bus.
- *Expansion Bus*—a standard interface for expansion cards, such as the ISA bus, the EISA bus, or Micro Channel.
- *Hardwired*—a dedicated interface to an embedded controller, such as the i960 family of embedded RISC processors. The controller could also be an ASIC.

The Accelerator has an input signal, MULTICHIP#, to inform microcontroller software that it is in a multi-chip system. The microcontroller can test this condition in the CRB register

Multi-chip systems must contend with two issues. During training, when a new prototype is allocated, they must collect D_{min} (the city-block distance between the input vector and the nearest prototype vector of a different class) from all Accelerators to find the global D_{min} used to initialize the threshold radius of the prototype. The second issue occurs during probabilistic classification. The probability densities from each chip must be uploaded to the host to be combined.

4.6 Classification Timing

The timing of the classification pipeline varies with the number of features in the input vector and the number of valid prototypes. If the latency is short enough, the main source of delay will be I/O.

Figure 4-44 shows pipelined processing of three vectors with up to 500 prototypes. If the host system is not a limiting factor or if the number of features is low, filling one bank of the IRAM will be very quick compared to processing the vector. As soon as the first vector is loaded, it can be dispatched for processing while the second vector is loaded.

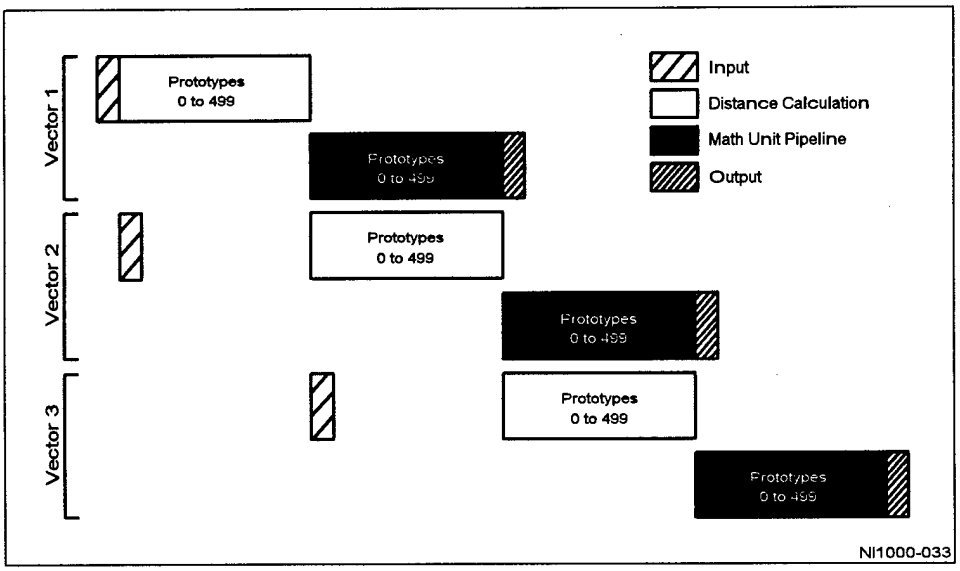


Figure 4-44. Pipeline Usage (For Up To 500 Prototypes)

After the first vector is processed by the distance calculation units, the IRAM buffers can swap and the third vector can be loaded. Meanwhile, the first vector is in the pipeline comprising the MU and MURAMs and the second vector is in distance calculation. As each vector is processed by the MU pipeline, it can quickly be loaded into the ORAM.

Figure 4-45 shows pipeline operation for over 500 prototypes. Operation of the MU pipeline is fully overlapped with distance calculation. Up to 1000 prototypes are run against the input vector in two phases, up to 500 per phase. Up to 500 features are compared every two clocks.

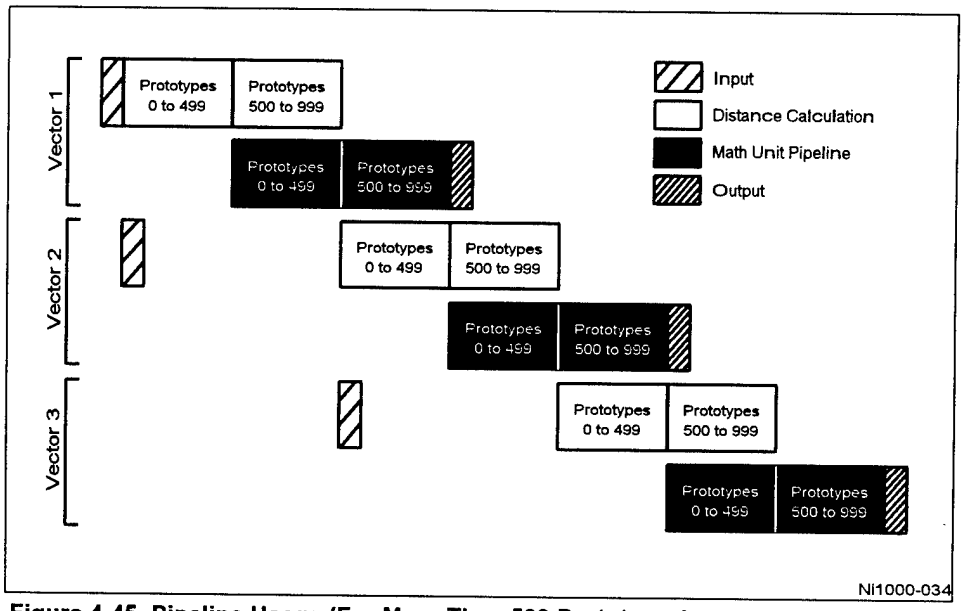


Figure 4-45. Pipeline Usage (For More Than 500 Prototypes)

The number of clocks required by each stage of the pipeline can be estimated from the following expressions.

Input—(8L / B) + 3
Distance Calculation—2L + I, if P <= 512
Distance Calculation—2(2L + I), if P > 512
MU Pipeline—P + I
Output—(CR / B) + C + 3

The parameters of these expressions are shown below.

L—Vector length.
P—Number of valid prototypes.
B—Bus size (32 or 64).
C—Number of classes required.
R—Size of classification results (8 for RCE, 16 or 32 for PRCE)
I—Initialization time, about 5 clocks for each block.

4.7 Computational Precision

The computational precision of variables is listed in Table 4-7. Floating-point probability densities appear in one of two formats, 16-bit internal or 32-bit IEEE, as described in the immediately following sections.

Table 4-7. Computational Precision

Variable	Mantissa		Exponent		Smallest Representable Non-Zero Value*	Largest Representable Value*	Actual Smallest Non-Zero Value*	Actual Largest Value*
	bits	format	bits	format				
v	5	00000	x	x	1	2^5-1	x	x
d	13	0...0	x	x	1	$2^{13}-1$	x	x
λ	13	0...0	x	x	1	$2^{13}-1$	x	x
K_{off}	4	0000	x	x	0	8	x	x
k	4	0000	4	s000	2^{-20}	$15*2^9$	2^{-20}	$15*2^{-2}$
kd	10	.0...0	6	s00000	2^{-20}	$2^{32}-2^{22}$	2^{-20}	$2^{18}-2^{14}$
2^{-kd}	10	.0...0	6	s00000	2^{-37}	$2^{32}-2^{22}$	2^{-37}	1
C	16	0000	x	x	1	$2^{16}-1$	1	$2^{16}-1$
$C*2^{-kd}$	10	.0...0	6	s00000	2^{-37}	$2^{32}-2^{22}$	2^{-37}	$2^{16}-1$
$\Sigma C*2^{-kd}$	20	.0...0	6	s00000	2^{-37}	$2^{32}-2^{12}$	2^{-37}	$2^{26}-2^{10}$
IEEE-754 single precision real	23	1.0...0	8	s0...0	2^{-127}	2^{128}	x	x

s = sign bit
x = not applicable or not available
* = Representable Value—Value that can be represented by the internal number format.
Actual Value—Value that is supported by the chip, external sotware or microcode must enforce limits on K_e and K_{off}

4.7.1 16-Bit Internal Format

Figure 4-46. shows the MU internal format used for floating-point numbers. In this format, the 6-bit exponent uses 2's complement to represent negative numbers. The number represented is $0.F^* \times 2^{\pm E^*}$, where the MSB of F^* is 1 unless the entire number is zero.

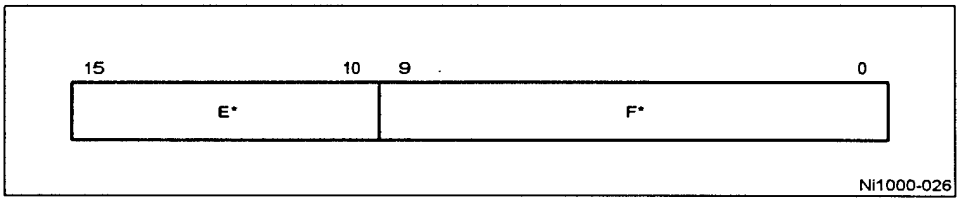


Figure 4-46. Internal 16-Bit Floating-Point Format

4.7.2 32-Bit IEEE Format

Figure 4-47 shows the IEEE 32-bit format for floating-point numbers. The number represented is $(-1)^S \times 1.F \times 2^{E-127}$. There is no restriction on the MSB of F.

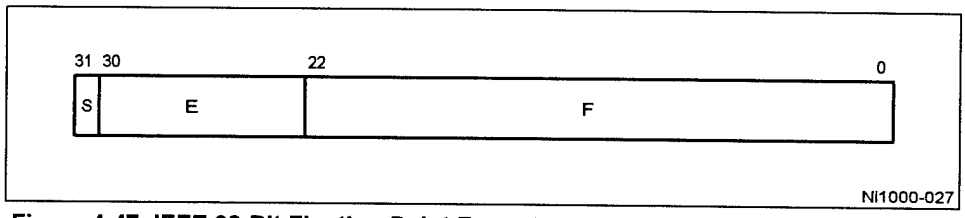


Figure 4-47. IEEE 32-Bit Floating-Point Format

Bit 4 of the CRB register selects the output format. If set, the IEEE 32-bit format is used. If clear, the internal 16-bit format is used. The conversion from the 16-bit format to the 32-bit format is accomplished through the mapping:

$$S = 0$$
$$E = E^*_* + (5e)_{16}, \quad \text{if } E^*_* \text{ is negative}$$
$$\quad\;\; E^*_* + (7e)_{16}, \quad \text{if } E^* \text{ is positive}$$
$$F = F^* \times (4000)_{16}$$

4.8 Software Modes and Configurations

This section discusses how to program the Accelerator. Adherence to the guidelines given below is necessary so as to avoid damage to the Accelerator.

There are specific instruction sequences to perform the operations on some logic blocks, such as I/O access, Prototype Array access, PPRAM access, PGFLASH programming, Classification and Learning. It is crucial that logic blocks are set into appropriate modes and certain registers or bits in registers are written with proper values. See the *Ni1000 Recognition Accelerator User's Guide* for details.

4.8.1 General Programming Guidelines

1. After power up, the Accelerator is in 'reset' status internally, even if the RESET# signal is deasserted. The host must take the Accelerator out of the reset state by writing a 0 to CMR[15] before any operation is attempted.
2. When the Accelerator is released from the reset state, the program counter (PC) contains a 1, so microcontroller code starts execution at location 1 in the PGFLASH (address F001h).
3. All volatile memories (RAMs, registers) are undefined after changing the hardware mode (see Table 5-1) and must be updated by the microcontroller or the host.
4. Anytime the Ni1000 is changed from *PG* mode to *NORMAL* mode, the Ni1000 will be reset and must be released by writing a 0 to CMR[15] after MC# is deasserted.
5. PGFLASH must contain a valid microcontroller program before use. Otherwise, the microcontroller may cause unexpected results or be damaged.
6. Data in PPRAMs are valid as long as RESET# is deasserted and power is up. PPRAM data may be lost after switching into *PG* mode.
7. Always remove IRAM and ORAM from their reset states before use, by writing a 0 to CRB[0] and CRB[1], respectively.

4.8.2 Software-Controlled Modes

Each logic block has its own software-controlled modes. These modes determine the status of the blocks and the functions they can provide. Setting the software-controlled mode is necessary since the hardware modes (see Table 5-1) only determine the pin grouping, the address mapping and the bus operations; they have little control over individual blocks. Tasks such as classification or learning can only be performed with appropriate combinations software-controlled and hardware modes. Accessing the software-controlled modes is accomplished by writing to appropriate registers associated with each logic block.

Table 4-8 summarizes the software-controlled modes for the relevant logic blocks, along with the register values for the corresponding modes. Usually, setting the software-controlled mode is one of many steps in the operation of a logic block.

Table 4-8. Software Mode Configuration

Logic Block	Register			Register Value Under Software-Controlled Mode (hex)		
	Name	Address (hex)	Written by	Classify	MC	Global Reset
PADCU	CSA	3001	MC	6C00	8000	0000
	CSB	3040	MC	6000	8000 8800 (FD-Read)	0000
	MODE	3002	MC	0100	0040 (set flag) 0020 (clear flag) 0080 (read flag)	0000
	AUX	3020	MC	0000	0000	0000
MURAMs	MURAM_CR	6100	MC	Bit 1=1 Bits{2:5]= 0000 - 1000	0000	0000
PPRAMs	PPRAM_CR	4381	MC	4000	8000	0000
I/O	CRA	0040	MC or Host	CRA[0]=1	CRA[0]=0	FFFF
	CRB	0048	MC or Host	CRB[0]=0 CRB[1]=0 CRB[5]=1 CRB[6]=1	CRB[0]=0 CRB[1]=0 CRB[5]=0 CRB[6]=0	CRB[0]=1 CRB[1]=1
All Blocks	CMR	0000	MC or Host	CMR[15]=0	CMR[15]=0	CMR[15]=1

MC = Ni1000 on-chip microcontroller
PADCU = Prototype Array and Distance Calculation Unit
MURAMs = Math Unit RAMs
PPRAMs = Prototype Parameter RAMs
I/O = IRAM, ORAM, and I/O Registers
FD-Read = DCU Used flags and distance read, a submode of MC software-controlled mode.

5. Bus Operations

This chapter discusses the interaction between a host system and the Ni1000 Recognition Accelerator through various bus operations. Figure 5-1 shows the Accelerator's buses. Externally, the I/O unit (IRAM, ORAM, and I/O registers) connects to the host system through the signal pins. Internally, the I/O unit connects to the following buses:

- *Data I/O Bus* (DIO Bus)—Connects to IRAM, ORAM and I/O registers. The data I/O datapath is either 32- or 64-bit wide.
- *Internal Address and Data Buses* (Abus and Dbus)—Serves as the data path of the microcontroller. The Abus is a 16-bit address bus, and the Dbus is a 16-bit data bus.
- *Microcontroller Program Address and Data Buses* (PAbus and PDbus)—Serves as the internal instruction path of the microcontroller. The PAbus is a 16-bit address bus, and the PDbus is a 16-bit instruction (data) bus.

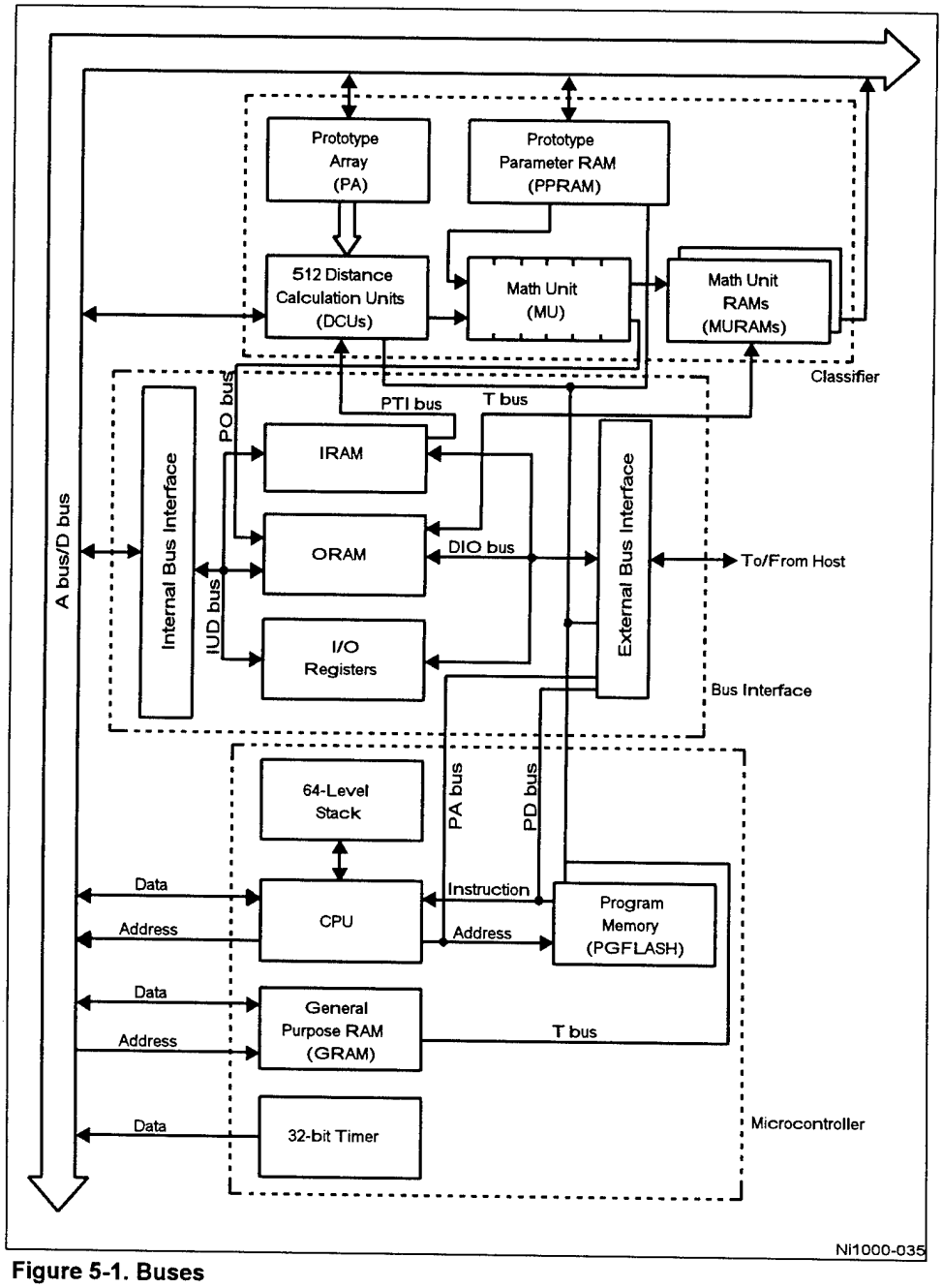


Figure 5-1. Buses

5.1 Hardware Modes

The Ni1000 Accelerator supports 32-bit or 64-bit data output at a bus clock of up to 25 MHz. A shared clock synchronizes the bus interface between the Accelerator and host. A set of modes for access to the Accelerator determines which internal buses are accessible to the host, and which pin groups are used in data transfers with the host. The following list summarizes the modes supported by the Accelerator:

NORMAL—Used for classification and learning.
PG—Used to read or load microcontroller programs into program memory (PGFLASH).
RESET—Used to suspend precharging and most latching, and re-initialize state machines.

After power up, the Accelerator idles in a reset state and must be removed by the host by writing a 0 to bit 15 of the CMR register (see the "Architecture" chapter for details), in addition to setting the appropriate signals as described in the following. The address space accessible by the host in these modes may be one of the following:

- I/O unit (access IRAM through address 2000h, ORAM through 2800h, and I/O registers).
- PGFLASH and associated registers.

Other memory locations and registers can only be accessed by the microcontroller. See the "Architecture" chapter for details. The address pins A[0:15] are mapped to the corresponding internal address space. Table 5-1 summarizes the control-signal settings and mapping of data and address pins for the access modes.

Table 5-1. Hardware Modes

Mode	Access Mode Control Signals					Data Bus[3]			Address Bus
	CS#	W/R#	MC#	RESET#	64/32#	D[0:15]	D[16:31]	D[32:63]	A[0:15]
NORMAL Write	0	1	1	1	1 or 0	DIO[0:15]	DIO[16:31]	DIO[32:63]	AIO[0:15]
NORMAL Read	0	0	1	1	1 or 0	DIO[0:15]	DIO[16:31]	DIO[32:63]	AIO[0:15]
PG[1] Write	0	1	0	1	0	PDBUS[0:15]	undefined	undefined	PABUS[0:15]
PG[1] Read	0	0	0	1	0	PDBUS[0:15]	undefined	undefined	PABUS[0:15]
RESET[2]	X	X	X	0	X	inactive	inactive	inactive	inactive

1. During the PG mode, the internal logic (except PGFLASH contents and registers) is reset.
2. During RESET, the internal logic (including PGFLASH registers) and the external bus interface are reset, except the non-volatile PGFLASH contents.
3. When 64/32# is asserted, 32-bit data bus (D[0:31]) is selected. When 64/32# is deasserted, 64-bit data bus (D[0:63]) is selected.

When used for output, the data-bus bits are grouped as one of the following:

- 8-bit data for firing class IDs.
- 16-bit data for unformatted probabilities.
- 32-bit data for floating-point numbers.
- 64-bit data for two floating-point numbers.

When used for input, the data-bus bits are grouped as:

- 8-bit data for input vector components.
- 16-bit data for register contents and microcontroller instructions.

See the "Architecture" chapter for details. Input vector components are aligned to the high order 5 bits for each byte. For example, the first 5-bit component in each input vector should be transferred to the Accelerator on pins D[3:7], with D7 receiving the most-significant bit of the component. The three least significant bits, D[0:2], are ignored.

The Accelerator does not always require the use of all 64 data I/O bits for transfers with the host. Examples include:

- The Accelerator is operated in a system with a 32-bit data path. In this case, the upper 32 data pins are not used in the data transfer.

- The internal resources being accessed have a 16-bit data path. This is the case in the PG modes; the PDbus and the Dbus are only 16 bits wide.

5.1.1 Normal Mode (NORMAL)

NORMAL mode has the following characteristics:

- The Accelerator may perform classification, learning or housekeeping.
- The Accelerator acts as a slave processor in interactions with the host.
- The Accelerator uses its SRQ# signal to request service. See Chapter 5 of the *Ni1000 Recognition Accelerator User's Guide* for details of microcontroller interrupt handling.
- Input vectors may be sent to the Accelerator and classification results may be received by the host.
- Commands may be written to the Accelerator and status read by the host.
- The classification pipeline may be enabled or the Accelerator may operate under control of the microcontroller for learning or housekeeping.

The host writes and reads via the I/O unit's buffers, namely the IRAM, ORAM, and I/O registers. The width of the data path could be either 64 or 32 bits, as indicated by the 64/32# signal. See Chapter 5 for the accessible address spaces. Accessing addresses other than those for the IRAM, ORAM and I/O registers will yield invalid results.

NORMAL mode is initiated by deasserting the MC# signals. The direction of data transfer (input or output) is indicated by the W/R# signal.

5.1.2 PGFLASH Access Modes (PG Modes)

There are two PG modes, as described in the following sections. In these modes, the host can write or read the PGFLASH control registers through the microcontroller. This allows control, reading, programming, and erasing of the PGFLASH in order to upload or download the microcontroller program. In these modes, the internal logic (except PGFLASH contents and

registers) is reset. The external bus interface is not reset. The reset condition is maintained when NORMAL mode is re-entered from PG mode.

5.1.2.1 PGFLASH Program Mode

In this mode, the microcontroller is halted while its PGFLASH memory is written by the host. This mode is initiated by asserting MC# and driving W/R# high (using Vpp) to indicate a write operation. After MC# is subsequently deasserted, the microcontroller remains in the reset state until a command is written to the CMR register that clears bit 15 of that register to zero.

The A[0:15] pins are used to address the memory locations into which instructions are to be written. Addresses drive the internal Program Address Bus (PAbus) and are latched into a program address register associated with the PGFLASH. Addresses are latched coincident with writing of the instruction into a second program address register. Once an address has been latched, a series of control words must be written to the two 16-bit control registers associated with the PGFLASH. The required sequence is described in the PGFLASH programming section of Chapter 5 in the *Ni1000 Recognition Accelerator User's Guide.*

Only 16 bits (D[0:15]) are used for data transfer, since the PDbus is 16-bits wide. Data on D[16:63] should be set to 1 on input.

5.1.2.2 PGFLASH Read Mode

In this mode, the microcontroller is halted while its PGFLASH memory is read by the host. The internal Program Data Bus (PDbus) drives the external data pins, D[0:63].

This mode is initiated by asserting MC# and driving W/R# low to indicate a read operation. As in the case of load operation, only 16 bits (D[0:15]) are used for data transfer.

5.1.3 Reset Mode (RESET)

Bit 15 of the CMR register is the Accelerator's reset latch. Any of the following conditions can reset all or parts of the Accelerator:

- The RESET# signal is asserted at the rising edge of CLK.
- The MC# signal is asserted at the rising edge of CLK.
- A value of 1 is written to bit 15 of the CMR register (by the host or the microcontroller).

The first and third condition above reset the entire Accelerator, which includes the external bus interface, PGFLASH, and other internal logic. The second condition alone resets the internal logic, except the contents of PA, PGFLASH and the PGFLASH registers.

When the Accelerator is reset, all precharging and most latching is suspended. An active clock is still distributed to most blocks, but the blocks become idle. All state machines are reinitialized. There are two exceptions to these reset actions:

- The external bus interface is only forced into its initialization state when the RESET# signal is asserted on a rising edge of CLK.
- The PGFLASH is not reset if the MC# signal is asserted.

The first exception allows the external bus interface to function, so that the host can write a 0 into bit 15 of the CMR register. The second exception enables the PGFLASH to be manipulated while all other units on the Accelerator (except the external bus interface) are deactivated during the PG modes.

After the Accelerator is reset, operation can only be restored by the host deasserting RESET# and writing a 0 to bit 15 of the CMR register. The Accelerator is put into NORMAL mode if MC# and RESET# are deasserted, and CS# is asserted. The microcontroller starts executing instructions from location 1 in the PGFLASH which is at address F001h in the memory space.

5.2 Bus Cycles

In NORMAL and PG modes, 64, 32 or 16 bits can be transferred. The Accelerator supports both burst and non-burst transfers. Both the host and the Accelerator can terminate a bus cycle, but only the host can initiate one. The timing diagrams in this section illustrate the read and write cycles that can be performed in the NORMAL and PG modes.

Table 5-2 shows the signals used to control bus cycles. A cycle starts when ADS# and CS# are both asserted (not necessarily at the same time) by the host at a rising edge of CLK. At the same time, the host drives the address on A[0:15] and the W/R# signal to define a read or write.

Data (whether input or output) are not transferred unless the Accelerator asserts RDY# or BRDY#. If the Accelerator asserts RDY#, the cycle is terminated by the Accelerator after a single bus-width of data is transferred. If the Accelerator asserts BRDY# during the first transfer, additional transfers can be made in that cycle. The BLAST# signal, when asserted by the host at the beginning of a cycle, indicates either a single-bus-width (non-burst) cycle or the last data transfer of a burst cycle. If the Accelerator asserts BRDY# and the host asserts BLAST# at the same time, the host is terminating the cycle. If the Accelerator asserts neither RDY# nor BRDY#, data are not transferred. If the Accelerator asserts BERR#, data are not transferred and the cycle is terminated. The BLAST# signal always indicates the last transfer of any cycle (burst or non-burst); it is not required for a burst transfer, although when asserted by the host it always indicates the end of a cycle from the hosts viewpoint.

Table 5-2. Cycle-Definition and Control Signals for Normal and PG Modes

Pins	Driven By	Description
ADS#	Host	**Address Strobe**. When asserted by the host on a rising edge of CLK, this signal causes the Accelerator to sample CS# and the address on A[0:15], thereby initiating a bus cycle.
CS#	Host	**Chip Select**. Asserted by the host to indicate that the Accelerator is being addressed. The signal must be held asserted throughout the bus cycle. The signal is used to select one of potentially multiple Ni1000 Accelerators.
BLAST#	Host	**Burst Last**. When asserted by the host, this signal indicates the last data transfer in the current cycle, whether burst or non-burst. For burst cycles, the host holds BLAST# negated until the last data transfer of the cycle, during which it asserts BLAST#. For non-burst cycles, the host asserts BLAST# during the first (and only) data transfer. The signal is compatible with the x86 BLAST# architecture; however, only a maximum of 64 bits can be burst to or from the Ni1000 Accelerator.
W/R#	Host	**Write or Read**. Driven by the host on the same rising clock edge as ADS#, CS#, and BLAST#, to indicate that the current bus cycle is a write (high) or read (low).
RDY#	Ni1000 Accelerator	**Non-Burst Ready**. When asserted by the Accelerator, this signal indicates that the data on D[0:63] is valid (for output) or accepted (for input) and that it is the last data transfer in the current bus cycle. The signal terminates the bus cycle. The Accelerator cannot pull this signal high.
BRDY#	Ni1000 Accelerator	**Burst Ready**. When asserted by the Accelerator, this signal indicates that the data on D[0:63] is valid (for output) or accepted (for input) and that more data may be transferred in the current burst bus cycle. The signal does not terminate the current bus cycle. The Accelerator cannot pull this signal high.
BERR#	Ni1000 Accelerator	**Bus Error**. When asserted by the Accelerator, this signal indicates that illegal bus-definition conditions have occurred. For example, the host may attempt to write to the input buffer when the Accelerator is not in an appropriate mode or when the buffer is full, or the host may attempt to access the output buffer before data is available. The signal also terminates the current bus cycle. The Accelerator cannot pull this signal high.
64/32#	Host	**64-Bit or 32-Bit Data Bus**. Driven by the host to select 64-bit (high) or 32-bit (low) operation on the D[0:63] bus. Data alignment is described in Section 4.1.
MC#	Host	**Microcontroller**. Asserted by the host on the same rising clock edge as ADS# and CS# to read or write the Accelerator's microcontroller-program memory (PGFLASH).

5.2.1 I/O-Register Read or Write

The I/O registers can be read or written by the host in single (non-burst) cycles when the Accelerator is operating in the NORMAL access mode. Figure 5-2 shows an I/O-register read cycle followed by an I/O-register write cycle. The addresses of the I/O registers are given in the "Architecture" chapter.

A read cycle begins when the host asserts ADS# and CS# and the Accelerator samples them at the rising edge of CLK. At the same time, the host drives the address, A[0:15], and it drives W/R# low. The host also asserts BLAST# at the beginning of the cycle to indicate a single or non-burst data transfer, i.e., the first data transfer is the last transfer expected in the cycle. When the Accelerator responds by placing valid data on D[0:31], the Accelerator asserts RDY# indicating that the host should sample the data.

A write cycle, shown in the right side of Figure 5-2, follows essentially the same protocol as the read cycle except that the host drives W/R# high and it drives the data on D[0:31] at the beginning of the cycle, at the same time that it drives ADS#, CS#, and the address. The Accelerator terminates the cycle in the same manner as it terminates a read: by asserting RDY#.

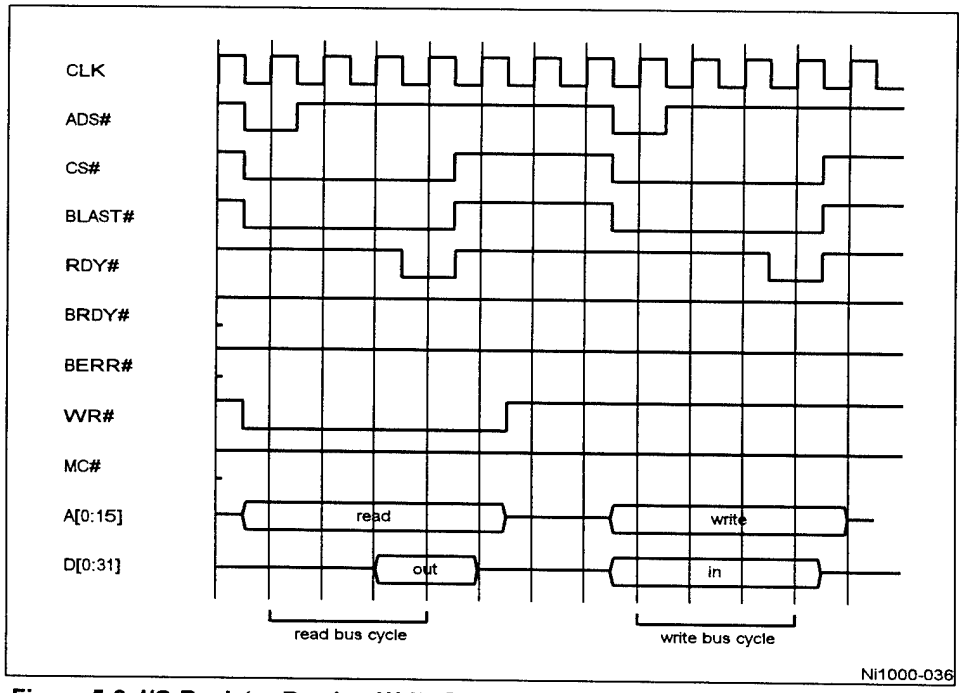


Figure 5-2. I/O Register Read or Write By Host

5.2.2 PGFLASH Read or Write

The Program Flash Memory (PGFLASH) can be read or written by the host in single (non-burst) cycles when the Accelerator is operating in one of the two PG access modes. Figure 5-3 shows a PGFLASH read cycle followed by a PGFLASH write cycle. The PGFLASH is accessed at addresses F000h through FFFFh. In a read cycle, the host begins by driving ADS#, CS#, the address, W/R#, and BLAST#. Three clocks later, the Accelerator places the data on D[0:31] and asserts RDY#. In a write cycle, the Accelerator samples the data and asserts RDY# two clocks after the host begins the cycle.

The protocol for reading and writing PGFLASH is the same as for reading and writing an I/O register; however, the timing of reads in the PGFLASH is three clocks longer than for an I/O register, and the timing of writes is one clock shorter.

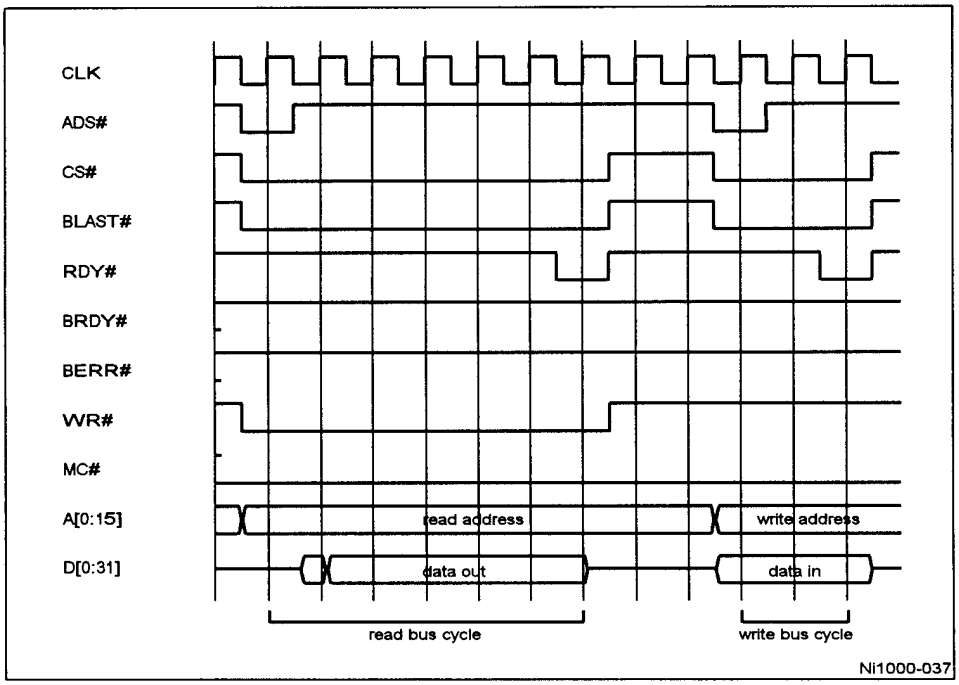


Figure 5-3. PGFLASH Read or Write By Host

5.2.3 IRAM Non-burst Write

The IRAM can be written by the host in single (non-burst) or burst cycles while the Accelerator is operating in the NORMAL access mode. IRAM is accessible at address 2000h. When writing input vectors into IRAM for classification, the BERR# signal will be asserted if the host attempts to write more data than specified in the DIM register. See the "Architecture" chapter for details.

Figure 5-4 shows two sequential single (non-burst) writes to the IRAM. The timing is identical for both writes, and is also identical to the timing for PGFLASH writes: two clocks after the host starts the cycle, the Accelerator samples the data and asserts RDY# to terminate it. Burst writes to the IRAM are shown in Figure 5-6.

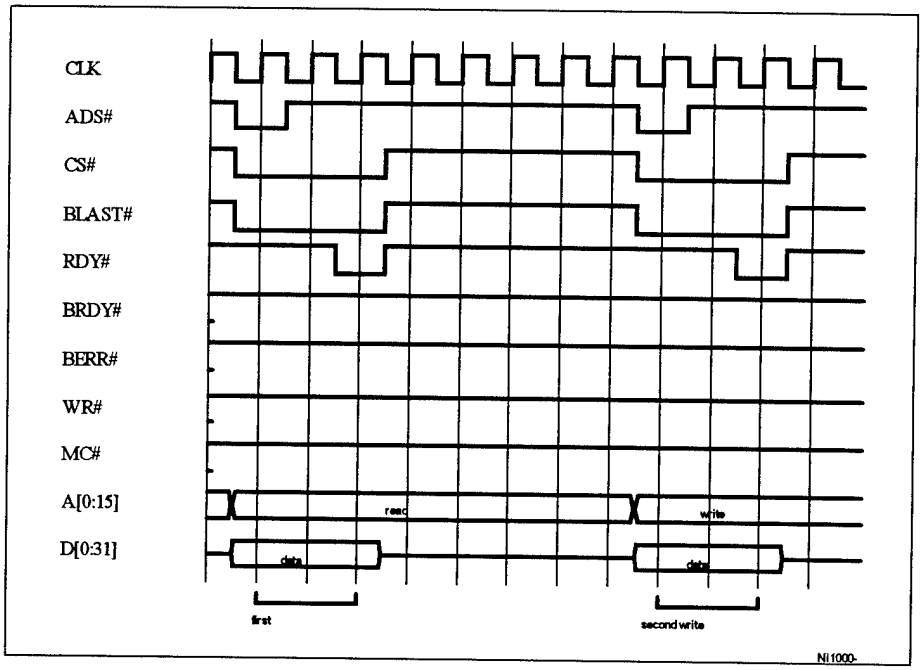


Figure 5-4. IRAM Non-burst Write By Host

5.2.4 ORAM Non-burst Read

The ORAM can be read by the host in single or burst cycles while the Accelerator is operating in the NORMAL access mode. ORAM external output port is accessed at address 2800h.

Figure 5-5 shows two types of single (non-burst) ORAM reads. The first read occurs when the ORAM has just been filled with classification results. The cycle takes five clocks to complete. This is two clocks longer than a read that occurs when the ORAM has been previously accessed (i.e., not just filled with classification results). Burst reads of the ORAM are shown below in Figures 5-7 and 5-8.

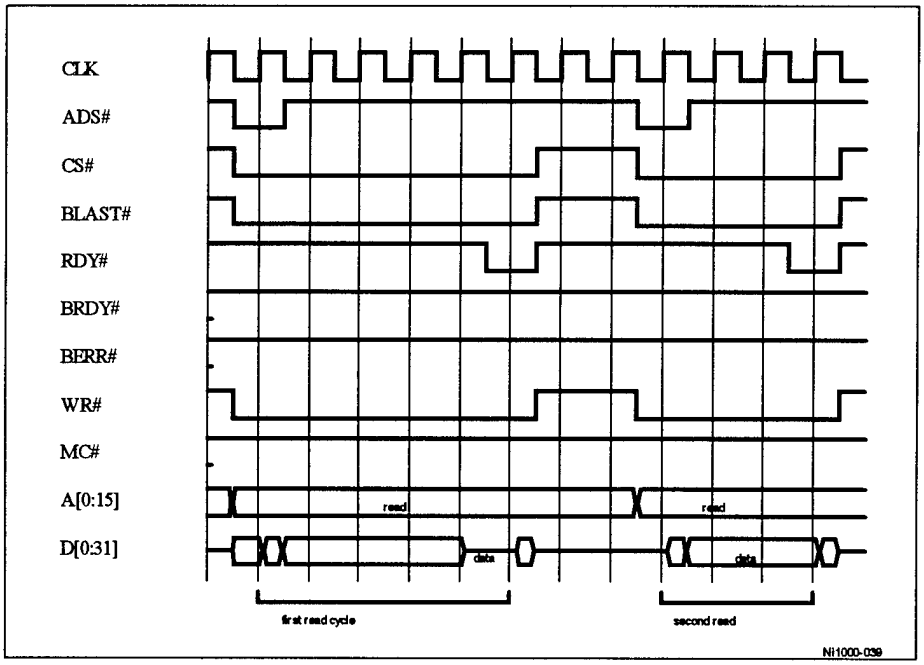


Figure 5-5. ORAM Non-burst Read By Host

5.2.5 IRAM Burst Write

Figure 5-4 showed a single write to the IRAM. Figure 5-6, below, shows a four-transfer burst write to the IRAM. In burst cycles, the Accelerator asserts BRDY# instead of RDY# to indicate each successful data transfer in the multi-transfer sequence. Unlike RDY#, BRDY# does not terminate the cycle.

The host begins the cycle with BLAST# deasserted. It keeps BLAST# deasserted through the third transfer, indicating to the Accelerator that these transfers are not expected to be the last transfer of the cycle (i.e., that this will be a burst cycle). In response, the Accelerator holds BRDY# asserted from the first data transfer through the last transfer (although RDY# can be substituted for BRDY# in the last transfer). The end of the cycle occurs when the host asserts BLAST# while the Accelerator asserts either BRDY# or RDY#.

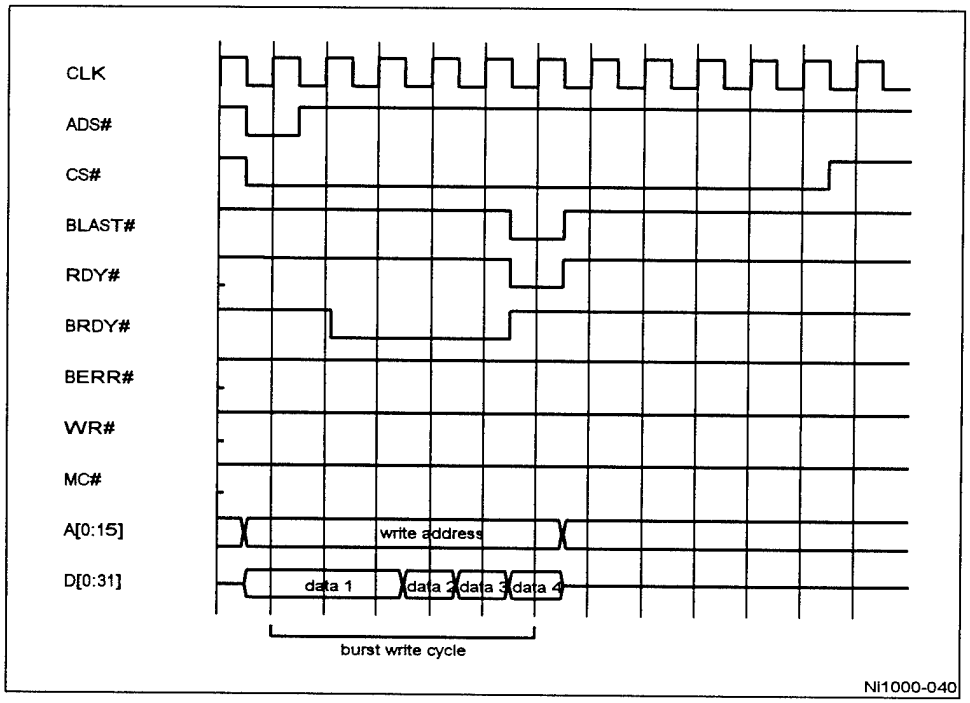


Figure 5-6. IRAM Burst Write By Host

5.2.6 ORAM Burst Read

Figure 5-5 previously showed the two types of single (non-burst) reads to the ORAM: the first type applies to the ORAM immediately after is has been filled with classification-results data, and the second type applies to the ORAM when it is filled prior to the read access and waiting idly. Figures 5-7 and 5-8, below, show four-transfer burst reads of the ORAM under the same respective situations. As in burst cycles that access IRAM, the Accelerator asserts BRDY# instead of RDY# to indicate each successful data transfer in the multi-transfer sequence. Unlike RDY#, BRDY# does not terminate the cycle.

Figure 5-7 shows a burst read to a newly filled ORAM. The host begins the cycle with BLAST# deasserted. It keeps BLAST# deasserted through the third transfer, indicating to the Accelerator that these transfers are not expected to be the last transfer of the cycle (i.e., that this will be a burst cycle). In response, the Accelerator holds BRDY# asserted from the first data transfer through the last transfer (although RDY# can be substituted for BRDY# in the last transfer). The end of the cycle occurs eight clocks later when the host asserts BLAST# while the Accelerator asserts either BRDY# or RDY#.

Figure 5-8 shows a burst read to a partially read ORAM. The same protocol is used as in Figure 5-7 (except in this case, RDY# instead of BRDY# is asserted by the Accelerator to terminate the cycle). However, the cycle is shorter by two clocks than for accesses to a newly filled ORAM.

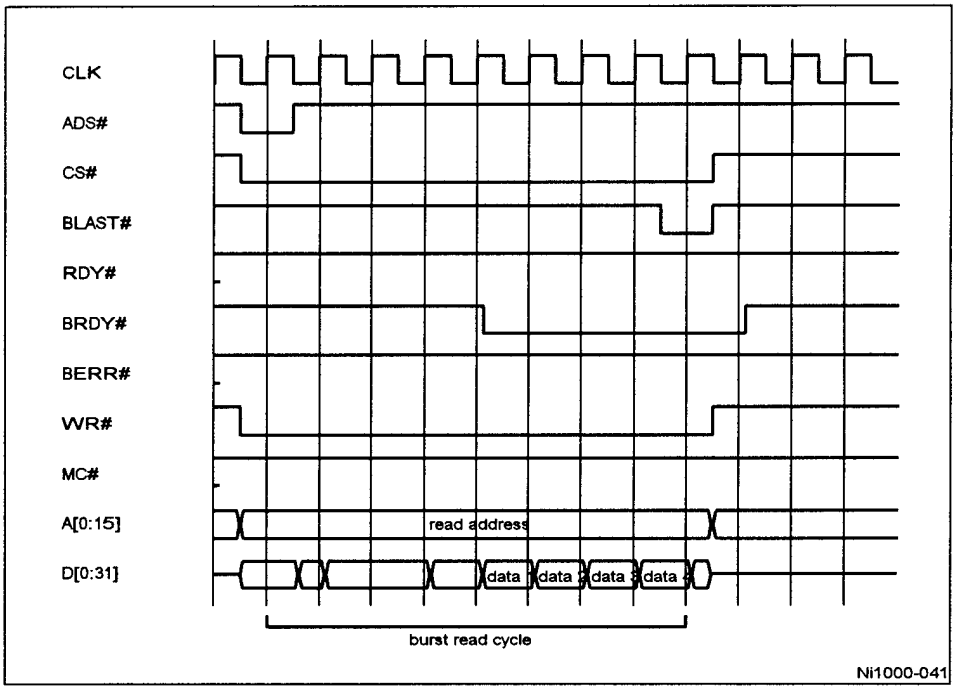


Figure 5-7. ORAM Burst Read By Host

5.2.7 Reset

All or parts of the Ni1000 Accelerator can be reset by the host asserting RESET# or MC#, or by the host writing a 1 to bit 15 of the CMR register. Figure 5-9 shows the cycle, once reset, the Accelerator can only be brought back to normal operation by the host writing a 0 to bit 15 of the CMR register (not shown in Figure 5-9), provided that MC# is not asserted and RESET# is deasserted.

The top half of Figure 5-9 shows that, when RESET# is asserted while MC# is deasserted at the rising edge of CLK, the entire Accelerator is reset, which includes the external bus interface, PGFLASH, and other internal logic. When RESET# is deasserted, all units remain reset except the external bus interface, since this interface is reset only when RESET# is asserted. When MC# is asserted, PGFLASH is no longer in reset condition, but the other internal logic remains reset. When MC# is deasserted, PGFLASH re-enters the reset condition.

The bottom half of Figure 5-9 shows that when MC# is asserted while RESET# is deasserted at the rising edge of CLK, only the internal logic is reset, not including PGFLASH. The external bus interface is reset when RESET# is held low. When MC# is deasserted, PGFLASH enters the reset condition. Other internal logic remains reset throughout the cycle.

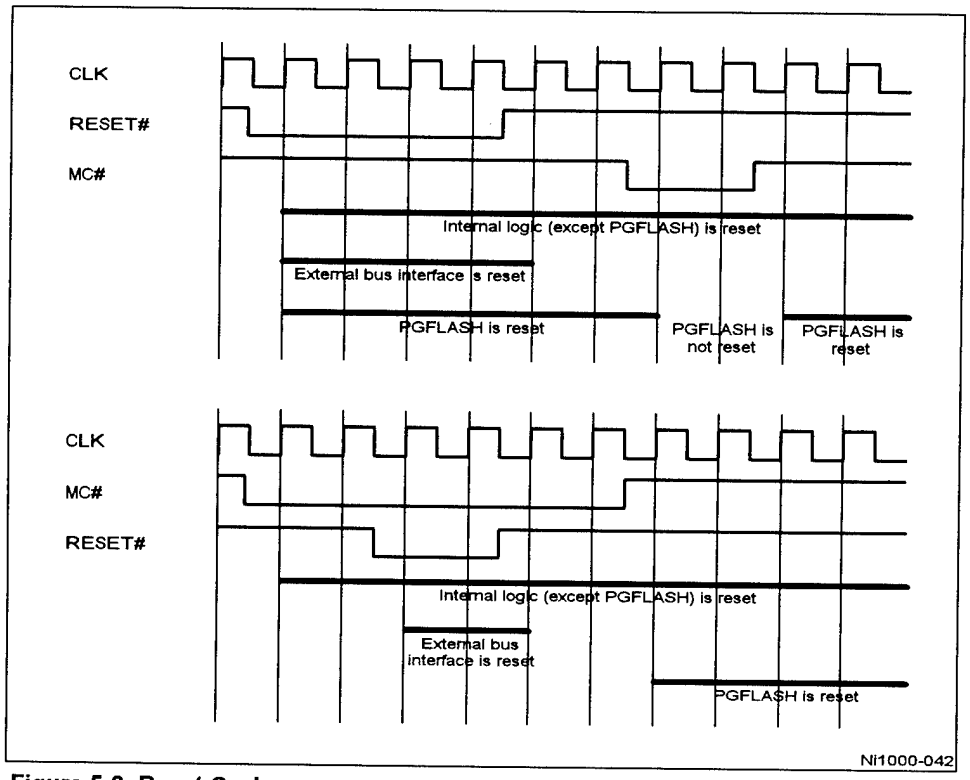


Figure 5-8. Reset Cycle

6. Electrical Characteristics

6.1 Absolute Maximum Ratings

Operating Temperature (ambient) ..... 0°C to +70°C
Storage Temperature ..... -55°C to +140°C
Voltage on Inputs and Outputs with Respect to V_{SS} ..... –0.5V to +6.5V
V_{cc}, V_{cx} Supply Voltages with Respect to V_{SS} ..... –0.5V to +6.5V
V_{pp} Supply Voltage with Respect to V_{SS} ..... –0.5V to +13.5V

NOTICE: *Stresses above those listed under Absolute Maximum Ratings may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any conditions above those indicated in the operational sections of this specification is not implied. Exposure to Absolute Maximum Rating conditions for extended periods may affect device stability. All specifications contained within the following tables are subject to change.*

6.2 D.C. Characteristics

Symbol	Parameter	Min	Max	Notes
Vil	Input LOW Voltage	-0.3V	0.8V	tested at 1 MHz
Vih	Input HIGH Voltage	2.0V	Vcc+0.3V	tested at 1 MHz
Vol	Output LOW Voltage	n/s	0.45V	tested @ 1 MHz, 4mA
Voh	Output HIGH Voltage	2.4V	n/s	tested @ 1 MHz, 4 mA
Icc	Supply Current	n/s	700mA	Vcc = 5.25 V, CLK = 25 MHz
Ioh	Output Current HIGH Voltage	n/s	4 mA	
Iol	Output Current LOW Voltage	n/s	4 mA	
Ili	Input Leakage Current	n/s	+/-10 µA	
Ilo	Output Leakage Current	n/s	+/-10 µA	
Cin	Input Capacitance	n/s	15 pF	exc. CLK
Co	Output Capacitance	n/s	15 pF	
Cclk	Clock Capacitance	n/s	20 pF	

Symbol	Parameter	Min	Max	Notes
Ipp1	Program Current	n/s	30 mA	Vpp = Vpph, Programming in progress
Ipp2	Erase Current	n/s	30 mA	Vpp = Vpp(max), Erasure in progress
Ipps	Vpp Leakage Current	n/s	+/-10 mA	Vpp = Vpp(min), No programming or erase in progress
Vcc	Vcc Voltage	4.75V	5.25V	
Vcx	Vcx Voltage	4.75V	5.25V	
Vpp	Vpp Voltage During P/E Operations	11.4V	12.6V	
Tpap	Pulse Width to Program PA	10µs	50µs	
Nppa	Number of Pulses to Program PA	1	200	
Tpae	Pulse Width to Erase PA	1ms	5 ms	
Nepa	Number of Pulses to Erase PA	1	2000	
Tpgp	Pulse Width to Program PGFLASH	10µs	50µs	
Nppg	Number of Pulses to Program PGFLASH	1	200	
Tpge	Pulse Width to Erase PGFLASH	2ms	10 ms	
Nepg	Number of Pulses to Erase PGFLASH	1	2000	
NCpa	Number of Program/Erase Cycles for PA	1000	n/a	
NCpg	Number of Program/Erase Cycles for PGFLASH	1000	n/a	

n/a - not applicable
n/s - not specified

6.3 A.C. Characteristics

Operating Conditions: Vcc = 5V ± 5%, Vcx = 5V ± 5%, Ta = 0°C to 70°C, I/O Levels: 0, 3.5V .

Pin Array Location	Signal Name	Load (pF)	Min (ns)	Max (ns)
Q11	CLK Period (-25) (-10)	n/a n/a	40 100	500 500
Q11	CLK Frequency Variation (cycle-to-cycle)	n/a	n/a	0.1%
Q11	Phase Variation	n/a	n/a	5%
Q11	CLK High Time	n/a	13	n/s
Q11	CLK Low Time	n/a	13	n/s
Q11	CLK Rise Time	n/a	n/s	2
Q11	CLK Fall Time	n/a	n/s	2
Various	A[0:15] Setup Time	n/a	15	n/a
Various	A[0:15] Hold Time	n/a	6	n/a
Various	D[0:31] Setup Time	n/a	15	n/a
Various	D[0:31] Hold Time	n/a	6	n/a
Various	D[32:63] Setup Time	n/a	15	n/a
Various	D[32:63] Hold Time	n/a	6	n/a
A12	Chip Reset Time	n/a	5 clocks	n/a
A13	MCINT# Setup Time	n/a	15	n/a
A13	MCINT# Hold Time	n/a	6	n/a
A14	ERROR# Setup Time	n/a	15	n/a
A14	ERROR# Hold Time	n/a	20	n/a
A14	ERROR# Delay Time	n/a	10	n/a
B11	MULTCHIP# Setup Time	n/a	13	n/a
B11	MULTCHIP# Hold Time	n/a	20	n/a
C11	MC# Setup Time	n/a	15	n/a
C11	MC# Hold Time	n/a	5 clocks	n/a

Pin Array Location	Signal Name	Load (pF)	Min (ns)	Max (ns)
C13	IACK# Setup Time	n/a	15	n/a
C13	IACK# Hold Time	n/a	6	n/a
E15	BLAST# Setup Time	n/a	15	n/a
E15	BLAST# Hold Time	n/a	6	n/a
F15	W/R# Setup Time	n/a	15	n/a
F15	W/R# Hold Time	n/a	6	n/a
G15	ADS# Setup Time	n/a	15	n/a
G15	ADS# Hold Time	n/a	6	n/a
H15	CS# Setup Time	n/a	15	n/a
H15	CS# Hold Time	n/a	5 clocks	n/a
Q12	RDY# Clk -> Output	30	n/s	12
R12	BRDY# Clk -> Output	30	n/s	12
S12	BERR# Clk -> Output	30	n/s	12
S13	SRQ# Clk -> Output	30	n/s	12
Dn	D0-D63 Clk -> Output	30	n/s	12

n/a - not applicable
n/s - not specified

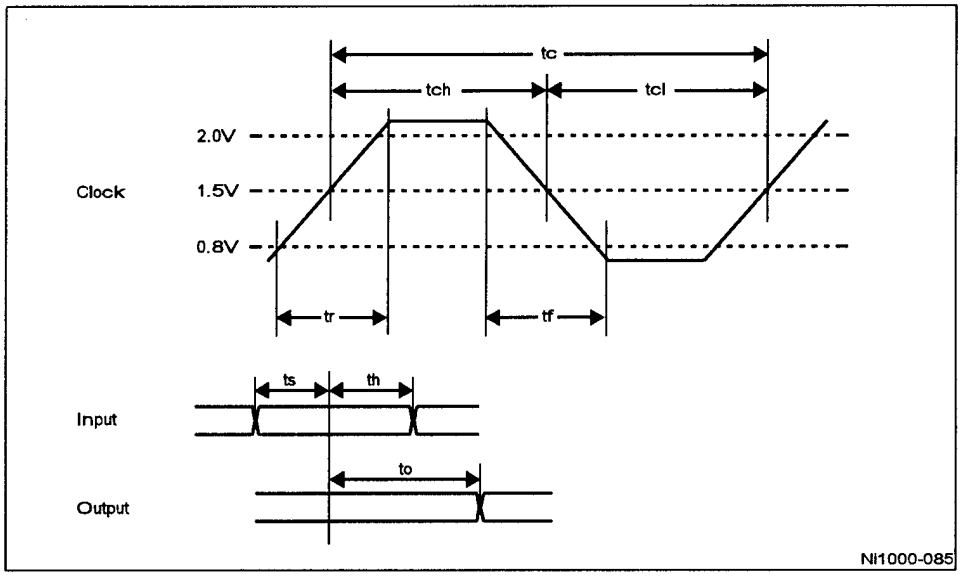


Figure 6-1. A.C. Waveforms

Table 6.1 Legend for A.C. Waveforms

Symbol	Meaning
Tc	CLK Cycle Time
Tch	CLK High Time
Tcl	CLK Low Time
Tr	CLK Rise Time
Tf	CLK Fall Time
Ts	INPUT Setup Time
Th	INPUT Hold Time
To	OUTPUT Delay Time

7. Mechanical and Thermal Characteristics

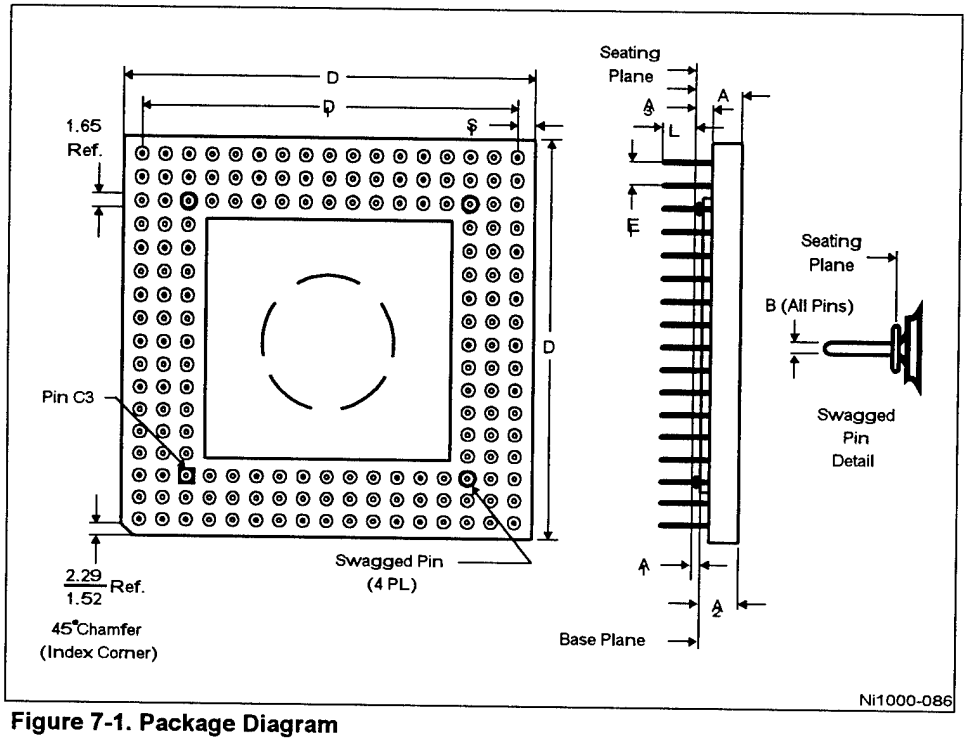


Figure 7-1. Package Diagram

Table 7.1 Package Dimensions

Ceramic Pin Grid Array Package						
Symbol	Millimeters			Inches		
	Min	Max	Notes	Min	Max	Notes
A	3.56	4.57		0.140	0.180	
A_1	0.64	1.14	Solid Lid	0.025	0.045	Solid Lid
A_2	2.79	3.56	Solid Lid	0.110	0.140	Solid Lid
A_3	1.14	1.40		0.045	0.055	
B	0.43	0.51		0.017	0.020	
D	44.07	44.83		1.735	1.765	
D_1	40.51	40.77		1.595	1.605	
E_1	2.29	2.79		0.090	0.110	
L	2.54	3.30		0.100	0.130	
N (pins)	168			168		
S_1	1.52	2.54		0.060	0.110	

The Ni1000 Accelerator is specified for operation when the case temperature, Tc, is within the range of 0°C-85°C. Tc may be measured in any environment to determine whether the Accelerator is within this range, but the measurement should be made at the center of the top surface on the opposite side from the pins.

Given power dissipation and thermal resistance information from Table 7.2, the following equations can be used to related junction, case and ambient temperatures:

$$Tj = Tc + P * \theta jc$$
$$Ta = Tj - P * \theta ja$$
$$Tc = Ta + P * [\theta ja - \theta jc]$$

where,

Tj = junction temperature
Ta = ambient temperature
Tc = case temperature
θjc = junction-to-case thermal resistance
θjz = junction-to-ambient thermal resistance
P = maximum power consumption

The values for θjc and θja are given in Table 7-2. Note that Ta is greatly improved by attaching a heat sink to the package. The maximum power consumption, P, is calculated by using the maximum Icc at 5V as tabulated in the *D.C. Characteristics* section.

Table 7.2 Thermal Resistance (°C/W)

	θjc	θja vs. Airflow—ft/min(m/sec)					
		0 (0)	200 (1.01)	400 (2.03)	600 (3.04)	800 (4.06)	1000 (5.07)
Without Heat Sink	1.5	17	14.5	12.5	11.0	10.0	9.5
With Heat Sink	2.0	13	8.0	6.0	5.0	4.5	4.25

* 0.350" high unidirectional heat sink (Al allow 6063, 40 mil fin width, 155 mil center-to-center fin spacing).

8. Glossary

Bayes Rule—A statistical approach used in pattern classification when overlap exists among the fields of influence of prototype classes. The mathematical form of the *Bayes* rule is: $P(\omega|x) = p(x|\omega)\ P(\omega)\ /\ p(x)$, where the *a priori* probability $P(\omega)$ and the conditional probability density $p(x|\omega)$ are known. In the context of the Ni1000 Accelerator, ω is a classification class, and x is an input pattern. *Bayes* rule shows how the input pattern x changes the *a priori* probability $P(\omega)$ to the *a posterior* probability $P(\omega|x)$. Under this rule, input patterns are assigned to the class with the highest *PD*, $P(\omega|x)$.

City Block Distance—Also called "Manhattan Distance", or "L1 Norm" in mathematical terms, a measure of the distance between an input vector and a prototype vector. Absolute differences between corresponding components of the two vectors are summed to form this distance. "City Block" refers to the orthogonality in the distance computing. It is used in the Ni1000 Recognition Accelerator to simplify the implementation.

Class—A category into which prototypes are grouped and input patterns are classified. The Ni1000 Recognition Accelerator supports up to 64 classes.

Class Firing—If at least one prototype of a class fires (see *prototype firing*), the class is said to fire. Each prototype has a class ID number to indicate the class to which it belongs.

DCU—One of the distance calculation units. There are 512 DCUs in the Ni1000 Accelerator.

Deterministic Radial Basis Function—A radial basis function that does not overlap with an RBF of another class so that classifications done with it are deterministic. Compare *Probabilistic Radial Basis Function.*

Deterministic RBF—See *Deterministic Radial Basis Function.*

Dimension—The number of components in the input and prototype vectors. The Ni1000 Recognition Accelerator supports vectors with up to 256 components (or dimensions).

Feature Space—The complete range of possible patterns of input data. A point in a multidimensional feature space corresponds to an input vector. Classification defines regions within the feature space. The Ni1000 Accelerator supports feature space of up to 256 dimensions and combines small units of feature space into complex regions representing *classes.*

Field of Influence—A region in the *feature space* associated with each prototype, subsequently with each class, and indicated by the value of the *threshold distance*. An input pattern is within the field of influence of a class if its *city block distance* to one of the prototype vectors in the class is less than the threshold distance of that prototype.

Flash EPROM—Electrically erasable and programmable read-only memory. Unlike conventional EPROM which requires physical removal from the computer and UV exposure, erasing and programming on the flash memory may be done in-system by applying high electrical voltage to the memory cells. The contents of the flash memory are preserved after

power-down. The Ni1000 Recognition Accelerator uses flash memory to store microcontroller program (*PGFLASH*) and prototype array (PA).

GRAM—General-purpose random-access memory, used by the on-chip microcontroller of the Ni1000 Accelerator.

Incremental Learning—Addition of new prototypes to the existing base and/or adjustment of parameters. This is the normal learning mode on the chip, assuming initialization has occurred, at least one prototype is stored, and some training had been done.

Influence Field—Same as *field of influence*.

IRAM—The on-chip input buffer random access memory.

Lambda (λ)—Same as *threshold radius*.

Learning—Also called training or adaptation, a process of selectively choosing a set of prototypical vectors from the training data and adjusting appropriate parameters associated with those vectors. Ideally, the prototypical vectors chosen are best indicators of output based on input. The parameters associated with each prototypical vector include the threshold distance for RCE, the amplitude constant and decay constant of the exponential distribution function for PRCE.

Manhattan Distance—See *city block distance*.

MC—The on-chip microcontroller.

Minimum Threshold Distance—A prescribed global constant, intended to limit the number of prototypes selected in the learning process by forcing each prototype to have a minimum region of influence in the pattern space.

MU—The on-chip mathematical unit.

MURAM—The on-chip mathematical unit RAM. There are two MURAMs.

Neural Network Classifier—An implementation of an algorithm that accepts input patterns and outputs classification information. The Ni1000 Recognition Accelerator is such a classifier that uses the *RCE* and/or *PRCE* algorithms.

Neuron—In biology, a cell in the brain that produces an output signal in response to multiple input signals. In the Ni1000 Recognition Accelerator, it is a structure that computes the *city block distance* between an input vector and a pre-stored prototype vector. For *RCE,* this distance is then compared with the radial threshold distance associated with the prototype vector to produce a binary-valued output. For *PRCE,* this distance is used to compute a floating-point number which enters the *PDF* calculation.

ORAM—The on-chip output buffer random access memory.

PA—The on-chip prototype array.

PADCU—*PA* and *DCU*s.

Parzen-Windows—A technique to compute probability density functions. It assumes that within a small region of the feature space, the density function does not vary appreciably, and the probability that a pattern of class C falls within the region is simply the number of vectors in class C in the region, Kc, divided by the total number of vectors in the feature space. In this technique, $Kc = \Sigma\, \phi(p0 - p(k))$, where the sum is over all patterns p(k) in class C, and p0 is the center of the region. The window function $\phi(f - f(k)) = 1$, if $|f - f(k)|$ is less than a threshold value, and 0 otherwise. The threshold value defines the region of estimation.

Pattern—An input vector to be classified. The Ni1000 Recognition Accelerator can accept input vectors with up to 256 dimensions, each of which has a 5-bit resolution.

PDF—See *probability density function.*

PGFLASH—Flash memory used in the Ni1000 Accelerator to store the program for the microcontroller. Default microcontroller program supports *RCE* or *PRCE,* and *PNN* algorithm. PGFLASH is erasable and programmable. See *Flash EPROM.*

PNN—See *probabilistic neural network.*

PPRAM—The on-chip prototype parameter random-access memory. There are three PPRAMs to store, for each prototype vector, its class ID, decay constant, receptive field radius, count, and flags.

PRCE—See *probabilistic Restricted Coulomb Energy.*

Probabilistic Neural Network—A pattern-recognition algorithm that classifies patterns using probability distribution and *Bayes rule.* All training patterns are stored and used to estimate the probability density functions. Learning is rapid (one pass through the training set) and the continuous-valued (Gaussian) estimators perform spatial averaging, resulting in improved PDF estimates in regions of low sample density. A drawback is that memory usage is inefficient so that large data sets require large networks.

Probabilistic Radial Basis Function—A radial basis function that overlaps with an RBF of another class so that classifications done with it are probabilistic. Compare *Deterministic Radial Basis Function.*

Probabilistic RBF—See *Probabilistic Radial Basis Function.*

Probabilistic Restricted Coulomb Energy—A pattern-recognition algorithm that combines *RCE* and *PDF* estimation. The Ni1000 Accelerator computes simultaneously the RCE (firing class IDs) and PRCE (PDFs) results, and the user can select to output either or both. The classification decision is made using *Bayes rule* in the case of multiple firing classes.

Probability Density Function—A specification of the dependence of prototype-classes distribution on an input pattern, used with *Bayes rule* to determine the class to which the input pattern belongs. Functional forms can vary, and are chosen to be the sum of decaying exponentials in the Ni1000 Accelerator.

Prototype—A stored vector that serves to represent the typical features of the patterns to be classified. The Ni1000 Recognition Accelerator supports up to 1024 such vectors of 256 *dimensions* each, and 8000 vectors of lower dimensions. These vectors are selected in the *learning* process, and are grouped into 64 *classes*.

Prototype Firing—An indication of a match between an input vector and a stored prototype. Firing occurs when the input vector is within the *influence field* of the prototype.

Radial Basis Function—A radially symmetric function that has a maximum at some point in its input space and that falls off to zero rapidly at large distances from that point. The region around the symmetry point can be thought of as an influence (or receptive) field since input vectors which fall near this point will result in non-zero response from the RBF logic. The RBF used in the Ni1000 Accelerator is a decaying exponential function.

RBF—See *radial basis function*.

RCE—See *Restricted Coulomb Energy*.

Restricted Coulomb Energy—A pattern-recognition algorithm that is supported by the Ni1000 Accelerator. Training is supervised and selective, in the sense that not all training vectors are committed as prototypes. Several passes of the entire training set may be required. Classification is done using a set of *RBF*s for the prototype vectors. The advantage is that *RCE* may not require as large a network as *PNN*.

Threshold Radius—A parameter associated with each prototype that defines the prototype's field of influence. It is determined during the learning process. When the *city block distance* of an input vector to the prototype is less than the threshold radius, a match is found and the prototype fires. See also *field of influence*, and *minimum threshold distance*.

9. Bibliography

Bellegarda, J.R., and D. Nahamoo. *Tied Mixture Continuous Parameter Modeling for Continuous Speech Recognition*, IEEE Transactions on Acoustics, Speech and Signal Processing, 38:, 1990. pp. 2033-2045.

Bengio, Y., and R. De Mori. *Connectionist Models and their Application to Automatic Speech Recognition*, Artificial Neural Networks and Statistical Pattern Recognition, North-Holland, 1991. pp. 175-194.

Burrascano, P. *Learning Vector Quantization for the Probabilistic Neural Network*, IEEE Trans. on Neural Networks, vol. 2, July 1991. pp. 458-461.

Cooper, L.N. , C. Elbaum, D. L. Reilly, and C.L. Scofield. *Parallel, Multi-Unit Adaptive Nonlinear Pattern Class Separator and Identifier.* U.S. Patent #4,760,604. Filed 9/12/85, Issued 7/26/88.

Deuser, L.M. and S.D. Beck. *Comparison of Artificial Neural Networks for Classifying Sonar Signals and Images*, Government Microcircuit Application Conference, Nov. 1992, pp. 67-70.

He, X., and A. Lapedes. *Nonlinear Modeling and Prediction by Successive Approximation Using Radial Basis Functions*, sub. to Physica D

Huang, Z. D., and M.A. Jack. *Semi-Continuous Hidden Markov Models for Speech Signals*, Computer Speech and Language, 3(3):239-252, July 89

Hubel, D. H., and T.N. Wiesel. *Receptive Fields of Single Neurones in the Cat's Striate Cortex*, J. Physiology (London) 148:574-591, 1959

Lee, Y. *Handwritten Digit Recognition Using K Nearest-Neighbor, Radial Basis Function and Backpropagation Neural Networks*, Neural Computation, Vol. 3, 1991. p440-449.

Leonard, J. A., and M.A. Kramer, L.H. Ungar. *Using Radial Basis Functions to Approximate a Function and its Error Bounds*, IEEE transactions on Neural Networks, Vol. 3, No. 4, July 92.

Lippmann, R.P. *A Critical Overview of Neural Network Pattern Classifiers*, Proceedings of the 1991 Workshop on Neural Networks for Signal Processing, pp. 266-275.

Morgan, D.P., and C.L. Scofield. *Neural Networks for Speech Processing*, Kluwer Academic Publishers, Boston, 1991. pp. 63-69.

Moody, J., and C. Darken. *Learning with Localized Receptive Fields*, Proceedings of the 1988 Connectionist Models Summer School, Morgan Kaufmann, San Mateo, Ca., 1989. pp. 133-143.

Parzen, E. *On Estimation of a Probability Density Function and Mode*, Ann. Math. Statist., Vol. 33, 1962. pp. 1065-1076.

Reilly D. L., and L.N. Cooper, C. Elbaum. *A Neural Model for Category Learning*, Biol. Cybern., vol. 45, 1982, pp. 35-41.

Renals, S., and N. Morgan, H. Bourlard. *Probability Estimation by Feed-Forward Networks in Continuous Speech Recognition*, Proceedings of the 1991 IEEE Workshop on Neural Networks for Signal Processing, pp. 309-318.

Scofield, C. L. *N-Dimensional Coulomb Neural Network Which Provides for Cumulative Learning of Internal Representations*. U.S. Patent #4,897,811. Filed 1/19/88, Issued 1/30/90.

Scofield, C.L., D.L. Reilly et al. *Into Silicon: Real Time Learning in a High Density RBF Neural Network*, International Joint Conference on Neural Networks, July 1991, Seattle WA, pp. I551-556.

Scofield, C. L., and L. Kenton, J. Chang. *Multiple Neural Net Architectures for Character Recognition*, COMPCON Spring '91, San Francisco, Ca., pp. 487-491.

Seligson, D., and M. Griniasty, D. Hansel, N. Shoresh. *Computing with a Difference Neuron*, Network, Vol 3, 1992. pp. 187-204.

Specht, D.F. *A General Regression Neural Network*, IEEE Transactions on Neural Networks, Vol. 2, No. 6, Nov. 1991.

Specht D. F. *Enhancements to Probabilistic Neural Networks*, International Joint Conference on Neural Networks, Baltimore, Md., June 1992.

Specht, D.F. *Generation of Polynomial Discriminant Functions for Pattern Recognition*, IEEE Transactions on Elec. Computers, Vol. EC-16, 1967, pp. 308-319.

Specht, D.F. *Probabilistic Neural Networks*, Neural Networks, Vol. 3, 1990. pp. 109-118.

Specht, D.F. *Probabilistic Neural Networks for Classification, Mapping or Associative Memory*, IEEE International Conference on Neural Networks, June 1988, Vol. I, pp, 525-532.

Appendix E
Ni1000 Recognition Accelerator User's Guide

Ni1000 Recognition Accelerator
User's Guide

Copyright

Rev. 2.0, November 1995 - Nestor Document No PC-UG-Ni1000-1195
Rev. 1.1, July 1994 - Nestor Document No PC-UG-Ni1000-0694

© Copyright Nestor Incorporated, 1994, 1995. All rights reserved. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language without the written permission of Nestor, Inc.

U.S. GOVERNMENT RESTRICTED RIGHTS. This computer software and documentation are provided with RESTRICTED RIGHTS. Use, duplication or disclosure by the Government is subject to restrictions as set forth in the governing Rights in Technical Data and Computer Software clause - subdivision (b)(3)(B) of DAR 7-104.9(a) (May 1981) or subdivision (b)(3) of DOD FAR Supplemental 252.227-7013 (May 1981). Contractor/Manufacturer is Nestor, Inc. of One Richmond Square, Providence, RI 02906 USA.

The information in this document is subject to change without notice and does not represent a commitment on the part of Nestor, Inc.

Trademarks

The Nestor logo, Ni1000, Ni1000 Recognition Controller, Recognition Controller and NestorACCESS are trademarks of Nestor, Incorporated in the United States and other countries. (Ni1000 Microcontroller and Ni1000 Emulator are not trademarked)

MS-DOS, Windows 3.1 , Visual C++ and Excel are the trademarks of Microsoft Corporation.

Every effort has been made to treat trademarks and service marks in accordance with the United States Trademark Commission's guidelines. Any omissions are unintentional and should not be regarded as affecting the validity of any trademark or service mark.

Communications

Nestor, Inc.
One Richmond Square
Providence, RI 02906

For customer service and technical support please call (401) 331-9640 Monday through Friday, 9am -5pm EST or EDT. Facsimile: (401) 331-7319. Compuserve: 76476,1072. Nestor Bulletin Board: (401) 331-2160. See also Appendix D, Customer Support.

Nestor, Inc. is the world leader in providing neural network-based systems for pattern recognition applications. The company was founded in 1983 through the collaborative efforts of Dr. Leon Cooper (Nobel Laureate) and Dr. Charles Elbaum (former Chair, Department of Physics, Brown University) to pursue commercial applications of neural network technology. Principals of the company hold 27 patents in 12 countries and have published several hundred scientific works and articles related to recognition technologies.

Printed in the United States of America
First Printing November, 1995

TABLE OF CONTENTS

1. INTRODUCTION **1-1**

2. PRINCIPLES OF OPERATION **2-1**

2.1. PATTERN RECOGNITION 2-1
2.2. LEARNING AND CLASSIFICATION ALGORITHMS 2-4
2.2.1. Restricted Coulomb Energy (RCE) *2-5*
2.2.2. Probabilistic RCE (PRCE) *2-6*

3. HARDWARE ARCHITECTURE **3-1**

3.1. THE CLASSIFIER 3-3
3.1.1. Distance Calculation Units (DCUs) *3-5*
3.1.2. Prototype Array (PA) *3-5*
3.1.3. Prototype Parameter RAM (PPRAM) *3-7*
3.1.4. Math Unit (MU) *3-10*
3.1.5. Math Unit RAMs (MURAMs) *3-13*
3.2. BUS INTERFACE 3-15
3.2.1. I/O Registers *3-17*
3.2.2. Input RAM (IRAM) *3-18*
3.2.3. Output RAM (ORAM) *3-21*
3.3. COMPUTATIONAL PRECISION 3-22
3.3.1. 16-Bit Internal Format *3-23*
3.3.2. 32-Bit IEEE Format *3-24*
3.4. MICROCONTROLLER 3-24
3.4.1. Memory-Map Overview *3-24*
3.4.2. Architecture *3-24*
3.4.3. Registers *3-27*
3.4.4. Flags *3-28*
3.4.5. Instruction Set *3-28*
3.4.6. Program Memory (PGFLASH) *3-29*
3.4.7. Timer *3-30*
3.4.8. Reset Initialization *3-31*
3.4.9. Interrupts *3-31*
3.4.10. Errors *3-32*
3.5. SYSTEM-LEVEL ARCHITECTURE 3-32
3.6. CLASSIFICATION TIMING 3-33
3.7. SIGNAL DESCRIPTIONS 3-35

4. BUS OPERATIONS **4-1**

4.1. HARDWARE-CONTROLLED ACCESS MODES 4-1
4.1.1. Normal Mode (NORMAL) *4-5*
4.1.2. PGFLASH Access Modes (PG Modes) *4-5*
4.1.3. Reset Mode (RESET) *4-6*
4.2. BUS CYCLES 4-7
4.2.1. I/O-Register Read or Write *4-7*
4.2.2. PGFLASH Read or Write *4-10*
4.2.3. IRAM Non-Burst Write *4-11*
4.2.4. ORAM Non-burst Read *4-12*
4.2.5. IRAM Burst Write *4-13*
4.2.6. ORAM Burst Read *4-14*
4.2.7. Reset *4-15*

5. OPERATION 5-1

5.1. I/O TO AND FROM HOST 5-2
5.1.1. I/O Registers *5-3*
5.1.2. IRAM *5-14*
5.1.3. IRAM Read and Write by the Microcontroller *5-17*
5.1.4. IRAM Write by the Host *5-18*
5.1.5. ORAM *5-19*
5.1.6. ORAM Read and Write by the Microcontroller *5-22*
5.1.7. ORAM Read by the Host *5-23*
5.1.8. Retrieving Both Class and Probabilistic Data from ORAM by the Host *5-24*
5.2. MICROCONTROLLER OPERATIONS 5-25
5.2.1. Internal Mapped Memory *5-25*
5.2.2. PGFLASH *5-29*
5.2.3. PGFLASH Programming *5-34*
5.2.4. GRAM *5-37*
5.2.5. TIMER *5-37*
5.2.6. Interrupt Handling *5-38*
5.2.7. Error Handling *5-42*
5.2.8. Multiple Chip Support *5-42*
5.3. CLASSIFIER ACCESS AND CONTROL 5-42
5.3.1. Prototype Array *5-43*
5.3.2. PA Access *5-53*
5.3.3. PPRAM *5-57*
5.3.4. PPRAM Access *5-61*
5.3.5. MURAMs *5-62*
5.3.6. Classification *5-67*
5.3.7. Learning *5-70*

6. NI1000 MICROCONTROLLER ARCHITECTURE 6-1

6.1. INTRODUCTION 6-1
6.2. MICROCONTROLLER REGISTERS AND FLAGS 6-1
6.3. ADDRESSING MODES 6-2

6.4. INSTRUCTION SUMMARY (BY FUNCTIONAL GROUP) 6-4
6.4.1. Conditional Jumps *6-4*
6.4.2. Subroutine Calls *6-5*
6.4.3. Stack Operations *6-5*
6.4.4. Flag Operations *6-5*
6.4.5. Data Transfer Operations *6-6*
6.4.6. Arithmetic and Logical Operations *6-7*
6.5. INSTRUCTION SET (ALPHABETICAL) 6-7
6.6. FLAGS CROSS REFERENCE 6-48
6.7. INTERRUPT HANDLING 6-58

7. MICROCONTROLLER SOFTWARE 7-1

7.1. OVERVIEW 7-1
7.1.1. Code Design Goals *7-1*
7.1.2. Key Functions *7-1*
7.1.3. Code Structure *7-1*
7.1.4. Learning Code *7-1*
7.1.5. Principal External Interfaces *7-1*
7.1.6. Error Processing *7-1*
7.1.7. Program Flow *7-2*
7.1.8. Ni1000 Microcontroller Code Function Blocks *7-3*
7.2. MEMORY UTILIZATION 7-4
7.2.1. Constants and Variables *7-4*
7.2.2. Important GRAM Locations *7-4*
7.2.3. Command Jump Table *7-4*
7.2.4. Command Queue *7-5*
7.2.5. Host Message Queue *7-5*
7.2.6. Instruction Memory *7-5*
7.2.7. Interrupt Vector *7-5*
7.2.8. Initialization Code Entry Point *7-5*
7.2.9. Microcontroller Stack *7-5*
7.2.10. Configuration Table Data *7-5*
7.3. PROGRAMMING CONVENTIONS 7-6
7.3.1. Register Use *7-6*
7.4. ERROR SUPPORT 7-6
7.4.1. Error Codes *7-6*
7.5. DEBUGGING SUPPORT 7-9
7.6. HOST TO CHIP COMMUNICATION 7-9
7.6.1. Microcontroller Initialization *7-9*
7.6.2. Microcontroller Interrupts *7-9*
7.6.3. I/O Register Usage *7-10*
7.6.4. Host Service Requests *7-10*
7.6.5. Protocol Rules *7-11*
7.6.6. Command Opcodes *7-12*
7.7. PROTOTYPE ARRAY MANAGEMENT 7-19

7.7.1. Bad Column Table *7-19*
7.7.2. Sectors *7-20*
7.7.3. PA Configuration Table *7-20*
7.7.4. Backed Up Networks *7-22*
7.7.5. PA Usage Field *7-25*
7.8. ADDING CUSTOMIZED MICROCODE 7-26

8. GLOSSARY 8-1

9. INDEX 9-1

LIST OF FIGURES

FIGURE 1-1. NI1000 RECOGNITION ACCELERATOR BLOCK DIAGRAM 1-2
FIGURE 1-2. TYPICAL MULTICHIP ADD-IN BOARD 1-3
FIGURE 2-1. HYPOTHETICAL PATTERN RECOGNITION SYSTEM 2-2
FIGURE 2-2. PDFS FOR NUT AND BOLT WEIGHTS 2-3
FIGURE 2-3. RBF EXAMPLES 2-3
FIGURE 2-4. APPROXIMATING FEATURE SPACE WITH RBFS 2-4
FIGURE 2-5. PD CONTRIBUTION OF A SINGLE PROTOTYPE 2-7
FIGURE 2-6. HYPOTHETICAL PDS FOR A TWO-CLASS PROBLEM 2-8
FIGURE 3-1. INTERNAL-ARCHITECTURE BLOCK DIAGRAM 3-3
FIGURE 3-2. CLASSIFIER 3-4
FIGURE 3-3. DISTANCE CALCULATION UNIT (DCU) 3-5
FIGURE 3-4. CONCEPTUAL VIEW OF PROTOTYPE ARRAY (PA) AND DCUS 3-7
FIGURE 3-5. PA DATA FORMAT 3-7
FIGURE 3-6. PA DURING PROGRAMMING 3-8
FIGURE 3-7. PROTOTYPE PARAMETER RAM (PPRAM) 3-8
FIGURE 3-8. PPRAM WORD FORMAT 3-9
FIGURE 3-9. MATH UNIT (MU) PIPELINE 3-11
FIGURE 3-10. MATH UNIT RAMS (MURAMS) 3-14
FIGURE 3-11. CLASS-LIST MURAM WORD 3-14
FIGURE 3-12. PROBABILITY MURAM WORD 3-15
FIGURE 3-13. BUS INTERFACE 3-16
FIGURE 3-14. INPUT RAM (IRAM) 3-18
FIGURE 3-15. IRAM PRE-WRITE LATCH 3-19
FIGURE 3-16. OUTPUT RAM (ORAM) 3-21
FIGURE 3-17. ORAM PRE-WRITE LATCH 3-22
FIGURE 3-18. INTERNAL 16-BIT FLOATING-POINT FORMAT 3-23
FIGURE 3-19. IEEE 32-BIT FLOATING-POINT FORMAT 3-24
FIGURE 3-20. MEMORY-MAP OVERVIEW 3-25
FIGURE 3-21. MICROCONTROLLER DATAPATH 3-26
FIGURE 3-22. BUS ARCHITECTURE 3-27
FIGURE 3-23. PGFLASH DURING PROGRAMMING 3-30
FIGURE 3-24. TIMER 3-31
FIGURE 3-25. PIPELINE USAGE (FOR UP TO 500 PROTOTYPES) 3-33
FIGURE 3-26. PIPELINE USAGE (FOR MORE THAN 500 PROTOTYPES) 3-34
FIGURE 4-1. NI1000 RECOGNITION ACCELERATOR BUSES 4-3
FIGURE 4-2. I/O REGISTER READ OR WRITE BY HOST 4-10
FIGURE 4-3. PGFLASH READ OR WRITE BY HOST 4-11
FIGURE 4-4. IRAM NON-BURST WRITE BY HOST 4-12
FIGURE 4-5. ORAM NON-BURST READ BY HOST 4-13

FIGURE 4-6. IRAM BURST WRITE BY HOST 4-14
FIGURE 4-7. ORAM BURST READ BY HOST 4-15
FIGURE 4-8. RESET CYCLE 4-16
FIGURE 5-1. THE CMR REGISTER 5-4
FIGURE 5-2. THE DIM REGISTER 5-5
FIGURE 5-3. THE IDR REGISTER 5-5
FIGURE 5-4. THE SSR REGISTER 5-6
FIGURE 5-5. THE HS1 REGISTER 5-7
FIGURE 5-6. THE HS2 REGISTER 5-8
FIGURE 5-7. THE XIR REGISTER 5-10
FIGURE 5-8. THE IIR REGISTER 5-10
FIGURE 5-9. THE CRA REGISTER 5-11
FIGURE 5-10. THE CRB REGISTER 5-12
FIGURE 5-11. THE OP REGISTERS 5-14
FIGURE 5-12. IRAM PRE-WRITE LATCH SCENARIO 5-16
FIGURE 5-13. IRAM ADDRESS ASSIGNMENT 5-16
FIGURE 5-14. DATA ALIGNMENT ON 32-BIT EXTERNAL BUS 5-17
FIGURE 5-15. DATA ALIGNMENT ON 64-BIT EXTERNAL BUS 5-17
FIGURE 5-16. INTERNAL BUS DATA ALIGNMENT 5-17
FIGURE 5-17. ORAM PRE-WRITE LATCH SCENARIO 5-20
FIGURE 5-18. ORAM BIT ASSIGNMENT 5-20
FIGURE 5-19. FORMAT OF RCE CLASSIFICATION RESULTS 5-21
FIGURE 5-20. FLOATING-POINT FORMATS 5-22
FIGURE 5-21. THE PGF_ADR REGISTER 5-30
FIGURE 5-22. THE PGF_DR REGISTER 5-31
FIGURE 5-23. THE PGF_CR1 REGISTER 5-31
FIGURE 5-24. THE PGF_CR2 REGISTER 5-33
FIGURE 5-25. THE PGF_SR REGISTER 5-34
FIGURE 5-26. TIMER 5-38
FIGURE 5-27. THE CSA REGISTER 5-47
FIGURE 5-28. THE CSB REGISTER 5-47
FIGURE 5-29. THE MODE REGISTER 5-48
FIGURE 5-30. THE AUX REGISTER 5-48
FIGURE 5-31. THE ARR REGISTER 5-49
FIGURE 5-32. PROTOTYPE ARRAY SEGMENTATION 5-50
FIGURE 5-33. EXAMPLE A OF SPECIFYING A WINDOW IN PA 5-50
FIGURE 5-34. EXAMPLE B OF SPECIFYING A WINDOW IN PA 5-51
FIGURE 5-35. THE DCU_DIM REGISTER 5-52
FIGURE 5-36. THE NCA REGISTER 5-52
FIGURE 5-37. THE NCB REGISTER 5-52
FIGURE 5-38. WORD FORMAT PPRAM 5-58
FIGURE 5-39. PPRAM REGISTERS 5-60
FIGURE 5-40. CLASS-LIST MURAM WORD 5-64
FIGURE 5-41. MURAM PROBABILITY WORD 5-65
FIGURE 5-42. THE MURAM_CR REGISTER 5-66

LIST OF TABLES

TABLE 3-1. EXAMPLE OF IRAM VECTOR-ADDRESSING LSBS 3-20
TABLE 3-2. COMPUTATIONAL PRECISION 3-23
TABLE 4-1. HARDWARE-CONTROLLED ACCESS MODES 4-4
TABLE 4-2. CYCLE-DEFINITION AND CONTROL SIGNALS FOR NORMAL AND PG MODES 4-9
TABLE 5-1. LOGIC BLOCK MODE CONFIGURATION 5-2
TABLE 5-2. I/O REGISTER MAP 5-3
TABLE 5-3. IRAM ACCESS ADDRESSES IN MICROCONTROLLER MODE 5-15
TABLE 5-4. ORAM ACCESS ADDRESSES IN MICROCONTROLLER MODE 5-19
TABLE 5-5. ORAM OUTPUT POSSIBILITIES 5-21
TABLE 5-6. MEMORY AND REGISTER ADDRESS MAP 5-26
TABLE 5-7. PGFLASH REGISTERS 5-30
TABLE 5-8. CLASSIFIER LOGIC BLOCK MODE CONFIGURATION 5-43
TABLE 5-9. BAD COLUMN TABLE 5-45
TABLE 5-10. PADCU REGISTERS 5-46
TABLE 5-11. PPRAM AND REGISTERS 5-57
TABLE 5-12. MURAMS AND REGISTERS 5-62
TABLE 6-1. MICROCONTROLLER REGISTERS 6-1
TABLE 6-2. MICROCONTROLLER FLAGS 6-2
TABLE 6-3. FLAGS CROSS-REFERENCE 6-49

PREFACE

This manual provides detailed documentation of the Ni1000 Recognition Accelerator chip and its hardware and software interfaces. The manual is written for system designers who have experience in pattern recognition and microprocessors.

Terminology and Notation

- *Accelerator* or *Recognition Accelerator*—The Ni1000 Recognition Accelerator chip described in this book.
- *Active-Low Signal Names*—Signal whose names are followed by the symbol, #, indicate active-low signals that are asserted at low voltage and negated at high voltage.
- *Buses*—The notation *SIGNAL[m:n]* represents bits *m* through *n* of a bus.
- *Fields*—The notation *FIELD[m]* represents bit *m* of a field, and the notation *FIELD[n:m]* represents bits *n* through *m* of a field.
- *Bit Values*—Bits can be *set* to 1 or *cleared* to 0.
- *Reserved Bits and Signals*—When bits are marked as *undefined* or *reserved,* it is essential for compatibility with future processors that software treat these bits as having a future, though unknown, effect. Programs that read registers with undefined bits must mask off those values. Programs that write to registers with undefined bits must first read the register and then change only the desired defined bits before writing back to the register.
- *Data Quantities*—A *word* is 16 bits (two bytes), a *dword* or doubleword is 32 bits (four bytes), and a *qword* or quadword is 64 bits (eight bytes).
- *Data Abbreviations*—The following notation is used for bits and bytes:
  Bits .....b
  Bytes ..... B
  Kilo (10^3)..... K
  Mega (10^6) .....M
  Giga (10^9) .....G
- *Hexadecimal Numbers*—Hexadecimal numbers are followed by an *h,* unless the context makes this notation unnecessary.
- *Set Inclusion*—A square bracket, [or], indicates that the adjacent point is included in the set being defined. A parenthesis, (or), indicates that the adjacent point is not included in the set.

Little-Endian Convention

The 80160NC Recognition Accelerator is a *little endian* machine. This means that the bytes within a word are numbered starting from the least significant byte. Pictures of data structures in memory show the smallest addresses at the bottom and the highest addresses at the top. Bit positions are numbered from right to left. Figure i-1 illustrates these conventions using a 32-bit register as an example. The numerical value represented by a bit that is set (1) is equal

to two raised to the power of the bit position. The bit notation in a 32-bit register corresponds directly to the bit notation on a 32-bit data bus when data items are aligned to 32-bit boundaries in memory.

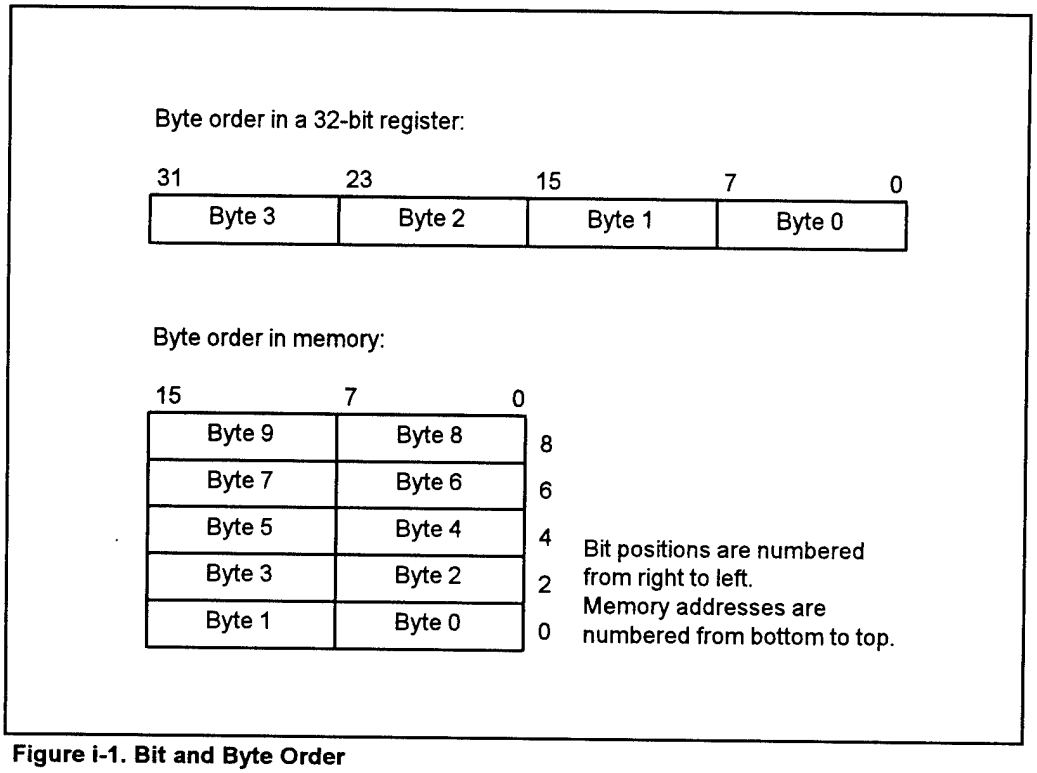


Figure i-1. Bit and Byte Order

1. INTRODUCTION

The Ni1000 Recognition Accelerator chip provides a high-performance solution for pattern recognition problems. The Ni1000 will be available in two versions: the Ni1000-25 and the Ni1000-10. Except as noted, all references to processing rates refer to the Ni1000-25. The Ni1000-10 is rated at approximately 40% of the speed of the Ni1000-25. The Ni1000 has the following features:

- Neural Network Pattern Recognition Accelerator
- Fully Digital Design
- 16-bit, On-Chip Microcontroller with 4K-Word FLASH Program Memory
- 32-Bit or 64-Bit Host Interface
- RCE, PRCE and PNN are Learning and Classification Supported on-chip
- Programmable to Support other RBF Paradigms
- Supports 64-Class Problems
- Outputs Both Class and Probability Data
- 222-Feature Input Vectors with 5-Bit Resolution per Feature
- 1024 Prototype Locations in 1.3-Mbit On-Chip FLASH EEPROM (1000 Prototypes usable)
- 12.4 Billion Operations per Second at 25 MHz
- Supports Multi-chip Operations
- 168-Pin PGA Package

The Ni1000 Accelerator supports classification of over 10,000 patterns per second, with real-time adaptation. The chip is compatible with commonly used Radial Basis Function (RBF) paradigms, including Restricted Coulomb Energy (RCE), Probabilistic RCE (PRCE), Probabilistic Neural Networks (PNN) and other algorithms. The flexible, on-chip microcontroller with its 4K x 16-bit non-volatile microcode memory, also permits implementation of custom algorithms.

The Accelerator accepts input vectors with a maximum of 222 features, each with 32 levels of resolution, and produces up to 64 class IDs and/or probabilities. High-speed parallel processing units compute the city-block distance between an input vector and up to 1000 stored prototypical examples. The Accelerator's high speed is suitable for computationally intensive applications like optical character recognition, fingerprint identification and industrial inspection.

Pattern recognition is the process of sorting input data into categories or classes that are significant to the user. The prototypical values of differentiating traits (features) of each class must first be loaded into the chip's memory. The contents of the chip's memory can be developed manually or extracted from examples of data typical to the problem, using a learning algorithm. Feature sets are problem-specific and may consist partially or completely of stored data, such as historical records, or of direct sensor inputs. Once learning is complete, the system is ready to classify input data. The Ni1000 Recognition Accelerator supports incremental learning in the field, which may be necessary to further adapt the recognition system to its environment.

A high-level diagram of the Ni1000 Recognition Accelerator architecture appears in Figure 1-1. The on-chip, custom, 16-bit microcontroller has separate program and data memories, i.e. a Harvard architecture. The 4K x 16-bit non-volatile FLASH EPROM program memory can hold learning algorithms, chip maintenance routines, and other software required by the application. A general-purpose 256 x 16-bit RAM is also available to the microcontroller.

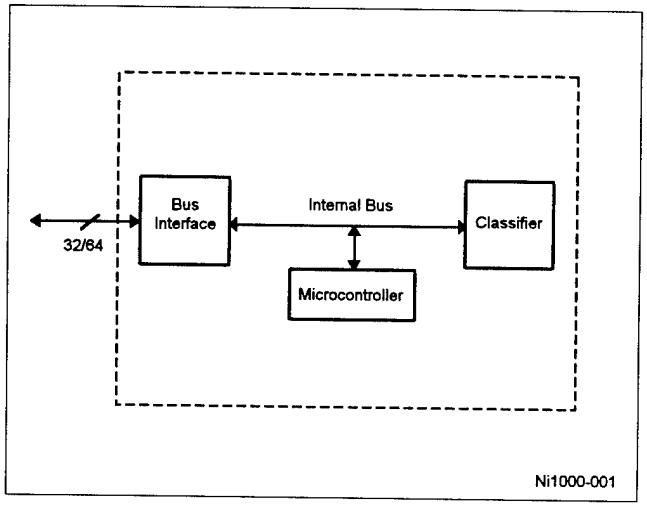


Figure 1-1. Ni1000 Recognition Accelerator Block Diagram

The microcontroller can enable an automatic classification mode in which a series of logic blocks arranged as a pipeline process data and output the results to the host. The classification pipeline consists of input buffers, distance calculation units, a large FLASH prototype array that stores the results from the learning process, a mathematical unit and its output memories, and output buffer. At 25 MHz, the pipeline can classify over 10,000 input vectors per second, in which each input vector has up to 222 features with 5-bit resolution for each feature. The performance is made possible by the Ni1000 parallel architecture, which can execute over 12 billion operations per second. A typical Von Neumann machine would need to execute more than 40 billion instructions per second to approach the processing rate achieved by the Ni1000 Recognition Accelerator.

In most applications, the Accelerator will reside on a bus that is shared with a host CPU and perhaps other Ni1000 Accelerators, as shown in Figure 1-2. The Accelerator will typically be a slave device on the host bus and will not initiate transactions on the host bus. Both the host and the on-chip microcontroller have the ability to interrupt each other.

Figure 1-2 shows a local host CPU on an add-in card for personal computers and workstations. The CPU manages the flow of data through the Accelerator(s). The CPU may also have other functions, such as preprocessing data, interpreting classification results, or coordinating the operation of multiple chips. Other implementations may rely on the CPU of the system board for these functions.

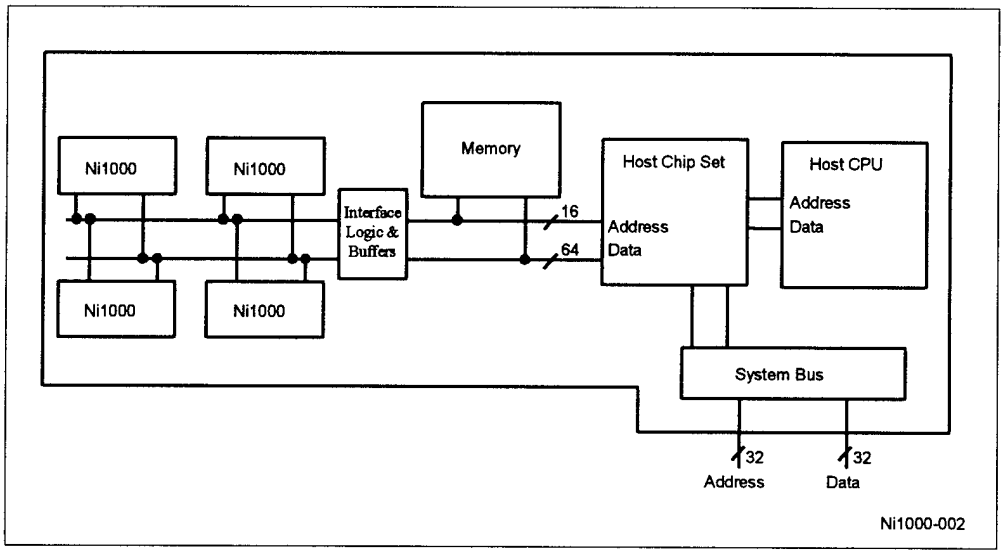


Figure 1-2. Typical Multichip Add-In Board

Overview of Major Changes in the Ni1000 Users' Guide at Rev 2.0.

1. The addition of Ni1000-25 specifications and discontinuation of the Ni1000-33.
2. Reduction in the number of input vector features to 222 on all versions of the Ni1000 and the use of padding in input vectors.
3. Updated classification rate specifications.
4. Limitations on the use of address relocation.

2. PRINCIPLES OF OPERATION

The Ni1000 Recognition Accelerator is optimized for use in systems that require fast classification capabilities. Classification is the process of associating input data with categories or *classes*. In an optical character recognition application, for example, classification would consist of identifying the characters represented by data scanned off an image sensor. The input data would be patterns of on and off pixels from the scanner and output could be ASCII character codes corresponding to the scanned images.

A neural network based recognition system is trained to perform the required classification. It stores what it has learned in a *memory*. Various learning algorithms can be employed to produce a neural network memory. In general, the algorithms are provided with a sample of inputs typical to the problem, called the *training set*, through which the neural network learns the differences among the classes based on their characteristics. Of course, the training must have each input pattern (or vector) *labeled* with the correct class.

Learning is usually a phase of product development only. Once trained, the neural network is usually placed into *classification* mode, in which it performs recognition operations based on what it has learned. In most cases, the neural network memory remains unchanged. However, the Ni1000 Recognition Accelerator provides on-chip learning, allowing it to be used in systems that require retraining in the field.

2.1. Pattern Recognition

To illustrate the recognition process, consider a hypothetical industrial inspection task of separating nuts from bolts out of a stream of hardware moving on a conveyor past a set of sensors. The task of the recognition system developer is to devise a system that takes data from the sensors and assigns it a *class*, e.g., nut or bolt. A system that can accomplish the task appears in Figure 2-1. Before entering the classifier, data output by the sensors may need to pass through a *pre-processor* that extracts *features,* such as the weight, size, shape, or aspect ratio of each piece of hardware. Each property is then a component of the *feature vector*. For example, the size is one feature in the feature vector; shape is another. The classification engine processes the vector and completes the recognition by making a sorting decision.

Components of feature vectors tend to have random individual values. All bolts do not have the same weight, but their weight does have both upper and lower limits. Weighing a representative sample of bolts and charting the results to show the weight variation among bolts produces a Probability Density Function (PDF).

In our example, nuts are generally lighter than bolts. The two hypothetical PDF graphs appear on the same axes in Figure 2-2. Using Bayes Rule, the system sorts the parts into the class with the higher PDF value. In Figure 2-2, pieces with weights below point C are likely to be

nuts. Those heavier than C are probably bolts. Since virtually no bolts weigh less than B and no nuts weigh more than C, on the [A, B] and [C, D] intervals the decision is fairly simple.

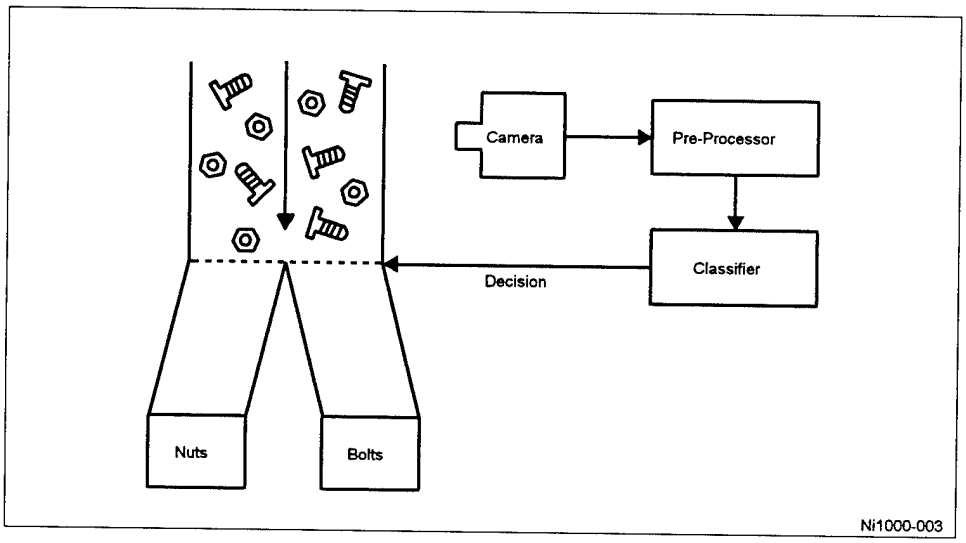


Figure 2-1. Hypothetical Pattern Recognition System

However, on [B, C] both choices are possible, and the system picks the one with the higher probability density. If the input is outside the interval [A, D], it is something unexpected, and the system classifies it as neither a bolt nor a nut. The system thus exhibits novelty detection capability, known as generalization.

Additional features provide additional information to the classifier. The Ni1000 Recognition Accelerator can handle input vectors with up to 222 features.

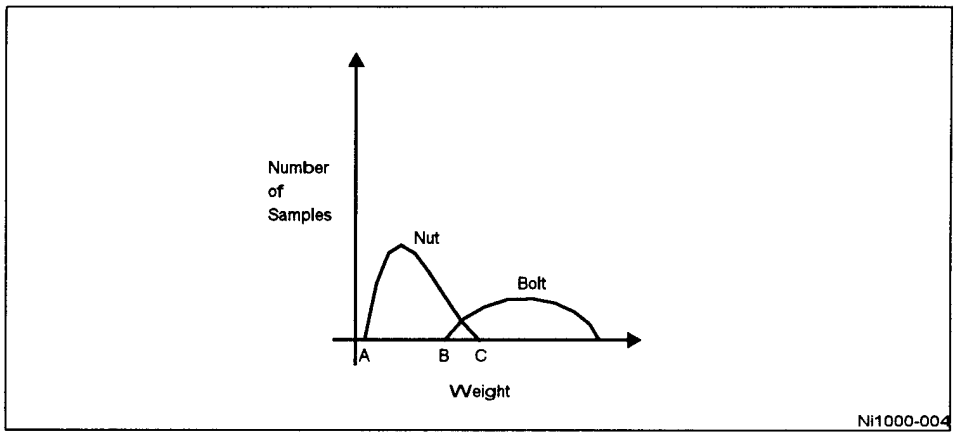


Figure 2-2. PDFs For Nut and Bolt Weights

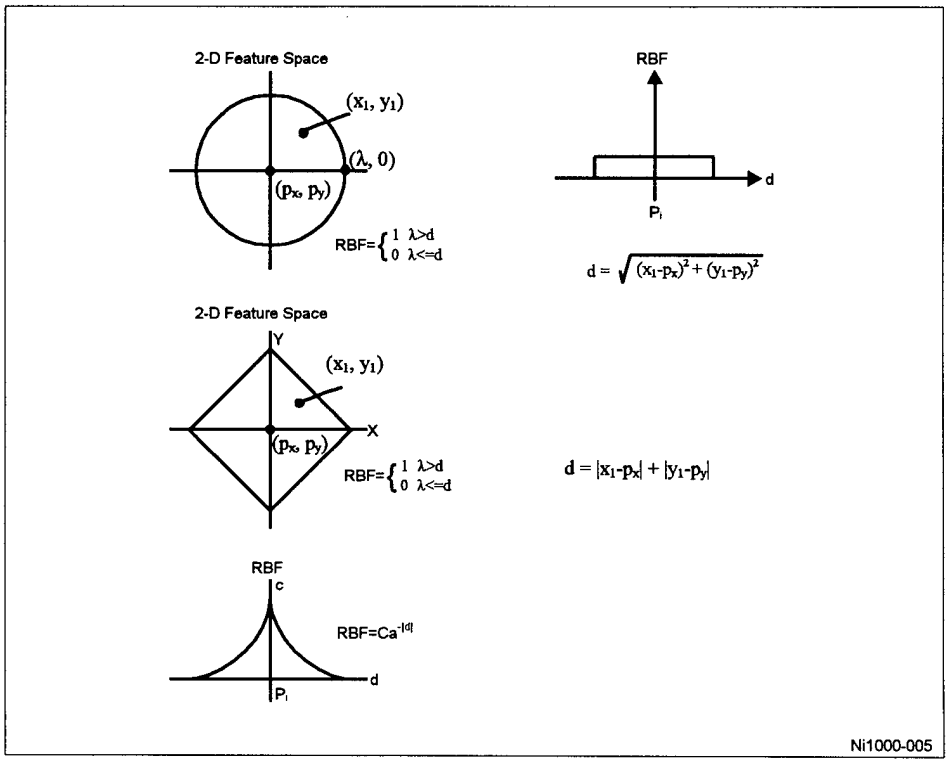


Figure 2-3. RBF Examples

Input vectors describe points in a multidimensional *feature space*, which is the complete range of possible patterns of input data. In the example of the nut and bolt sorter, the axes of the

feature space are the inputs from the sensors; they may include size, color, weight, and shape of each piece of hardware. Features belonging to each class tend to cluster into regions just as nuts in the example have similar weights. The learning process approximates the locations of the regions in feature space using Radial Basis Functions (RBF). Three examples of RBFs appear in Figure 2-3. An input close to the center of an RBF elicits a large response. Inputs far from the center produce insignificant responses. Figure 2-4 illustrates a two dimensional class region approximated using circular *fields of influence* centered on stored examples.

The classification process maps input vectors onto feature space. The classifier then outputs the class of the region into which the input fell. If multiple classes fire, as may occur when an input falls into overlapping regions, probabilistic information can help resolve ambiguities.

Examples of radial fields of influence are common in biological systems. Neuroscientists have found that a ganglion cell in the retina responds only to light detected by a small, circular area called the cell's receptive field. The cell produces no response, or it is actually inhibited from producing a response if it detects light outside the area. Receptive fields of adjacent cells tile the retina in overlapping RBF receptive areas to cover regions in feature space.

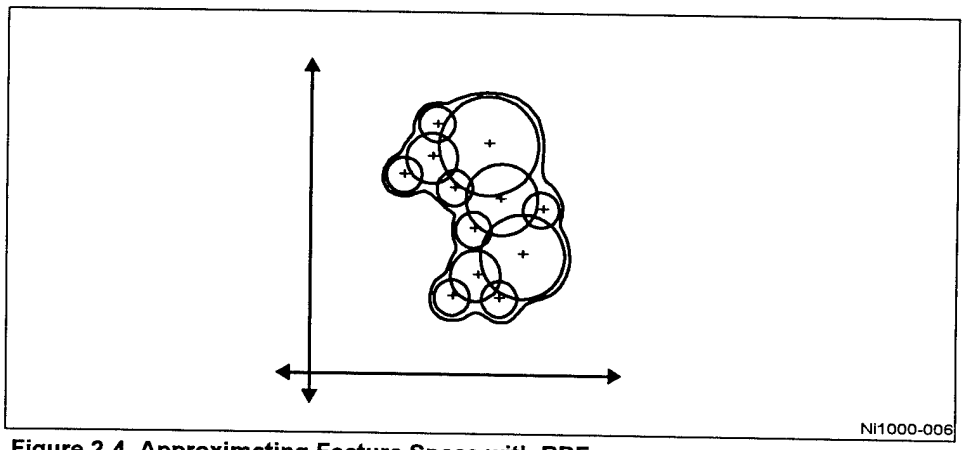


Figure 2-4. Approximating Feature Space with RBFs

2.2. Learning and Classification Algorithms

During the learning process, the recognition system develops a memory of class-region approximations and probability-density-function estimates for each point in feature space. The system studies a *training set* of input examples, along with their class labels, and learns to distinguish among the classes under the control of a learning algorithm.

The Ni1000 Recognition Accelerator's on-chip microcontroller is intended to execute the learning algorithm code. It facilitates incremental learning outside the factory to adapt and customize the chip's memory to special circumstances that arise in the field. The Accelerator supports algorithms like Restricted Coulomb Energy (RCE) and Probabilistic RCE (PRCE) as

well as other radial basis function algorithms like Probabilistic Neural Networks (PNN) and custom algorithms.

Alternatively, the prototypes and their parameters can be loaded into the chip from outside. They may be the result of learning performed off chip, or they may simply consist of data that can take advantage of the calculations performed by the classification pipeline.

2.2.1. Restricted Coulomb Energy (RCE)

In RCE and other algorithms compatible with the Ni1000 Recognition Accelerator, learning is a process of approximating class regions in feature space with radial basis functions. The algorithm selectively stores a set of prototypical inputs, called *prototypes*, that are obtained from the training data, and it assigns a field of influence to each prototype, defined by a threshold radius .

Before learning begins, the designer specifies minimum and maximum radius values, r_{min} and r_{max}. During learning, the Accelerator computes the distance between each input vector and any existing prototypes. Distances are defined as the sum of the differences between each feature of an input vector, *(u)*, and the corresponding feature of a stored prototype vector, *(p)*. This metric, *d,* is called the *city-block distance* or *Manhattan distance* and is computed as:

$$d = \sum_{0 \le i \le 256} |u_i - p_i| \qquad (1)$$

The algorithm then compares the distances with each prototype's radius to determine whether or not the training input vector (which has an associated class) is within that prototype's field of influence. If the input vector does not fall within the field of influence of any prototype, the Accelerator stores it as a new prototype along with its class label and sets its radius to r_{max}. If, however, the input is within the field of influence of a prototype in a class different than the one with which the input is tagged, the input is stored and both its radius and the prototype's radius are set to the distance between them. The algorithm does not store the input as a new prototype if it only falls within the field of influence of one or more prototype(s) in the same class. Instead, it increments the count for each firing prototype. Iterations through the data set continue until prototype storage and radius adjustments stop.

The pseudo-code for a typical RCE or PRCE training procedure is shown below. In the procedure, the italicized steps are for PRCE only. All others apply to both RCE and PRCE. The on-chip microcode supplied with the development system performs both RCE and PRCE specifications during learning.

```
{ // Learn RCE/PRCE
Set r_min and r_max
do
    { // begin epoch
    Reset C_k's
    do
        { // learn vector
        Input next vector and its associated class (class_i).
        Compute the input vector's distance to each of the stored prototypes.
        Compare distances to corresponding prototype's lambdas and determine firings.
        Compute D_min using the D_min calculation procedure shown below.
        If ((no prototypes of class_i exist) or (no prototypes fire))
            store input vector with radius = D_min
        else for k = 1 to k_highest stored
            if (P_k firing and (class of P_k = class_i)) then C_k = C_k + 1
            if (P_k firing and (class of P_k != class_i) and (radius of P_k != r_min )) then
                {
                if (distance to P_k > r_min ) then radius of P_k = distance to P_k
                else radius of P_k = r_min
                }
            }
        } while more input vectors are available // learn vector
    } while ((new prototypes were stored) or (any radius changed)   // end of epoch
} // Learn RCE/PRCE
```

In the above pseudo-code, D_{min} is calculated as:

```
D_min = r_max
For k = 1 to k_highest stored
{   if ((distance to P_k< D_min) and (class_i != class of P_k))
    {   if (distance to P_k < r_min)
            D_min = r_min
        else
            D_min = distance to P_k
    }
}
```

During classification, the algorithm computes the distances from the input vector to each of the prototype vectors stored during learning. If the distance to a prototype is less than the prototype's lambda, the input receives the prototype's class label. The result of the RCE classification is a union of all firing classes. Probabilistic methods like PRCE described below can resolve ambiguities in case of multiple firings.

2.2.2. Probabilistic RCE (PRCE)

PRCE outputs the probability that an input belongs to a given class. While learning, the classifier stores prototypes and computes radii using the RCE algorithm. In classification, it computes probability density estimates throughout feature space for each class. The Ni1000

Accelerator performs the probabilistic calculations in parallel with the class firing calculations used in RCE.

The PRCE algorithm finds its roots in Bayes Decision Theory. A Bayesian classifier computes an input's probability of belonging to each class by using corresponding class probability density functions (PDF's). In a simple two-category problem in which class A and class B are the possible categories, an input vector *u* is a member of:

Class A when $C_A * f_A(u) > C_B * f_B(u)$ (2a)

or

Class B when $C_A * f_A(u) < C_B * f_B(u)$ (2b)

where C_A and C_B are the *a priori* probabilities of pattern occurrence from category A and B, respectively. The *a priori* probability C_A is the ratio of the number of training patterns belonging to class A, to the total number of training patterns, and $C_B = 1 - C_A$. Functions f_A and f_B are the probability density functions for class A and class B, respectively.

The construction of decision boundaries requires knowledge of the underlying PDF's, which must be determined through training. The PDF for each class may be constructed from a linear combination of a family of radially symmetric distribution functions, each centered on a prototype stored during training. Decaying exponentials shown in Figure 2-5 approximate each prototype's contribution to the probability density (PD) of its class:

$$PD_{Class} = \sum_{P_j \in Class} C_j \cdot 2^{-k_j \cdot d_j} \qquad (3)$$

where *d* is the Manhattan distance between the input and the prototype, *k* is a decay constant specified by the host program before initiating PRCE classification, and C_j is the *a priori* rate obtained during training. C_j is the number of training patterns that belong to a class that fell within the distance r_j of the prototype p_j. An example of the resultant PDFs appears Figure 2-6.

The PRCE classification is a mapping of the input vector onto feature space, resulting in a set of PDs (one value for each class). The host can then select the largest PD for each vector to determine its class.

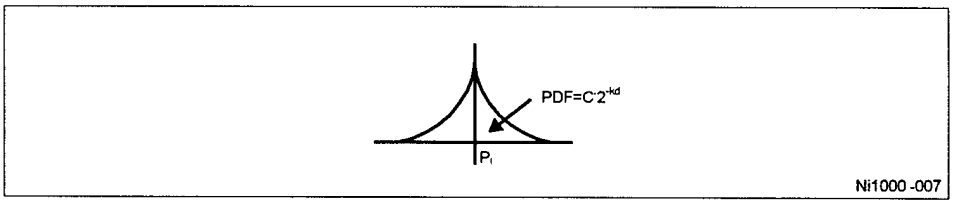


Figure 2-5. PD Contribution of a Single Prototype

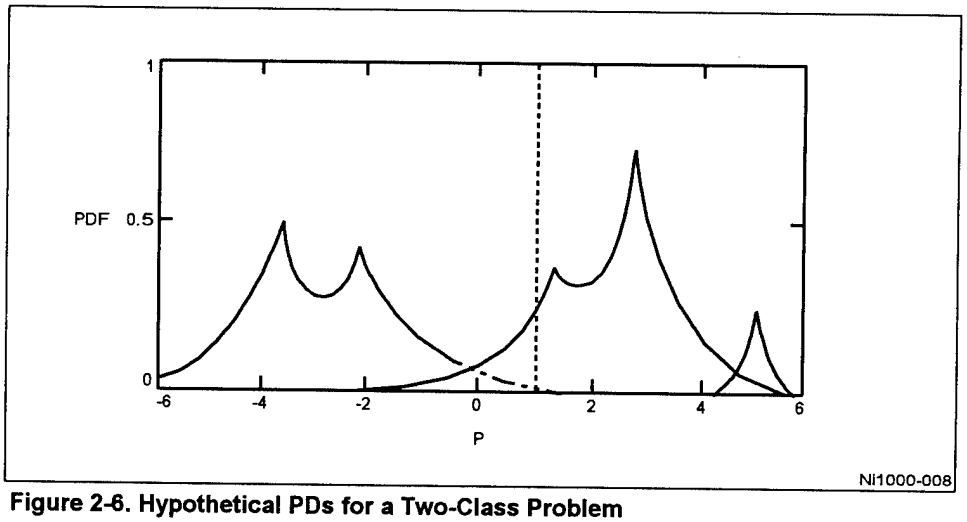


Figure 2-6. Hypothetical PDs for a Two-Class Problem

3. HARDWARE ARCHITECTURE

The Ni1000 Recognition Accelerator consists of two main parts: a dedicated *classifier* engine and a general-purpose 16-bit *microcontroller*. The classifier implements the model described in the previous chapter, while the microcontroller code implements the on-chip learning algorithms and interacts with software running on the host.

Figure 3-1 is a block diagram of the internal hardware architecture. The upper part of Figure 3-1 shows the classifier, the middle part shows the interface to the host , and the bottom part shows the microcontroller.

In an application environment, the classifier receives data from the host system through the bus interface, processes it, and sends the classification results back through the bus interface to the host. The classifier exploits both array and pipeline parallelism to perform over 10,000 classifications per second. The parallel hardware of the Distance Calculation Units and their tight coupling to the Prototype Array (PA) are responsible for much of this processing power. The Prototype Array holds 1024 (raw prototypes) x 256 (feature values) x 5 (bits per feature), for a total of 1.3 million non-volatile Flash storage bits. Note that the 1024 x 256 physical array provides 1000 x 222 storage locations usable for classification. Additional prototype storage is possible using multiple Ni1000 Accelerators and a higher effective input feature resolution is possible using two or more Ni1000 features to represent each feature.

Each of the 512 parallel Distance Calculation Units calculates a city-block distance (see Chapter 2) by summing the differences produced by its absolute value subtractor. The subtraction is performed on each feature of the input vector and the corresponding feature of one of the prototype vectors stored in the Prototype Array. The DCUs are multiplexed twice in time to achieve a sustainable processing rate of over 12 billion operations per second and a bandwidth of over 30 Gbps.

The classifier's Math Unit (MU) calculates probability densities and results classes concurrently. It processes floating-point data and computes the exponential and other mathematical functions that appear in equations (1) and (3) of Chapter 2. The MU uses a six-stage pipeline with a resolution of 16-bits for floating-point computations (10-bit mantissa and 6-bit exponent). It places results into one of two static RAMs. This double-buffering scheme allows the Math Unit to continue processing a second vector without interrupting the classification pipeline. The Prototype Parameter RAMs (PPRAMs) hold parameters like the radius(r), smoothing factor (k), and Count(C), described in Chapter 2.

The bottom part of Figure 3-1 shows the microcontroller. It is a fully custom, 16-bit, Harvard-architecture microcontroller that supervises learning, performs chip maintenance tasks, and maintains communication with the host. It can also exchange interrupts with the host. The 4k x 16-bit PGFLASH Flash memory stores the microcontroller programs. All memory devices are memory-mapped to the microcontroller's address space, with the exception of the the microcontroller's program memory (PGFLASH). Other facilities available to the

microcontroller include 256 words of general-purpose static RAM (GRAM) and a free-running 32-bit timer. Classification must stop while the microcontroller accesses these memories.

The microcontroller can enable an automatic classification mode in which a series of logic blocks, arranged as a pipeline, process data and output the results to the host. The classification pipeline consists of input buffers, distance calculation units, a large FLASH prototype array that stores the results from the learning process, a mathematical unit and its output memories, and an output buffer. At 25MHz, the pipeline can classify over 10,000 input vectors per second, in which each input vector has up to 222 5-bit features. The performance is made possible by the Ni1000 parallel architecture, which can execute over 12 billion operations per second at 25 MHz. A typical Von Neumann machine would need to execute more than 40 billion instructions per second to approach the processing rate achieved by the Ni1000 Recognition Accelerator.

The middle part of Figure 3-1 shows the interface to the host, which consists of input buffers (IRAM), an output buffer (ORAM), and sixteen I/O control registers. The external data bus can be either 32 or 64 bits wide and will perform single-clock burst transfers. The input stage buffers two full-sized vectors. The outputs can be either in IEEE standard 32-bit floating-point format or the internal 16-bit floating-point format. Both the host and the Accelerator's on-chip microcontroller can access the sixteen 16-bit I/O control registers. The registers contain various control parameters for the Accelerator and provide a general channel for communication between the microcontroller and the host.

The middle part of Figure 3-1 shows the interface to the host, which consists of input buffers (IRAM), an output buffer (ORAM), and sixteen I/O control registers. The external data bus can be either 32 or 64 bits wide and will perform single-clock burst transfers. The input stage buffers two full-sized vectors. The outputs can be either in IEEE standard 32-bit floating-point format or the internal 16-bit floating-point format. Both the host and the Accelerator's on-chip microcontroller can access the sixteen 16-bit I/O control registers. The registers contain various control parameters for the Accelerator and provide a general channel for communication between the microcontroller and the host.

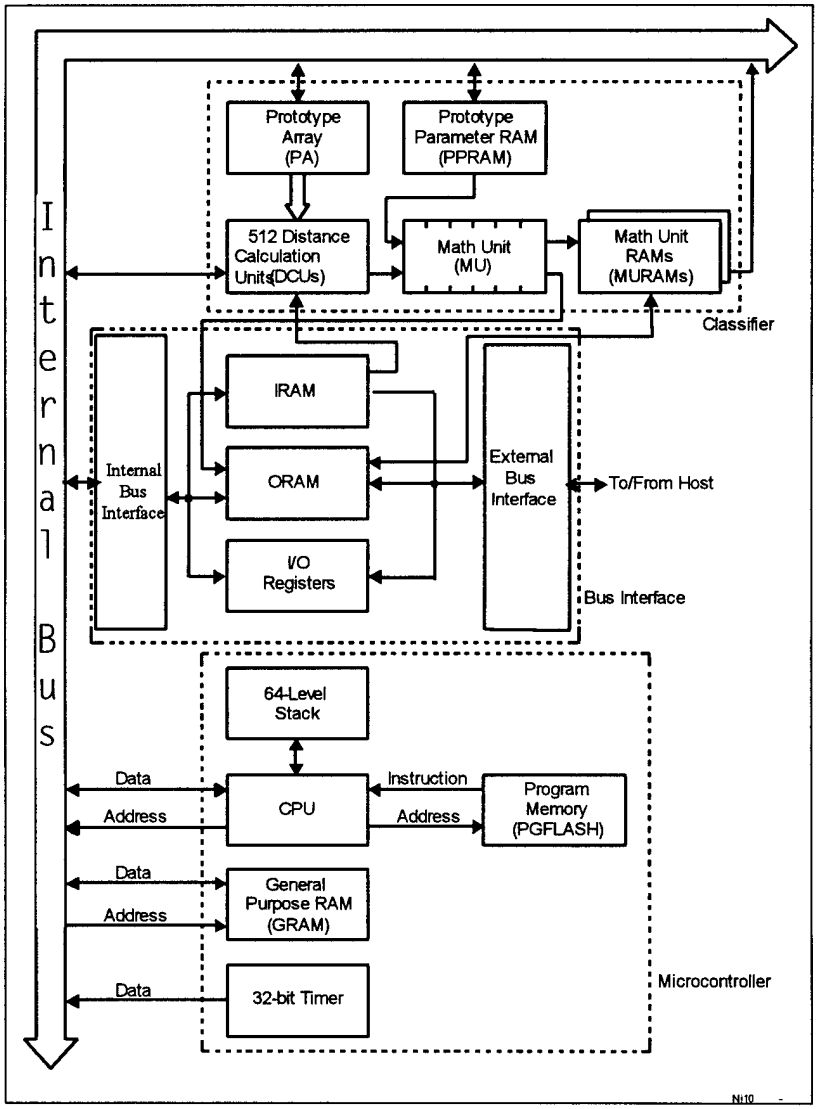


Figure 3-1. Internal-Architecture Block Diagram

3.1. The Classifier

The classifier consists of the pipeline shown in Figure 3-2. While data is loaded into the double buffer at the input of the pipeline or read from the output of the pipeline, the classifier can compare a previously loaded input vector against the prototype vectors in the prototype array.

The classifier consists of the following units:

- *Input RAM (IRAM)*—a double buffer consisting of two 256 x 5 memories. Each memory can store one input vector. The IRAM is part of the bus interface unit, which is described in Section 3.2.
- *Prototype Array (PA)*—a flash memory that holds the prototype vectors allocated during learning, i.e. the coordinates of the Radial Basis Function (RBF) centers.
- *Distance Calculation Units (DCUs)*—a 512-processor array that performs the distance calculations between an input vector and each prototype vector in the PA.
- *Prototype Parameter RAMs (PPRAMs)*—a memory that holds all of the data that defines an RBF except its prototype vector (which is stored in the PA). This data includes the RBF radius, number of vectors it recognized during exposure to the training set, etc. Unlike the PA, the PPRAM is not flash memory; it is static RAM. Typically, it is loaded during power-on initialization from off chip or from a reserved section of the prototype array (PA).
- *Math Unit (MU)*—a six-stage pipelined processor that implements the decay function for calculating probability densities. It also applies the threshold function, to decide whether an input vector falls within an RBF's field of influence.
- *Math Unit RAMs (MURAMs)*—a set of memories that receives the class IDs that are classified as similar to the input vector. It also holds the accumulated probability densities for each class.
- *Output RAM (ORAM)*—a buffer that receives the classification results for a vector and optionally reformats the probability densities from the internal 16-bit floating-point format into a format compatible with the standard IEEE 32-bit floating-point format. The ORAM is also part of the bus interface unit, which is described in Section 3.2.

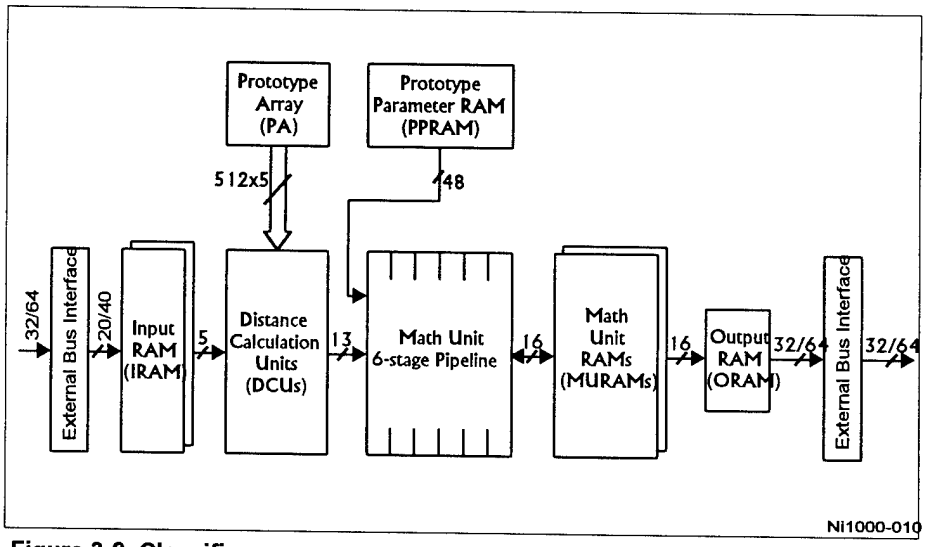


Figure 3-2. Classifier

3.1.1. Distance Calculation Units (DCUs)

Figure 3-3 shows an individual distance calculation unit (DCU) from the 512-unit array. The DCUs are statically associated with PA columns. Due to redundant array elements and back up storage allocation, as few as 500 DCUs may be used for classification at any time. The DCU computes the absolute value of the difference between a feature (dimension) of the input vector and the corresponding feature of a prototype. The DCU then accumulates that value into a running tally of the distance between the input vector and the prototype. Such a distance is calculated between the input vector and each valid (i.e. not disabled) prototype.

The DCU accumulates a sum of the absolute differences, called *city-block distances,* between the input vector and each prototype vector in each dimension. The following equation expresses the city-block distance, *d,* between an input vector *U* with *i* dimensions and a prototype vector *P*.

$$d = | u_0 - p_0 | + | u_1 - p_1 | + \ldots | u_i - p_i |$$

The DCU has two accumulators and is used in a two-cycle mode, in which half of the prototypes in the PA are processed in one cycle, and the other half in the following cycle. When there are 500 prototypes or less, a one-cycle mode is used that does not require the second accumulator. When there are more than 500 prototypes, the two-cycle mode is used, with the other accumulator being used during the second cycle. Section 3.6 describes the timing of the DCUs and the classification pipeline.

At the end of a classification pass through the prototype array, the values in the accumulators represent the city-block distance between the input vector and each valid prototype vector stored in the PA. They are used by the MU to perform the probability and threshold functions.

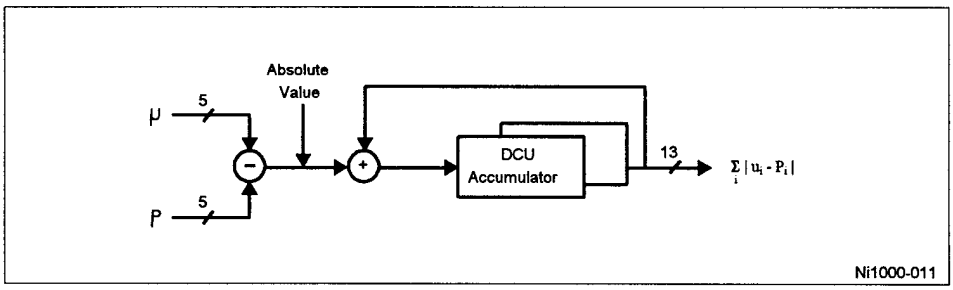


Figure 3-3. Distance Calculation Unit (DCU)

3.1.2. Prototype Array (PA)

The PA is a flash memory holding up to 222 five-bit features for each of up to 1000 prototype vectors (or 8000, for problems with 26 dimensions or less) plus array management and network backup data. Flash memory is a form of non-volatile electrically-erasable memory. See Section 3.2.4 for a description of writing to the PA.

Figure 3-4 shows a conceptual view of the PA being accessed by the DCUs. An input vector is presented as a stream of 5-bit integers. These are the individual features of the input

vector. The 500 DCUs compare one feature of the input vector against the corresponding feature in up to 500 prototype vectors simultaneously. After the last prototype has been processed, the DCUs pass their accumulated city-block distances to the next stage of processing, the math unit (MU) pipeline.

PA flash memory can only be programmed by the microcontroller. Programming PA requires a 12V voltage applied to the Vpp pin.

Two address ranges in the microcontroller's address space are used to read the PA. An address in the range from B000h to B3FFh is used to specify one of the 1K prototype vectors to read. Note that, due to redundancy and remapping, a column address and prototype number may not match. Another range of 256 addresses from B800h to B8FFh specifies which features of the selected prototype vector are to be read. Reading is a two-step process in which a first read specifies either the vector or the feature, and a second read specifies the remaining quantity. Either the vector or the feature can be specified first. Valid data is returned on the second read. The upper six bits of the data are undefined . The lower ten bits are the value of one 5-bit feature in both true and complement form. Figure 3-5 shows the alignment of a 5-bit feature, p[0:4]. Complemented values are indicated by a bar over their bit names.

Figure 3-6 shows the architecture of the PA during programming. There are four registers used to control the PA during programming: the control and status registers, *CSA* and *CSB*, and the hardware mode setting registers, *AUX* and *MODE*. See Section 5.1.6 for details.

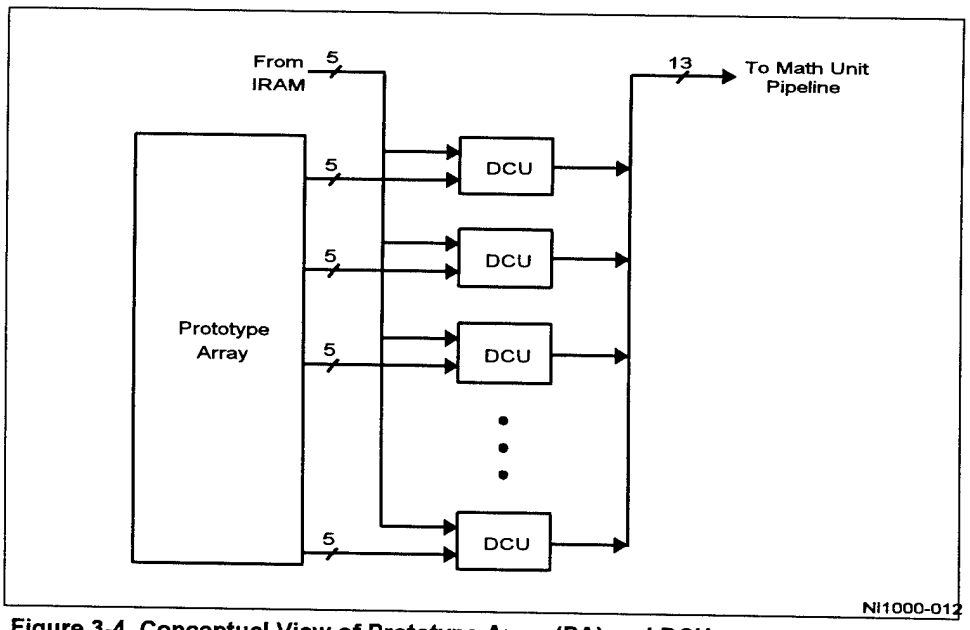


Figure 3-4. Conceptual View of Prototype Array (PA) and DCUs

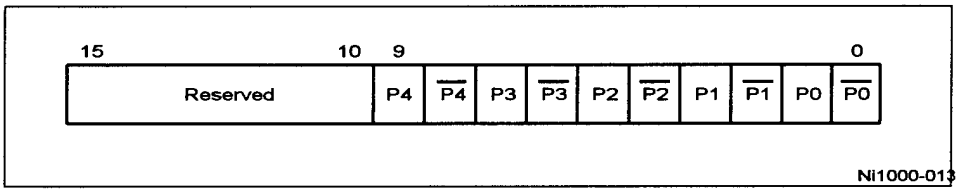


Figure 3-5. PA Data Format

3.1.3. Prototype Parameter RAM (PPRAM)

As each 13-bit city-block distance enters the MU, it is accompanied by 48 bits of parameters for its RBF, which come from the PPRAM. These parameters include several fields, such as the RBF radius. The PPRAM can only be written by the microcontroller, and appears as three 16-bit banks in the microcontroller's address space. Figure 3-7 shows the PPRAM.

The addresses of the three banks, PPRAM1, PPRAM2, and PPRAM3, are given in Chapter 5. The fields of the 48-bit word passed to the MU pipeline, broken down by bank, are given in Figure 3-8.

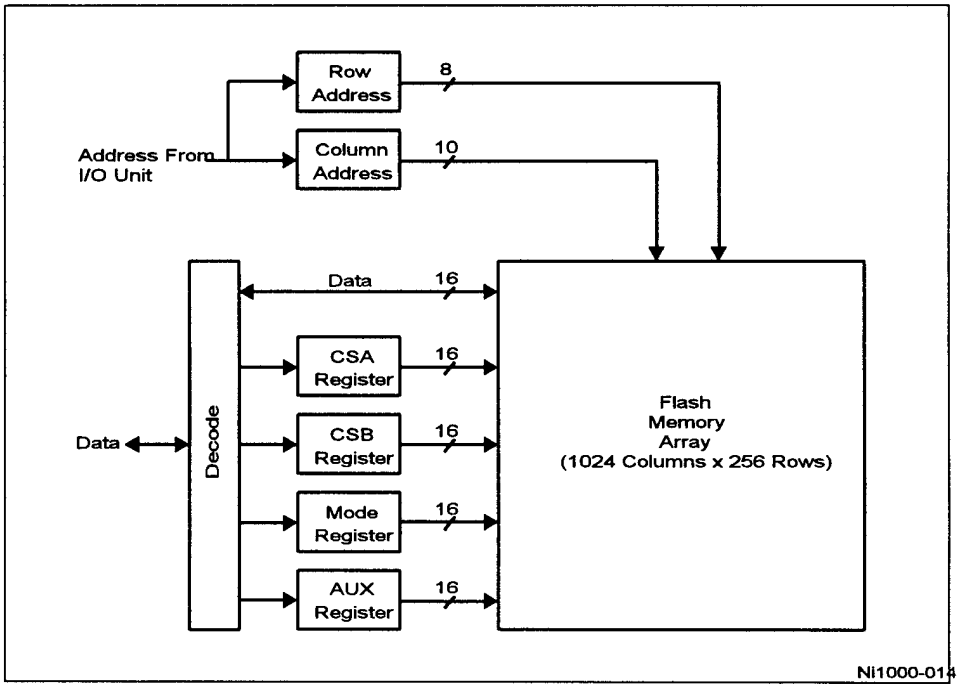


Figure 3-6. PA During Programming

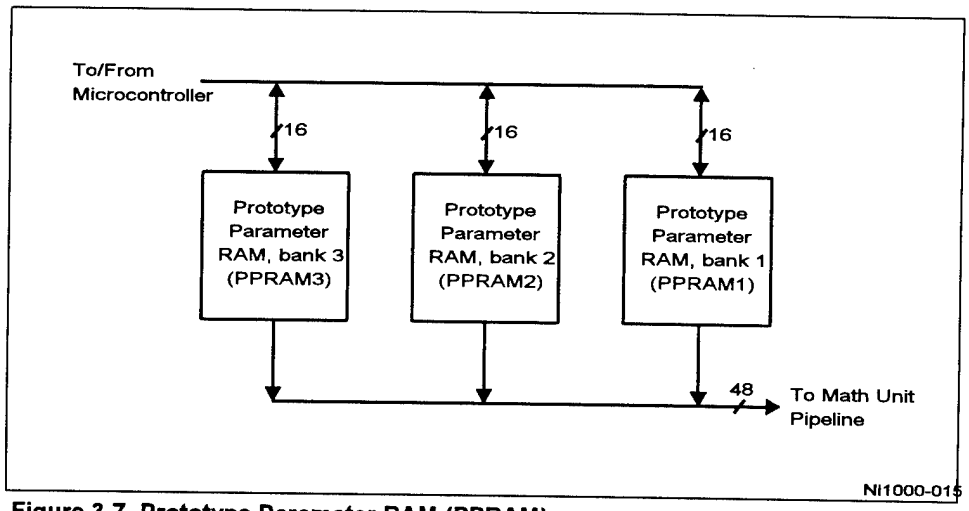


Figure 3-7. Prototype Parameter RAM (PPRAM)

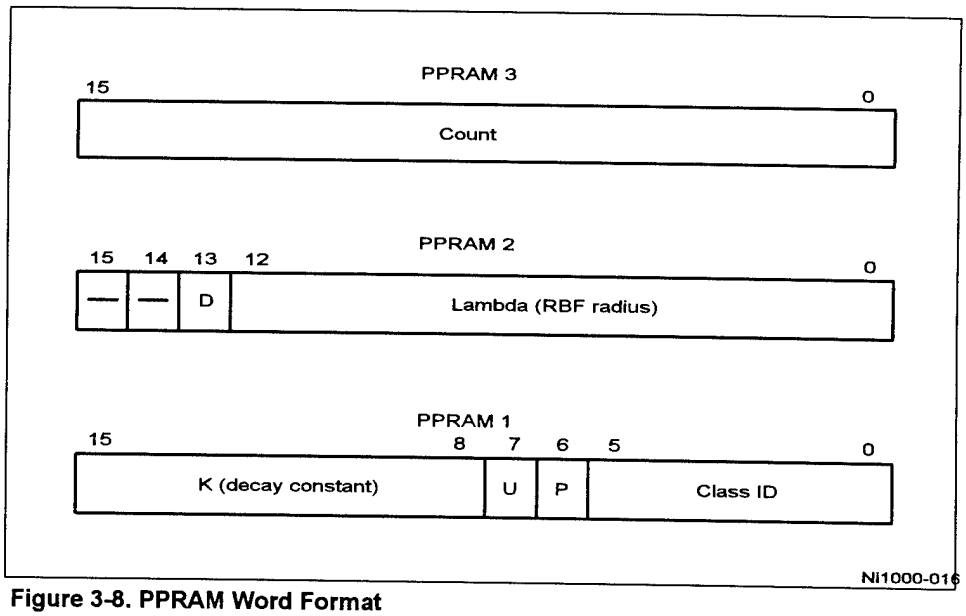


Figure 3-8. PPRAM Word Format

The fields of a PPRAM word are:

- *Count—C[0:15]*—the number of training vectors that fall within this RBF during learning; used as a factor during classification when calculating probability density.
- *Disable Flag—D*—set to disable this prototype.
- *Radius—R[0:12]*—the RBF threshold distance.

- *Decay-Constant Mantissa—$K_m[0:3]$*—unsigned mantissa of the decay constant of the exponential function.
- *Decay-Constant Exponent—$K_e[4:7]$*—signed exponent of the decay constant of the exponential function.
- *"Used" Flag—U*—set when the PPRAM word is loaded with a valid prototype.
- *Probabilistic—P*—indicates that the RBF threshold distance for this prototype is the minimum allowed threshold distance. This bit is passed through to the class identification result to indicate that only probabilistic, not deterministic, classification is possible with this prototype.
- *Class—S[0:5]*—the class ID of the prototype.

All prototypes that have their *Used* flag set to 1 will be processed by the classifier. To avoid the possibility of processing spurious data, all locations in the PPRAM should be written whenever the chip is loaded with a new set of prototypes, and unused prototypes should have their *Used* flag cleared to 0.

3.1.4. Math Unit (MU)

The inputs to the first stage of the MU pipeline are the fields described above for the PPRAM, accompanied by D, the 13-bit city-block distance calculated between the input vector and the prototype by the DCUs. The MU pipeline performs two functions based on the point in feature space defined by the input vector: it reports the class(es) of prototype(s) whose field of influence includes that point and it calculates the probability density for every class. Figure 3-9 shows the architecture of the MU pipeline. The Math Unit transfer function is described in Section 5.1.8.

The MU pipeline and the next functional block of the classifier, the math unit RAMs (MURAMs), are closely tied together. Several stages of the pipeline access data in the MURAMs. The ports shown in Figure 3-9 are connected to the MURAMs, as shown in Figure 3-10. The first stage of the MU pipeline writes to the *Flag MURAM*. The second stage writes to the *Class List MURAMs*. The fifth stage reads from the *Probability MURAMs*, and a result is accumulated which is written back to the *Probability MURAMs* following the sixth pipeline stage.

Recognition of whether an input vector falls within a prototype's field of influence is indicated in the first stage of the MU pipeline by subtracting the threshold radius from the city-block distance and making a decision based on the sign of the result. This indication is used to update the 1-bit flag MURAM to indicates that the network associated that output class with the input vector. If this prototype is the first of its class to indicate recognition of the vector, a counter is incremented, the address it issues is used to allocate the next entry in the class-list MURAM and the prototype's *P* bit is copied into the result for that class.

Also in the first stage of the pipeline, the smoothing factor (k) and the city-block distance (D) are multiplied, and this floating-point product is split into three components for separate processing. This split is done for efficiency, allowing each component to be handled by a circuit that is easy to implement in hardware, then the output components are recombined.

After the first stage of the MU pipeline (See Figure 3-7), the four-bit exponent is processed by aligning its bits to the final result. The six most-significant bits (MSBs) of the mantissa are processed by a ROM lookup of the reciprocal of the exponential. The six least-significant bits are multiplied by a constant factor, the natural logarithm of 2 (i.e. ln 2). The computation of the exponential decay function occurs in the second and third stages, with recombination occurring in the third stage.

If an output class was indicated in the first stage as being recognized for the first time, its class ID is written to the class-list MURAM one cycle later. The address for this cycle comes from one of two class counters, and the data comes from the latch for the class ID in the second stage of the pipeline. Two counters are provided so that when the probability and class-list MURAMs swap (they are both double buffers), the counters can also swap, giving the circuits that control the ORAM a value for the number of classes in the class list.

The third stage of the pipeline produces the floating-point probability density for an RBF at the point in feature space described by the input vector.

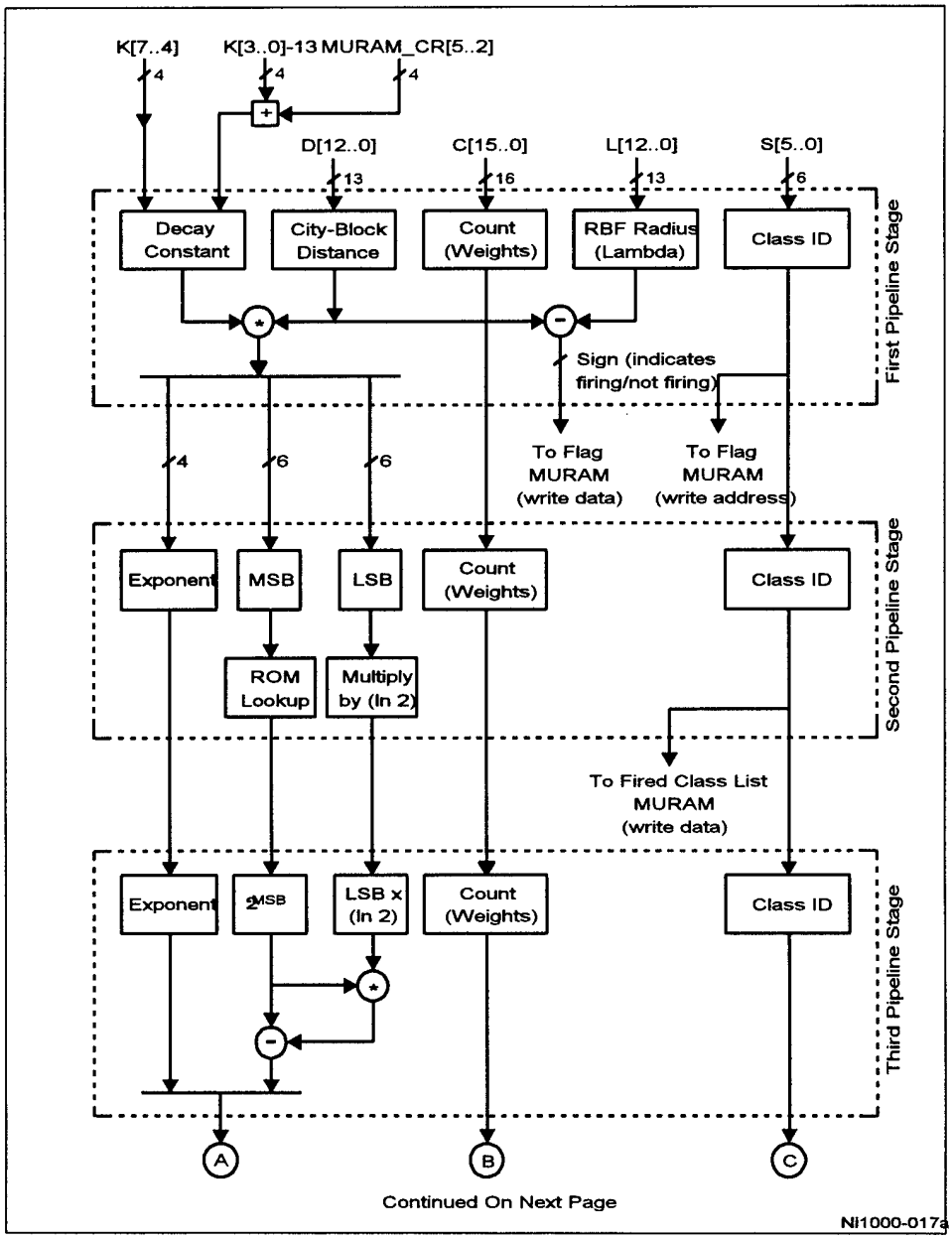


Figure 3-9. Math Unit (MU) Pipeline

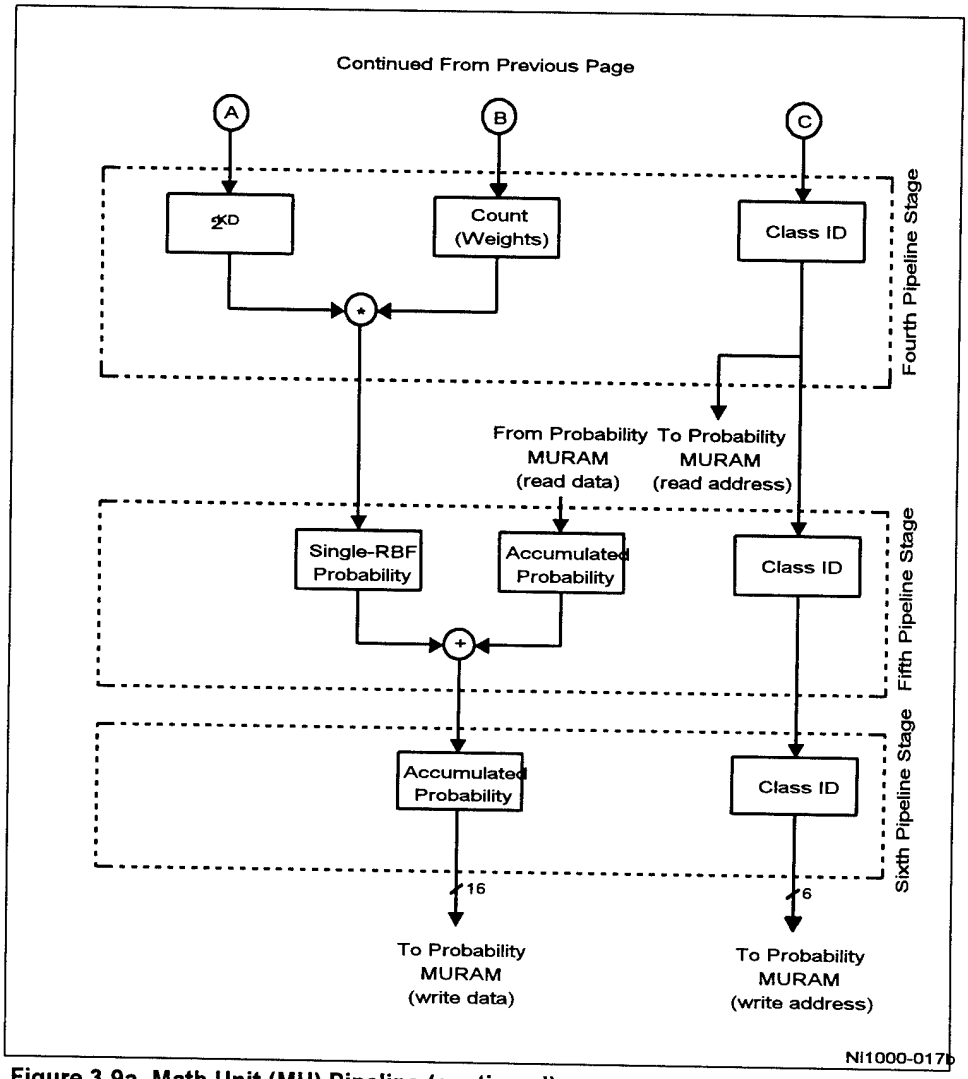


Figure 3-9a. Math Unit (MU) Pipeline (continued)

In the fourth stage of the pipeline, the probability value generated in the previous stage is scaled by the prototype's *Count*, which is typically a count of the number of times a vector in the training set fell within the radius of that prototype, although some algorithms may establish this factor using other means. The class ID from this stage is then sent to the probability MURAM.

In the fifth stage of the pipeline, the scaled probability density of a single prototype is added to the accumulated value for all prototypes of the same class. This floating-point sum is written to the probability MURAM after the sixth stage of the pipeline, addressed by the sixth-stage class ID.

Once the last probability calculation has been performed for the input vector, the double-buffered MURAMs reverse roles, so that the classification results for the current vector can be uploaded to the host through ORAM while the next vector is processed. Both the class list and probability density values are computed simultaneously, so either or both can be uploaded to the host without re-running the classification.

3.1.5. Math Unit RAMs (MURAMs)

The output of the MU pipeline is loaded into the MURAMs. The architecture of the MURAMs, shown in Figure 3-10, is intimately tied to the pipeline. Four of the six stages of the pipeline shown in Figure 3-9 either provide an MURAM address, read data from an MURAM, or write data to an MURAM. The MURAM memories include:

- *Flag MURAM*—a 1 x 64 memory used as a table of classes that have already recognized the input vector being presented. Each entry corresponds to one of the 64 possible classes.
- *Class List MURAMs*—an 8 x 64 x 2 double buffer holding a list of the class IDs of recognized classes. A new byte is allocated every time a new class is encountered. (Unlike the other MURAM memories, the memories in this buffer are not indexed by class ID; they are addressed by counters, so they grow up from address zero.)
- *Probability MURAMs*—a 16 x 64 x 2 double buffer that accumulates the probability density for each class. As with the flag MURAM, each MURAM address corresponds to one of the 64 classes.

The MU pipeline sends its data to the set of class-list and probability MURAMs currently waiting to receive input (while the other set is available to unload data into the ORAM, discussed later). One set of MURAMs may be written with the data for the input vector being presented, while the other set passes data, if available, for the previous input vector to the ORAM or the microcontroller. After the input vectors have been processed, the MURAMs change roles.

The flag MURAM is not accessible to the ORAM, so it is re-used every cycle. It is indexed by the class ID. When a class is recognized, the bit addressed by the class ID is set. If the bit previously was clear, the class had not yet been associated with this input vector. This causes the class counter to be incremented and allocates a word in the class-list MURAM. The counter keeps a running tally of the number of classes, which is used to address the class-list MURAM when a new word is allocated.

The class-list MURAMs are 8 bits wide, consisting of a six-bit class ID, a seventh bit to indicate that the first prototype (highest numbered prototype) to recognize this input vector had been shrunk to minimum radius, and an eighth bit to indicate validity. When a prototype shrinks to the minimum radius, it is a probabilistic prototype. This bit aids in combined deterministic and probabilistic classification when prototypes are reordered to take advantage of this, i.e. that the deterministic prototypes of any class are all located at higher numbered prototypes than any of the probabilistic prototypes of that same class. Figure 3-11 shows the format of a byte in the class-list MURAMs.

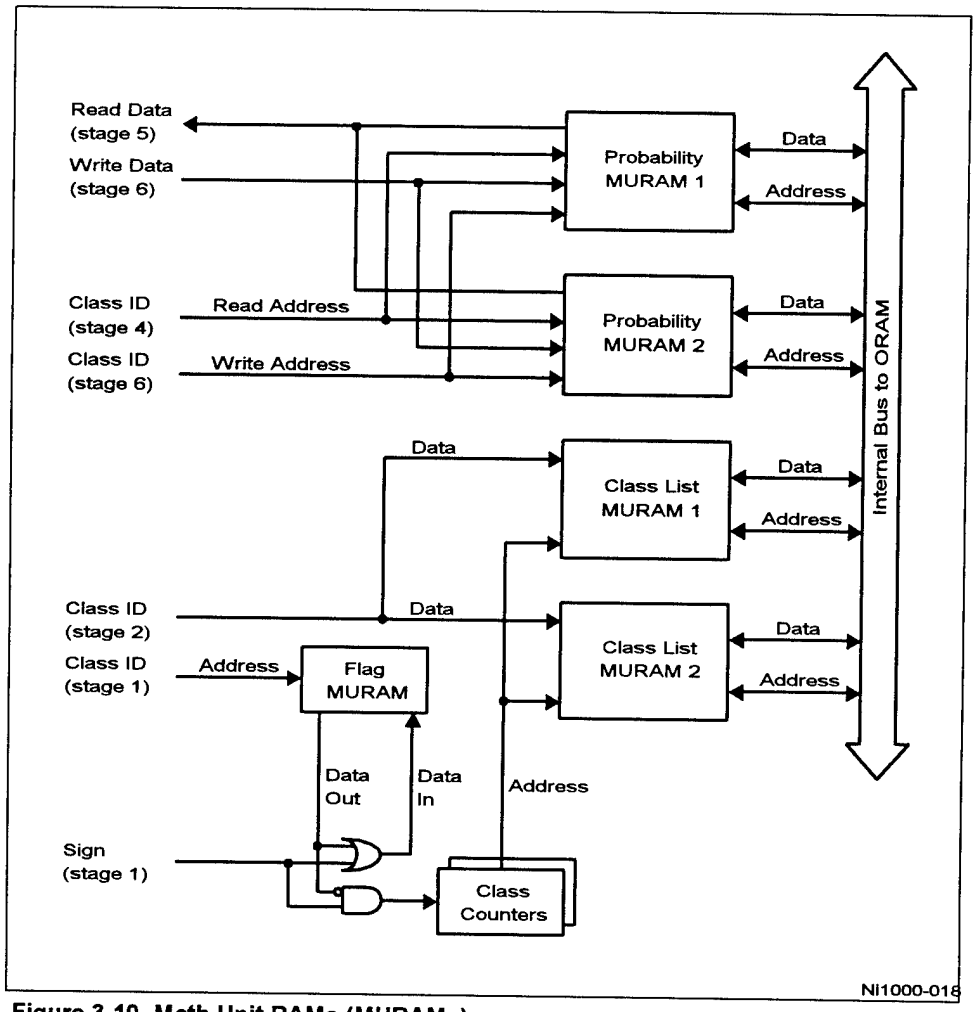


Figure 3-10. Math Unit RAMs (MURAMs)

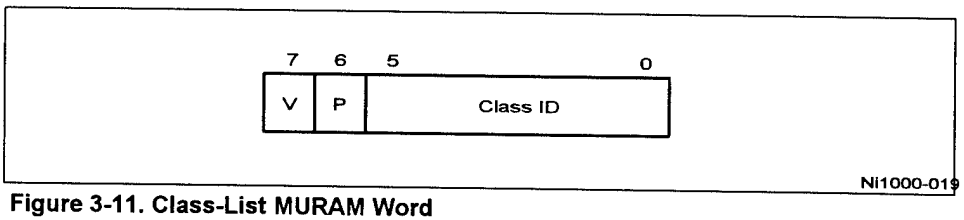


Figure 3-11. Class-List MURAM Word

The fields of a class-list MURAM word are:

- *Class ID—S[0:5]*—Class ID of a class that includes the input vector.
- *Probabilistic—P*—a prototype with the minimum radius recognized the input vector. This indicates that the first prototype to classify this vector was a probabilistic prototype.
- *Valid—V*—this word has been written since MURAM initialization.

The class-list MURAMs are addressed by a counter. The counter begins at zero and increments as new classes are encountered.

The probability MURAMs consist of a 16-bit floating-point accumulator in the internal format of the Ni1000 Accelerator. The internal format may be passed through ORAM or translated into an IEEE-compatible format as it passes out the ORAM. Figure 3-12 shows the internal format of a word in the probability MURAMs.

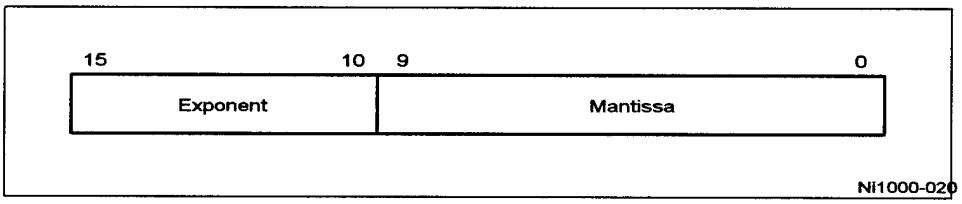


Figure 3-12. Probability MURAM Word

The fields of a probability MURAM word are:

- *Exponent*—six-bit 2's-complement exponent.
- *Mantissa*—10-bit fractional mantissa (i.e. 0 <= mantissa < 1).

3.2. Bus Interface

The bus interface is used for communication between the host and either the microcontroller or the classifier. It is used by the host to program the flash memory, write vectors to the input buffer, read classification results from the output buffer, and access a set of registers used to interact with the microcontroller.

The bus interface has an input and an output side, as shown in Figure 3-13. The external bus interface handles bus cycles to the host, and the internal bus interface handles bus cycles from the microcontroller. These interfaces are used to access these resources:

- *I/O Registers*—a set of sixteen 16-bit registers used for communication between the host and the microcontroller.
- *Input RAM (IRAM)*—a double buffer consisting of two 256 x 5 memories. Each memory can store one input vector. The IRAM is also integrated into the architecture of the classifier's pipeline, shown in Figure 3-2.

- *Output RAM (ORAM)*—a buffer that receives the contents of the current MURAM, and optionally reformats the probability values from the internal 16-bit floating-point format into the standard IEEE 32-bit floating-point format. The ORAM is also integrated into the architecture of the classifier's pipeline, shown in Figure 3-2.

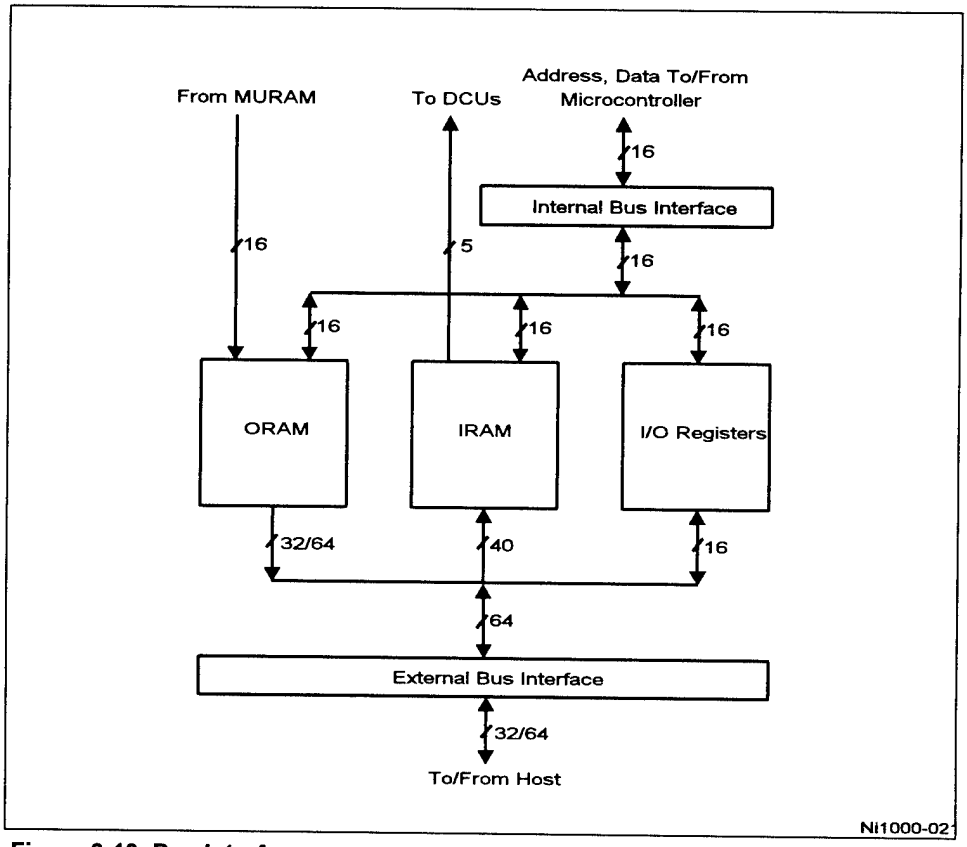


Figure 3-13. Bus Interface

Both 32- and 64-bit data bus widths are available, as selected by the 64/32# signal. This signal is not allowed to change dynamically. The chip must be reset following a change.

The Ni1000 Accelerator may appear to external hardware as a block of memory. However, the host can only access IRAM and ORAM through addresses that act like I/O ports, in which the same address is accessed over and over, until all data has been transferred. Register bits indicate when the buffer memories are about to overflow or underflow, and a bit in the CRA register can be programmed to cause assertion of the service request SRQ# output to the host when the ORAM is full (see Chapter 5 for a description of the CRA register).

The IRAM, ORAM and virtually all other memories in the Ni1000 are mapped into the microcontroller's address space and can be accessed by the microcontroller when the classifier is not running. Refer to Chapter 5 for the addresses of these resources.

The main signals of the bus interface are:

- *CLK*—clock input.
- *A[0:15]*—address bus, input from host.
- *D[0:63]*—64-bit bidirectional data bus.
- *ADS#*—address/data strobe input from host.
- *W/R#*—read/write input from host.
- *RDY#*—bus cycle termination output to host.
- *BRDY#*—bus cycle termination output with burst-mode request.
- *BLAST#*—input from host indicating the last data transfer of a cycle.

The latter two signals, BRDY# and BLAST#, are used for burst cycles, in which one 32- or 64-bit word is transferred per clock period. Burst cycles begin like non-burst cycles (which take a minimum of two clock periods each, if the external logic can support this speed by returning RDY# in the second clock) with the assertion of ADS#. However, the assertion of BRDY# by the chip allows the host to enter a bus mode in which each additional data transfer requires only one additional cycle. Host support for burst mode is optional. The Accelerator can accommodate vectors with up to 222 features, with each feature in a separate byte on the bus (the 5-bit feature must be placed in the high order 5 bits of a byte). See Section 3.7 for a detailed description of the bus signals. See Chapter 4 for the timing diagrams of bus cycles.

3.2.1. I/O Registers

There are sixteen 16-bit I/O registers. All of them can be read by the host and the microcontroller. Some registers are read-only, and others can be written from only one side of the interface. The detailed description of the addresses and bit assignments for these registers appears in Chapter 5.

Uses of some of these registers are defined by the microcontroller's software. By convention software running on the host and the microcontroller will use specific registers or fields within registers for particular purposes. These registers and fields mostly involve mode settings and network or algorithm parameters. The conventions defined by the standard microcontroller software are defined in Chapter 7.

Many of these registers may be used to send requests to the microcontroller software. Customer-specific software may redefine the meaning of some of these registers and fields. A few registers, however, are hardwired to critical control and status functions, and they will be used for the same purpose in all applications. Some of these functions are:

- *Interrupt*—interrupt the microcontroller.
- *Reset*—reset the Ni1000 Accelerator.
- *Reset IRAM*—separate reset bit for the IRAM autosequencer.
- *Reset ORAM*—separate reset bit for the ORAM autosequencer.
- *IRAM Full*—indicates the IRAM is full. Writing to a full IRAM causes an error condition.

- *IRAM Not Full*—indicates the IRAM has not yet been loaded with a complete input vector, and it is waiting for more data.
- *ORAM Empty*—indicates the ORAM is empty. Reading from an empty ORAM causes an error condition.
- *ORAM Not Empty*—indicates the ORAM has results waiting to be unloaded.

These bits give the host some control over the microcontroller, at least to the extent of interrupting it and resetting it. They also allow the host to poll the status of the input and output buffers without going through the microcontroller.

The Ni1000 Accelerator is always a slave to external bus cycles, so the host (or external control logic around the Ni1000 Accelerator, such as a bus-master interface controller) must initiate bus cycles. The host system may use the hardwired status flags for the IRAM and ORAM for flow control to prevent overflowing or underflowing any buffers.

The microcontroller also has a mechanism to interrupt the host. The mechanism is described later in this chapter.

3.2.2. Input RAM (IRAM)

The IRAM is shown in Figure 3-14. It is a double buffer consisting of two 32 x 40 banks. Each bank can store one 222-feature input vector (padded to 256 features), with 8 five-bit features packed into each 40-bit word. When the classifier is running, these banks are inaccessible to the microcontroller, however the host can load data to the IRAM. When the classifier is not running, the microcontroller can directly access the IRAM. The IRAM is mapped into the microcontroller's address space (see Chapter 5 for the microcontroller's memory map).

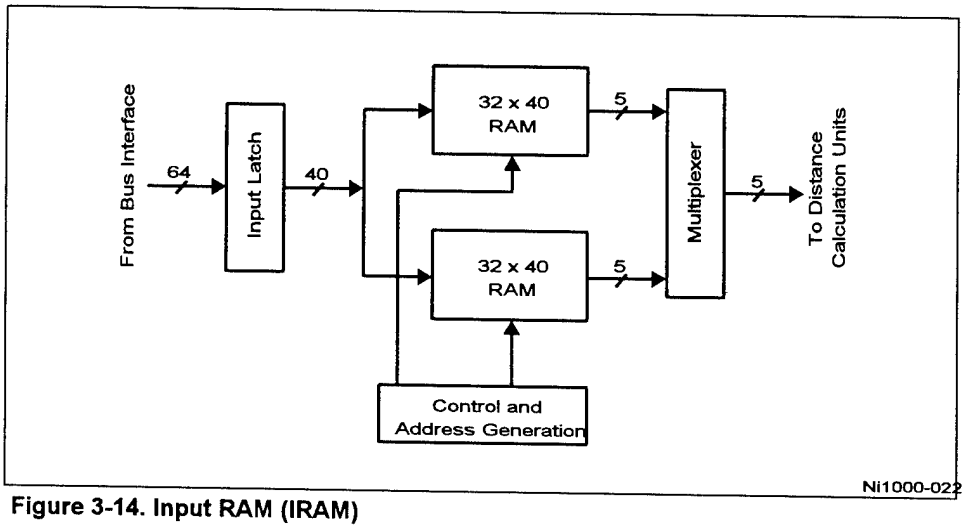


Figure 3-14. Input RAM (IRAM)

Microcontroller writes to the IRAM require loading a write latch, illustrated in Figure 3-15. The write latch is addressable in the microcontroller's address space. Note that unless the write latch is entirely filled with valid data prior to writing to IRAM, unknown data in those portions of the latch will be written into the IRAM. Writing to a location in the IRAM loads the locations' entire row with the contents of the latch. See section 5.1.3 for details.

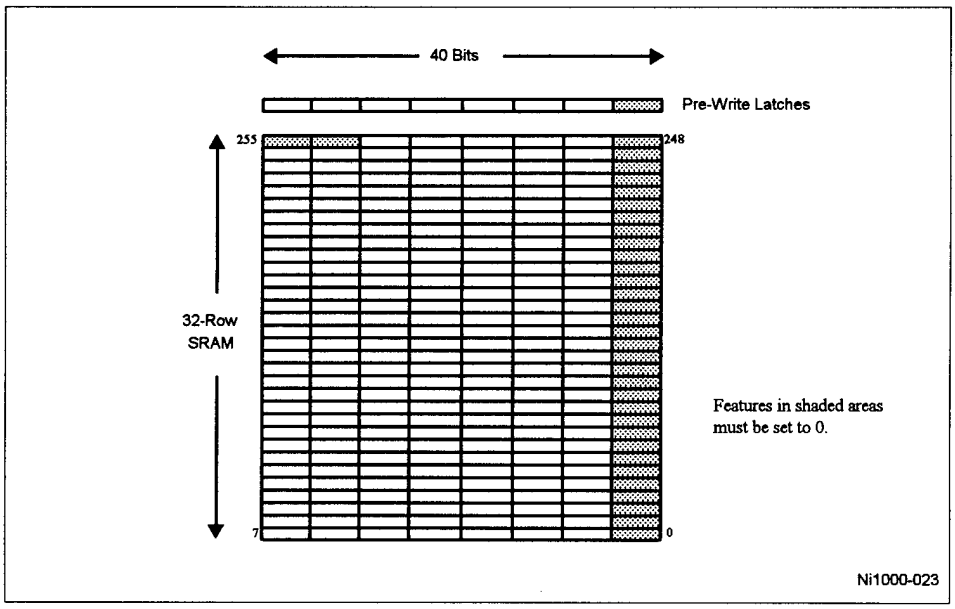


Figure 3-15. IRAM Pre-Write Latch

Chapter 5 contains the addresses for accessing the IRAM. Each of the two banks of the IRAM has its own set of latches and addresses.

The features of the input vector, as received from the bus interface, are five-bit quantities aligned to the five most-significant bits of each byte on the bus interface. The lower three bits of each byte are ignored.

The autosequencing logic for loading the IRAM works differently for 32- and 64-bit external data bus width. The vector loaded into the IRAM, as visible to the microcontroller, has a different organization depending on whether 32- or 64-bit bus width is selected.

Table 3-1 shows the least significant bits for the addressing of a 17-feature vector in the IRAM for both 32- and 64-bit modes. It shows the mapping of each feature of the input vector and where the data is stored by the autosequencing hardware in the IRAM. The autosequencer loads the first feature of the vector at an address in the IRAM, and works toward lower addresses. These addresses are only relevant to the host program when it is accessing data in the IRAM using microcontroller mode.

Table 3-1. Example of IRAM Vector-Addressing LSBs

Address LSBs	Vector Feature (32-Bit Mode)	Vector Feature (64-Bit Mode)
00000	Unused	Unused
00001	Unused	Unused
00010	Unused	Unused
00011	10000	Unused
00100	01111	Unused
00101	01110	Unused
00110	01101	Unused
00111	01100	10000
01000	01011	01111
01001	01010	01110
01010	01001	01101
01011	01000	01100
01100	00111	01011
01101	00110	01010
01110	00101	01001
01111	00100	01000
10000	00011	00111
10001	00010	00110
10010	00001	00101
10011	00000	00100
10100	Unused	00011
10101	Unused	00010
10110	Unused	00001
10111	Unused	00000

In 32-bit mode, the last address to be loaded is at location 000xx (binary) where xx are the inverse of the two least significant bits after subtracting one from the vector length. In the example in Table 3-1, the vector length is 17 (decimal) or 10001 (binary). Subtracting one from 10001 gets 10000. The last two bits are then inverted, so xx is 11 (binary) and therefore the last address to be loaded is 00011. The first address to be loaded is the last address plus the vector length minus 1. In this example, that is 00011 plus 10000 which is 10011.

In 64-bit mode, the last address to be loaded is at location 00xxx (binary). In the example in Table 3-1, xxx is 111 (binary) and so the last address to be loaded is 00111. The first address to be loaded is 00111 plus 10000 which is 10111.

3.2.3. Output RAM (ORAM)

The ORAM is shown in Figure 3-16. It is a small buffer, with a pre-write latch between it and the MURAMs. The MURAMs are more than simple double buffers; they feed back their contents to the MU pipeline. However, once an input vector has been read into one of the two MURAM buffers and completely processed, the contents of that buffer are read out to the ORAM while a new input vector is being read into the other MURAM buffer.

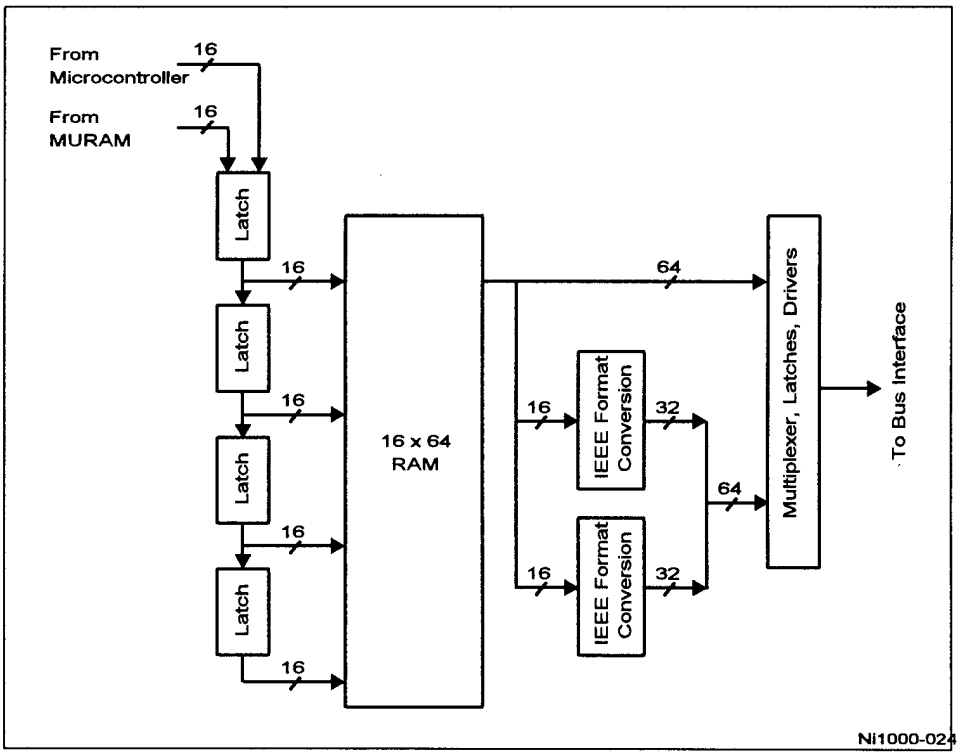


Figure 3-16. Output RAM (ORAM)

The ORAM is 16 x 64-bits in size, so it can hold all of the data generated by classifying an input vector.

When the classifier is running, the ORAM is accessible to the host. The number of valid reads that can be made from the MURAM selected for output depends on the mode. If the class-list MURAM is being uploaded, the number of entries in the ORAM will be equal to the number of

firing classes. If the probability MURAM is being uploaded, the number of entries will be specified by a byte in the DIM register (see Chapter 5).

In microcontroller mode (classifier not running), the microcontroller may directly access the ORAM. On reads, the ORAM is mapped into the microcontroller's address space (see Chapter 5 for the microcontroller's memory map).

Microcontroller writing to the ORAM requires loading a pre-write latch, illustrated in Figure 3-17. The write latch is also addressable in the microcontroller's address space. Like the IRAM, there is a special range of addresses for referencing the destination of a write. Unlike the IRAM, the ORAM write latch is a four-word shift register. Writing less than four words of data to the write latch before invoking a write operation may result in data appearing in the wrong position within a word.

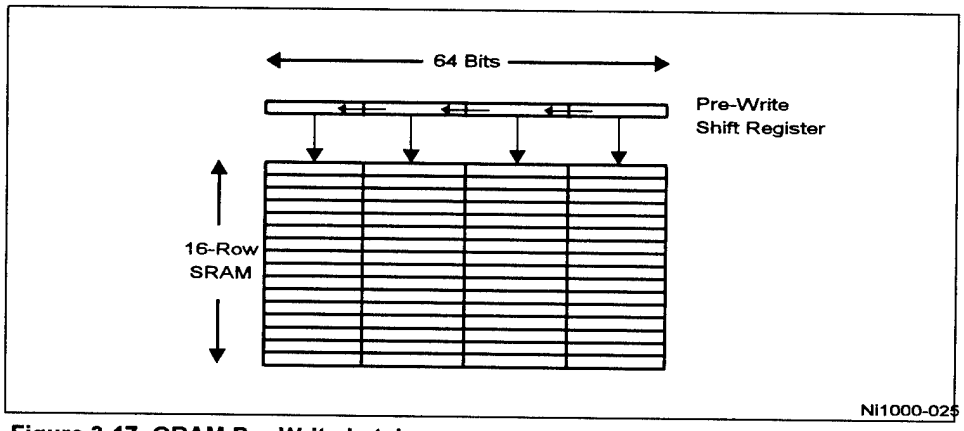


Figure 3-17. ORAM Pre-Write Latch

3.3. Computational Precision

The computational precision of variables is listed in Table 3-2. Floating-point probabilities appear in one of two formats, 16-bit internal or 32-bit IEEE, as described in the immediately following sections.

Table 3-2. Computational Precision

Variable	Mantissa		Exponent		Smallest Representable Non-Zero Value*	Largest Representable Value*	Actual Smallest Non-Zero Value*	Actual Largest Value*
	bits	format	bits	format				
v	5	00000	x	x	1	2^5-1	x	x
d	13	0...0	x	x	1	2^{13}-1	x	x
λ	13	0...0	x	x	1	2^{13}-1	x	x
K_{off}	4	0000	x	x	0	8	x	x
k	4	0000	4	s000	2^{-20}	$15*2^9$	2^{-20}	$15*2^{-2}$
kd	10	.0...0	6	s00000	2^{-20}	$2^{32}-2^{22}$	2^{-20}	$2^{18}-2^{14}$
2^{-kd}	10	.0...0	6	s00000	2^{-37}	$2^{32}-2^{22}$	2^{-37}	1
C	16	0000	x	x	1	2^{16}-1	1	2^{16}-1
$C*2^{-kd}$	10	.0...0	6	s00000	2^{-37}	$2^{32}-2^{22}$	2^{-37}	2^{16}-1
$\Sigma C*2^{-kd}$	20	.0...0	6	s00000	2^{-37}	$2^{32}-2^{12}$	2^{-37}	$2^{26}-2^{10}$
IEEE-754 single precision real	23	1.0...0	8	s0...0	2^{-127}	2^{128}	x	x

s = sign bit
x = not applicable or not available
* = Representable Value—Value that can be represented by the internal number format.
Actual Value—Value that is supported by the chip, external sotware or microcode must enforce limits on K_e and K_{off}.

3.3.1. 16-Bit Internal Format

Figure 3-18 shows the MU internal format used for floating-point numbers. In this format, the 6-bit exponent uses 2's complement to represent negative numbers. The number represented is $0.F^* \times 2^{\pm E^*}$, where the MSB of F^* is 1 unless the entire number is zero.

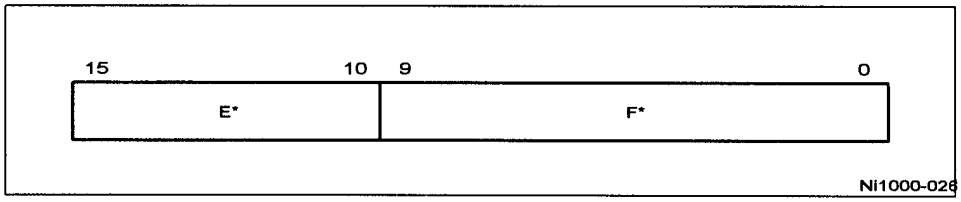


Figure 3-18. Internal 16-Bit Floating-Point Format

3.3.2. 32-Bit IEEE Format

Figure 3-19 shows the IEEE 32-bit format for floating-point numbers. The number represented is $(-1)^S \times 1.F \times 2^{E-127}$. There is no restriction on the MSB of F.

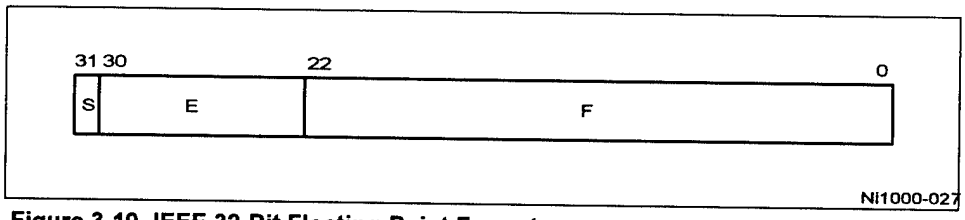


Figure 3-19. IEEE 32-Bit Floating-Point Format

Bit 4 of the CRB register selects the output format. If set, the IEEE 32-bit format is used. If clear, the internal 16-bit format is used. The conversion from the 16-bit format to the 32-bit format is accomplished through the mapping:

$S = 0$
$E = E^* + (5e)_{16}$, if E* is negative
$E^* + (7e)_{16}$, if E* is positive
$F = F^* \times (4000)_{16}$

3.4. Microcontroller

The 16-bit, custom microcontroller (MC) has a Harvard architecture (i.e. physically separate instruction and data memories). It is supported with 4K words of flash program memory, 256 words of general-purpose data RAM (GRAM) and a 32-bit timer.

3.4.1. Memory-Map Overview

Figure 3-20 gives an overview of the Accelerator's memory map, most of which is accessible to the microcontroller. For a detailed memory map, see Chapter 5.

3.4.2. Architecture

Figure 3-21 shows the architecture of the microcontroller datapath. It has four general-purpose registers and a simple instruction set. Instructions consist of one or two 16-bit words.

There are four important microcontroller buses, shown in Figure 3-22:

- *PABUS*—program memory address bus.
- *PDBUS*—program memory data bus.
- *ABUS*—general-purpose address bus.
- *DBUS*—general-purpose data bus.

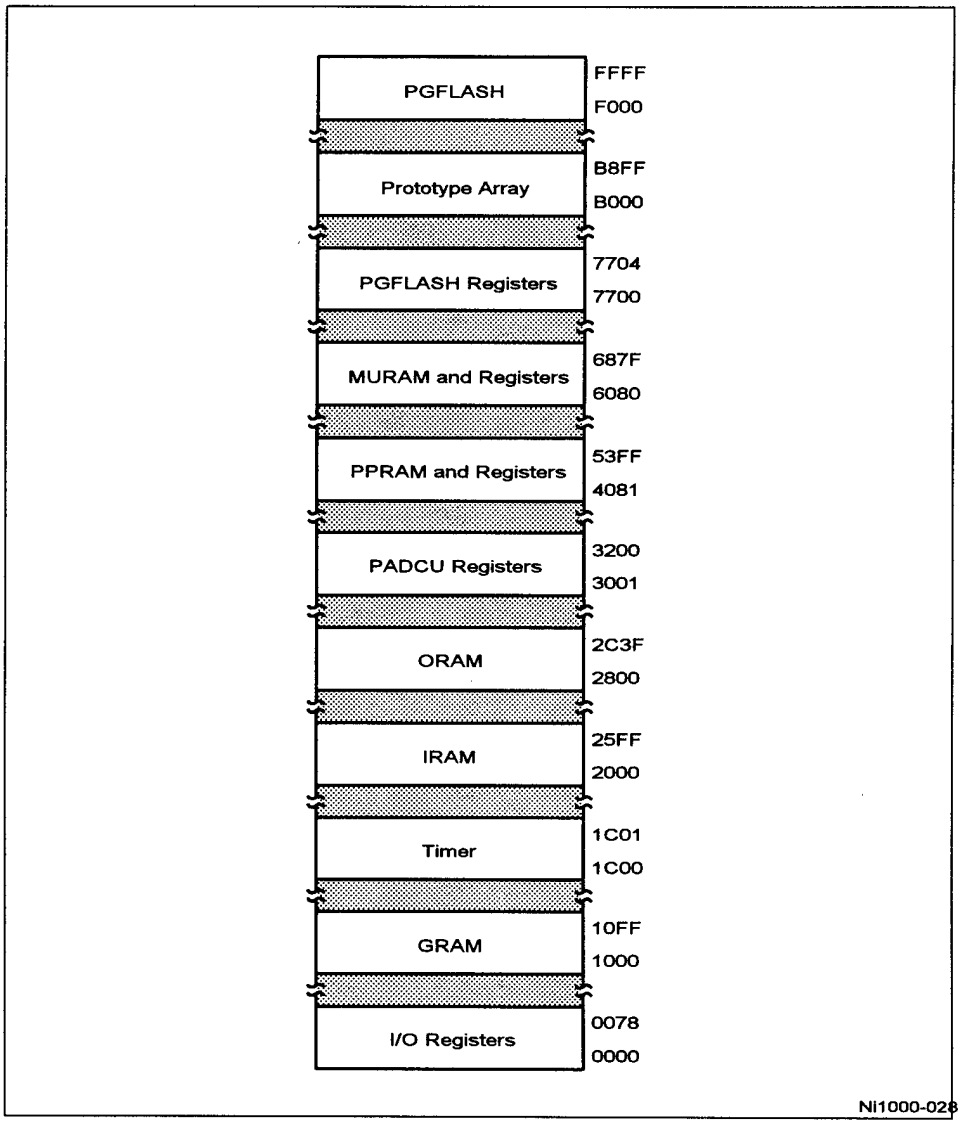


Figure 3-20. Memory-Map Overview

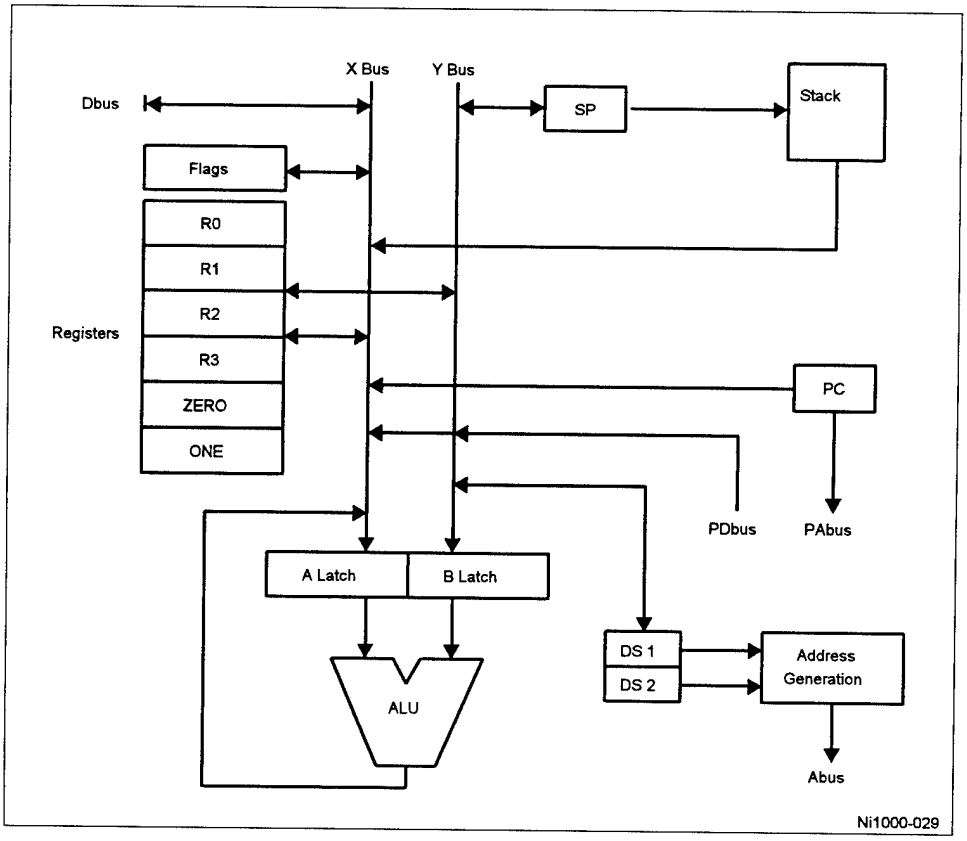


Figure 3-21. Microcontroller Datapath

The PABUS and PDBUS are used to access the microcontroller's flash memory. The ABUS and DBUS access the microcontroller's 256-word RAM, 32-bit timer and almost all of the registers and memories within the classifier. The 16-bit external data interface to the Bus Control Unit is for PG access modes, since the PGFLASH is 16 bits wide.

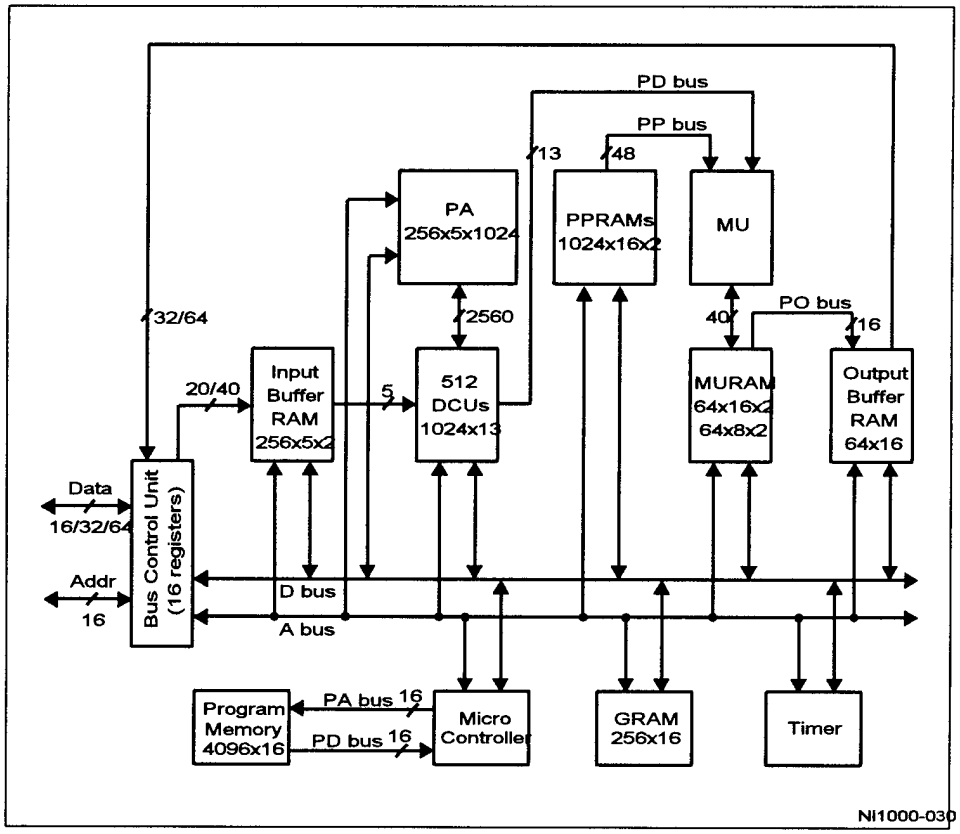


Figure 3-22. Bus Architecture

3.4.3. Registers

Most instructions have one or two 4-bit fields to specify registers. The available microcontroller registers are:

R0—General-purpose register 0.
R1—General-purpose register 1.
R2—General-purpose register 2.
R3—General-purpose register 3.
Zero—Reserved always reads as zero.
One—Reserved always reads as one.
DS1—Data segment register 1.
DS2—Data segment register 2.
SP—Stack pointer.

The data segment registers are used for address generation. The stack is 64 levels deep, so only the low six bits of the stack pointer are used; the remaining bits are reserved.

3.4.4. Flags

All conditional jump instructions test the microcontroller status flags for the following conditions (encoded with a four-bit binary field in the instruction word):

Carry (C)
Zero (Z)
Negative (N)
Positive (P)
Overflow (O)
Interrupt Request (IR)
Interrupt Enable (IE)
Stack Error (SE)
General Error (GE)
Multi-Class Firing (MC)
FLASH-Write (MC)
MURAM1 Ready (M1)
MURAM2 Ready (M2)
PADCU Busy (DC)

Bits in the HS1 register correspond to the above list of flags and the host can read these flags by reading HS1. In addition to the conditional jump instructions, three other instructions contain a flag-specifier field: of them, SFLG and CFLG, allow any flag to be set or cleared and WAIT suspends execution until a specified flag is set.

3.4.5. Instruction Set

The instruction set includes the following basic arithmetic and logical instructions:

- *Double-Operand Arithmetic Instructions*—add (ADD), add with carry (ADC), subtract (SUB), and compare (CMP).
- *Single-Operand Arithmetic Instructions*—increment (INC) and decrement (DEC).
- *Double-Operand Logical Instructions*—and (AND), or (OR), and exclusive-or (XOR).
- *Single-Operand Logical Instructions*—complement (NOT), shift left (SHL), shift right (SHR), rotate left (ROTL), and rotate right (ROTL).

Arithmetic and logical instructions only operate on 16-bit register operands. There are no operations on memory operands, other than reading or writing data to a register. See Chapter 5 for a table of the instruction opcodes, mnemonics and operations.

The six data movement instructions are: move (MOV), load (LD), read (RD), write (WR), push (PUSH), and pop (POP). The move instruction only transfers data from one register to another. The load instruction puts a 16-bit immediate operand following the instruction word into a register. The read and write instructions transfer a word between a memory location and a register. There are many flavors of read write due to the variety of addressing modes. The push and pop instructions transfer data between a register and the top of the stack.

Eight conditional jumps utilize the flag field that are described in Section 3.4.4. Both positive and negative polarities are supported for each condition. The address of the target of the jump can be calculated using a register or a 16-bit immediate operand following the instruction. The address can be absolute or Program Counter (PC) relative; *i.e.* the address can be used directly, or it can be added to the program counter. All combinations of these three options (condition, register/immediate and absolute/PC relative) are supported.

Ten conditional jumps first test common microcontroller conditions then jump to an address specified with a PC-relative 8-bit address embedded in the instruction. Four unconditional jumps result from combinations of where the address comes from (i.e. register or immediate) and how it is applied to the PC (i.e. absolute or relative).

Five "unconditional jump to subroutine" (JS) instructions are provided. They push the program counter on the stack, then jump to an absolute address specified by a register or 16-bit immediate, or a PC-relative address specified by a register, 16-bit immediate, or 8-bit field embedded in the instruction word.

Chapter 5 gives more detailed information on addressing modes, instruction mnemonics, syntax and flag cross reference.

3.4.6. Program Memory (PGFLASH)

The microcontroller program memory is stored in a 4K x 16 flash memory. This is a form of non-volatile electrically-erasable memory. PGFLASH can only be programmed from the host side of the interface. The microcontroller can only read PGFLASH for instructions.

When programming PGFLASH using a development system or other external programming method, a mode is entered by asserting the MC# signal. In this mode, a new register set appears for controlling flash memory program cycles. After programming and negating MC#, the Accelerator remains in reset mode until the host clears the reset bit in the CMR register. Programming is controlled by registers that are only accessible to external logic during programming mode. Chapter 5 describes the CMR register and the programming procedure.

Figure 3-23 shows the architecture of the PGFLASH during programming. The memory is programmed by accessing a set of registers:

- *Data Register*—stores data to write to PGFLASH.
- *Address Register 1*—stores a read address for PGFLASH.
- *Address Register 2*—stores a read/write address for PGFLASH. A multiplexer selects one of the address registers to drive PGFLASH. When address register 2 is read, the data returned is the output of the multiplexer. The multiplexer is controlled by a bit in user control register 2 (described below).
- *User Control Register 1*—controls the operation of drivers and sense amplifiers for the flash array.
- *User Control Register 2*—controls verification of the programming of the array and controls the multiplexer select line for addressing the flash array.
- *Status Register*—reports status of voltage levels against on-chip voltage references.

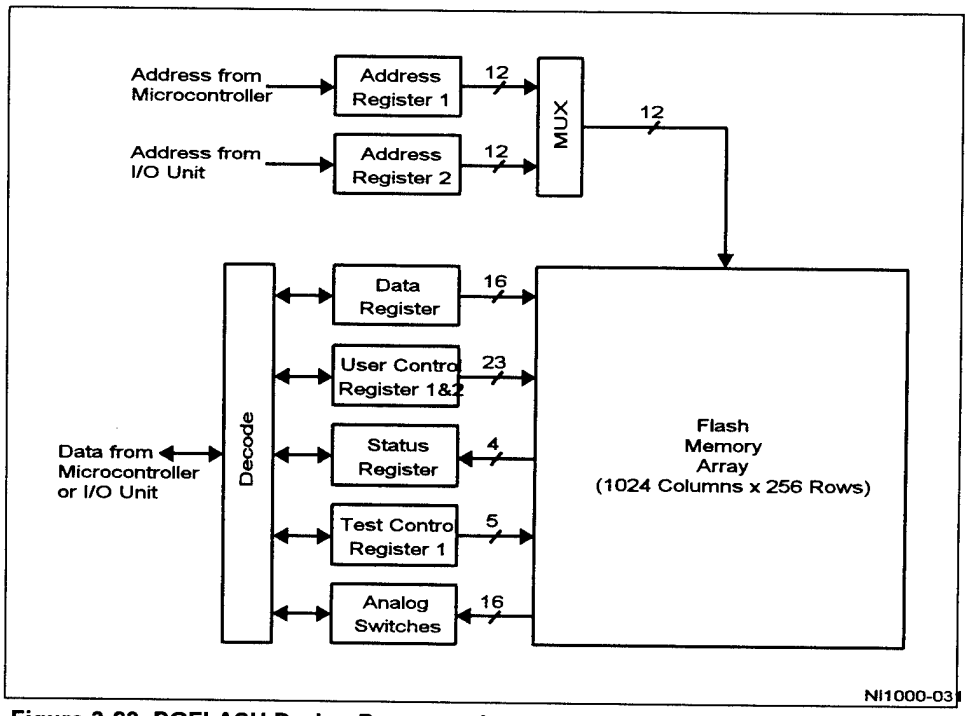


Figure 3-23. PGFLASH During Programming

3.4.7. Timer

The microcontroller has a timer accessed as a pair of 16-bit words. Figure 3-24 shows the structure of the timer. The timer is free-running, clocked by the system clock. At 25 MHz it reaches its terminal count and wraps around in 171.8 seconds. Note that instruction latency should be added to the timer values.

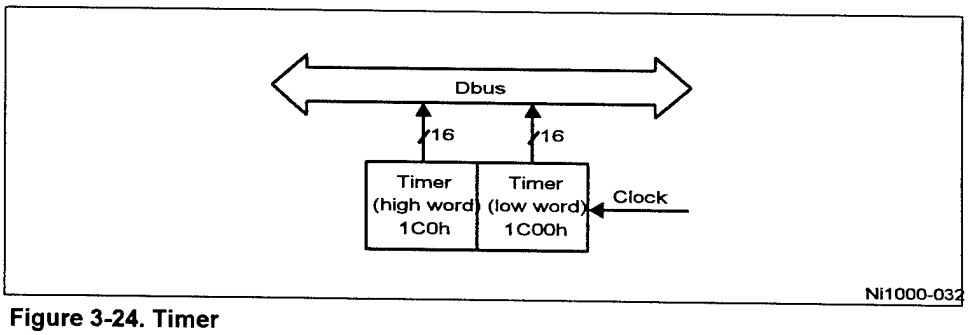


Figure 3-24. Timer

3.4.8. Reset Initialization

When the microcontroller is reset, the PC is initialized to address 0000h in the PGFLASH to begin execution. The microcontroller can be reset by setting bit 15 of the chip mode register (CMR). Setting this bit puts the entire chip into reset mode, clearing the bit clears the reset condition. The bit is set automatically when the RESET# signal is asserted. See Chapter 5 for a detailed description of the CMR register, or Section 3.7 for a description of the signals.

There are separate reset bits for the IRAM and ORAM. The IRAM and ORAM are unavailable to the microcontroller or the classifier while in reset mode. The reset bits must be cleared to enable operation of the classifier. Before they are cleared, the following things must be initialized:

- *32/64-Bit Bus Width*—the width of the external data bus, as defined by the level on the 64/32# signal.
- *Output Mode*—whether class IDs or class probabilities are the output, controlled by a bit in the CRA register.
- *Vector Dimensionality*—the number of features in an input vector, loaded into the DIM register. For probabilistic mode, the number of desired classes to upload must also be initialized in this register.
- *Floating-Point Format*—for probabilistic mode, whether the native 16-bit floating-point format or the IEEE-compatible 32-bit format is used for output, controlled by a bit in the CRB register.

3.4.9. Interrupts

Three forms of interrupts exist between the Ni1000 Accelerator and the host system:

- *Programmed Host-to-Microcontroller*—the host writes to the IIR register or CMR register. If interrupts are enabled, a subroutine jump to address is triggered as a side-effect of this write. See Chapter 5 for more information about the IIR and CMR registers.
- *Hardwired Host-to-Microcontroller*—the host (or external logic) asserts the MCINT# signal or SRQ# signal to invoke an interrupt service routine. If interrupts are enabled, the microcontroller performs a subroutine jump to address 0000h.
- *Hardwired Microcontroller-to-Host*—the microcontroller writes to the XIR register. See Chapter 5 for more information about the XIR register. The service request signal (SRQ#) is automatically asserted as a result of this write. The host responds by asserting IACK, to cause the microcontroller to negate SRQ#.

A bit in the microcontroller control and status register (HS1[6]) is used to enable interrupts to the microcontroller from the host. This bit is clear following reset, so interrupts are initially disabled.

The IIR register may be read by the microcontroller's interrupt handler routine to find the source of the error and respond appropriately. This registercontains a set of condition flags as described in Chapter 5; it is not an interrupt vector.

3.4.10. Errors

Three types of errors can occur:

- *External Error*—an attempt by the host to write to a full IRAM or read from an empty ORAM, or the assertion by external logic of the SRQ# signal. The BERR# signal is asserted.
- *Internal Stack Error*—microcontroller has overrun or underrun the stack space by popping an empty stack or pushing onto a full stack. This causes stack-error flag to be asserted as described in Section 3.4.4.
- *Internal General Error*—microcontroller has overrun or underrun a buffer, which asserts the general-error flag described in Section 3.4.4.

3.5. System-Level Architecture

Because the Accelerator is addressed like memory, it will always be a bus slave. There are several system design options available for the Ni1000 Accelerator:

- *Processor Bus*—placement directly on the bus with the processor.
- *Local Bus*—interface through a local bus standard, such as PCI or VL-Bus.
- *Expansion Bus*—a standard interface for expansion cards, such as the ISA bus, the EISA bus, or Micro Channel.
- *Hardwired*—a dedicated interface to an embedded controller, such as the i960 family of embedded RISC processors. The controller could also be an ASIC.

The Ni1000 Accelerator has an input signal, MULTICHIP#, to inform microcontroller software that it is in a multichip system. The microcontroller can test this condition in the CRB register (see Chapter 5).

Multichip systems must contend with two issues. During training, when a new prototype is allocated, they must collect Dmin (the city-block distance between the input vector and the nearest prototype vector of a different class) from all Accelerators to find the global Dmin used to initialize the radius of the prototype. When doing probabilistic classification, probability densities will have to be uploaded to the host from all chips that they can be combined.

3.6. Classification Timing

The timing of the classification pipeline varies with the number of features in the input vector and the number of valid prototypes. If the latency is short enough, the main source of delay will be I/O.

Figure 3-25 shows the pipelined processing of three vectors with up to 500 prototypes. If the host system is not a limiting factor or if the number of dimensions is low, filling one bank of the IRAM will be very quick compared to processing the vector. As soon as the first vector is loaded, it can be dispatched for processing while the second vector is being loaded.

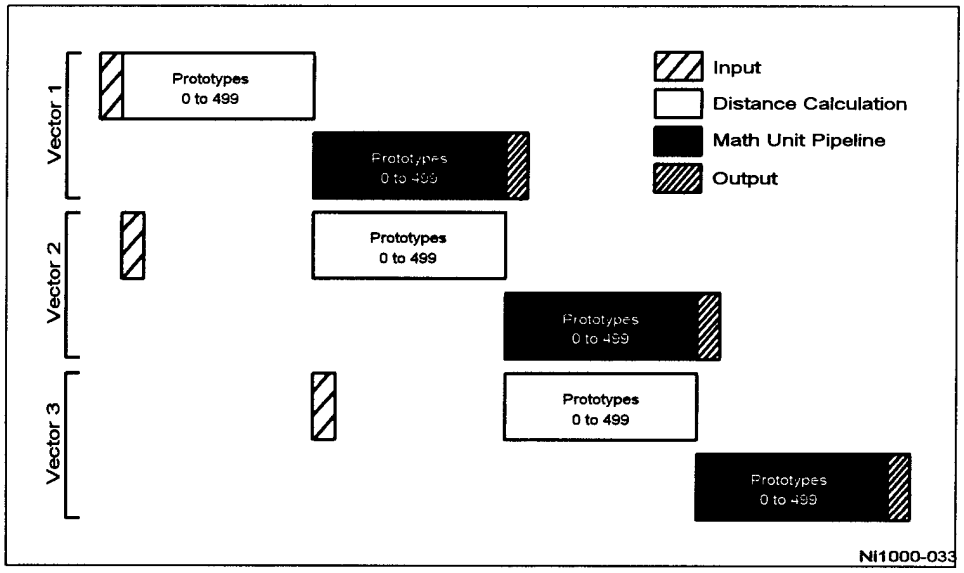


Figure 3-25. Pipeline Usage (For Up To 500 Prototypes)

After the first vector is processed by the distance calculation units, the IRAM buffers can swap and the third vector can be loaded. At this point, the first vector will be in the math unit pipeline and the second vector will be in distance calculation unit. As each vector is processed by the MU pipeline, it can quickly be loaded into the ORAM.

Figure 3-26 shows pipelined operation for more than 500 prototypes. Operation of the MU pipeline is fully overlapped with distance calculation. One feature of the input vector can be compared to the corresponding features of up to 500 prototypes in two cycles. Two additional cycles are required when over 500 prototypes are used.

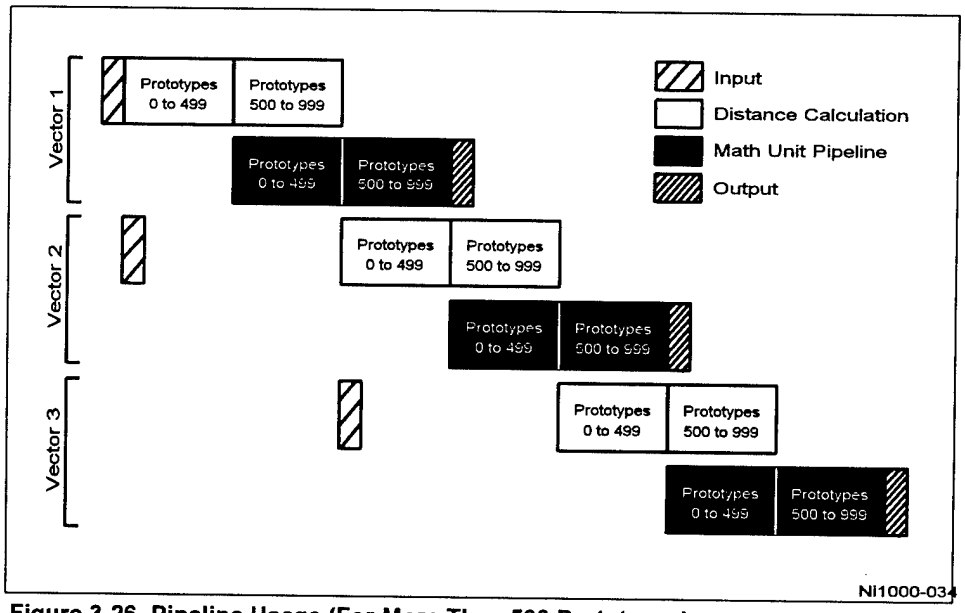


Figure 3-26. Pipeline Usage (For More Than 500 Prototypes)

The number of clocks required by each phase of the pipeline can be estimated from the following expressions.

Input—(8L / B) + I
Distance Calculation—2L + I, if P <= 500
Distance Calculation—2(2L + I), if P > 500
MU Pipeline—P + I
Output—(CR / B) + C + I

The parameters of these expressions are shown below.

L—Vector Length
P—Number of valid Prototypes
B—Bus width in bits (32 or 64)
C—Number of Classes required
R—Results size in bits (8 for RCE, 16 or 32 for PRCE)
I—Initialization time, about 5 clocks for each block

3.7. Signal Descriptions

Name	Type	Description
Clock, Address, and Data (Synchronous)		
CLK	I	**Clock.** This clock must be shared with or divided down from the host's clock, so that all bus transactions are synchronous with it.
A[0:15]	I	**Address**. Driven by the host to access the Accelerator's microcontroller program memory (PGFLASH), prototype-array memory (PA), prototype-parameter memory (PPRAM), and control and status registers. Detailed memory and register address maps are given in Chapter 5.
D[0:63]	I/O	**Data**. As inputs, the host writes feature vectors for classification, control information, and microcontroller programs on this bus. The inputs include 5-bit input vector components; 16-bit data, register contents, and microcontroller instructions; or 64-bit multiple-input vectors. As outputs, the host reads vector classifications or probabilities, status information, and microcontroller-program verification. The outputs include 8-bit classes; 16-bit data, register contents, and microcontroller instructions; 32-bit IEEE standard floating point values; or 64-bit groups of classes or probabilities.
Bus-Cycle Definition and Control (Synchronous)		
ADS#	I	**Address Strobe**. When asserted by the host on a rising edge of CLK, this signal causes the Accelerator to sample CS# and the address on A[0:15], thereby initiating a bus cycle.
CS#	I	**Chip Select**. Asserted by the host to indicate that the Accelerator is being addressed. The signal must be held asserted throughout the bus cycle. The signal is used to select one of potentially multiple Ni1000 Accelerators.
BLAST#	I	**Burst Last**. When asserted by the host, this signal indicates the last data transfer in the current cycle, whether burst or non-burst. For burst cycles, the host must hold BLAST# negated until the last data transfer of the cycle, at which time it asserts BLAST#. For non-burst cycles, the host asserts BLAST# during the first (and only) data transfer. The signal is compatible with the x86 BLAST# architecture; however, only a maximum of 64 bits can be burst to or from the Ni1000 Accelerator.
W/R#	I	**Write or Read**. Driven by the host on the same rising clock edge as ADS#, CS#, and BLAST#, to indicate that the current bus cycle is a write (high) or read (low).
RDY#	O	**Non-Burst Ready**. When asserted by the Accelerator, this signal indicates that the data on D[0:63] are valid (for output) or accepted (for input) and that it is the last data transfer in the current bus cycle. The signal terminates the bus cycle. For a burst cycle, RDY# is only asserted on the last transfer of the burst.

BRDY#	O	**Burst Ready**. When asserted by the Accelerator, this signal indicates that the data on D[0:63] is valid (for output) or accepted (for input) and that more data may be transferred in the current bus cycle. The signal does not terminate the current bus cycle and is not asserted on the last transfer of a burst; instead, RDY# is asserted.
BERR#	O	**Bus Error**. When asserted by the Accelerator, this signal indicates that illegal bus-definition conditions have occurred. For example, the host may attempt to write to the input buffer when the Accelerator is not in an appropriate mode or when the buffer is full, or the host may attempt to access the output buffer before data is available. The signal also terminates the current bus cycle. This signal is open collector.
64/32#	I	**64-Bit or 32-Bit Data Bus**. Driven by the host to select 64-bit (high) or 32-bit (low) operation on the D[0:63] bus. Data alignment is described in Chapter 4.
MC#	I	**Microcontroller**. Asserted by the host on the same rising clock edge as ADS# and CS# to read or write the Accelerator's microcontroller-program memory (PGFLASH).
MULTCHIP#	I	**Multi-Chip Operation.** Asserted by the host or tied to ground when multiple Ni1000 Accelerator chips are to operate in parallel on the same address and data bus. When asserted, the Accelerator alters its data flow, primarily during learning.
Interrupt Control (Asynchronous)		
SRQ#	O	**Service Request.** Asserted by the Accelerator's microcontroller to indicate that valid output is available on the data bus, an error has occurred, or some other action by the host is needed. The signal is also asserted when the microcontroller writes to the XIR register, and held asserted until the host asserts the IACK# signal.
IACK#	I	**Interrupt Acknowledge.** Asserted by the host to acknowledge that it sampled the Accelerator's assertion of SRQ#.
MCINT#	I	**Microcontroller Interrupt.** Asserted by the host to force the Accelerator's microcontroller to jump to a specified program address.
ERROR#	I/O	**Error.** As an input, asserted by the host to indicate interrupt to the microcontroller. On receiving an error, the Accelerator will identify the reason for the interrupt by reading the IIR register. As an output, asserted by the Accelerator to indicate that an internal error, such as data underflow or overflow in an I/O buffer. On receiving an error, the host should read the status register, XIR, to determine the nature of the error. This signal is open collector.

RESET#	I	**Reset.** Asserted by the host to halt and reinitialize the Accelerator. To exit the Reset state, the host must subsequently write a 0 to bit 15 of the CMR, whereupon the microcontroller begins executing instructions in NORMAL mode from location 1 in the PGFLASH (address F001h). The host can also reset the Accelerator by writing a 1 to bit 15 of the CMR register.
System and Power		
V_{cx}	P	**+5 Volt Memory Supply.** Used during normal operation by the prototype array (PA) and the microcontroller's program flash memory (PGFLASH).
V_{pp}	P	**+12 Volt Programming Supply.** Used during programming by the prototype array (PA) and the microcontroller's program flash memory (PGFLASH).
V_{cc}	P	**+5 Volt Supply.**
V_{ss}	P	**Ground.**

Type: I = Input, O = Output, P = Power or Ground.

4. BUS OPERATIONS

This chapter discusses the interaction between a host system and the Ni1000 Recognition Accelerator through various bus operations. Figure 4-1 shows the Accelerator's buses. Externally, the I/O unit (IRAM, ORAM, and I/O registers) connects to the host system through the signal pins. Internally, the I/O unit connects to the following buses:

- *Data I/O Bus* (DIO bus)—Connects to IRAM, ORAM and I/O registers. The I/O data path is either 32- or 64-bits wide.
- *Internal Address and Data Buses* (Abus and Dbus)—Serves as the data path of the microcontroller. Abus is a 16-bit address bus, and Dbus is a 16-bit data bus.
- *Microcontroller Program Address and Data Buses* (PAbus and PDbus)—Serves as the internal instruction path of the microcontroller. PAbus is a 16-bit address bus, and PDbus is a 16-bit instruction (data) bus.

4.1. Hardware-Controlled Access Modes

The Ni1000 Accelerator supports 32-bit or 64-bit data output at a bus clock of up to 25 MHz. The Accelerator interface is designed for synchronous operation using the bus clock as the Accelerator clock. A set of modes for access to the Accelerator determines which internal buses are accessible to the host, and which pin groups are used in data transfers with the host. The following list summarizes the access modes supported by the Accelerator:

NORMAL—Used for classification and learning.
PG—Used to read from program memory or load microcontroller programs into program memory (PGFLASH).
RESET—Used to suspend precharging and most latching and to initialize state machines.

After power up, the Accelerator microcontroller idles until the host releases it from the reset state by writing a 0 to bit 15 of the CMR register. In addition, the appropriate access-mode control signals, shown in Table 4-1, must be stable before taking the chip out of the reset state. The access mode control signals configure the address space to provide access to one of the following:

- I/O unit (access IRAM through address 2000h, ORAM through 2800h, and I/O registers).
- PGFLASH and associated registers.

Other memory locations and registers can only be accessed by the microcontroller. See Chapter 5 for details. The address pins A[0:15] are mapped to the corresponding internal address space. Table 4-1 summarizes the control-signal settings and mapping of data and address pins for the access modes.

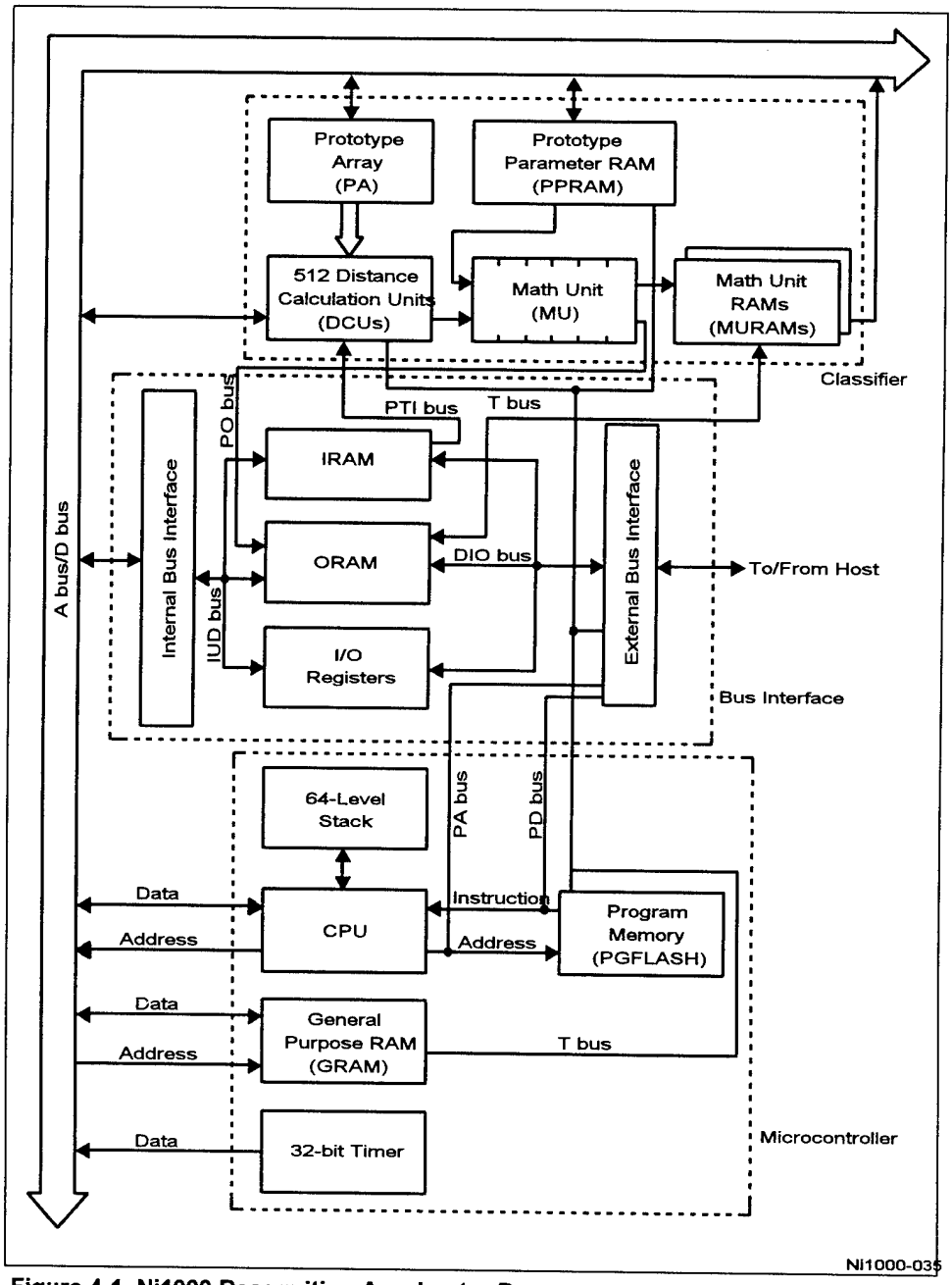


Figure 4-1. Ni1000 Recognition Accelerator Buses

Table 4-1. Hardware-Controlled Access Modes

Mode	Access Mode Control Signals					Data Bus[3]			Address Bus
	CS#	W/R#	MC#	RESET#	64/32#	D[0:15]	D[16:31]	D[32:63]	A[0:15]
NORMAL Write	0	1	1	1	1 or 0	DIO[0:15]	DIO[16:31]	DIO[32:63]	AIO[0:15]
NORMAL Read	0	0	1	1	1 or 0	DIO[0:15]	DIO[16:31]	DIO[32:63]	AIO[0:15]
PG[1] Write	0	1	0	1	0	PDBUS[0:15]	undefined	undefined	PABUS[0:15]
PG[1] Read	0	0	0	1	0	PDBUS[0:15]	undefined	undefined	PABUS[0:15]
RESET[2]	X	X	X	0	X	inactive	inactive	inactive	inactive

1. In PG mode, the internal logic (except PGFLASH contents and registers) is reset.
2. When RESET, the internal logic (including PGFLASH registers) and the external bus interface are reset, except the non-volatile PGFLASH contents.
3. When 64/32# is asserted, 32-bit data bus (D[0:31]) is selected. When 64/32# is deasserted, 64-bit data bus (D[0:63]) is selected.

When used for output, the data-bus bits are grouped as one of the following:

- 8-bit data for firing class IDs.
- 16-bit data for internal format probability densities.
- 32-bit data for IEEE floating-point format probability densities.
- 64-bit data for two IEEE floating-point numbers.

When used for input, the data-bus bits are grouped as:

- 8-bit data for input vector components.
- 16-bit data for register contents and microcontroller instructions.

See Chapter 5 for details.

Input vector components are aligned to the high-order 5 bits for each byte. For example, the first 5-bit component in each input vector should be transferred to the Accelerator on pins D[3:7], with D7 receiving the most-significant bit of the component. The three least significant bits, D[0:2], are ignored. 16-bit data is expected in the lowest 16 bits of a 32- or 64-bit transfer.

The Accelerator does not always require the use of all 64 data I/O bits for transfers with the host. Examples include:

- The Accelerator is operated in a system with a 32-bit data path. In this case, the upper 32 data pins are not used in the data transfer.

- The internal resources being accessed have a 16-bit data path. This is the case in the PG modes; the PDbus and the Dbus are only 16 bits wide.

4.1.1. Normal Mode (NORMAL)

NORMAL mode has the following characteristics:

- The Accelerator may perform classification, learning or housekeeping.
- The Accelerator acts as a slave processor in interactions with the host.
- The Accelerator uses its SRQ# signal to request service. See Chapter 5 for details of microcontroller interrupt handling.
- Input vectors may be sent to the Accelerator and classification results may be received by the host.
- Commands may be written to the Accelerator and status read by the host.
- The classification pipeline may be enabled or the Accelerator may operate under control of the microcontroller for learning or housekeeping.

The host writes and reads via the I/O unit's buffers, namely the IRAM, ORAM, and I/O registers. The width of the data path could be either 64 or 32 bits, as indicated by the 64/32# signal. See Chapter 5 for the accessible address spaces. Accessing addresses other than those for the IRAM, ORAM and I/O registers will yield invalid results.

NORMAL mode is initiated by deasserting the MC# signals. The direction of data transfer (input or output) is indicated by the W/R# signal.

4.1.2. PGFLASH Access Modes (PG Modes)

There are two PG modes, as described in the following sections. In these modes, the host can write or read the PGFLASH control registers through the microcontroller. This allows control, reading, programming, and erasing of the PGFLASH in order to upload or download the microcontroller program. In these modes, the internal logic (except PGFLASH contents and registers) is reset. The external bus interface is not reset. The reset condition is maintained when the NORMAL mode is re-entered from the PG mode.

4.1.2.1. PGFLASH Program Mode

In this mode, the microcontroller is halted while its PGFLASH memory is written by the host. This mode is initiated by asserting MC# and driving W/R# high to indicate a write operation. After MC# is subsequently deasserted, the microcontroller remains in the halted state until a command is written to the CMR register that clears bit 15 of that register to zero.

The A[0:15] pins are used to address the memory locations into which instructions are to be written. Addresses drive the internal Program Address Bus (PAbus) and are latched into a program-address register associated with the PGFLASH. Addresses are latched coincident with writing of the instruction into a second program address register. Once an address has been latched, a series of control words must be written to the two 16-bit control registers associated with the PGFLASH. The required sequence is described in the PGFLASH programming section of Chapter 5.

Only 16 bits (D[0:15]) are used for data transfer, since the PDbus is 16-bits wide. Data on D[16:63] should be set to 1 on input.

4.1.2.2. PGFLASH Read Mode

In this mode, the microcontroller is halted while its PGFLASH memory is read by the host. The internal Program Data Bus (PDbus) drives the external data pins, D[0:63].

This mode is initiated by asserting MC# and driving W/R# low to indicate a read operation. As in the case of load operation, only 16 bits (D[0:15]) are used for data transfer.

4.1.3. Reset Mode (RESET)

Bit 15 of the CMR register is the Accelerator's reset latch. Any of the following conditions can reset all or parts of the Accelerator:

- The RESET# signal is asserted at the rising edge of CLK.
- The MC# signal is asserted at the rising edge of CLK.
- A value of 1 is written to bit 15 of the CMR register (by the host or the microcontroller).

The first and third condition above reset the entire Accelerator, which includes the external bus interface, PGFLASH, and other internal logic. The second condition alone resets the internal logic, except the contents of PA, PGFLASH and the PGFLASH registers.

When the Accelerator is reset, all precharging and most latching is suspended. An active clock is still distributed to most blocks, but the blocks become idle. All state machines are re-initialized. There are two exceptions to these reset actions:

- The external bus interface is only forced into its initialization state when the RESET# signal is asserted on a rising edge of CLK.
- The PGFLASH is not reset if the MC# signal is asserted.

The first exception allows the external bus interface to function, so that the host can write a 0 into bit 15 of the CMR register. The second exception enables the PGFLASH to be manipulated while all other units on the Accelerator (except the external bus interface) are deactivated during the PG modes.

After the Accelerator is reset, operation can only be restored by the host deasserting RESET# and writing a 0 to bit 15 of the CMR register. The Accelerator is put into NORMAL mode if MC# and RESET# are deasserted, and CS# is asserted. The microcontroller starts executing instructions from location 1 in the PGFLASH which is at address F001h in the memory space.

4.2. Bus Cycles

In NORMAL and PG modes, 64, 32 or 16 bits can be transferred. The Accelerator supports both burst and non-burst transfers. Both the host and the Accelerator can terminate a bus cycle, but only the host can initiate one. The timing diagrams in this section illustrate the read and write cycles that can be performed in the NORMAL and PG modes.

Table 4-2 shows the signals used to control bus cycles. A cycle starts when ADS# and CS# are both asserted by the host at a rising edge of CLK. At the same time, the host drives the address on A[0:15] and the W/R# signal to define a read or write.

Data (whether input or output) are not transferred unless the Accelerator asserts RDY# or BRDY#. If the Accelerator asserts RDY#, the cycle is terminated by the Accelerator after a single bus-width of data are transferred. If the Accelerator asserts BRDY# during the first transfer, additional transfers can be made in that cycle. The BLAST# signal, when asserted by the host at the beginning of a cycle, indicates either a single-bus-width (non-burst) cycle or the last data transfer of a burst cycle. If the Accelerator asserts BRDY# and the host asserts BLAST# at the same time, the host is terminating the cycle. If the Accelerator asserts neither RDY# nor BRDY#, data are not transferred. If the Accelerator asserts BERR#, data are not transferred and the cycle is terminated. The BLAST# signal always indicates the last transfer of any cycle (burst or non-burst); it is not required for a burst transfer, although when asserted by the host it always indicates the end of a cycle from the hosts viewpoint.

4.2.1. I/O-Register Read or Write

The I/O registers can be read or written by the host in single (non-burst) cycles when the Accelerator is operating in the NORMAL hardware-controlled access mode. Figure 4-2 shows an I/O-register read cycle followed by an I/O-register write cycle. The addresses of the I/O registers are given in Chapter 5.

A read cycle begins when the host asserts ADS# and CS# and the Accelerator samples them at the rising edge of CLK. At the same time, the host drives the address, A[0:15], and it drives W/R# low. The host also asserts BLAST# at the beginning of the cycle to indicate a single or non-burst data transfer, i.e., the first data transfer is the last transfer expected in the cycle. When the Accelerator responds by placing valid data on D[0:31], the Accelerator asserts RDY# indicating that the host should sample the data.

A write cycle, shown in the right side of Figure 4-2, follows essentially the same protocol as the read cycle except that the host drives W/R# high and it drives the data on D[0:31] at the beginning of the cycle, at the same time that it drives ADS#, CS#, and the address. The Accelerator terminates the cycle in the same manner as it terminates a read: by asserting RDY#.

Table 4-2. Cycle-Definition and Control Signals for Normal and PG Modes

Pins	Driven By	Description
ADS#	Host	**Address Strobe**. When asserted by the host on a rising edge of CLK, this signal causes the Accelerator to sample CS# and the address on A[0:15], thereby initiating a bus cycle.
CS#	Host	**Chip Select**. Asserted by the host to indicate that the Accelerator is being addressed. The signal must be held asserted throughout the bus cycle. The signal is used to select one of potentially multiple Ni1000 Accelerators.
BLAST#	Host	**Burst Last**. When asserted by the host, this signal indicates the last data transfer in the current cycle, whether burst or non-burst. For burst cycles, the host holds BLAST# negated until the last data transfer of the cycle, during which it asserts BLAST#. For non-burst cycles, the host asserts BLAST# during the first (and only) data transfer. The signal is compatible with the x86 BLAST# architecture; however, only a maximum of 64 bits can be burst to or from the Ni1000 Accelerator.
W/R#	Host	**Write or Read**. Driven by the host on the same rising clock edge as ADS#, CS#, and BLAST#, to indicate that the current bus cycle is a write (high) or read (low).
RDY#	Ni1000 Accelerator	**Non-Burst Ready**. When asserted by the Accelerator, this signal indicates that the data on D[0:63] is valid (for output) or accepted (for input) and that it is the last data transfer in the current bus cycle. The signal terminates the bus cycle. For a burst cycle, RDY# is only asserted on the last transfer of the burst.
BRDY#	Ni1000 Accelerator	**Burst Ready**. When asserted by the Accelerator, this signal indicates that the data on D[0:63] is valid (for output) or accepted (for input) and that more data may be transferred in the current burst bus cycle. The signal does not terminate the current bus cycle. The Accelerator cannot pull this signal high.
BERR#	Ni1000 Accelerator	**Bus Error**. When asserted by the Accelerator, this signal indicates that illegal bus-definition conditions have occurred. For example, the host may attempt to write to the input buffer when the Accelerator is not in an appropriate mode or when the buffer is full, or the host may attempt to access the output buffer before data is available. The signal also terminates the current bus cycle. This signal is open collector.
64/32#	Host	**64-Bit or 32-Bit Data Bus**. Driven by the host to select 64-bit (high) or 32-bit (low) operation on the D[0:63] bus. Data alignment is described in Section 4.1.
MC#	Host	**Microcontroller**. Asserted by the host on the same rising clock edge as ADS# and CS# to read or write the Accelerator's microcontroller-program memory (PGFLASH).

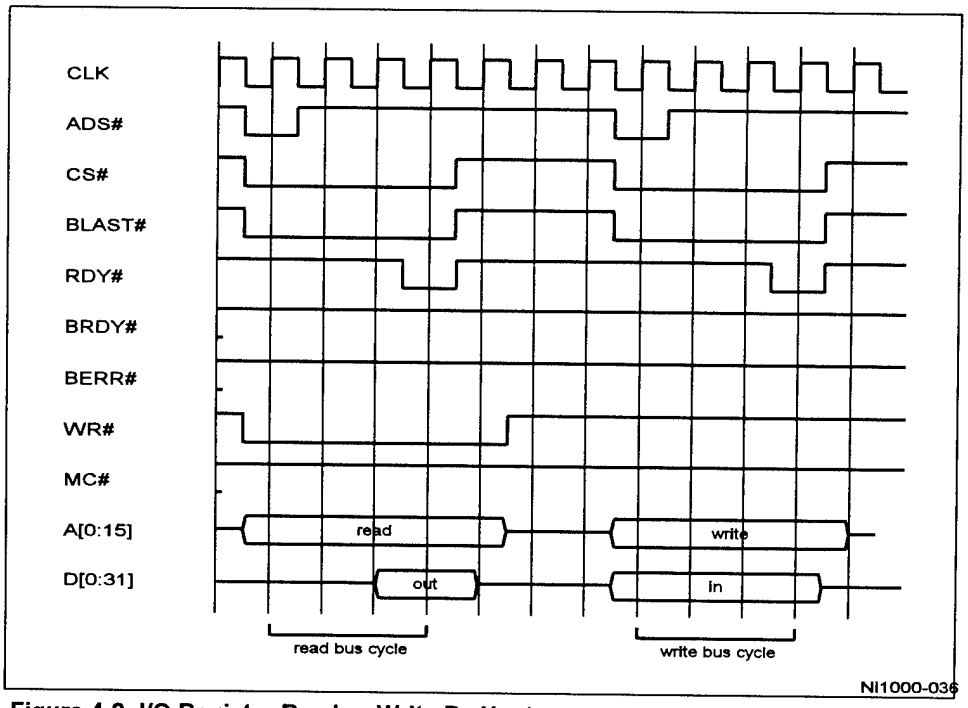


Figure 4-2. I/O Register Read or Write By Host

4.2.2. PGFLASH Read or Write

The PGFLASH can be read or written by the host in single (non-burst) cycles when the Accelerator is operating in one of the two PG hardware-controlled access modes. Figure 4-3 shows a PGFLASH read cycle followed by a PGFLASH write cycle. The PGFLASH is accessed at addresses F000h through FFFFh.

Figure 4-3 shows a PGFLASH read or write cycle initiated by the host. In a read cycle, the host begins by driving ADS#, CS#, the address, W/R#, and BLAST#. Six clocks later, the Accelerator places the data on D[0:31] and asserts RDY#. In a write cycle, the Accelerator samples the data and asserts RDY# two clocks after the host begins the cycle.

The protocol for reading and writing PGFLASH is the same as for reading and writing an I/O register; however, the timing of reads in the PGFLASH is three clocks longer than for an I/O register, and the timing of writes is one clock shorter, as evident from Figure 4-2 and 4-3.

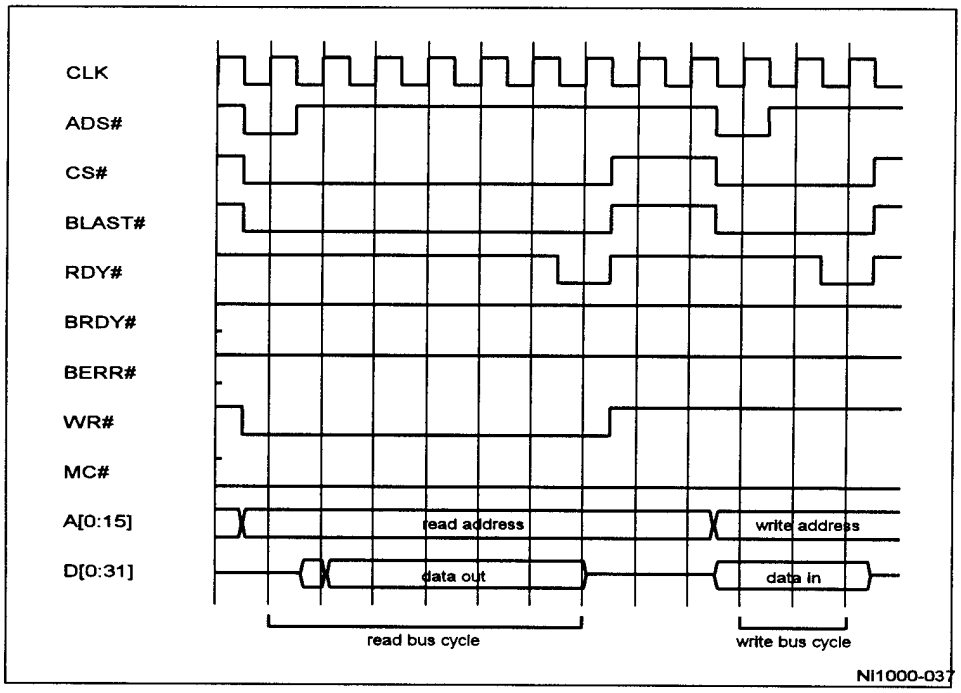


Figure 4-3. PGFLASH Read or Write By Host

4.2.3. IRAM Non-Burst Write

The IRAM can be written by the host in single (non-burst) or burst cycles while the Accelerator is operating in the NORMAL hardware-controlled access mode. IRAM is accessible at address 2000h. When writing input vectors into IRAM for classification, the BERR# signal will be asserted if the host attempts to write more data than specified in the DIM register. See Chapter 5 for details.

Figure 4-4 shows two sequential single (non-burst) writes to the IRAM. The timing is identical for both writes, and is also identical to the timing for PGFLASH writes: two clocks after the host starts the cycle, the Accelerator samples the data and asserts RDY# to terminate it. Burst writes to the IRAM are shown in Figure 4-6.

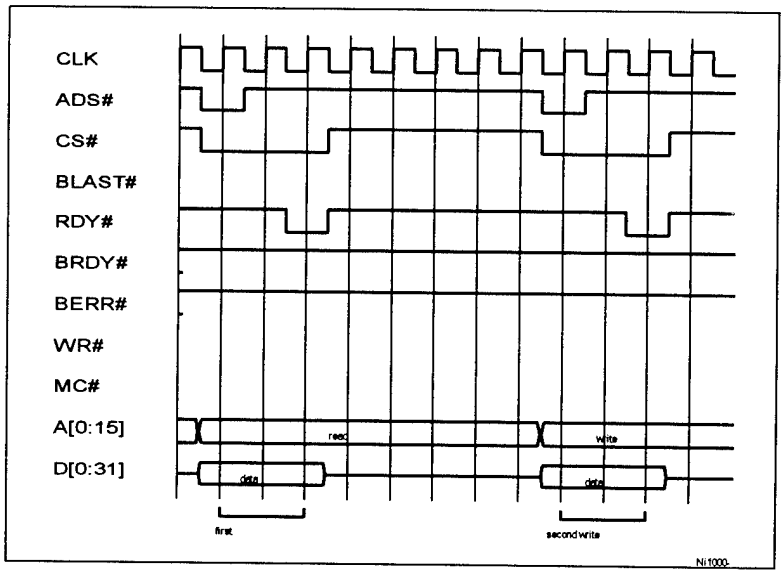


Figure 4-4. IRAM Non-burst Write By Host

4.2.4. ORAM Non-burst Read

The ORAM can be read by the host in single or burst cycles while the Accelerator is operating in the NORMAL access mode. The ORAM external output port is accessed at address 2800h.

Figure 4-5 shows two types of single (non-burst) ORAM reads. The first read occurs when the ORAM has just been filled with classification results. The cycle takes five clocks to complete, which is two clocks longer than a read that occurs when the ORAM has been previously accessed (i.e., not just filled with classification results). Burst reads of the ORAM are shown below in Figure 4-7.

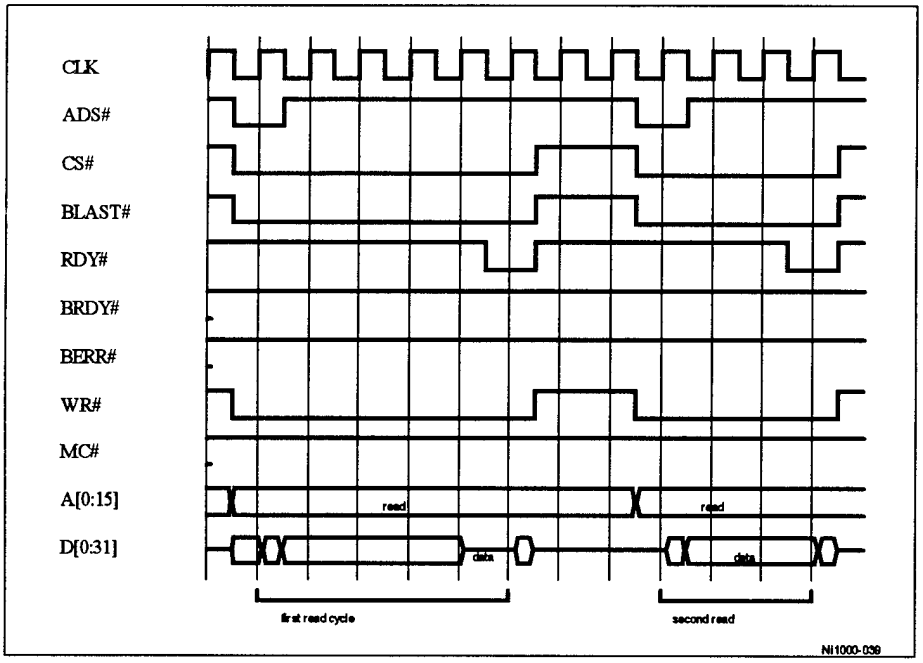


Figure 4-5. ORAM Non-Burst Read By Host

4.2.5. IRAM Burst Write

Figure 4-4 showed a single write to the IRAM. Figure 4-6, shows a four-transfer burst write to the IRAM. In burst cycles, the Accelerator asserts BRDY# instead of RDY# to indicate each successful data transfer in the multi-transfer sequence. Unlike RDY#, BRDY# does not terminate the cycle.

The host begins the cycle with BLAST# deasserted. It keeps BLAST# deasserted through the third transfer, indicating to the Accelerator that these transfers are not expected to be the last transfer of the cycle (i.e., that this will be a burst cycle). In response, the Accelerator holds BRDY# asserted from the first data transfer through the last transfer (although RDY# can be substituted for BRDY# in the last transfer). The end of the cycle occurs when the host asserts BLAST# while the Accelerator asserts either BRDY# or RDY#.

The Accelerator can accommodate vectors with up to 222 dimensions, at one byte per dimension.

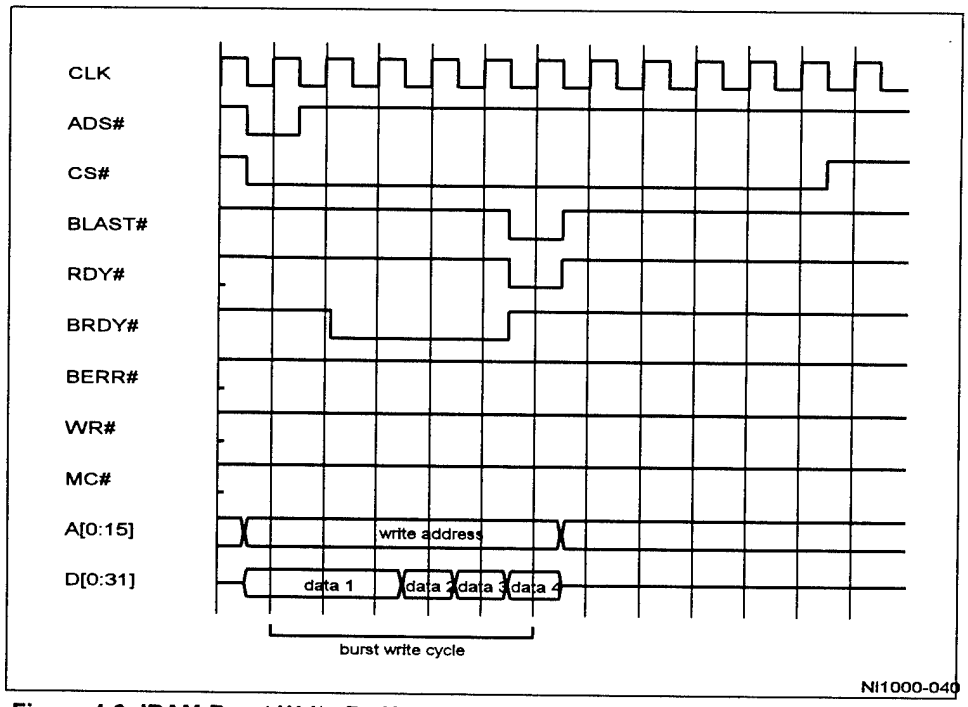


Figure 4-6. IRAM Burst Write By Host

4.2.6. ORAM Burst Read

Figure 4-5 previously showed the single (non-burst) read to the ORAM. Figure 4-7, below, shows a four-transfer burst read of the ORAM. As in burst cycles that access IRAM, the Accelerator asserts BRDY# instead of RDY# to indicate each successful data transfer in the multi-transfer sequence.

Figure 4-7 shows a burst read to a newly filled ORAM. The host begins the cycle with BLAST# deasserted. It keeps BLAST# deasserted through the third transfer, indicating to the Accelerator that these transfers are not expected to be the last transfer of the cycle (i.e., that this will be a burst cycle). In response, the Accelerator holds BRDY# asserted from the first data transfer through the last transfer (although RDY# can be substituted for BRDY# in the last transfer). The end of the cycle occurs eight clocks later when the host asserts BLAST# while the Accelerator asserts either BRDY# or RDY#.

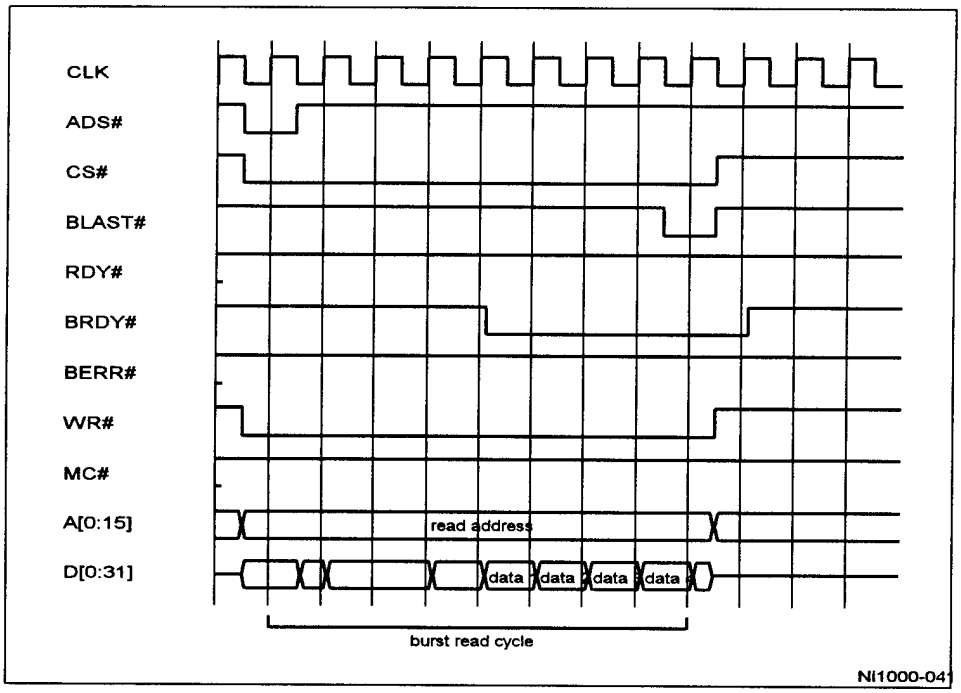


Figure 4-7. ORAM Burst Read By Host

4.2.7. Reset

As described in Section 4.1, all or parts of the Ni1000 Accelerator can be reset by the host asserting RESET# or MC#, or by the host writing a 1 to bit 15 of the CMR register. Figure 4-8 shows the cycle. Once reset, the Accelerator can only be brought back to normal operation by the host writing a 0 to bit 15 of the CMR register (not shown in Figure 4-8), provided that MC# is not asserted and RESET# is deasserted.

The top half of Figure 4-8 shows that, when RESET# is asserted while MC# is deasserted at the rising edge of CLK, the entire Accelerator is reset, which includes the external bus interface, PGFLASH, and other internal logic. When RESET# is deasserted, all units remain reset except the external bus interface, since this interface is reset only when RESET# is asserted. When MC# is asserted, PGFLASH is no longer in reset condition, but the other internal logic remains reset. When MC# is deasserted, PGFLASH re-enters the reset condition.

The bottom half of Figure 4-8 shows that when MC# is asserted while RESET# is deasserted at the rising edge of CLK, only the internal logic is reset, not including PGFLASH. The external bus interface is reset when RESET# is held low. When MC# is deasserted, PGFLASH enters the reset condition. Other internal logic remains reset throughout the cycle.

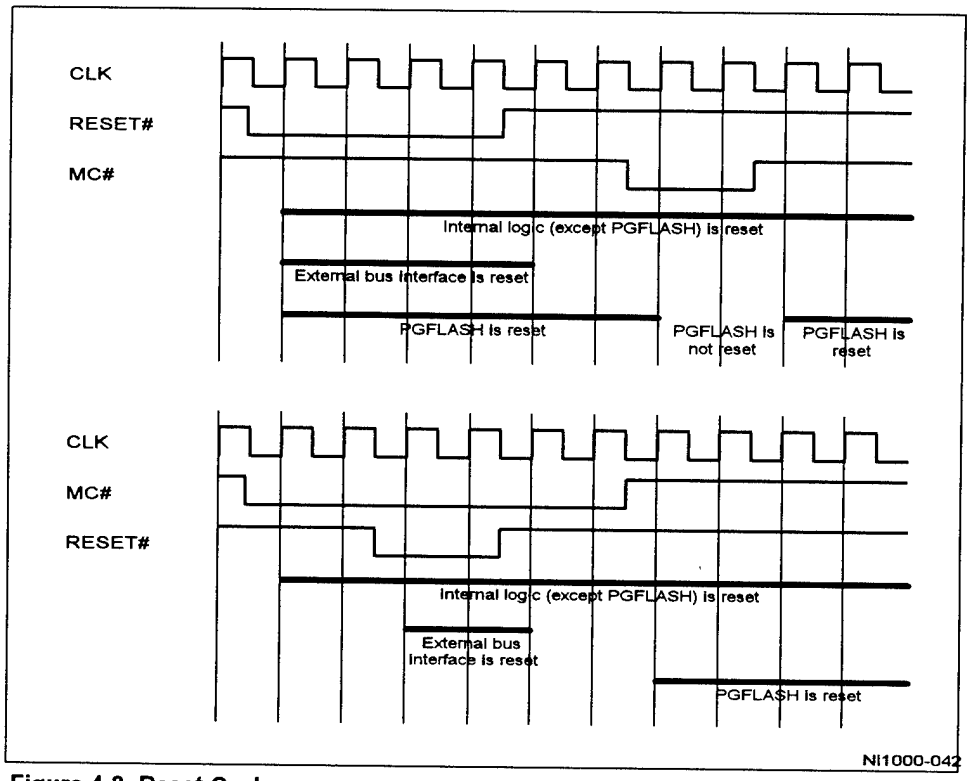


Figure 4-8. Reset Cycle

5. OPERATION

The Ni1000 has three access modes established by the host by controlling I/O pins; **Normal** mode, **PG** mode and **Reset** Mode.

1. **NORMAL** mode established by CS# = 0, MC# = 1, and RESET# = 1 is used for Classification, Learning and general housekeeping.

2. **PG** mode or PGFLASH ACCESS mode established by CS# = 0, MC# = 0, and RESET# = 1 is used to load and save PGFLASH.

3. **RESET** mode established by RESET# = 0 resets the Ni1000 and suspends precharging and most latching.

The Microcontroller software that implements several classification algorithms, controls the chip and communicates with the host, is available with the Ni1000. Custom programs for the Microcontroller can also be written using the Microcontroller instruction set provided.

The host-to-Microcontroller communication is mediated through the I/O unit: the I/O registers, the IRAM, and the ORAM. With the exception of accessing PGFlash, all communication with the Ni1000 is in NORMAL mode.

This section discusses the low level operation and programming of the Ni1000. Adherence to the guidelines given below is necessary to achieve reliable performance and avoid damage to the Accelerator.

General Programming and Operational Guidelines:

1. After power up, the Accelerator is in 'reset' state internally, even if the RESET# signal is deasserted. The host must activate the Accelerator (release from the reset state) by writing a 0 to CMR[15] before any operation is attempted.
2. After the Accelerator is activated (by writing a 0 to CMR[15]), Microcontroller code starts execution at location 1 in the PGFLASH (address F001h).
3. PGFLASH must contain a valid Microcontroller program before use. Otherwise, the Microcontroller may cause unexpected results or be damaged.
4. After changing from the *PG* to the *NORMAL* access mode, activate the Accelerator by writing a 0 to CMR[15] after MC# is deasserted.
5. All volatile memories (RAMs, registers) are undefined after power on and after switching between NORMAL and PG modes.
6. Data in PPRAM are valid as long as RESET# is deasserted and power is up. PPRAM data may be lost after switching into *PG* mode.
7. Always activate IRAM and ORAM before use, by writing a 0 to CRB[0] and CRB[1], respectively.
8. When interfacing with or modifying the Microcontroller software, the Protocol rules in Chapter 7, Microcontroller Software, supersede **any conflicting rules (e.g. #5 & #7,**

above), since the rules in this section refer to the native chip and the user-visibility can be modified using microcontroller code.

In addition to the three access modes described previously for the entire Accelerator each logic block (see Table 4-1), has its own register-controlled or software-controlled modes. These software-controlled modes determine the state of the blocks and the functions they can support. Accessing the software-controlled modes is accomplished by writing to appropriate registers associated with each logic block.

Table 5-1 summarizes the software-controlled modes for the relevant logic blocks, along with the register values for the corresponding modes. Usually, setting the software-controlled mode is one of many steps in the operation of a logic block. See Sections 5.1.3, 5.1.4, and 5.1.6 through 5.1.8 for the sequences of instructions.

Table 5-1. Logic Block Mode Configuration

LOGIC BLOCK	MODE	REGISTER VALUES	WRITTEN BY
I/O, IRAM & ORAM	RESET CLASSIFY MC (Microcontroller)	CMR[15] = 1 CRB[6] = 1 CRB[6] = 0	HOST or M/C HOST or M/C HOST or M/C
PPRAM	IDLE CLASSIFY MC (Microcontroller)	PPRAM_CR = 0h PPRAM_CR = 4000h PPRAM_CR = 8000h	Microcontroller Microcontroller Microcontroller
MURAM	CLASSIFY MC (Microcontroller)	MURAM_CR Bit = 1 and bits[2:5] = 1000h MURAM_CR = 0000h	Microcontroller Microcontroller
PADCU	DISABLED CLASSIFY MC (Microcontroller)	CSA = 0000h & CSB = 0000h CSA = 6000h & CSB = 6000h CSA = 8000h & CSB = 8800h	Microcontroller Microcontroller Microcontroller

I/O = IRAM, ORAM, and I/O Registers
M/C = Ni1000 on-chip Microcontroller
MURAM = Math Unit RAM
PPRAM = Prototype Parameter RAM
PADCU = Prototype Array and Distance Calculation Unit

5.1. I/O TO AND FROM HOST

Input and output between the host and Ni1000 while in NORMAL access mode utilizes the I/O registers, IRAM and ORAM. Table 5-2 shows the I/O Register Map.

Table 5-2. I/O Register Map

Address (Hex)	Name	Host W/R	MC W/R	Description
0000	CMR	W/R	R	Chip Mode Register.
0008	DIM	W/R	W/R	Vector Dimension Register.
0010	IDR	R	R	Chip ID Register.
0018	SSR	W/R	W/R	Software Status Register.
0020	HS1	R	R	Hardware Status Register 1.
0028	HS2	R	R	Hardware Status Register 2.
0030	XIR	R	W/R	External Interrupt Register.
0038	IIR	W/R	R	Internal Interrupt Register.
0040	CRA	W/R	W/R	Control Register A.
0048	CRB	W/R	W/R	Control Register B.
0050	OP0	W/R	W/R	General-purpose operand register 0.
0058	OP1	W/R	W/R	General-purpose operand register 1.
0060	OP2	W/R	W/R	General-purpose operand register 2.
0068	OP3	W/R	W/R	General-purpose operand register 3.
0070	OP4	W/R	W/R	General-purpose operand register 4.
0078	OP5	W/R	W/R	General-purpose operand register 5.

5.1.1. I/O Registers

The 16-bit I/O registers occupy addresses 0000h to 0078h in the memory. They can be read by both the host and the Microcontroller, but not all bits in all registers can be written. Their functions include:

- Control the operating mode of the IRAM and ORAM.
- Provide a location from which to read the hardware status signals.
- Provide communication between the host and the Microcontroller, which includes host commands to the Microcontroller, or input/output parameters between the host and the Microcontroller.

5.1.1.1. CMR (Chip Mode Register)

This 16-bit register is generally used to input commands from the host to the Microcontroller. The Microcontroller must not write to this register. A write by the host sets IIR[15], which causes an internal interrupt to the Microcontroller if the Interrupt Enable flag (HS1[6]) is set. When this happens, the interrupt request flag, IR, is set and visible to the Microcontroller. Figure 5-1 shows the register, followed by its bit assignments.

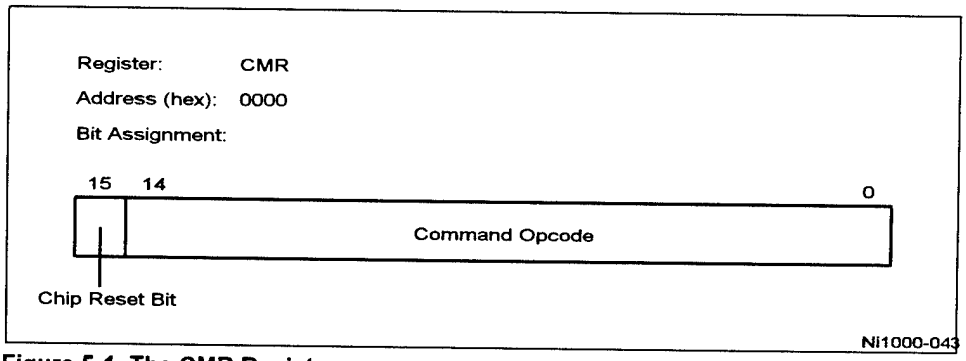


Figure 5-1. The CMR Register

bits [0:14] — *Command Opcode*
(read and write by host or Microcontroller)
User command opcode. The opcode is interpreted by the Microcontroller software.

bit 15 — *Chip Reset*
(read and write by host or Microcontroller, initialized to 1 upon chip reset.)
1 = Accelerator is reset. This bit is also set when the RESET# or MC# pin is asserted.
0 = Accelerator is not reset. The value can be written only by the host.

5.1.1.2. DIM (Dimension Register)

This 16-bit register contains the number of features (0-255) and the desired number of classes (0-63) in PRCE output. Both the host and the Microcontroller can read from and write to this register with no immediate side-effects. However, the value in the register must be stable before and throughout the classification process. Figure 5-2 shows the register, followed by its bit assignments.

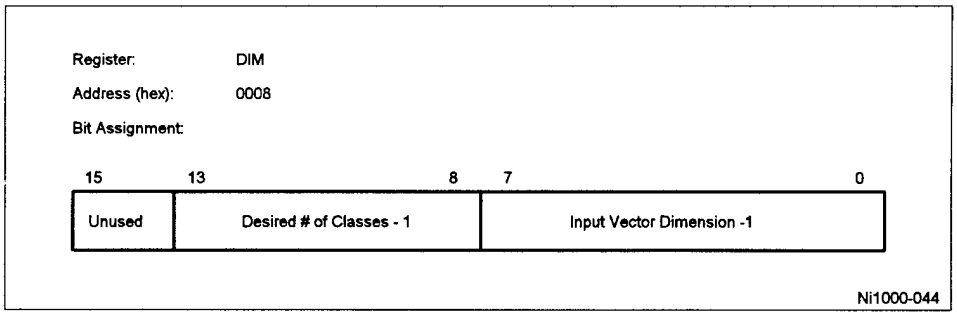


Figure 5-2. The DIM Register

bits [0:7] *Number of Input Features*
(read and write by host or Microcontroller)
The number of input features per input vector. The number to be entered here is the actual number of features minus 1.

bits [8:13] *Number of Output Classes*
(read and write by host or Microcontroller)
The desired number of classes for PRCE output. The number to be entered here is the number of desired output classes minus 1.

bits [14:15] *Reserved.*

5.1.1.3. IDR (ID Register)

This 16-bit register is read-only by both the host and the Microcontroller and hard-coded with the value 315Bh. It is the chip identification. Figure 5-3 shows the register.

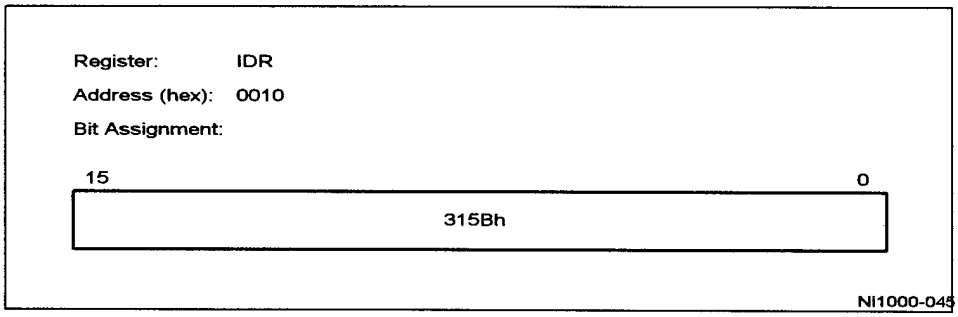


Figure 5-3. The IDR Register

5.1.1.4. SSR (Software Status Register)

This 16-bit register is intended to reflect the status of the microcontroller's software. Although the host could write this register, the intention is that it should only be written by the microcontroller. It has no effect on the Accelerator's operation. Figure 5-4 shows the register.

Bit-assignments must be defined in the Microcontroller's program. For how it is used in the standard Microcontroller program that is shipped with the chip, see Chapter 7, *Microcontroller Software*.

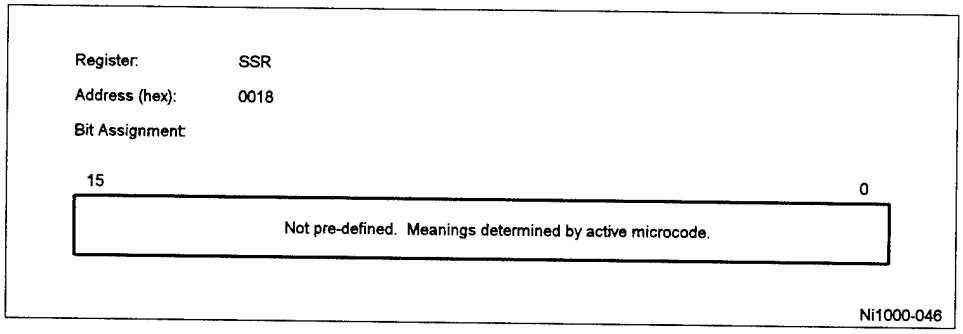


Figure 5-4. The SSR Register

5.1.1.5. HS1 (Hardware Status Register 1)

This 16-bit register is used to store the states of the microcontroller's flags, which are sampled at each clock cycle. It is intended to be read by the host. Reading of this register by the Microcontroller is less efficient than using the built-in Microcontroller flag-testing instructions. *CSW* is the Microcontroller flag register. See Section 5.4.1 for its bit assignments. Figure 5-5 shows the HS1 register, followed by its bit assignments, which are identical to that of CSW. Microcontroller flags are not the only flags that are used by the Accelerator. PPRAM (Prototype Parameter RAMs) have *Used* and *Disabled* flags, and DCUs also have *Used* flags. See Section 5.1.6 and 5.1.7 for details.

Register: HS1

Address (hex): 0020

Bit Assignment:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved		DC	M2	M1	FW	MC	GE	SE	IE	TR	O	P	N	Z	C

NI1000-047

Figure 5-5. The HS1 Register

All bits are read-only by both the host and the Microcontroller. 1 = set, 0 = clear.

bit 0	*C*	*Carry*
bit 1	*Z*	*Zero*
bit 2	*N*	*Negative*
bit 3	*P*	*Positive*
bit 4	*O*	*Overflow*
bit 5	*IR*	*Interrupt Request*
bit 6	*IE*	*Interrupt Enable*
bit 7	*SE*	*Stack Error*
bit 8	*GE*	*General Error*
bit 9	*MC*	*Multi-Class Firing*
bit 10	*FW*	*FLASH-Write*
bit 11	*M1*	*MURAM1 Ready*
bit 12	*M2*	*MURAM2 Ready*
bit 13	*DC*	*PADCU Busy*
bit 14		*Reserved*
bit 15		*Reserved* (always cleared to 0)

5.1.1.6. HS2 (Hardware Status Register 2)

This 16-bit register is used to indicate the status of the hardware units other than the Microcontroller. Bits 10 through 15 are particularly important, since they indicate the mode of the Accelerator and the full or empty status of IRAM and ORAM. They should be checked by the host before loading input vectors for classification, and before reading classification results. Figure 5-6 shows the register, followed by its bit assignments.

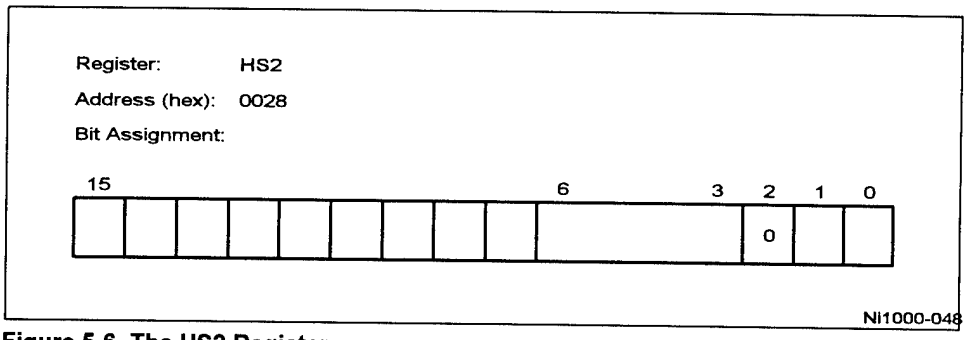


Figure 5-6. The HS2 Register

bit 0 — *PADCU Busy*
(read-only by host)
1 = PADCU is busy.
0 = PADCU is not busy.

bit 1 — *MU Busy*
(read-only by both the host and the Microcontroller)
1 = MU is busy
0 = MU is not busy.

bit 2 — *Complement of SRQ# Output*

bits [3:6] — *Last I/O Register Written*
(read-only by both the host and the Microcontroller)
Contain bits [3:6] of the address of the last I/O register written, either by the Microcontroller or host. Reading HS2 and masking its value with 0078h yields the last I/O register into which data was written. This provides communication between the host and the Microcontroller.

bit 7 — *64/32# Status*
(read-only by both the host and the Microcontroller)
1 = Host data bus is 64-bit.
0 = Host data bus is 32-bit.

bit 8 — *MULTICHIP# Status*
(read-only by both the host and the Microcontroller)
1 = The MULTICHIP# is asserted.
0 = The MULTICHIP# is deasserted.

bit 9 — *Multiple Firing Classes*
(read-only by both the host and the Microcontroller)
1 = ORAM contains multiple firing classes.
0 = ORAM does not contain multiple firing classes.

bit 10 — *ORAM Fully Read*
(read-only by both the host and the Microcontroller)
1 = ORAM has been fully read by the host.
0 = ORAM has not been fully read by the host.
This bit is necessary because multiple data transfers may be needed to read all of the data in ORAM.

bit 11 *IRAM Fully Written*
(read-only by both the host and the Microcontroller)
1 = IRAM has been fully written by the host.
0 = IRAM has not been fully written by the host.
This bit is necessary because multiple data transfers may be needed to write all of the data into IRAM.

bit 12 *ORAM Mode*
(read-only by both the host and the Microcontroller)
1 = ORAM is in Classify mode. The host can read ORAM if it is not empty.
0 = ORAM is in Microcontroller mode. The Microcontroller can access (read and write) ORAM. Any read attempt by the host is illegal and causes the Accelerator to assert BERR#.

bit 13 *IRAM Mode*
(read-only by both the host and the Microcontroller)
1 = IRAM is in *Classify* mode. The host can write to IRAM if it is not full.
0 = IRAM is in *Microcontroller* mode. The Microcontroller can access (read and write) IRAM. Any write attempt by the host is illegal and causes the Accelerator to assert BERR#.

bit 14 *ORAM Full*
(read-only by both the host and the Microcontroller)
1 = ORAM is full. The host can read ORAM if bit 12 has value 1.
0 = ORAM is not full. Any read attempt by the host is illegal and causes the Accelerator to assert BERR#.

bit 15 *IRAM Full*
(read-only by both the host and the Microcontroller)
1 = IRAM is not full. The host can write another vector to IRAM if bit 13 has value 1.
0 = IRAM is full. Any write attempt by the host is illegal, and causes the Accelerator to assert BERR#.

XIR (External Interrupt Register)

This 16-bit register is used to identify the reason for a service request from the Microcontroller to the host. It can be written only by the Microcontroller and is intended to be read by the host. A special value of FFFFh indicates that ORAM is full. The Microcontroller initiates a service request by writing to XIR. As a result, the Service Request pin ,SRQ#, is asserted and stays that way until the host asserts the interrupt acknowledge pin, IACK#.

Figure 5-7 shows the register. There are no specific bit assignments for the XIR register. The host and the Microcontroller should follow the same convention to code or decode its contents. For how it is used in the standard Microcontroller program that is shipped with the chip, see the Chapter 7, *Microcontroller Software*.

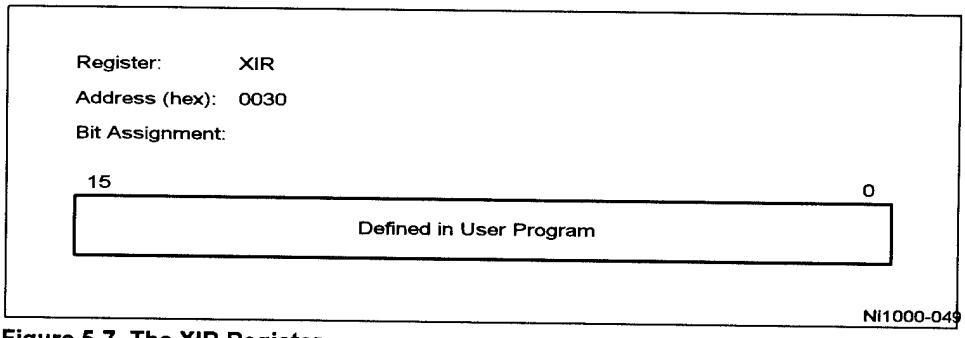


Figure 5-7. The XIR Register

5.1.1.7. IIR (Internal Interrupt Register)

This 16-bit register is used to identify the reason for an interrupt request from the host to the Microcontroller. It can only be written by the host. However, the Microcontroller responds to this interrupt only when its interrupt-enable (IE) flag is set. The contents of IIR may be changed by on-chip hardware conditions, such as the loading of the CMR register, or the assertion of the MCINT# pin by the host. Each bit position represents a different hardware condition, except that IIR[2:3] may be used by host software to specify the reason for an interrupt. The value in IIR is only an interrupt identifier, not an interrupt vector (see Section 5.2.7 for interrupt handling). Figure 5-8 shows the register, followed by its bit assignments.

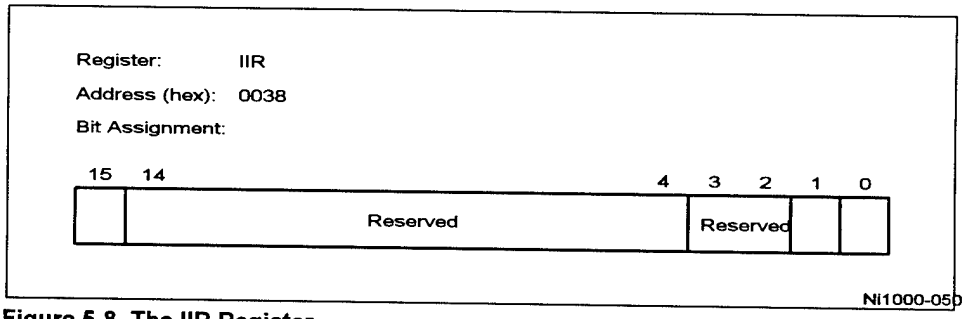


Figure 5-8. The IIR Register

bit 0 — *MCINT# Status*
(read and write only by host)
1 = The MCINT# pin is asserted by the host.
0 = The MCINT# pin is deasserted.

bit 1 — *ERROR# Status*
(read and write only by host)
1 = The ERROR# pin is asserted by the host.
0 = The ERROR# pin is not asserted.

bits [2:3] — *Reserved.*

bits [4:14] — *Reserved*
(Always reads zero.)

bit 15 *CMR Written*
(read-only by host)
1 = The CMR register has been written by the host.
0 = The CMR register has not been written by the host.

5.1.1.8. CRA (Control Register A)

This 16-bit register is used by the host to monitor and control the behavior of the IRAM and ORAM when the Accelerator is operating in *Classify* mode. However, it can be written by both the host and the Microcontroller. Figure 5-9 shows the register, followed by its bit assignments.

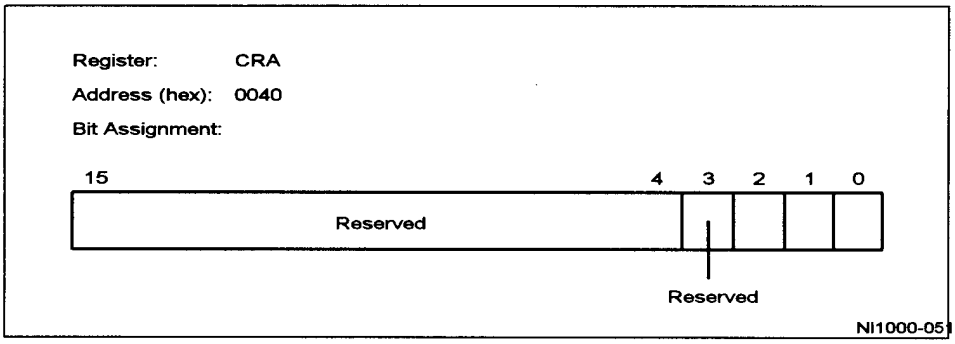


Figure 5-9. The CRA Register

bit 0 *ORAM Mode*
(read and write by both the host and the Microcontroller)
(Changing the value of this bit initiates an internal reset of the ORAM, followed by re-activation; HS2[14] is cleared to 0 to mark ORAM empty.)
1 = Results from MURAM are automatically loaded into ORAM whenever ORAM is empty.
0 = Results from MURAM are not automatically loaded into ORAM when ORAM is empty..

bit 1 *RCE/PRCE*
(read and write by both the host and the Microcontroller)
(Changing the value of this bit initiates an internal reset of the ORAM, followed by reactivation; HS2[14] is cleared to 0 to mark ORAM empty.)
1 = Results from MURAM are PRCE results, probability densities.
0 = Results from MURAM are RCE results, firing class IDs.

bit 2 *ORAM Service Request*
(read and write by both the host and the Microcontroller)
(Initialized to 1 upon chip reset.)
1 = The SRQ# pin will not be asserted and FFFFh will not be loaded into XIR when ORAM becomes full.
0 = The SRQ# pin will be asserted when ORAM becomes full and FFFFh will be loaded into the XIR register. ORAM full is indicated by HS2[14] = 1.

bits [3:15] *Reserved*

5.1.1.9. CRB (Control Register B)

This 16-bit register is used to control the software modes of the ORAM and the IRAM. It is written by the Microcontroller. Figure 5-10 shows the register, followed by its bit assignments.

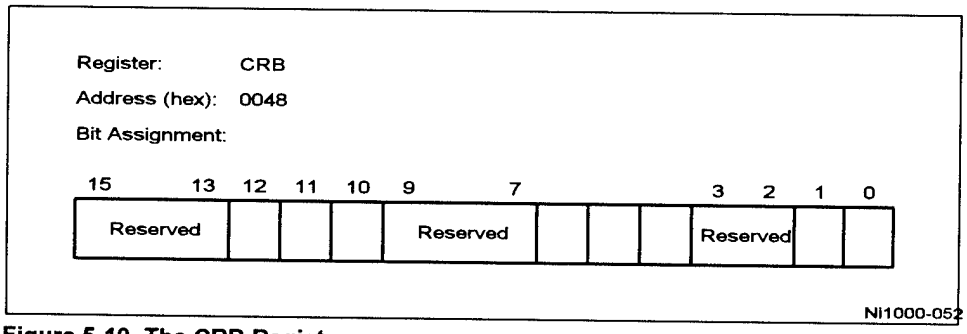


Figure 5-10. The CRB Register

bit 0 — *IRAM Reset*
(read and write by both the host and the Microcontroller)
(Initialized to 1 upon chip reset.)
1 = IRAM is reset.
0 = IRAM is active. The value must be written to bring IRAM out of reset.

bit 1 — *ORAM Reset*
(read and write by both the host and the Microcontroller)
(Initialized to 1 upon chip reset.)
1 = ORAM is reset.
0 = ORAM is active. The value must be written to bring ORAM out of reset.

bits [2:3] — *Reserved*

bit 4 — *Floating-Point Conversion*
(read and write by both the host and the Microcontroller)
(Initialized to 1 upon chip reset.)
1 = ORAM converts the PRCE probabilities obtained from MU into IEEE-754 32-bit format.
0 = PRCE data read out of the ORAM are in the MU internal 16-bit format.

bit 5 — *ORAM Mode*
(read and write by both the host and the Microcontroller)
(Initialized to 1 upon chip reset.)
1 = ORAM is in the Classify mode.
0 = ORAM is in the *Microcontroller* mode.

bit 6 *IRAM Mode*
(read and write by both the host and the Microcontroller)
(Initialized to 1 upon chip reset.)
1 = IRAM is in the Classify mode.
0 = IRAM is in the *Microcontroller* mode.

bits [7:9] *Reserved.*

bit 10 *IRAM1 Full*
(read and write by both the host and the Microcontroller)
1 = IRAM1 is full.
0 = IRAM1 is not full.

bit 11 *IRAM2 Full*
(read and write by both the host and the Microcontroller)
1 = IRAM2 is full.
0 = IRAM2 is not full.

bit 12 *ORAM Full*
(read and write by both the host and the Microcontroller)
1 = ORAM is full.
0 = ORAM is not full.

bits [13:15] *Reserved*

5.1.1.10.OP[0:5] (General Operand Registers)

These six 16-bit registers are storage locations used to pass parameters between the host and Microcontroller. They have no side-effects on hardware when written. Conventions for using these registers must be established and followed by both the host and the Microcontroller. For how they are used in the standard Microcontroller program that is shipped with the chip, see the Chapter 7, *Microcontroller Software*. Figure 5-11 shows the register. The bit assignments are defined in the user program.

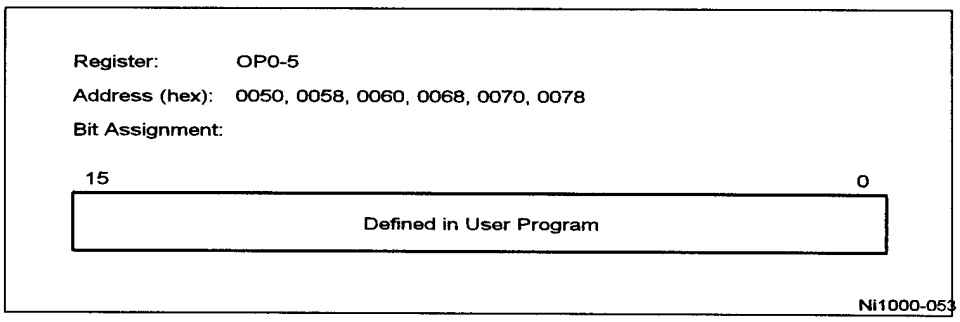


Figure 5-11. The OP Registers

5.1.2. IRAM

IRAM is a double buffer consisting of two alternating 256 x 5-bit banks, IRAM1 and IRAM2. Each bank can store one input vector with up to 256 stored features (up to 222 input features). IRAM has three software-controlled modes:

- *Reset*—This mode prohibits both the host and the Microcontroller from accessing IRAM. The mode is set by the host by writing a 1 to CMR[15], or by either the host or the Microcontroller by writing a 1 to CRB[0]. See Section 3.5.8 for required initialization before clearing the reset bit for normal operation.
- *Classify*—This mode allows the host to write into IRAM. The mode is set by the host by writing a 1 to CRB[6].
- *Microcontroller*—This mode allows the Microcontroller to write into or read from IRAM. The mode is set by the host by writing a 0 to CRB[6].

Instruction sequences for host and Microcontroller access to IRAM are given in Section 5.3.1. IRAM access addresses in the *Microcontroller* mode are shown in Table 5-3.

Table 5-3. IRAM Access Addresses in Microcontroller Mode

Location	Starting Address	Ending Address	# of Addresses	Resolution of Data
IRAM1 Readable Addresses	2000h	20FFh	256	5 bit
IRAM2 Readable Addresses	2100h	21FFh	256	5 bit
IRAM1 Latchable (Pre-Write) Addresses	2000h	20FFh	8 (3 least-significant address bits determine position)	5 bit (5 MSBs of the *low* byte on the DBUS)
IRAM2 Latchable (Pre-Write) Addresses	2100h	21FFh	8 (3 least-significant address bits determine position)	5 bit (5 MSBs of the *low* byte on the DBUS)
IRAM1 Writable Addresses	2400h	24FFh	32 (higher address bits judge RAM *row* to be written)	40 bits (written simultaneously)
IRAM2 Writable Addresses	2500h	25FFh	32 (higher address bits determine RAM *row* to be written)	40 bits (written simultaneously)

The latchable (pre-write) addressing scheme is created because of the specialized nature of the IRAM. Since the external data bus is 64-bit wide and the internal data bus 16-bit wide, multiple writes are required for internal data bus access. Figure 5-12 gives a more graphical explanation for this.

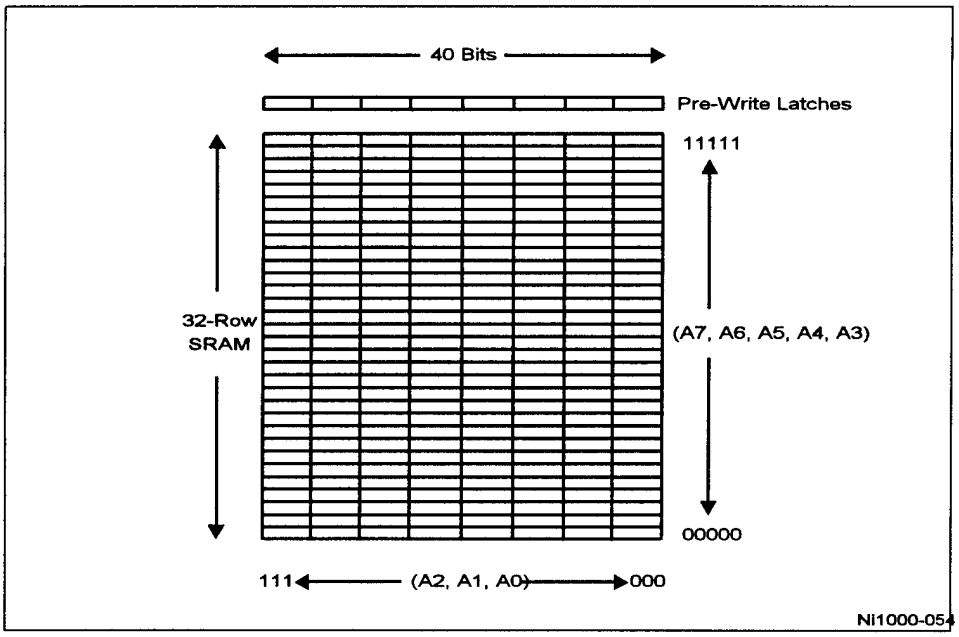


Figure 5-12. IRAM Pre-Write Latch Scenario

The bit assignments corresponding to Figure 5-12 are given in Figure 5-13.

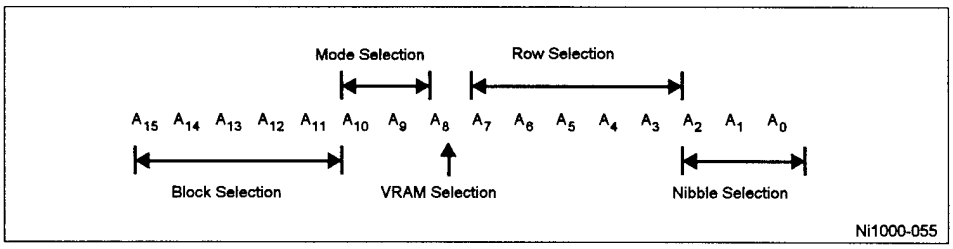


Figure 5-13. IRAM Address Assignment

Writing to the IRAM involves first writing to a set of latches. The contents of these latches may eventually be written into the IRAM itself, but the operation must be explicitly ordered. You may always choose to use the writable addresses, in which case some unwanted data may be written into other memory locations on that line of the RAM. Due to the orientation of the block, you must write 8 times to ensure that a line in the IRAM contains valid data.

When the host loads data into IRAM in *Classify* mode, each feature of the input vector is byte-aligned high. Only the upper 5 bits of each byte are used. The lower 3 bits are ignored.

Figure 5-14 and 5-15 show the data alignments for 32-bit and 64-bit external buses, respectively.

```
(MSB) dddddxxx dddddxxx dddddxxx ddddxxx (LSB)

d=valid data bit
x=invalid bit
```

Ni1000-056

Figure 5-14. Data Alignment on 32-Bit External Bus

```
(MSB) dddddxxx dddddxxx dddddxxx dddddxxx
      dddddxxx dddddxxx dddddxxx dddddxxx (LSB)

d=valid data bit
x=invalid bit
```

Ni1000-057

Figure 5-15. Data Alignment on 64-Bit External Bus

When the Microcontroller accesses IRAM (read or write), data appear in the most significant 5 bits of the least significant byte, as shown in Figure 5-16.

```
(MSB) xxxxxxxx ddddxxx (LSB)

d=valid data bit
x=invalid bit
```

Ni1000-058

Figure 5-16. Internal Bus Data Alignment

5.1.3. IRAM Read and Write by the Microcontroller

The host writes input vector components to IRAM for classification in the following steps:

1. Write to the following registers:

Register	Address (hex)	Data	Description
CRB[0]	0048	0	Unreset IRAM.
CRB[6]	0048	0	Set IRAM to *Microcontroller* mode.

2. Read from addresses 2000h to 21FFh, or write to addresses 2400h through 25FFh. Valid data appear as the most significant 5 bits of the least significant byte.

5.1.4. IRAM Write by the Host

1. Write to the following registers:

Register	Address (hex)	Data	Description
CRB[0]	0048	0	Unreset IRAM.
CRB[6]	0048	1	Set IRAM to *Classify* mode.
DIM[0:7]	0008	input vector features	The lower byte of DIM holds the number of input vector features minus 1.

2. Write an input vector to address 2000h, which can have up to 222 input features (in up to 256 input bytes). Each feature is byte-aligned high and only uses the upper 5 bits of each byte for data. The first feature value in each group of 8 must be set to 0 and the last two feature values (#254 and #255), if used, must be set to 0. High-features are written first, padded with empty (all 0s) bytes if necessary. Each write contains data for 4 (8) features if the 64/32# signal is asserted (deasserted).

3. Repeat step 2 until the data for all features of the vector are written.

For example, writing a vector with 5 features to IRAM on a 32-bit bus (64/32# asserted) will result in the following memory storage:

Address (hex)	Input Vector Dimension Number	Bit Pattern
2000	0	dddddxxx
2001	1	dddddxxx
2002	2	dddddxxx
2003	3	dddddxxx
2004	4	dddddxxx
2005	5	dddddxxx
2006	empty	00000xxx
2007	empty	00000xxx

d = valid data bit
x = invalid data bit

The status of IRAM is indicated in the following ways:

- A bus error occurs (the BERR# signal asserted) if a write to IRAM is attempted when both IRAMs are full. That is, the error will occur if the write is attempted while HS2[15] = 0.
- CRB[10] and CRB[11] show the status of IRAM1 and IRAM2, respectively:
  1 = The bank is full.
  0 = The bank is not full.
- HS2[15] indicates the status of entire IRAM:
  1 = IRAM is not full, i.e., at least one full-size vector maybe written.
  0 = IRAM is full and cannot take more input vectors.
- HS2[11] indicates whether data is being transferred into IRAM:
  1 = IRAM has been fully written by the host..
  0 = IRAM has not been fully written by the host and more transfers are required.

5.1.5. ORAM

ORAM is a buffer 64-words deep and 16-bits wide. It can hold all of the data generated by classifying an input vector. ORAM has three software-controlled modes:

- *Reset*—This mode prohibits both the host and the Microcontroller from accessing ORAM. The mode is set by the host by writing a 1 to CMR[15], or by either the host or the Microcontroller by writing a 1 to CRB[1]. See Section 3.4.8 for the initialization that is required before clearing the reset bit for normal operation.
- *Classify*—This mode allows the host to read from ORAM. The mode is set by the host by writing a 1 to CRB[5].
- *Microcontroller*—This mode allows the Microcontroller to write into or read from ORAM. The mode is set by the host by writing a 0 to CRB[5].

Instruction sequences for host and Microcontroller access to the ORAM are given in Sections 5.1.6 through 5.1.9. ORAM access addresses in *Microcontroller* mode are shown in Table 5-4.

Table 5-4. ORAM Access Addresses in Microcontroller Mode

Location	Starting Address	Ending Address	Number of Addresses	Resolution of Data
ORAM Readable Addresses	2800h	283Fh	64 (One of 4 words in 16 rows)	16 bit
ORAM Latchable (Pre-Write) Addresses	2800h	283Fh	1 (the first block in the shift register)	8 or 16 bits (selected with RCE#)
ORAM Writable Addresses	2C00h	2C3Fh	16 (possible rows in the RAM)	64 bits (written simultaneously)

The latchable (pre-write) addressing scheme exists because of the specialized nature of the ORAM. Figure 5-17 gives a more graphical explanation for this anomaly. The bit assignments corresponding to Figure 5-17 are given in Figure 5-18.

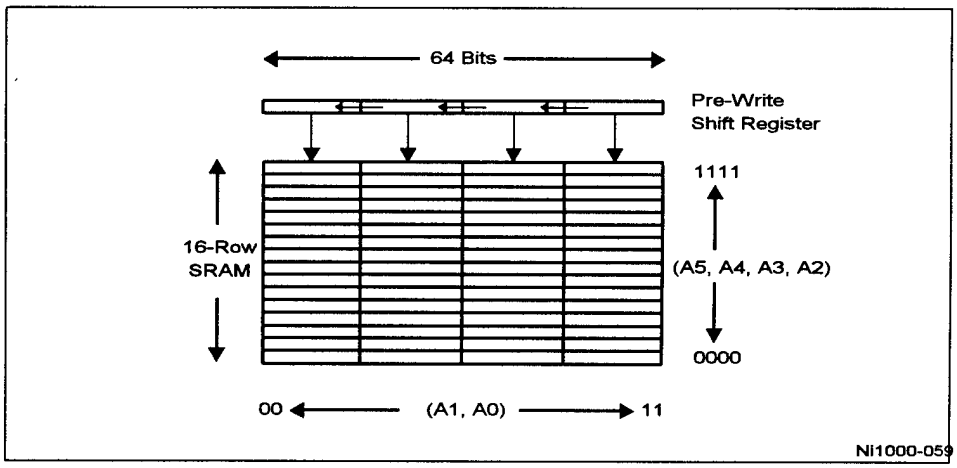


Figure 5-17. ORAM Pre-Write Latch Scenario

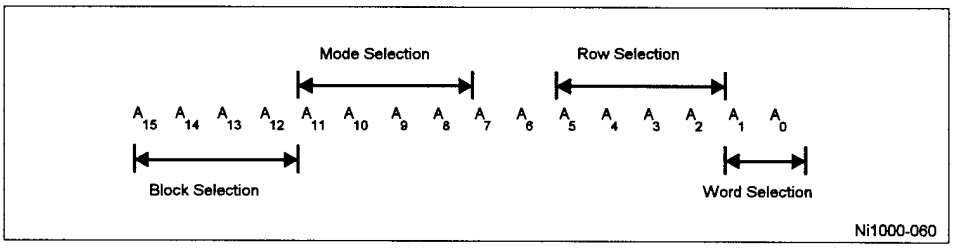


Figure 5-18. ORAM Bit Assignment

Writing to ORAM requires writing to a shift register first. The contents of this register may eventually be written into the static RAM itself. You may choose to always use the writable addresses, in which case some unwanted data may be written into other memory locations of the RAM. Due to the orientation of the block, you must write 4 times to ensure that a line in the RAM contains valid data. Any fewer write cycles may result in the desired data appearing *close to* but not *at* the desired memory location.

When the host reads the classification results from ORAM, the output is one of those summarized in Table 5-5.

Table 5-5. ORAM Output Possibilities

CRA[1]	CRB[4]	64/32#	Output (per bus cycle)	Maximum Possible Number of Bus Cycles
0	0	0	4 *Classes*	16
0	0	1	8 *Classes*	8
0	1	0	4 *Classes*	16
0	1	1	8 *Classes*	8
1	0	0	2 Unformatted *Probabilities*	32
1	0	1	4 Unformatted *Probabilities*	16
1	1	0	1 Formatted *Probabilities*	64
1	1	1	2 Formatted *Probabilities*	32

Class information is given in the format shown in Figure 5-19. The number of bytes output when *class* information is requested is determined by the number of firing classes, calculated by the math unit (MU). Regardless of the value of CRA[1], if the number of firing classes is greater than one, HS2[9] will be set to 1.

(MSB) vcnnnnnn vcnnnnnn vcnnnnnn vcnnnnnn (LSB)

n=firing class #
c=confidence bit (not meaningful on current rev of silicon)
v=valid byte (valid if HIGH, invalid if LOW)

Ni1000-061

Figure 5-19. Format of RCE Classification Results

Probabilities (actually probability densities) appear in one of the formats shown in Figure 5-20. The number of probabilities produced is determined by the *number of desired classes*, which you specify in the high byte of the DIM register (bits 8 through 15).

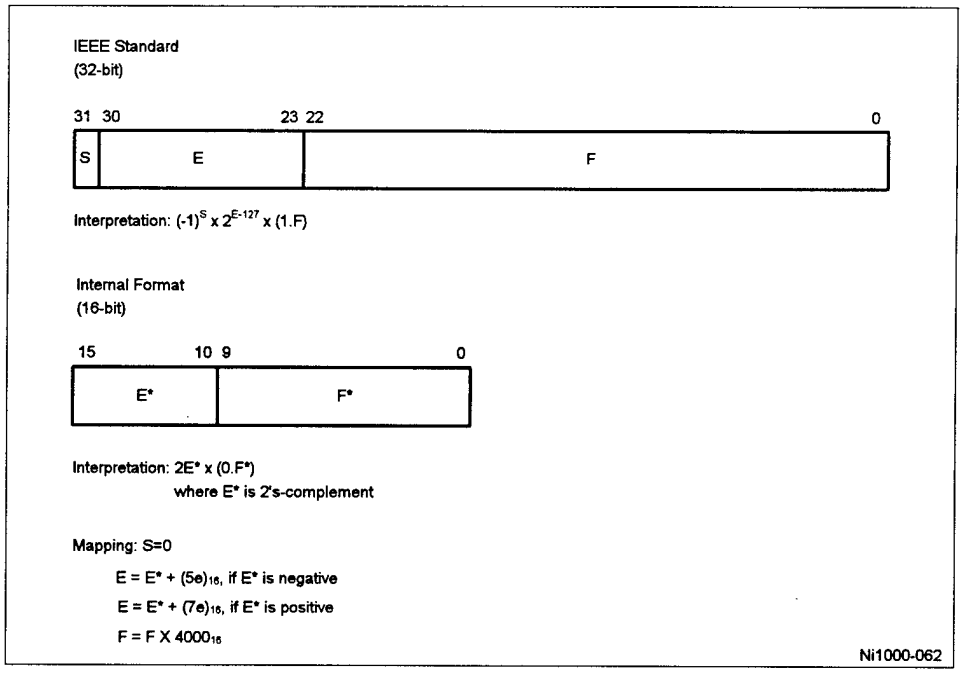


Figure 5-20. Floating-Point Formats

5.1.6. ORAM Read and Write by the Microcontroller

The host outputs classification results from ORAM in the following steps:

1. Write to the following registers:

Register	Address (hex)	Data	Description
CRB[1]	0048	0	Unreset ORAM.
CRB[5]	0048	0	Set ORAM to *Microcontroller* mode.

2. Read from addresses 2800h through 283Fh, or write to addresses 2C00h through 2C3F, respectively.

5.1.7. ORAM Read by the Host

1. Write to the following registers:

Register	Address (hex)	Data	Description
CRB[1]	0048	0	Unreset ORAM.
CRB[5]	0048	1	Set ORAM to *Classify* mode.
CRA[0]	0040	1	Set ORAM to *Classify* mode. Enable loading results from MURAM when ORAM is empty.
CRA[1]	0040	0 or 1	Select 0 for RCE firing classes as classification results. Select 1 for PRCE/PNN probability densities as classification results.
CRB[4]	0048	0 or 1	Select 0 for 32-bit output format. Select 1 for 16-bit output format.
DIM[8:15]	0008	desired class number	The upper byte of DIM holds the desired number of classes for the probability calculation.

2. Assert the 64/32# signal for 32-bit operation on data bus. Deasserted 64/32# for 64-bit operation.

3. Proceed to step 4 when data is available from ORAM, as indicated by CRB[12] = 1 or HS2[14] = 1.

4. Read data from address 2800h. See Section 5.1 for output types and floating-point formats.

5. Repeat step 4 for more data until one of the following occurs:

   - CRB[12] = 0 or HS2[14] = 0, indicating that ORAM has no more valid data.
   - The desired number of classes (for probabilistic results) is reached.
   - Invalid data bytes are encountered, for example, the MSB of a byte on the bus is zero for class IDs. Transmission must be terminated.
   - HS2[10] = 1, indicating that data transfer is completed.

5.1.8. Retrieving Both Class and Probabilistic Data from ORAM by the Host

If desired, both the firing class IDs and the probabilities may be retrieved from ORAM without reprocessing the data through the entire pipeline. This is accomplished by toggling bit 0 and 1 of the CRA register as in the following:

1. Write to the following registers:

Register	Address (hex)	Data	Description
CRA[0]	0040	0	Set ORAM to *Microcontroller* mode.
CRA[1]	0040	0	Select to first read firing-class IDs.

2. Proceed to step 3 when data is available from ORAM, as indicated by CRB[12] = 1 or HS2[14] = 1.

3. Read data from address 2800h, until there is no more data in ORAM, as indicated by CRB[12] = 0 or HS2[14] = 0. See Table 5-5 for number of outputs per bus cycle.

4. Read and mask the following register:

Register	Address (hex)	Data	Description
CRA[0]	0040	1	Set ORAM to *Classify* mode.
CRA[1]	0040	1	Select probabilities as outputs.

5. Write the masked value back to CRA.

6. Proceed to step 7 when data is available from ORAM, as indicated by CRB[12] = 1 or HS2[14] = 1.

7. Read data from address 2800h, until there is no more data in ORAM, as indicated by CRB[12] = 0 or HS2[14] = 0. See Table 5-5 for number of outputs per bus cycle, and Figure 5-20 for floating-point formats.

8. Read and mask the following register:

Register	Address (hex)	Data	Description
CRA[0]	0040	0	Set ORAM to *Microcontroller* mode.
CRA[1]	0040	0	Select firing-class IDs as output.

9. Write the masked value back to CRA.

10. Repeat from step 2 for subsequent vectors.

5.2. MICROCONTROLLER OPERATIONS

The Ni1000 utilizes a mapped memory scheme for access to all registers and memory locations. Access to the Microcontroller is accomplished via IRAM, ORAM and the I/O registers while the Ni1000 is in *NORMAL* access mode.

5.2.1. Internal Mapped Memory

Table 5-6 summarizes the memory and registers used in the Ni1000 Accelerator. The third column indicates whether the host can write and/or read the location. The fourth column indicates the same information for the Microcontroller (MC). A dash (-) indicates that the location is inaccessible.

Table 5-6. Memory and Register Address Map

Address (Hex)	Name	Host W/R	MC W/R	Description
I/O Registers				
0000	CMR	W/R	R	Chip Mode Register.
0008	DIM	W/R	W/R	Vector Dimension Register.
0010	IDR	R	R	Chip ID Register.
0018	SSR	W/R	W/R	Software Status Register.
0020	HS1	R	R	Hardware Status Register 1.
0028	HS2	R	R	Hardware Status Register 2.
0030	XIR	R	W/R	External Interrupt Register.
0038	IIR	W/R	R	Internal Interrupt Register.
0040	CRA	W/R	W/R	Control Register A.
0048	CRB	W/R	W/R	Control Register B.
0050	OP0	W/R	W/R	General-purpose operand register 0.
0058	OP1	W/R	W/R	General-purpose operand register 1.
0060	OP2	W/R	W/R	General-purpose operand register 2.
0068	OP3	W/R	W/R	General-purpose operand register 3.
0070	OP4	W/R	W/R	General-purpose operand register 4.
0078	OP5	W/R	W/R	General-purpose operand register 5.
GRAM				
1000-10FF	GRAM	-	W/R	Microcontroller general purpose memory, 256x16k.
TIMER				
1C00-1C01	TIMER	-	R	Clock count. Low word in 1C00, high word in 1C01. Cleared upon reset.

IRAM				
2000	IRAM_HW	W	-	IRAM host writable address.
2000-20FF	IR1_MCR	-	R	IRAM1 Microcontroller readable addresses.
		-	W	IRAM1 Microcontroller byte-oriented pre-write latches. Data occupy the high 5 bits of each byte.
2100-21FF	IR2_MCR	-	R	IRAM2 Microcontroller readable addresses.
		-	W	IRAM2 Microcontroller byte-oriented pre-write latches. Data occupy the high 5 bits of each byte.
2400-24FF	IR1_MCW	-	W	IRAM1 Microcontroller writable addresses.
2500-25FF	IR2_MCW	-	W	IRAM2 Microcontroller writable addresses.
ORAM				
2800	ORAM_HR	R	-	ORAM host readable address.
2800-283F	OR_MCR	-	R	ORAM Microcontroller readable addresses.
2C00-2C3F	OR_MCW	-	W	ORAM Microcontroller writable addresses.
PADCU Registers				
3001	CSA	-	W/R	PADCU Control and Status register.
3002	MODE	-	W/R	PADCU Mode register.
3004	DCU_DIM	-	W	PADCU Dimension register. It contains the number of features in the input vector. The value must be between 0 and 255, inclusive.
3008	NCA	-	W	PADCU register that contains the number of committed prototype vectors. The value must be between 0 and 999, inclusive.
3010	NCB	-	W	PADCU register that contains the MU clock count.
3020	AUX	-	W/R	PADCU auxiliary register.
3040	CSB	-	W/R	PADCU control and status register.
3200	ARR	-	W/R	PADCU Address Relocation register. Contains the lower-left position of the PA block in use. Value for starting row must be a multiple of 32, and column boundary must be a multiple of 128.

PPRAM and Registers				
4081	PPRAMCR3	-	W/R	PPRAM3 control register.
4101	PPRAMCR2	-	W/R	PPRAM2 control register
4201	PPRAMCR1	-	W/R	PPRAM1 control register
4381	PPRAM_CR	-	W	PPRAM global control register. It is used to write all three PPRAMs.
4400-47FF	PPRAM1	-	W/R	For each prototype vector, this RAM holds a 6-bit class ID, a 1-bit confidence flag, a 1-bit Used flag, and an 8-bit smoothing factor (mantissa plus exponent).
4800-4BFF	PPRAM2	-	W/R	For each prototype vector, this RAM holds a 13-bit threshold radius and a 1-bit Disabled flag.
5000-53FF	PPRAM3	-	W/R	For each prototype vector, this RAM holds a 16-bit count of the number of times that vector fired in the final epoch of the training process.
MURAMs and Registers				
6080	MURAM1	-	R	Firing class count for MURAM1.
60C0	MURAM2	-	R	Firing class count for MURAM2.
6100	MURAM_CR	-	W/R	MU mode-control register.
6200-623F	Flag MURAM	-	R	64 flags, used to indicate the firing classes for the current MURAM. Only the LSB is used.
6400-643F	Firing Class List MURAM1	-	W/R	One of the two alternating 64x8-bit buffers, reserved for the class IDs of firing classes.
6440-647F	Firing Class List MURAM2	-	W/R	One of the two alternating 64x8-bit buffers, reserved for the class IDs of firing classes.
6800-683F	Probability MURAM1	-	W/R	One of the two 64x16-bit buffers, used to accumulate the probability densities of the input vector for each of the 64 classes.
6840-687F	Probability MURAM2	-	W/R	One of the two 64x16-bit buffers, used to accumulate the probability densities of the input vector for each of the 64 classes.
PGFLASH Registers				
7700	PGF_DR	W/R	-	PGFLASH data register.
7701	PGF_CR1	W/R	-	PGFLASH control register 1.
7702	PGF_CR2	W/R	-	PGFLASH control register 2.
7703	PGF_SR	R	-	PGFLASH status register.
7704	PGF_ADR	W/R	-	PGFLASH address register.

Prototype Array				
B000-B3FF	PNUM DCU Used Flags DCU Distances	-	W/R	*PNUM*—Prototype number, which selects the prototype array column. The number must be between 0 and 999 inclusive. *DCU Used Flags*—1-bit (bit 13) flag for each of the 1000 prototype vectors. DCU hardware operates on a prototype vector only when the corresponding flag is set. *DCU Distances*—1000 City Block distances between an input vector and each of the prototype vectors. Each distance is 13-bit and aligned low. PADCU registers, MODE and AUX, are used for selection.
B800-B8FF	PDIM	-	W/R	Prototype dimension, which selects the prototype array row (feature). The number must be between 0 and 255, inclusive.
PGFLASH				
F000-FFFF	PGFLASH	W/R	R	Microcontroller program memory, 4Kx16-bit. The Microcontroller can only fetch instructions through Pdbus. The host can access PGFLASH only in PG mode.

5.2.2. PGFLASH

PGFLASH (Program FLASH) is the Microcontroller-program memory, occupying addresses F000h through FFFFh in the memory space. It is non-volatile flash memory with a size of 4K x 16-bits. PGFLASH modes are set by the host writing to appropriate PGFLASH registers. See Section 5.2.3 for the instruction sequences. The modes are:

- *Standby*—Disable PGFLASH outputs and high voltage circuits.
- *Read*—Allow the host to read a 16-bit word from the FLASH array or one of the six registers. This is the default mode after reset.
- *Erase*—Erase the entire FLASH array by setting all bits to 1.
- *Erase Verify*—A read operation that allows the host to verify that all bits in the FLASH array have been erased correctly.
- *Program*—Allow the host to write a 16-bit word in the FLASH array by selectively clearing bits to 0.
- *Program Verify*—A read operation that allows the host to verify that all bits in a 16-bit word have been programmed correctly in the FLASH array.

PGFLASH can only be downloaded (programmed) or uploaded (read) by the host when the Ni1000 is in ***PG*** access mode, which is entered by asserting the MC# signal. After programming and deasserting MC#, the Accelerator remains in **reset** mode until the host writes a 0 to CMR[15]. Downloading Microcontroller programs that exceed 4K x 16-bits in size will cause an error.

PGFLASH is programmed by accessing a set of registers listed in Table 5-7 and described in the following subsections. PGFLASH registers are used to store the address, data, control and status information.

Table 5-7. PGFLASH Registers

Address (Hex)	Name	Host W/R	MC W/R	Description
7700	PGF_DR	W/R	-	PGFLASH data register.
7701	PGF_CR1	W/R	-	PGFLASH control register 1.
7702	PGF_CR2	W/R	-	PGFLASH control register 2.
7703	PGF_SR	R	-	PGFLASH status register.
7704	PGF_ADR	W	-	PGFLASH address register.

5.2.2.1. PGF_ADR (Address Register)

This 12-bit register is used to store the address of the location to be read or written in PGFLASH. The register is write-only, and loaded automatically when the address is in the range between F000 and FFFF. Figure 5-21 shows the register.

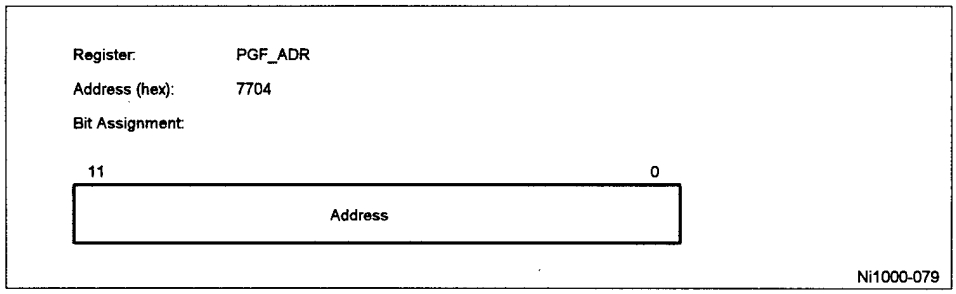


Figure 5-21. The PGF_ADR Register

5.2.2.2. PGF_DR (Data Register)

This 16-bit register is used to store the data to be programmed in the PGFLASH. Figure 5-22 shows the register.

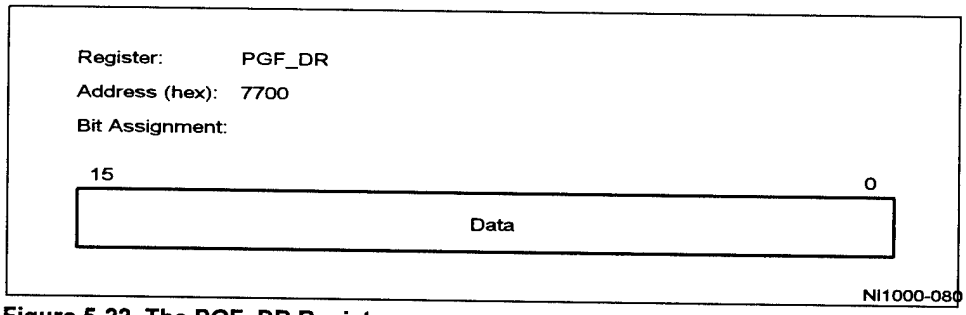


Figure 5-22. The PGF_DR Register

5.2.2.3. PGF_CR1 (User Control Register 1)

This 16-bit register is used to enable the software modes for PGFLASH. Figure 5-39 shows the register, followed by its bit assignments.

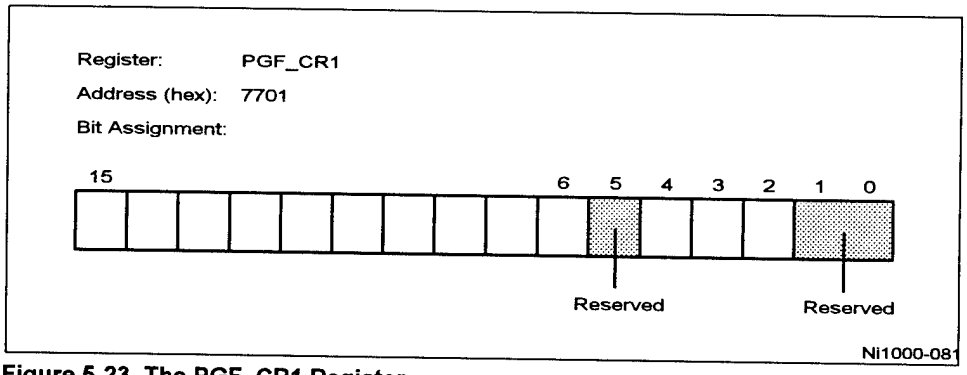


Figure 5-23. The PGF_CR1 Register

PGF_CR1 Register

Bit	Name	Description
0	CDSTSEB	Disables the pull down device on the array input node of the sense amplifier during VT, leaky column, program or program disturb operations.
1	CEROW	Ground all columns when PGFLASH is not selected and VPS is not high.
2	CNWL	Disable word lines during program disturb, standby and erase.
3	CYNBL	Deactivates the YSEL signals during erase.
4	CWNBL	Disables the WSEL signals during erase.
5	CIROWB	Ground Imprint Columns when Imprint Columns not used and CIREF not active.
6	CLVPSB	Ground the Array VPS.
7	CESAMPB	Enable sense amplifiers except anytime VPX tracks VPP during margin mode.
8	CREFRCEL	Gate single cell read reference to reference drain bias circuit or to sense node.
9	CREGATE	Enable the gate of the single cell read and erase verify reference cells.
10	CSCRLOD2	Enable 2:1 load ratio for reference of sense amplifier.
11	CIREF	Inhibit source of Imprint Column when VPX is high voltage, except program verify and HTRB II modes.
12	CHVPS	Bring the Array VPS to VPP during erase.
13	CSCRSELB	Gates SCRs to reference sense node.
14	CPDDIS	Disables all column pulldowns during modes in which all columns must be floated or controlled.
15	CEDIN	In program mode, enable data to be programmed.

5.2.2.4. PGF_CR2 (User Control Register 2)

This 8-bit register is used to enable the software modes of PGFLASH. Figure 5-40 shows the register, followed by its bit assignments.

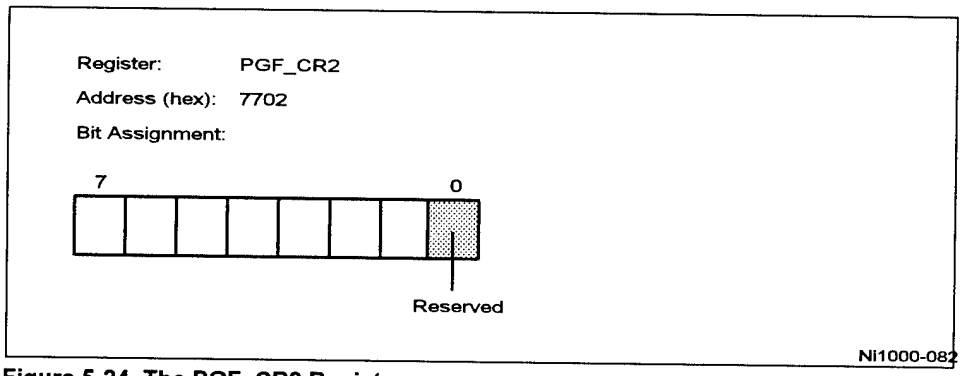


Figure 5-24. The PGF_CR2 Register

PGF_CR2 Register

Bit	Name	Description
0	RESERVED	Reserved.
1	CPGATE	Enable the gate of the single cell erase verify reference cell.
2	CREFPCEL	Gate single cell program verify cell to reference drain bias circuit, or to sense node.
3	CREFECEL	Gate single cell program verify cell to reference drain bias circuit, or to sense node.
4	CPGM	Bring VPX and VPY to high voltage during programming.
5	CPGMVER7	Enable VPX program verify generator to generate 7V during program verify.
6	CPGMVER	When CPGMVER7 is asserted, enable VPX program verify generator to generate 7.5V during program verify.
7	ADDREG1	Connect Address Register 1 to Address Mux.

PGF_SR (Status Register)

This 5-bit register is used to determine the status of key internal signals of the PGFLASH core circuitry. Figure 5-25 shows the register, followed by its bit assignments.

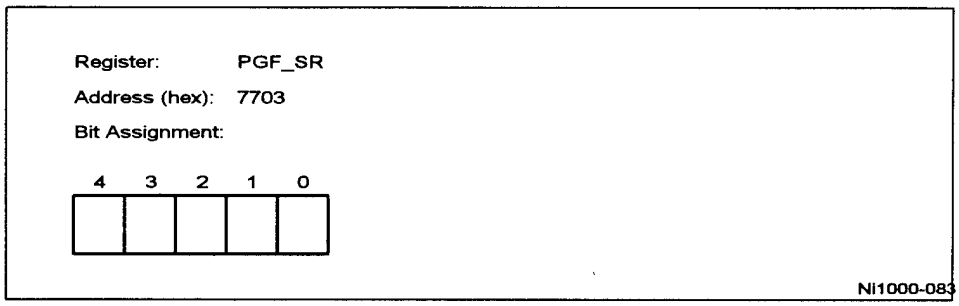


Figure 5-25. The PGF_SR Register

bit 0 Ground detected on Vps for PGFLASH.
bit 1 High voltage detected on Vpx.
bit 2 High voltage detected on Vpy.
bit 3 High voltage detected on Vpp.
bit 4 Low voltage detected on Vcc.

5.2.3. PGFLASH Programming

PGFLASH is the Microcontroller program memory. It can be programmed (downloaded) with Microcontroller programs only when the Accelerator is in the *PG* mode. The limitation on the size of the Microcontroller program is 4K x 16-bit. An error occurs when the size is exceeded. The software-controlled modes of PGFLASH are discussed in Section 5.2.2. The following instruction sequences give the steps to get into and out of various modes and the registers settings.

5.2.3.1. Standby

1. Write to the following registers to enable the Standby mode:

Register	Address (hex)	Data (hex)	Description
PGF_CR1	7701	2098	Sets PGF_CR1 and PGF_CR2 to
PGF_CR2	7702	0000	enable standby mode

5.2.3.2. Read

1. Write to the following registers:

Register	Address (hex)	Data (hex)	Description
PGF_CR1	7701	0700	Set PGF_CR1 and
PGF_CR2	7702	0086	PGF_CR2 to read mode

2. Write the address to be read in address register PGF_ADR at address 7704h.

3. Read PGFLASH data from data register PGF_DR at 7700h.

4. Go back to step 2 for new address; or proceed to step 5.

5. Enable *Standby* mode.

5.2.3.3. Erase

1. Write to the following registers:

Register	Address (hex)	Data (hex)	Description
PGF_CR1	7701	68BC	Set PGF_CR1 and
PGF_CR2	7702	0000	PGF_CR2 to erase mode

2. Write to the following register:

Register	Address (hex)	Data (hex)	Description
PGF_CR1	7701	78FC	Starts erase

3. Wait at least 2 milliseconds for erase.

4. Write to the following registers:

Register	Address (hex)	Data (hex)	Description
PGF_CR1	7701	68BC	Ends erase

5. Follow the procedure for *Erase Verify* and verify addresses F000 through FFFF. If anything was not erased, repeat this erase loop until everything in PGFLASH is erased, up to a maximum of 10 times.

6. Enable *Standby* mode.

5.2.3.3.1.Erase Verify

1. Write to the following registers:

Register	Address hex)	Data (hex)	Description
PGF_CR1	7701	200	Set PGF_CR1 and
PGF_CR2	7702	88	PGF_CR2 to erase verify mode

2. Read from desired address.
3. Read PGFLASH data from data register PGF_DR at 7700h.
4. Go back to step 2 for a new address; or proceed to step 5.
5. Enable *Standby* mode.

5.2.3.3.2.Program

1. Write to the following registers to put the PGFLASH in program mode:

Register	Address (hex)	Data (hex)	Description
PGF_CR1	7701	A888	Set PGF_CR1 and PGF_CR2 to
PGF_CR2	7702	10	enter program mode

2. Write the address to program in the address register:.

Register	Address (hex)	Data (hex)	Description
PGF_ADR	7704	address	address in PGFLASH to program

3. Write the data to be programmed into the data register:

Register	Address (hex)	Data (hex)	Description
PGF_DR	7700	data	data to be programmed into PGFLASH

4. Write to the following register:

Register	Address (hex)	Data (hex)	Description
PGF_CR1	7701	A880	Begin programming of PGFLASH

5. Wait 10 us for data to program.
6. Write to the following registers:

Register	Address (hex)	Data (hex)	Description
PGF_CR1	7701	A888	Set PGF_CR1 and PGF_CR2 to
PGF_CR2	7702	86	end program mode

7 Write to PGF_CR1 to enter program verify mode.

Register	Address (hex)	Data (hex)	Description
PGF_CR1	7701	200	Enter program verify mode

8. Without changing the address, read PGFLASH data from data register PGF_DR at 7700h.

9. If data was not programmed correctly, return to step 1 for the same data and address; or return to step 1 for new data and address; or proceed to step 10.

10. Enable *Standby* mode.

5.2.3.3.3.Program Verify

1. Write to the following registers:

Register	Address (hex)	Data (hex)	Description
PGF_CR1	7701	200	Set PGF_CR1 and PGF_CR2
PGF_CR2	7702	86	to enter program verify mode

2. Read from the desired address.

3. Read PGFLASH data from data register PGF_DR at 7700h.

4. Return to step 2 for new address; or proceed to step 5.

5. Enable *Standby* mode.

5.2.4. GRAM

GRAM is the general-purpose Microcontroller memory. It consists of 256 16-bit registers, occupying addresses 1000h through 10FFh. Constants and variables used by the Microcontroller software can be stored in GRAM.

5.2.5. TIMER

The 32-bit timer is accessed as a pair of 16-bit words, as shown in Figure 5-26. Location 1C00h is the lower half and 1C01h is the upper half. The timer is free-running, incremented at every clock. At 25 MHz, it wraps around to all zeros in 171.8 seconds. The timer is read only and is reset by the global reset pin, RESET#. Due to latency in reading the timer, it is best used for detecting non-terminating sequences as hangs rather than as a precise event timer.

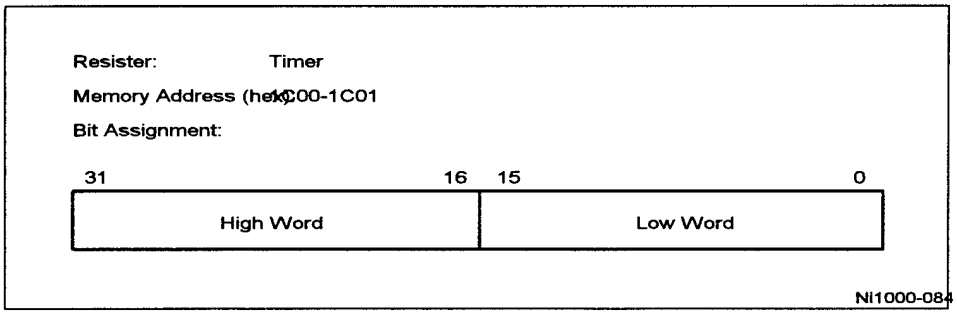


Figure 5-26. Timer

5.2.6. Interrupt Handling

Interrupt handling on the Ni1000 is accomplished by a combination of hardware and software. Since there is no specific hardware to queue multiple interrupts from the host, the system developer is responsible for implementing a protocol to ensure that interrupt requests are not overwritten or lost. The microcode program that is provided with the Ni1000 utilizes a protocol that is fully described in Chapter 7, *Microcontroller Software*.

Interrupts are enabled by setting Hardware Status Register #1 (HS1) bit 6, the interrupt enable (IE) flag to a 1. They are disabled by setting it to a 0. Interrupts are disabled following power on and initial activation of the chip. Interrupt requests are serviced immediately if the IE flag is set; otherwise, the request is deferred until the interrupt flag is set.

5.2.6.1. Host to Ni1000 Interrupts

Provided interrupts are enabled the Microcontroller interrupt request (IR) flag, which is visible to the host via CSW[5], is set when one of the following occurs:

- The host writes to the CMR register.
- The host writes to the IIR register.
- The host asserts the MCINT# pin.
- The host asserts the ERROR# pin.
- The Microcontroller clears the General Error flag (CSW[8]) after setting it.

The Ni1000 on-chip Microcontroller has a single interrupt vector at address F000h, which is the beginning of the Microcontroller program memory PGFLASH. The program counter PC is initialized to 1 instead of 0 for this reason. When the interrupt is requested, the Microcontroller immediately sets the host visible IR flag. This flag is visible via HS1[5]. The Microcontroller responds to the interrupt by reading the interrupt service routine (ISR) entry from the first location of PGFLASH, at address F000h, then performs a jump to subroutine ISR.

The interrupt request flag (IR) is cleared by the Microcontroller as it acknowledges the interrupt.

5.2.6.2. Ni1000 to Host interrupts

Interrupt request by the Microcontroller to the host is signaled by the service request pin, SRQ#. It is asserted when the Microcontroller writes to the XIR and deasserted when the host asserts the interrupt acknowledge pin, IACK#. Any outstanding service request is indicated by HS2[2].

If the IR flag is set by multiple events, only the one with the highest priority will be serviced by the Microcontroller. The priority order is:

1. The host asserts the Microcontroller Interrupt pin, MCINT# or writes a 1 to IIR[0].
2. The host asserts the Error pin, ERROR# or the host writes a 1 to IIR[1].
3. The host writes a 1 to IIR[2] or IIR[3].
4. The host writes to CMR[0:14].
5. The host writes a 0 to CMR[15] to reset the chip.

Interrupt Service Routine Example

The following is an example of an interrupt service routine (ISR) that calls a command interpreter and services hardware interrupts and other service requests. An example of the host service request is also provided. The following assumptions are made:

1. Since there is no acknowledge pin, the following example uses the host service request to gain acknowledge.

2. When the interrupt request is sent by the host writing to the CMR register, service from the Microcontroller is acknowledged through the general-purpose I/O registers.

3. The example does not handle any previously outstanding service requests. The actual implementation of the service routines are not provided, either. The example is given here for illustration purposes only and must not be used as a standard interrupt service routine. The user must implement his/her own routines to address application-specific requirements.

```
isrexam:                ; The physical address is F000h, which
                        ; is the first location in PGFLASH.

        cflgir          ; Clear the IR flag as soon as possible
                        ; to avoid missing new interrupts.

        push r0         ; Save registers.
        push r1
        rdi 38h, r0     ; Read IIR.

        ldi 0fh, r1     ; Get mask for interrupt request other
                        ; than that by writing CMR.
        and r0, r1
        jnz @isr1

        ldi 8000h, r1   ; Get mask for IR by writing CMR to
                        ; screening the IR by host clearing IIR.
```

```
        cmp r0, r1
        jnz @isrret         ; No service required for this IR.

                            ; Now IR by writing CMR.
        jsi cmrintr         ; Call Command Interpreter ---

                            ; interpreter for commands written in
                            ; CMR.
        jmp @isrret

@isr1:                      ; Provide acknowledge to host for IR
                            ; other than by writing CMR. If you
                            ; want to do the same for IR by writing
                            ; CMR, move this part up and reorganize
                            ; the code.

        ldi 4, r1           ; Mask for SRQ in HS2.

@isrw1:
        rdi 28h, r2         ; Read HS2.
        cmp r1, r2
        jnz @isrw2
                            ;
                            ; Put script here to notify SRQ

                            ; waiting for host or external hardware to
                            ; recognize that chip is waiting for
                            ; SRQ to be cleared.
                            ;
        jmp @isrw1

@isrw2:
        ldi 1111h,r1        ; Assume the host service request vector 1111h
                            ; is assigned for the Microcontroller
                            ; interrupt acknowledge.
        wri r1, XIR         ; Now the SRQ# pin is be asserted. The
                            ; host should clear IIR with this

                            ; request.
        mov one, r1         ; Get mask for interrupt request from
                            ; MCINT#.
        and r0, r1          ;
        jz @isr2

                            ; IR is from MCINT# pin or host writing
                            ; a 1 to IIR.
        jsi hwirsvc         ; Call service routine.
        jmp @isr4

@isr2:
        ldi 2, r1           ; Get mask for interrupt request from
```

```
                        ; ERROR# or the Microcontroller Error
                        ; flag.
        and r0, r1
        jz @isr3

                        ; IR is from ERROR# or the
                        ; Microcontroller error flag.
        jsi erirsvc     ; Call service routine.
        jmp @isr4

@isr3:                  ; IR is by the host writing IIR.
        jsi hsirsvc     ; Call service routine.

@isr4:                  ; Clear IR if by the host clearing IIR.
                        ; This can be removed.
        rdi 38h, r0
        cmp r0, rz
        jnz @isrret
        cflgir

@isrret:                ; Return processing.
        pop r2          ; Restore registers.
        pop r1
        pop r0
        sflgie          ; Enable interrupt.
```

5.2.7. Error Handling

The error pin ERROR# serves as both an input and an output. As an input, activating the ERROR# pin generates an interrupt to the Microcontroller provided the microcontroller's error flag is not active. The Microcontroller program, as supplied, causes it to freeze or suspend operations until the error has been reset by the host controller. As an output, the Microcontroller activates the ERROR# pin by activating its Error Flag, HS1[8].

Error handling is normally provided via the XIR register. Each time the Microcontroller completes a task from the host, it writes the command opcode it received into XIR[5-0] and a completion status or error code into XIR[14-8]. Error codes included with the Microcontroller software provided are listed in Chapter 7, *Microcontroller Software*.

5.2.8. Multiple Chip Support

The performance of an application may be improved linearly by using additional Ni1000 Accelerators. How many Ni1000s may be supported in parallel is primarily limited by the ability of the host to perform all of it's functions and sustain the I/O rates. Sustaining I/O rates is dependent on:

- I/O form (Burst vs. non-Burst)
- Number of input features. Vectors with fewer features classify faster.
- Number of desired classes or firing classes. More classes produce more output.
- Whether IEEE format output is required. The 16-bit non-IEEE format is faster.
- Bus size (16 or 32-bit width)
- Amount of processing performed by the host.

This section will discuss multi-chip support features of the Ni1000 and some hardware interface requirements.

The Chip Select pin, CS# may be used to select individual Ni1000s. A Multi-chip Pin , MULTICHIP# is available to notify the Ni1000 that it is being used in a multiple chip environment. The state of the Multi-chip pin is reflected in HS2[8].

5.3. CLASSIFIER ACCESS AND CONTROL

The Classifier consists of the *PA* (Prototype Array), *DCU* (Distance Calculation Unit), *PPRAM*, and *MURAM*. Operational modes and the register values used to establish them are shown in Table 5-8.

Table 5-8. Classifier Logic Block Mode Configuration

LOGIC BLOCK	MODE	REGISTER VALUES	WRITTEN BY
PA & DCU	DISABLED	CSA = 0000h & CSB = 0000h	Microcontroller
	CLASSIFY	CSA = 6000h & CSB = 6000h	Microcontroller
	MC (Microcontroller)	CSA = 8000h & CSB = 8800h	Microcontroller
PPRAM	IDLE	PPRAM_CR = 0h	Microcontroller
	CLASSIFY	PPRAM_CR = 4000h	Microcontroller
	MC (Microcontroller)	PPRAM_CR = 8000h	Microcontroller
MURAM	CLASSIFY	MURAM_CR Bit = 1 and bits[2:5] = 1000h	Microcontroller
	MC (Microcontroller)	MURAM_CR = 0000h	Microcontroller

MC = Ni1000 on-chip Microcontroller
MURAM = Math Unit RAM
PPRAM = Prototype Parameter RAM
PA & DCU = Prototype Array and Distance Calculation Unit

5.3.1. Prototype Array

The PA (Prototype Array) is a non-volatile flash memory. The PA occupies addresses B000h through B8FFh in the memory. It can store up to 1000 prototype vectors when they each have 222 features of five-bit resolution. This leaves 24 columns for the Bad Column Table, any bad columns and non-volatile memory for saving PPRAM for imbedded applications. When the prototype vectors have 32 features or less, the PA can store as many as 8000 such vectors. See the section describing the *ARR* register and the examples there.

The PA is organized as two 256 (row) x 512 (column) arrays. A row corresponds to a feature and a column corresponds to a prototype. Each array has 512 individually erasable blocks, each containing two columns. For example, column number 0 and column 512 are in the same block as are columns 1 and 513, etc. Data for either column (prototype) in a block can be written individually into the PA, but erasing operates on a block, potentially erasing two prototypes if both columns are used. It is the designer's responsibility to save the vector not intended for erasing. The standard microcontroller software saves and restores these columns when doing a COLUMNERASE command. When doing a COLUMNWRITE command, the host software must backup the "other" column . See Chapter 7, Microcontroller Software for more information.

Two addresses are required to access the PA: a column number, which has an address range of 0-1023, from B000h to B3FFh, and a row number, which has an address range of 0-255, from B800h to B8FFh. Access is a two-step process in which a read specifies the column number, followed by a read or write to the row. Valid data are returned on the second step. For example, to read row 10h of column number 21h, the following steps are required:

read address B021	column number 21h
read address B810	row 10h

It is not necessary to repeat the first read when a set of rows (features) of one column (prototype) is accessed. Two PADCU registers, CSA and CSB, hold the current column number and row number. They are updated when the upper 5 bits of address is 10110 for column number and 10111 for row number, respectively. For example, to read all features of a prototype at column number 20h with 6 features, the following steps are required:

read address B020	prototype number 20h
read address B800	first feature
read address B801	second feature
...	
read address B805	sixth feature

Each 5-bit feature is encoded as 10-bit data according to the following scheme:

Value	Encoded Data
0	01
1	10

The encoded data must be bitwise-inverted before being written into the PA. Reading the PA returns the bitwise-inverted encoded data. For example, if the desired value in PA is 14*d*, which is 01110 in binary, the encoded data is 01 10 10 10 01 (note, spaces inserted for clarity). The data actually written into the PA is 10 01 01 01 10 (again, spaces inserted for clarity and are not to be included in the actual data).

The use of the ARR affects PA addressing, as described later in this section. Access to the PA also requires setting of registers in appropriate sequences; see Section 5.3.2 for these instruction sequences. There are three PA operating modes: *Disabled*, *Classify* and *Microcontroller*, as shown in the list below. The PADCU registers must be set to proper values in each mode.

- *Disabled*—This mode prohibits access to the PA. The mode is entered by the Microcontroller by writing value 0000h to the CSA and CSB registers.
- *Classify*—This mode is used to perform classification. Other logic block modes must also be set properly. See Section 5.3.5 for details. The Microcontroller can access the PADCU registers in this mode. The mode is entered by the Microcontroller by writing values 6C00h and 6000h to the CSA and CSB registers, respectively.
- *Microcontroller*—This mode allows the Microcontroller to access the PA and to set, clear or read the DCU Used flags. A DCU is disabled if its Used flag is not set. The mode is entered by the Microcontroller by writing value 8000h to the CSA and CSB registers. Flag-and-Distance Read mode (FD Read) is a submode, which is entered by the Microcontroller by writing values 8000h and 8800h to the CSA and CSB registers, respectively.

5.3.1.1. Bad Column Table (BCT)

The last block of the PA (storage for columns 511 and 1023) is used as a Bad Column Table. If this block is faulty, the next block (for columns 510 and 1022) is used. The BCT stores the die's serial number, sort date, and faulty column locations in the PA flash memory. The BCT is organized as 256 10-bit numbers, as shown in Table 5-9. The BCT specifies locations of

faulty blocks (containing two prototype vectors if the number of features is 222 (padded to 256)), rather than prototype locations. The whole block must not be used if either prototype storage column is faulty. This avoids potential problems in PA programming and erasing.

Table 5-9. Bad Column Table

Entry #	Data
0	1001110000
1	0110001111
2-4	Serial Number
5-6	Sort Date
7	*Reserved*
8	Total Number of Bad Columns in the PA
9-15	*Reserved*
16-144	Bad Column Map
144	0110010000
145	1001101111
146-255	Bad Column List

Default Patterns—Since the column at which the BCT resides can also be faulty, four 10-bit code patterns at location 0, 1, 144 and 145, are used as validation codes. Any deviation from those shown in Table 5-9 means a faulty BCT.

Serial Number—Uses three 10-bit numbers.

Sort Date—Uses two 10-bit numbers. This information can be re-supplied within a limited period if accidentally erased.

Bad Column Map— Uses a 2-bit pattern for each of the 512 blocks in the PA:

10 a bad column
01 a good column

Each 10-bit number in the Bad Column Map stores data for four columns. The first two bits of the 10-bit number give the total number of bad columns among the four columns:

00 no bad column out of the four columns
01 one bad column out of the four columns
10 two bad columns out of the four columns
11 three or four bad columns out of the four columns

Bad Column List—A list of bad column numbers, each represented by a 10-bit number.

The following two examples illustrate what a 10-bit number in the Bad Column Map represents:

0001010101 : all four columns are good
0101100101 : the second of the four columns is bad

Use of the BCT varies, depending on the application software. For example, when new prototypes are committed and programmed into the PA during learning, the host must keep track of the locations of unused good columns. Otherwise, it may take a long time to locate the bad columns.

The PA and DCU registers are shown in Table 5-10. They are explained in the following section.

Table 5-10. PADCU Registers

Address (Hex)	Name	Host W/R	MC W/R	Description
3001	CSA	-	W/R	PADCU Control and Status register.
3002	MODE	-	W/R	PADCU Mode register.
3004	DCU_DIM	-	W	PADCU Dimension register. It contains the number of features of the input vector minus 1. The value is between 0 and 255, inclusive.
3008	NCA	-	W	PADCU register that contains the number of highest column containing a prototype (value between 0 and 1021, inclusive).
3010	NCB	-	W	PADCU register that contains the MU clock count.
3020	AUX	-	W/R	PADCU auxiliary register.
3040	CSB	-	W/R	PADCU control and status register.
3200	ARR	-	W/R	PADCU Address Relocation register. Contains the starting position of the PA block in use (lowest numbered row and column). Value for starting row # must be multiple of 32, and column boundary must be multiple of 128.

5.3.1.2. Control and Status Registers (CSA and CSB)

CSA and CSB are the 16-bit control and status registers of the PA and the DCU. Their contents are the status of the hardware FSM (Finite State Machine). They are initialized to 0000h upon power-up and chip reset (by either the host asserting the RESET# pin or writing a 0 to CMR[15]). The Microcontroller must write these registers to provide valid initial status of the machine or to change the mode of operation. The register settings in the three modes are shown in Figure 5-27 and Figure 5-28.

Register: CSA
Address (hex): 3001
Bit Assignment:

15			0
Register Value			
Software-Controlled Mode:	Disabled	Classify	Microcontroller
Register Value (hex):	0000	6C00	8000

NI1000-063

Figure 5-27. The CSA Register

Register: CSB
Address (hex): 3040
Bit Assignment:

15			0
Register Value			
Software-Controlled Mode:	Disabled	Classify	Microcontroller
Register Value (hex):	0000	6000	8000 8800 (for FD-read)

NI1000-064

Figure 5-28. The CSB Register

5.3.1.3. Hardware Setting Registers (MODE and AUX)

The MODE and AUX registers are 16-bit hardware-mode-setting registers. They are used for reading, erasing and programming the flash EPROM cells of the PA. They are initialized to 0000h. Only the values given in Figure 5-29 and Figure 5-30 are valid. Do not set them to any other values.

Register: MODE

Address (hex): 3002

Bit Assignment:

15			0
Register Value			

Software-Controlled Mode:	Disabled	Classify	Microcontroller
Register Value (hex):	0000	0100	0040 (set flags) 0020 (clear flags) 0080 (read flags)

NI1000-065

Figure 5-29. The MODE Register

Register: AUX

Address (hex): 3020

Bit Assignment:

15			0
Register Value			

Software-Controlled Mode:	Disabled	Classify	Microcontroller
Register Value (hex):	0000	0000	0000

NI1000-066

Figure 5-30. The AUX Register

5.3.1.4. Address Relocation Register (ARR)

The PA can store as many as 1000 prototype vectors with up to 222 features. When there are fewer than 1000 vectors or fewer than 222 features in each vector, the PA can be segmented into blocks. Each block may store the prototype vectors for a particular application problem. The 16-bit *ARR* register specifies the starting position in the PA of the region in use (lowest numbered row and column). Figure 5-31 shows this register followed by its bit assignments. The column offset gives the starting column number modulo of 128, and the row offset gives the starting row number modulo of 32. This representation results in a total of 56 possible blocks (8 of 64 theoretically possible include columns allocated for the BCT and PACT and are unavailable).

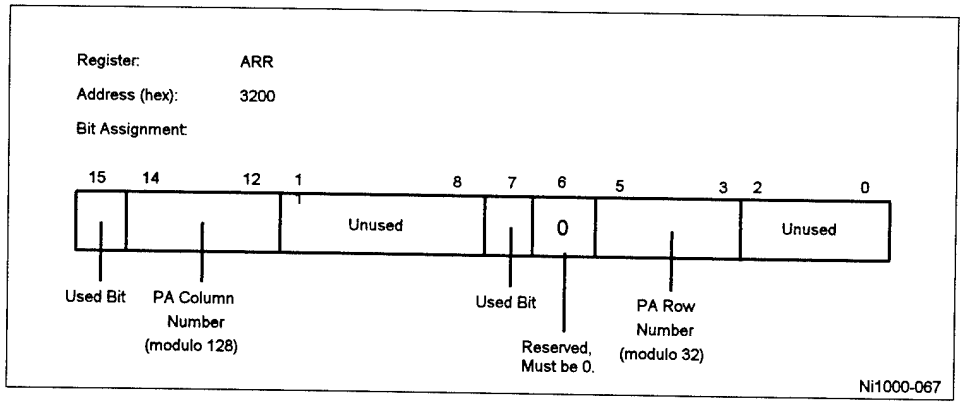


Figure 5-31. The ARR Register

bit(s)	*Function/Value*	**Description**
[0:2]	*Reserved*	
[5:3]	*Row Offset*	Starting row number, modulo 32.
[6]	*Reserved, must be 0.*	Forces high half of 512x512 memory if 1. Used for testing.
7	*Row Relocation Used Bit* 1 0	 PA row relocation is used. Row offset is given by bits [4:6]. PA row relocation is not used.
[8:11]	*Reserved*	
[12:14]	*Column Offset*	Starting column number in PA, modulo of 128.
15	*Column Relocation Used Bit* 1 0	 PA column relocation is used. Column offset is given by bits [12:14]. PA column relocation is not used.

Two additional registers specify the size of the block. *DCU_DIM* contains the dimensions of the prototype vectors in the block. *NCA* contains the index of the last prototype vectors in the region. This index is from 0 to 127, inclusive, when using address relocation in the low half of the prototype array. It is necessary to add 512 to the index when using address relocation in the upper half of the array. Figure 4-31 provides a graphical explanation. The starting position of a region (given by the ARR register) must coincide with the cross-points of the vertical and horizontal lines. The size of a block, however, is limited. When row relocation is enabled, a maximum of 32 features (28 usable) can be used and the block cannot cross the 511/512 boundary. When column relocation is enabled, a maximum of 128 columns are available. This number is reduced by bad columns or if the reserved columns fall within the block. Two examples are given in Figure 5-33 and Figure 5-34.

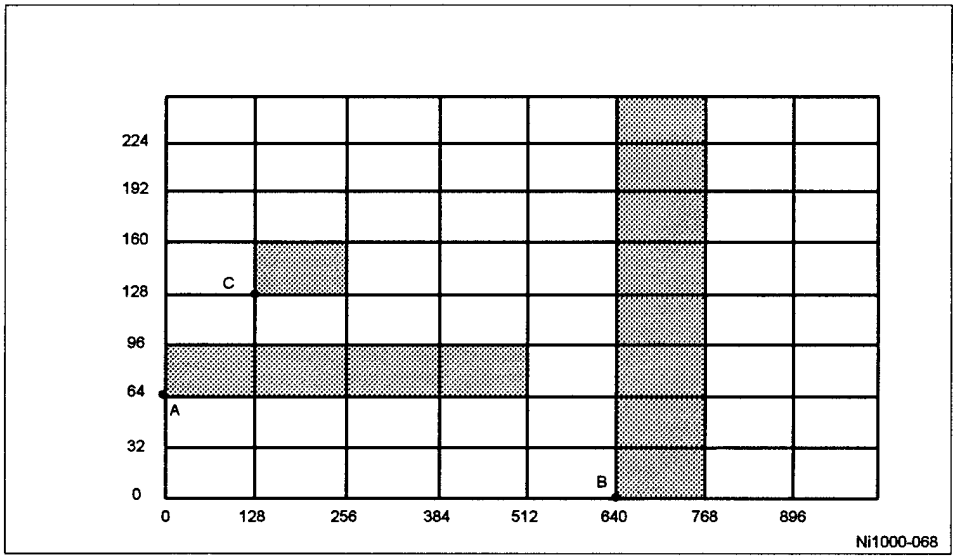


Figure 5-32. Prototype Array Segmentation

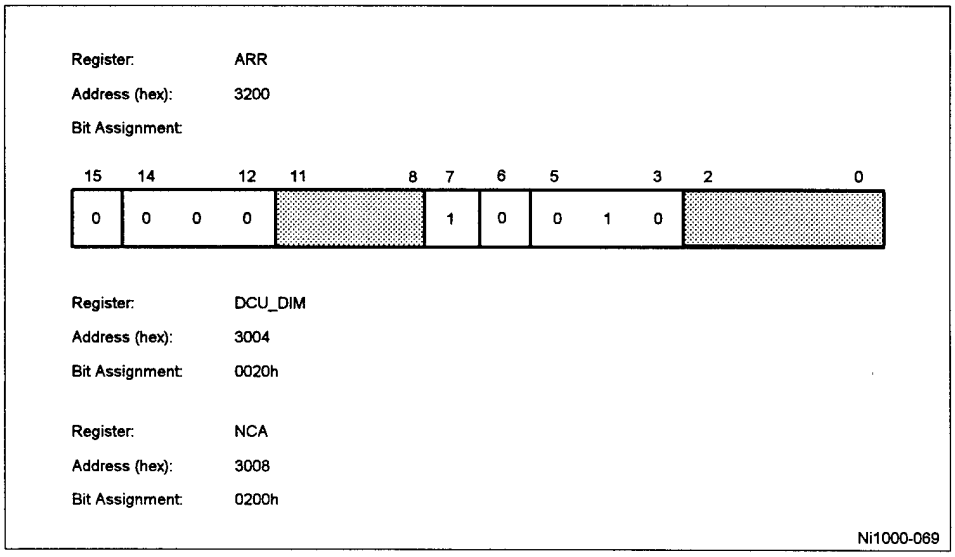


Figure 5-33. Example A of Specifying a Window in PA

The block is shown as the shaded area A in Figure 5-32.

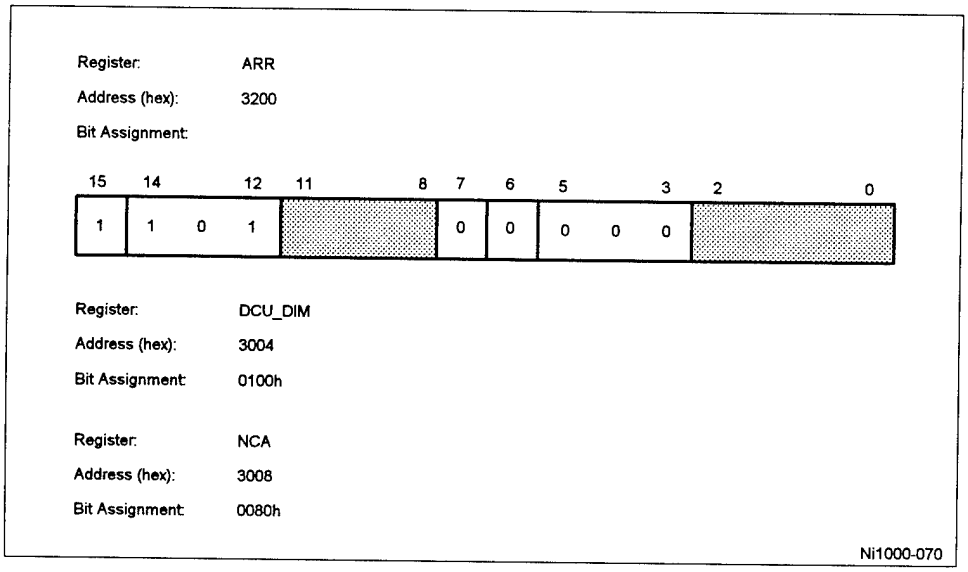


Figure 5-34. Example B of Specifying a Window in PA

The block is shown as the shaded area B in Figure 5-32.

The column and row offsets specified in the ARR register are relative to location (B000h, B800h). When the *Column* and *Row Relocation Used Bits* of ARR are clear, PA is addressed the same way as described at the beginning of this section. When the either of these bits is set, addressing is relative to the position in PA specified by the *Column* or *Row Offset* fields of ARR, respectively. For example, the value of ARR in Figure 5-33 specifies a PA block starting at column 0 (no offset), row 64 (2 * 32). Then, writing to location (B001h, B802h) will modify the PA entries at column 1, row 66.

5.3.1.5. Other Registers (DCU_DIM, NCA and NCB)

The DCU_DIM and NCA registers contain information about the window of prototype vectors in use in the PA. These registers are shown in Figure 5-35 and 5-36, respectively. A graphical explanation of the window is given in Figure 5-34, above. The NCB register shown in Figure 5-37, contains the clock count in the math unit. This value should be 8*h*.

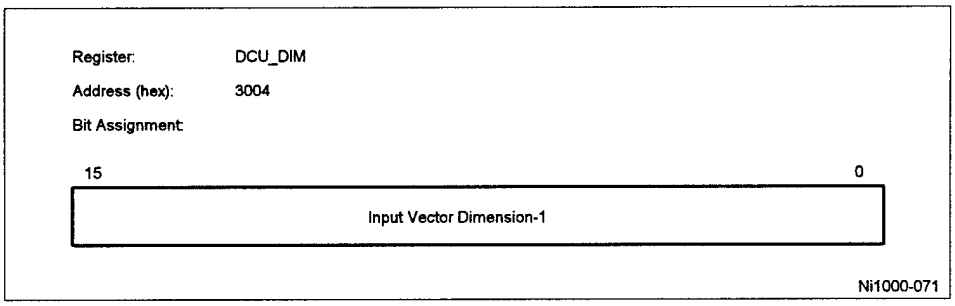


Figure 5-35. The DCU_DIM Register

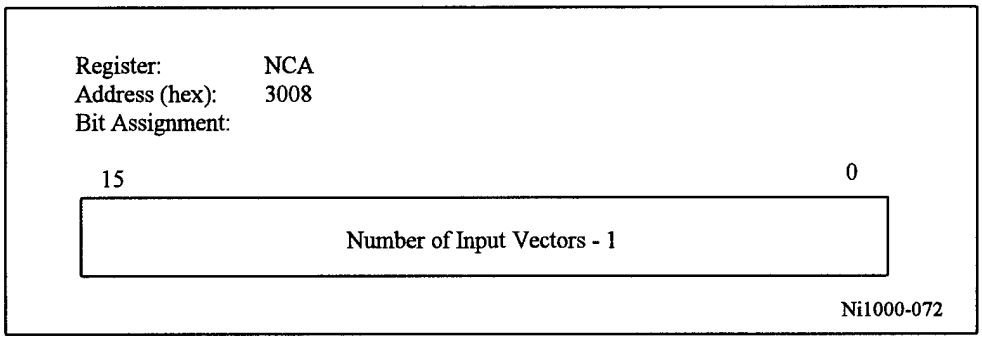


Figure 5-36. The NCA Register

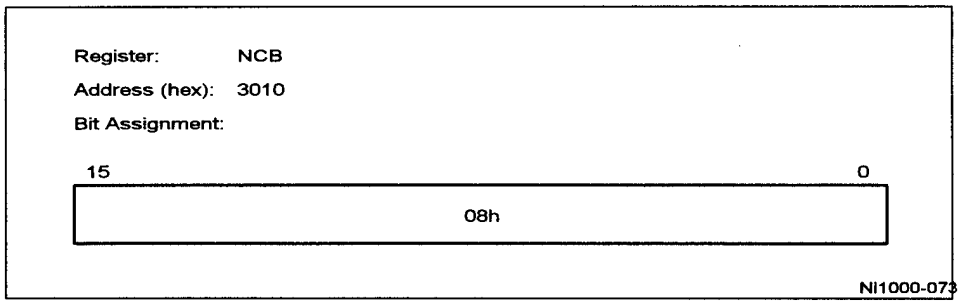


Figure 5-37. The NCB Register

5.3.2. PA Access

Only the Microcontroller can directly access the prototype array. The host must access PA through the Microcontroller. See Section 5.3.1 for how data are encoded and stored in PA.

5.3.2.1. Prototype Array Read by the Microcontroller

1. Write to the following registers:

Register	Address (hex)	Data	Description
CSA	3001	8000h	Set PA to *Microcontroller* mode.
CSB	0040	8000h	Set PA to *Microcontroller* mode.
MODE	3040	0000h	Initialize the MODE register.
AUX	3020	0000h	Initialize the AUX register.

2. Read from address (B000 | column number)$_h$, where "|" is the bitwise OR operation.

3. Read data from address (B800 | row number)$_h$. As many as 256 features can be read in sequence without re-entering the column number.

4. Write to the following registers:

Register	Address (hex)	Data	Description
MODE	3040	0000h	Reinitialize the MODE register.
AUX	3020	0000h	Reinitialize the AUX register.

5. Repeat from step 1 for another column.

   For example, the following sequence constitutes the above steps 2 and 3 for reading a prototype at column 20h containing 6 features:

```
read address B020
read address B800        first feature
read address B801        second feature
...
read address B805        sixth feature
```

5.3.2.2. Prototype Array Programming by the Microcontroller

1. Write to the following registers:

Register	Address (hex)	Data	Description
CSA	3001	8000h	Set PA to *Microcontroller* mode.
CSB	0040	8000h	Set PA to *Microcontroller* mode.
MODE	3040	0000h	Initialize the MODE register.
AUX	3020	0000h	Initialize the AUX register.

2. Read from address (B000 | column number)$_h$, where "|" is the bitwise OR operation.

3. Write data to address (B800 | row number)$_h$. As many as 256 rows can be written in sequence without re-entering the column number.

4. Write to the following registers:

Register	Address (hex)	Data	Description
MODE	3040	8110h	Program the flash cells.
AUX	3020	01D1h	Program the flash cells.

5. Wait 10 microseconds for programming.

6. Write to the following registers:

Register	Address (hex)	Data	Description
MODE	3040	0000h	Reinitialize the MODE register.
AUX	3020	0000h	Reinitialize the AUX register.

7. Repeat from step 1 for another column.

For example, the following sequence constitutes the above steps 2 and 3, for writing a prototype at column 20h containing 6 features:

```
read    address B020
write   address B800        first feature
write   address B801        second feature
...
write   address B805        sixth feature
```

5.3.2.3. Prototype Array Erase by the Microcontroller

1. Write to the following registers:

Register	Address (hex)	Data	Description
CSA	3001	8000h	Set PA to *Microcontroller* mode.
CSB	0040	8000h	Set PA to *Microcontroller* mode.
MODE	3040	0000h	Initialize the MODE register.
AUX	3020	0000h	Initialize the AUX register.

2. Read from address (B000 | column number)$_h$, where "|" is the bitwise OR operation.

3. Write to the following registers:

Register	Address (hex)	Data	Description
MODE	3040	4900h	Erase the flash cells.
AUX	3020	0001h	Erase the flash cells.

4. Wait 10 to 100 milliseconds for erasing.

5. Repeat from step 2 for another column.

5.3.2.4. Prototype Array Program-Verify and Erase-verify by the Microcontroller

The instruction sequence is the same as the read sequence, except that at step 4, the AUX register is set to the following values:

Program-verify: 2060h
Erase-verify: 1070h

5.3.3. PPRAM

The PPRAM (Prototype Parameter RAM) stores the parameters associated with the prototypes. These parameters are used by the MU (Math Unit) to compute the firing-class IDs and the probability densities. PPRAM and its associated registers are shown in Table 5-11 and described in the following section.

Table 5-11. PPRAM and Registers

Address (Hex)	Name	Host W/R	MC W/R	Description
4081	PPRAMCR3	-	W/R	PPRAM3 control register.
4101	PPRAMCR2	-	W/R	PPRAM2 control register
4201	PPRAMCR1	-	W/R	PPRAM1 control register
4381	PPRAM_CR	-	W	PPRAM global control register. It is used to write all three PPRAMs.
4400-47FF	PPRAM1	-	W/R	For each prototype vector, this RAM holds a 6-bit class ID, a 1-bit probabilistic flag, a 1-bit Used flag, and an 8-bit smoothing factor (exponent, mantissa)
4800-4BFF	PPRAM2	-	W/R	For each prototype vector, this RAM holds a 13-bit threshold radius and a 1-bit Disabled flag.
5000-53FF	PPRAM3	-	W/R	For each prototype vector, this RAM holds a 16-bit count of the number of times that vector fired in the last epoch of the training process.

Bit fields of the three PPRAMs are shown in Figure 5-32 and are described below. The PPRAMs occupy addresses 4400h through 53FFh.

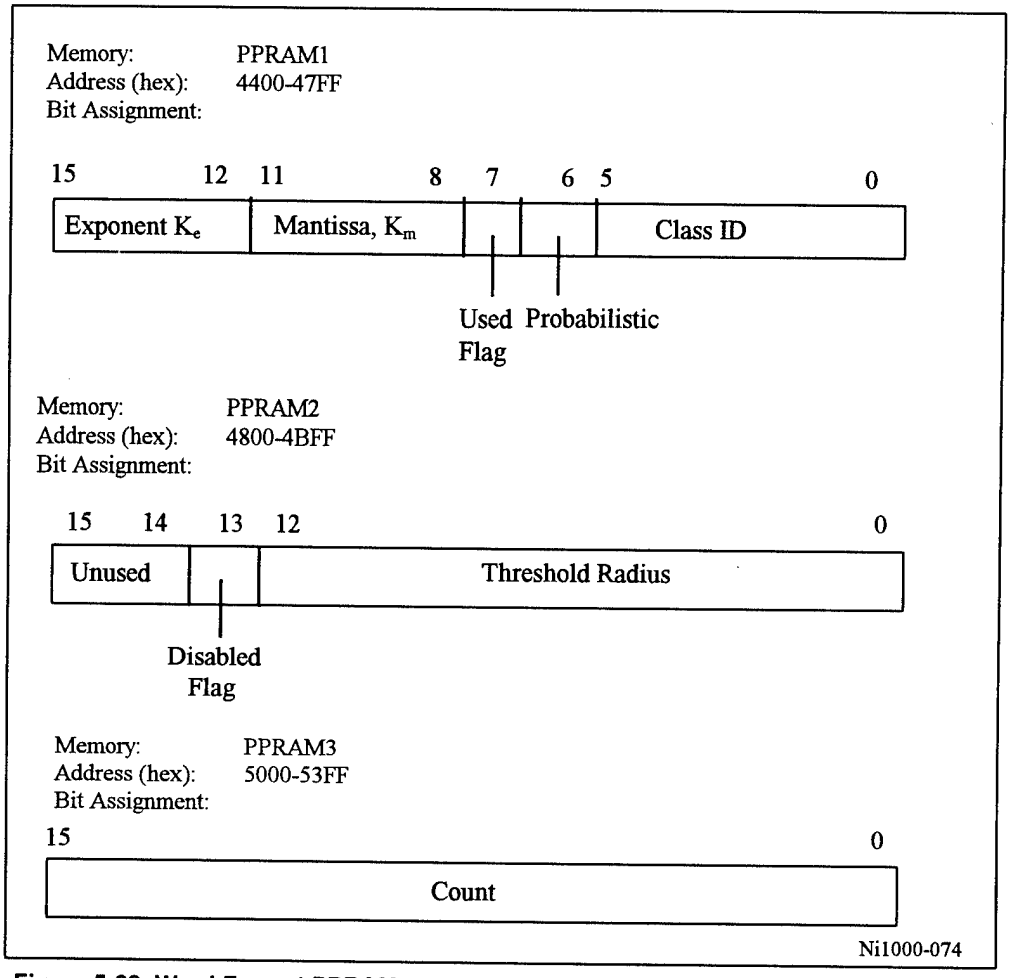


Figure 5-38. Word Format PPRAM

- *Count—C[0:15]*—the number of training vectors that fell within this prototype's influence field during the last epoch of learning; used as a factor during classification when calculating probability density.
- *Disable Flag—D*—set to disable this prototype.
- *Radius*—R*[1:12]*—the prototype threshold radius.
- *Smoothing Factor Mantissa—K_m[0:3]*—unsigned mantissa of the smoothing factor of the exponential function.
- *Smoothing Factor Exponent—K_e[4:7]*—signed exponent of the smoothing factor of the exponential function.
- *"Used" Flag—U*—set when the PPRAM word is loaded with a valid prototype.

- *Probabilistic—P*—indicates that the prototype's threshold radius is the minimum allowed. This bit is passed through to the class identification result to indicate that the first (highest number) prototype to fire for this class was probabilistic. If prototypes are ordered such that all deterministic prototypes have higher numbers than probabilistic prototypes for the same class, this bit flags that only probabilistic, not deterministic, classification is possible with this prototype.
- *Class—S[0:5]*—the class ID of the prototype.

The 4-bit signed exponent of the exponential function's smoothing factor is added to a built-in bias of negative 13 (i.e., 13 is subtracted from the stored value). For example, an exponent of 0 is really an exponent of -13. Since a value of -7 to +7 can be entered into this field, the effective exponent is -20 to -6. The 8-bit floating-point value for the smoothing factor has the following characteristics:

- The mantissa has only explicit bits, no implicit leading 1 as in the IEEE floating-point 32-bit format.
- The mantissa's binary point is to the right of the value (bbbb.).
- The exponent binary point is to the right, but the binary point is found moving the binary point left 13 places from the location indicated by the exponent. Thus, if all 4 bits of the exponent are zero, the mantissa is multiplied by 2^{-13}.
- Zero is represented by a zero mantissa, regardless of the exponent.
- The resulting number is always non-negative.
- The smallest non-zero value is represented by 1111 0001, or $2^{(-7-13)} * 1 = 2^{-20}$.
- The largest value is represented by 0111 1111, or $2^{(7-13)} * 15 = 2^{-6} * (2^4 - 1) = 2^{-2} - 2^{-6}$.
- This floating-point format is only used for the smoothing factor for PRCE and PNN calculations.

PPRAM entries correspond to the prototype vectors stored in the PA. For example, the parameters for prototype number 200h are stored in PPRAM at address 4400h+200h, 4800h+200h, and 5000h+200h.

The standard Microcontroller code that is shipped with the chip will copy the PPRAM *Used* flags to the PADCU. As a result, any prototype that has its *Used* flag set to 1 will be processed by the classifier. To avoid the possibility of uninitialized data being processed, all locations in the PPRAM should be written when they are loaded with new prototypes, and unused prototypes should have their *Used* flags cleared to 0.

PPRAMs have three operating modes, *Idle*, *Classify* and *Microcontroller*. These modes are set by the Microcontroller by writing to the control registers. When the Microcontroller writes to the global control register *PPRAM_CR,* the same value is written to the three individual control registers: *PPRAM1_CR, PPRAM2_CR* and *PPRAM3_CR.* The modes and the register settings under these modes are:

- *Idle*—This mode prohibits access to the PPRAM. It is entered by the Microcontroller by writing a value 0000h to the PPRAM_CR register.
- *Classify*—This mode is used to perform classification. Other logic blocks must also be in appropriate modes and certain pins must be set properly. The Microcontroller should not access the PPRAM during classification, since the data will be changing faster than the microcontroller can keep up with, resulting in unreliable access. This mode is entered by the Microcontroller by writing a value 4000h to the PPRAM_CR register.

- *Microcontroller*—This mode allows the Microcontroller to access the PPRAM. The mode is entered by the Microcontroller by writing a value 8000h to the PPRAM_CR register.

Memory: PPRAM_CR
Address (hex): 4381
Bit Assignment:

15	0
Register Value	

Memory: PPRAM1_CR
Address (hex): 4201
Bit Assignment:

15	0
Register Value	

Memory: PPRAM2_CR
Address (hex): 4101
Bit Assignment:

15	0
Register Value	

Memory: PPRAM3_CR
Address (hex): 4081
Bit Assignment:

15	0
Register Value	

Software-Controlled Mode:	Disabled	Classify	Microcontroller
Register Value (hex):	0000	4000	8000

Ni1000-075

Figure 5-39. PPRAM Registers

5.3.4. PPRAM Access

The host can only access the PPRAM through the Microcontroller by using the following instruction sequence:

5.3.4.1. PPRAM Read and Write by the Microcontroller

1. Write to the following register:

Register	Address (hex)	Data	Description
PPRAM_CR	4381	8000h	Set the PPRAM to *Microcontroller* mode.

2. For each prototype vector (with a column number in the range 0h-3FFh), read or write the following memory locations:

Memory	Address (hex)	Bit	Data
PPRAM1	4400 + column number	0:5	Class ID.
		6	Probabilistic flag.
		7	Used flag.
		8:15	Smoothing factor, K.
PPRAM2	4800 + column number	0:12	RBF radius.
		13	Disabled flag.
PPRAM3	5000 + column number	0:15	Count used in the PDF calculation.

5.3.5. MURAMs

The Math Unit RAMs and registers are shown in Table 5-12 and described in the following paragraphs.

Table 5-12. MURAMs and Registers

Address (Hex)	Name	Host W/R	MC W/R	Description
6080	MURAM1	-	R	Firing class count for MURAM1.
60C0	MURAM2	-	R	Firing class count for MURAM2.
6100	MURAM_CR	-	W/R	MU mode-control register.
6200-623F	Flag MURAM	-	R	64 flags, used to indicate the firing classes for the current MURAM. Only the LSB is used.
6400-643F	Firing Class List MURAM1	-	W/R	One of the two alternating 64x8-bit buffers, reserved for the class IDs of firing classes.
6440-647F	Firing Class List MURAM2	-	W/R	One of the two alternating 64x8-bit buffers, reserved for the class IDs of firing classes.
6800-683F	Probability MURAM1	-	W/R	One of the two 64x16-bit buffers, used to accumulate the probability densities of the input vector for each of the 64 classes.
6840-687F	Probability MURAM2	-	W/R	One of the two 64x16-bit buffers, used to accumulate the probability densities of the input vector for each of the 64 classes.

The MURAM memories include:

- *Flag MURAM*—a 1 x 64 memory used as a table of classes which have already recognized the input vector being presented. Each entry corresponds to one of the 64 possible classes. It occupies addresses 6200h through 623Fh.
- *Class List MURAMs*—an 8 x 64 x 2 double buffer holding a list of the class IDs of recognized classes. A new byte is allocated every time a new class is encountered. (Unlike the other MURAM memories, the memories in this buffer are not indexed by class ID; they are addressed by counters, so they grow up from address zero.) They occupy addresses 6400h through 647Fh.
- *Probability MURAMs*—a 16 x 64 x 2 double buffer which accumulates the probability value of the input vector for each class. As with the flag MURAM, each MURAM address corresponds to one of the 64 classes. They occupy addresses 6800h through 687Fh.

The flag MURAM is not accessible to the ORAM, so it is re-used every cycle. It is indexed by the class ID. When a class is recognized, the bit addressed by the class ID is set. If the bit previously was clear, that indicates the class had not yet been recognized during the processing of the input vector. This causes the class counter to be incremented and allocates a word in the class-list MURAM. The counter keeps a running tally of the number of classes, which is used to address the class-list MURAM when a new word is allocated.

The class-list MURAMs are 8 bits wide, consisting of a six-bit class ID and a bit to indicate that the first (highest numbered) prototype to recognize the input vector has a threshold radius equal to the minimum radius. If all prototypes with a threshold radius higher than the minimum are arranged so that they are at higher prototype numbers than those with the minimum radius, this bit can identify classifications that should be probabilistic. An eighth bit is undefined. Figure 5-40 shows the format of a byte in the firing-class-list MURAMs.

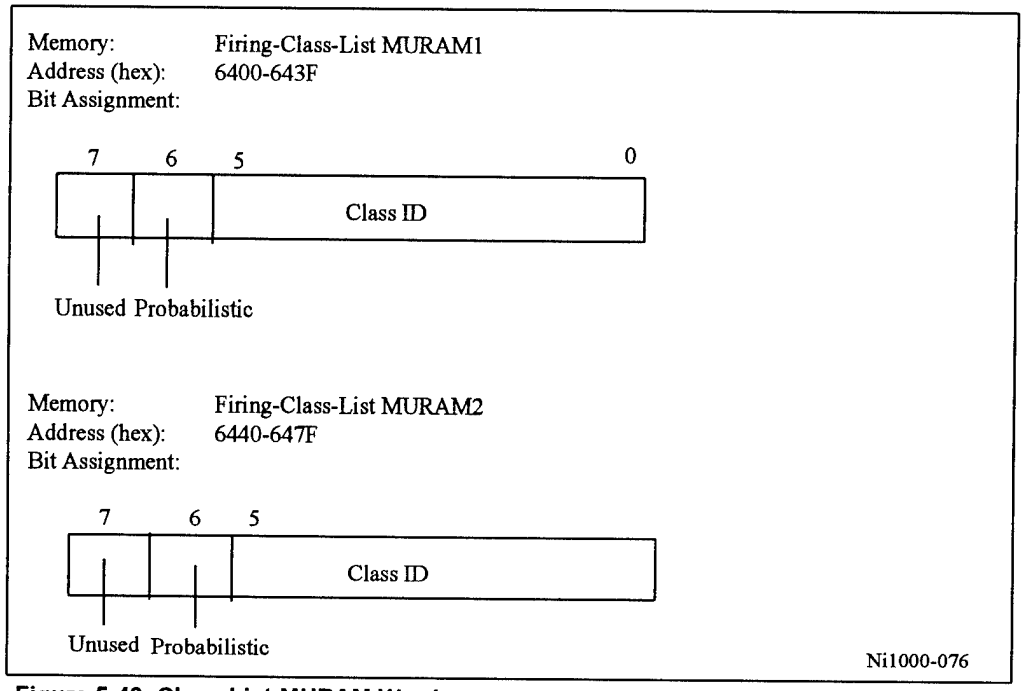


Figure 5-40. Class-List MURAM Word

The fields of a class-list MURAM word are:

- *Class ID S[5:0]*—Class ID of a class which includes the input vector.
- *Probabilistic*—a prototype with the minimum radius recognized the input vector. This indicates a reduced confidence in the ability to classify using deterministic classification and recommends the use of probabilistic classification. The *Probabilistic* bit is only meaningful if all *Probabilistic* prototypes of a given class are placed in lower numbered columns of the array than the deterministic prototypes for that class.
- *Valid*—this word has been written since reset initialization.

The class-list MURAMs are addressed by a counter. The counter begins at zero and increments as new classes are encountered.

The probability MURAMs consist of a 16-bit floating-point accumulator in the internal format of the Ni1000 Accelerator. The internal format is passed directly through ORAM or translated into an IEEE-compatible format as it passes through ORAM. Figure 5-41 shows the internal format of a word in the probability MURAMs.

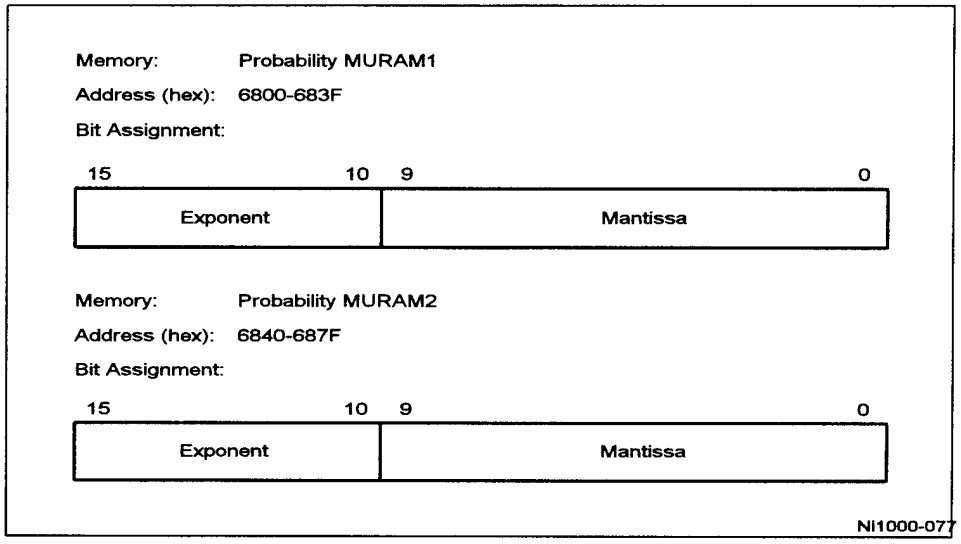


Figure 5-41. MURAM Probability Word

The fields of a probability MURAM word are:

- *Exponent*—six-bit 2's-complement exponent.
- *Mantissa*—10-bit fractional mantissa (i.e. 0 <= mantissa < 1).

Once the last probability calculation has been performed for the current input vector, the pairs of class-list and probability MURAMs are swapped. This allows the processing of a new vector to begin immediately, while the old vector is uploaded to the host. Both the class list and probability densities are computed and are available.

MURAMs have two operating modes: *Classify* and *Microcontroller*. A control register, *MURAM_CR,* is used to set the mode for all MURAMs. Its values under these modes are shown in Figure 5-36. Do not set the register to any other value.

- *Classify*—This mode is used to perform classification. Other logic block modes must also be set properly. See Section 5.3.6 for details. The microcontroller enters this mode by writing a 1 to bit 1 and a value in the range 0000b through 1000b to bits [2:5] of the MURAM_CR register.
- *Microcontroller*—This mode allows the Microcontroller to access the MURAMs. The mode is entered by the Microcontroller by writing a value 0000h to the MURAM_CR register.

Register: MURAM_CR
Address (hex): 6100
Bit Assignment:

15 … 6	5 … 2	1	0
	Exponent Offset		

Software-Controlled Mode:	Classify	Microcontroller
Bit 1	1	0
Bits [2-5]	0000-1000	0000

Ni1000-078

Figure 5-42. The MURAM_CR Register

The Math Unit smoothing function is:

$$2^{-K_m \cdot d \cdot 2^{(K_e + MURAM_CR[2:5]-13)}}$$

$$0 \le K_m \le 15$$

where,

K_m = unsigned mantissa of the smoothing factor, K, obtained from PPRAM,
K_e = signed exponent of the smoothing factor, K, obtained from PPRAM,
d = the calculated city-block distance,

Bits [2:5] of the MURAM_CR register specify the offset in the exponent of the Math Unit smoothing function.

5.3.6. Classification

1. Write to the following memory locations and registers:

Memory or Register	Address (hex)	Bit	Data	Description
MURAM	6400 - 647F, 6800 - 687F	N/A	0000h	Clear MURAM.
DIM	0008	0:7	input vector features - 1	the number of features for input vectors minus 1.
CRB	0048	0:15	0	Clear CRB.

2. Load PPRAM. See Section 5.3.4.

3. Write to the following registers:

Register	Address (hex)	Bit	Data	Description
PPRAM_CR	4381	N/A	0002h	Set PPRAM to *Classify* mode.
MURAM_CR	6100	1 2:5	1 0000h - 1000h	Set MURAM to *Classify* mode. Specify an offset to the exponent of the smoothing factor.

4. Set, clear or read the DCU *Used* flags by writing to the following registers and memory locations in order, where "p" is the prototype number of each vector to be used in classification. "p >> 1" means to shift p one bit to the right:

Memory or Register	Address (hex)	Data (hex)	Description
CSA	3001	8000	Set PA to *Microcontroller* mode.
CSB	3040	8000 or 8800	Write 8000 to set PA to *Microcontroller* mode. Write 8800 to read the Used flags.
AUX	3020	0000	Initialize the AUX register.
MODE	3002	0000	Initialize the MODE register.
Memory	B800 | (p >> 1)	0000	Select PA row.
Memory	B000 | p	0000	Select PA column.
MODE	3002	0020 or 0040 or 0080	Write 0020h to clear flags. Write 0040h to set flags. Write 0080h to read flags.
MODE	3002	0000	Reinitialize the MODE register.

Repeat the last four writes for another prototype.

5. Write to the following registers:

Register	Address (hex)	Data (hex)	Description
AUX	3020	0000	Set PA to *Classify* mode.
MODE	3002	0100	Set PA to *Classify* mode.
DCU_DIM	3004	input vector features - 1	Specify a window in PA. See Section 5.3.1.
MURAM_CR	6100	00xx	Set bits [2:5] to an appropriate value. See Section 5.3.5.
NCA	3008	number of used prototype vectors	Specify a window in PA. See Section 5.3.2.
NCB	3010	0008	Specify clock count for MU.
ARR	3200	column and row offset, relocation used bits	Specify the starting position of the PA block in use. See Section 5.3.2.
CSA	3001	6C00	Set PA to *Classify* mode.
CSB	3040	6000	Set PA to *Classify* mode.

6. Write to the following I/O registers in order:

Register	Address (hex)	Data (hex)	Description
CRB	0048	0070	Unreset IRAM, ORAM. Set IRAM, ORAM to *Classify* mode. PRCE results are in 32-bit IEEE format.
CRA	0040	0001	Select firing-class IDs as outputs.

7. Write an input vector to IRAM. See instruction sequence *IRAM Write by the Host* in Section 5.1.4, step 2.

8. Read RCE results (firing class IDs) from ORAM when ORAM is full, as indicated by HS2[14] = 1. Continue reading until ORAM is empty, as indicated by HS2[10] = 1.

9. Write to the following I/O register:

Register	Address (hex)	Data (hex)	Description
CRA	0040	0003	Unreset ORAM. Set ORAM to *Classify* mode. Select probabilities as outputs.

10. Read PRCE results (probabilities) from ORAM when ORAM is full, as indicated by HS2[14] = 1. Continue reading until ORAM is empty, as indicated by HS2[10] = 1.

11. Write to the following I/O register:

Register	Address (hex)	Data (hex)	Description
CRA	0040	0001	Unreset ORAM. Set ORAM to *Classify* mode. Select firing class IDs as outputs.

12. Repeat from step 7 if there are more vectors to classify. Otherwise, proceed to step 13.

13. Exit from the classification by writing to the following registers:

Register	Address (hex)	Data (hex)	Description
CSB	3040	8000	Set PA to *Microcontroller* mode.
CSA	3001	8000	Set PA to *Microcontroller* mode.

Writing to the above registers stops the classifier by taking PADCU out of *Classify* mode. Other logic blocks, such as IRAM, ORAM, PPRAM and MURAM are still in *Classify* mode. If so desired, individual control registers for each logic block must be written with proper values to bring the block out of *Classify* mode. See Sections 5.1.3, 5.1.4, 5.1.7 and 5.1.8 for details.

5.3.7. Learning

The instruction sequence for a learning process depends on the particular algorithm implemented. For user-defined algorithms, it is the user's responsibility to define such sequences.

6. NI1000 MICROCONTROLLER ARCHITECTURE

6.1. Introduction

Most users of the Ni1000 do not have to program the microcontroller. Instead, the standard microcontroller program (a.k.a. standard microcode) that is shipped with the chip for learning, classification and host interface protocol provides all of the functions required for most applications. See Chapter 7, *Standard Microcontroller Software*.

The Ni1000 microcontroller has a Harvard architecture (separate data and code storage), provides 6 address modes and 6 groups of instructions. The entire instruction set is explained in this section after a brief description of the microcontroller registers, flags and addressing modes. A summary table cross-referencing the flags affected by each instruction is presented at the end of this section.

6.2. Microcontroller Registers and Flags

There are 11 microcontroller registers, listed in Table 6-1. A summary table cross-referencing the flags affected by each instruction is presented at the end of this section.

Table 6-1. Microcontroller Registers

Register	Size	Type	Code (hex)	Description
R0	16 bits	W/R	0	General purpose register 0.
R1	16 bits	W/R	1	General purpose register 1.
R2	16 bits	W/R	2	General purpose register 2.
R3	16 bits	W/R	3	General purpose register 3.
ZERO	16 bits	R	6	Always reads 0.
ONE	16 bits	R	7	Always reads 1.
DS1	16 bits	W/R	8	Data segment register 1.
DS2	16 bits	W/R	9	Data segment register 2.
SP	16 bits	W/R	A	Stack Pointer.
PC	12 bits	W/R	-	Program Counter.
CSW	15 bits	W/R	-	Control Status Word.

The stack is 64 locations deep. SP is unsigned, starting with value 0, and points to the location after the current one. The first stack address is 0, so only 63 stack locations are available.

The CSW register, shown in Table 6-1, is broken into the flags shown in Table 6-2. If the stack overflows or underflows, the SE and IR flags will be set. If the IE flag is also set, an interrupt will occur. The interrupt will jump to the interrupt service routine whose address is stored in location F000*h* in PGFLASH.

WARNING: One-word memory access instructions must not follow any other memory reference instruction. They need to be separated by other instructions, such as NOOPs.

Table 6-2. Microcontroller Flags

CSW Bit Number	Name	Abbreviation	Code ($flg)
0	Carry	C	0000
1	Zero	Z	0001
2	Negative	N	0010
3	Positive	P	0011
4	Overflow	O	0100
5	Interrupt Request	IR	0101
6	Interrupt Enable	IE	0110
7	Stack Error	SE	0111
8	General Error	GE	1000
9	Multi-Class Firing	MC	1001
10	Flash-Write	FW	1010
11	MURAM1 Ready	M1	1011
12	MURAM2 Ready	M2	1100
13	PADCU Busy	DC	1101

6.3. Addressing Modes

The read and write instructions support six addressing modes:

- *Indirect Off DS with 8-bit Offset*—the address is the sum of an 8-bit field in the instruction and either the DS1 or DS2 registers.
- *Indirect*—the address is in R0, R1, R2, or R3.
- *Indirect with Register Offset*—the address is the sum of R0, R1, R2, or R3 and DS1 or DS2.
- *Indirect Autoincrement with Register Offset*—the address is the sum of R0, R1, R2, or R3 and DS1 or DS2. The general register is incremented after the operation.
- *Indirect Autodecrement with Register Offset*—the address is the sum of R0, R1, R2, or R3 and DS1 or DS2. The general register is decremented before the operation.

The jump instructions have the following addressing modes:

- *Immediate*—the address is coded into the instruction.
- *Register*—the value stored in the named register is placed in PC.
- *Relative*—the value, either the named address offset or the value stored at the named register, is added to PC.

6.4. Instruction Summary (By Functional Group)

This section groups all the instructions according to their functions. The mnemonics and English descriptions are given.

6.4.1. Conditional Jumps

Flag Condition	*Short Immediate Relative*	*Register*	*Register Relative*	*Long Immediate*	*Long Immediate Relative*
Unconditional	JMP	JMPR	JMPRR	JMPI	JMPIR
Carry	JC	JCR	JCRR	JCI	JCIR
Zero	JZ	JZR	JZRR	JZI	JZIR
Negative	JN	JNR	JNRR	JNI	JNRI
Positive	JP	JPR	JPRR	JPI	JPIR
Overflow	JO	JOR	JORR	JOI	JOIR
Interrupt Request	JIR	JIRR	JIRRR	JIRI	JIRIR
Interrupt Enable	JIE	JIER	JIERR	JIEI	JIEIR
Stack Error	JSE	JSER	JSERR	JSEI	JSEIR
General Error	JGE	JGER	JGERR	JGEI	JGEIR
Multi-Class Firing	JMC	JMCR	JMCRR	JMCI	JMCIR
Flash Write	JFW	JFWR	JFWRR	JFWI	JFWIR
MURAM 1 Ready	JM1	JM1R	JM1RR	JM1I	JM1IR
MURAM 2 Ready	JM2	JM2R	JM2RR	JM2I	JM2IR
PADCUs Busy	JDC	JDCR	JDCRR	JDCI	JDCIR
No Carry	JNC	JNCR	JNCRR	JNCI	JNCIR
No Zero	JNZ	JNZR	JNZRR	JNZI	JNZIR
No Negative	JNN	JNNR	JNNRR	JNNI	JNNIR
No Positive	JNP	JNPR	JNPRR	JNPI	JNPIR
No Overflow	JNO	JNOR	JNORR	JNOI	JNOIR
No Interrupt Request	JNIR	JNIRR	JNIRRR	JNIRI	JNIRIR
No Interrupt Enable	JNIE	JNIER	JNIERR	JNIEI	JNIEIR
No Stack Error	JNSE	JNSER	JNSERR	JNSEI	JNSEIR
No General Error	JNGE	JNGER	JNGERR	JNGEI	JNGEIR
No Multi-Class Firing	JNMC	JNMCR	JNMCRR	JNMCI	JNMCIR
No Flash Write	JNFW	JNFWR	JNFWRR	JNFWI	JNFWIR
No MURAM 1 Ready	JM1	JM1R	JM1RR	JM1I	JM1IR
No MURAM 2 Ready	JM2	JM2R	JM2RR	JM2I	JM2IR
No PADCUs Busy	JDC	JDCR	JDCRR	JDCI	JDCIR

Note: JNRI does not follow the normal naming convention due to a conflict with JNIR.

6.4.2. Subroutine Calls

JS	Jump to Subroutine
JSR	Jump to Subroutine Register
JSRR	Jump to Subroutine Register Relative
JSI	Jump to Subroutine Immediate
JSIR	Jump to Subroutine Immediate Relative
RETS	Return from Subroutine

6.4.3. Stack Operations

PUSH	Push
POP	Pop

6.4.4. Flag Operations

RDFLG	Read Flags
WDFLG	Write Flags
SFLGC	Set Carry Flag
SFLGZ	Set Zero Flag
SFLGN	Set Negative Flag
SFLGP	Set Positive Flag
SFLGO	Set Overflow Flag
SFLGIR	Set Interrupt Request Flag
SFLGIE	Set Interrupt Enable Flag
SFLGSE	Set Stack Error Flag
SFLGGE	Set General Error Flag
SFLGMC	Set Multi-Class Firing Flag
SFLGFW	Set Flash Write Flag
SFLGM1	Set MURAM 1 Ready Flag
SFLGM2	Set MURAM 2 Ready Flag
SFLGDC	Set PADCUs Busy Flag
CFLGC	Clear Carry Flag
CFLGZ	Clear Zero Flag
CFLGN	Clear Negative Flag
CFLGP	Clear Positive Flag
CFLGO	Clear Overflow Flag
CFLGIR	Clear Interrupt Request Flag
CFLGIE	Clear Interrupt Enable Flag
CFLGSE	Clear Stack Error Flag
CFLGGE	Clear General Error Flag
CFLGMC	Clear Multi-Class Firing Flag
CFLGFW	Clear Flash Write Flag
CFLGM1	Clear MURAM 1 Ready Flag
CFLGM2	Clear MURAM 2 Ready Flag
CFLGDC	Clear PADCUs Busy Flag
WAIT	Equivalent to WAITIR
WAITC	Wait For Carry Flag
WAITZ	Wait For Zero Flag

WAITN	Wait For Negative Flag
WAITP	Wait For Positive Flag
WAITO	Wait For Overflow Flag
WAITIR	Wait For Interrupt Request Flag
WAITIE	Wait For Interrupt Enable Flag
WAITSE	Wait For Stack Error Flag
WAITGE	Wait For General Error Flag
WAITMC	Wait For Multi-Class Firing Flag
WAITFW	Wait For Flash Write Flag
WAITM1	Wait For MURAM 1 Ready Flag
WAITM2	Wait For MURAM 2 Ready Flag
WAITDC	Wait For PADCUs Busy Flag

6.4.5. Data Transfer Operations.

MOV	Move Register to Register
LDI	Load Immediate
RDI	Read Immediate
RDR	Read Indirect Register
RD1	Read Indexed Base Register using DS1 as base, 8 bit immediate as offset.
RD2	Read Indexed Base Register using DS2 as base, 8 bit immediate as offset.
RD1R	Read Indexed Base Register using DS1 as base, named register for offset.
RD2R	Read Indexed Base Register using DS2 as base, named register for offset.
RD1RI	Read Indexed Base Register using DS1 as base, named register for offset.
RD2RI	Read Indexed Base Register using DS2 as base, named register for offset.
RD1RD	Read Indexed Base Register using DS1 as base, named register for offset.
RD2RD	Read Indexed Base Register using DS2 as base, named register for offset.
WRI	Write Immediate
WRRI	Write Indirect Register Immediate
WRR	Write Indirect Register
WR1	Write Indexed Base Register using DS1 as base, 8-bit immediate as offset.
WR2	Write Indexed Base Register using DS2 as base, 8-bit immediate as offset.
WR1R	Write Indexed Base Register using DS1 as base, named register for offset.
WR2R	Write Indexed Base Register using DS2 as base, named register for offset.
WR1RI	Write Indexed Base Register using DS1 as base, named register for offset.
WR2RI	Write Indexed Base Register using DS2 as base, named register for offset.
WR1RD	Write Indexed Base Register using DS1 as base, named register for offset.
WR2RD	Write Indexed Base Register using DS2 as base, named register for offset.

6.4.6. Arithmetic and Logical Operations.

ADD	Add
ADC	Add with Carry
SUB	Subtract
CMP	Compare
INC	Increment
DEC	Decrement
NOT	Logical Negation
OR	Logical Or
XOR	Logical Exclusive Or
SHL	Shift Register Left
SHR	Shift Register Right
ROTL	Rotate Register Left
ROTR	Rotate Register Right
NOOP	No Operation

6.5. Instruction Set (Alphabetical)

The instructions of the microcontroller take 0, 1, or 2 arguments. Those instructions that take two arguments use the following syntax, so do the machine instructions after assembling:

```
operation source, destination
```

The following notation is used in the opcodes:

- aa—8-bit address offset, in hexadecimal.
- r—4-bit code for registers R0-R3, Zero, One (see Table 6-1)
- s—4-bit code for registers R0-R3, Zero, One, DS1, DS2 (see Table 6-1)

6.5.1.1.1.ADC Add with Carry

Assembler Syntax:	ADC *register, register*
Opcode:	33rr
Argument:	R0, R1, R2, R3, Zero, or One
Words:	1
Clocks:	2
Flags Affected:	Carry, Zero, Negative, Positive, Overflow
Description:	Add the contents of two registers and the carry bit, and place the sum in the second-named register. This is used to implement 32-bit arithmetic.
Example:	`ADC R2, R1`
Equivalent C code:	`R1 += R2 + Carry flag;`

6.5.1.1.2.ADD Add

Assembler Syntax:	ADD *register, register*
Opcode:	31rr
Argument:	R0, R1, R2, R3, Zero, or One
Words:	1
Clocks:	2
Flags Affected:	Carry, Zero, Negative, Positive, Overflow
Description:	Add the contents of one register to another. Place the sum in the second-named register.
Example:	`ADD R1, R2`
Equivalent C code:	`R2 += R1;`

6.5.1.1.3.AND Logical AND

Assembler Syntax:	AND *register, register*
Opcode:	34rr
Argument:	R0, R1, R2, R3, Zero, or One
Words:	1
Clocks:	2
Flags Affected:	Carry, Zero, Negative, Positive, Overflow
Description:	And two registers. Place the results in the second register.
Example:	`AND R0, R1`
Equivalent C code:	`R1 &= R0;`

6.5.1.1.4.CFLG[**] Clear Flag

Assembler Syntax:	CFLG[**]
Opcode:	See table below
Argument	None
Words	1
Clocks	2
Flags Affected	Specified flag (cleared)

Description
Sets the specified flag to 0.

Example	CFLGZ
Equivalent C code	CSW &= !0x0002; /* clear zero flag */

mnemonic	*Opcode*	*description*
CFLGC	0012	Clear Carry Flag
CFLGZ	0112	Clear Zero Flag
CFLGN	0212	Clear Negative Flag
CFLGP	0312	Clear Positive Flag
CFLGO	0412	Clear Overflow Flag
CFLGIR	0512	Clear Interrupt Request Flag
CFLGIE	0612	Clear Interrupt Enable Flag
CFLGSE	0712	Clear Stack Error Flag
CFLGGE	0812	Clear General Error Flag
CFLGMC	0912	Clear Multi-Class Firing Flag
CFLGFW	0A12	Clear Flash Write Flag
CFLGM1	0B12	Clear MURAM 1 Ready Flag
CFLGM2	0C12	Clear MURAM 2 Ready Flag
CFLGDC	0D12	Clear PADCUs Busy Flag

6.5.1.1.5.CMP Compare

Assembler Syntax:	CMP *subtrahend minuend*
Opcode:	39rr
Argument:	R0, R1, R2, R3, Zero, One
Words:	1
Clocks:	2
Flags Affected:	Carry, Zero, Negative, Positive, Overflow
Description:	Subtract the contents of the first-named register from the contents of the second-named register and set the appropriate flags. The remainder is not stored.
Example:	`CMP R1, R2`
Equivalent C code:	`(R2-R1)`

6.5.1.1.6.DEC Decrement

Assembler Syntax:	DEC *register*
Opcode:	28r-
Argument:	R0, R1, R2, R3, Zero, or One
Words:	1
Clocks:	2
Flags Affected:	Carry, Zero, Negative, Positive, Overflow
Description:	Subtract one from the named register.
Example:	`DEC R1`
Equivalent C code:	`R1--;`

6.5.1.1.7.INC Increment

Assembler Syntax:	INC *register*
Opcode:	32-r
Argument:	R0, R1, R2, R3, Zero, or One
Words:	1
Clocks:	2
Flags Affected:	Carry, Zero, Negative, Positive, Overflow
Description:	Add one to the named register.
Example:	`INC R1`
Equivalent C code:	`R1++;`

6.5.1.1.8.J[***] Jump on Condition

Assembler Syntax: J[***] *offset*

Opcode: See table below
Argument: [offset] value in memory or register to be added to PC
Words: 1
Clocks: 5
Flags Affected: Carry, Zero, Negative, Positive, Overflow

Description: This set of jump instructions does a relative jump of up to ±127 words. To jump more than 127 words, use a jump immediate instruction, described below.

Example:
```
JC 2Ah
```
Equivalent C code:
```
if (Carry) PC += 0x2A;
```

mnemonic	*opcode*	*condition of jump*	*mnemonic*	*opcode*	*condition of jump*
JMP	9Faa	unconditional			
JC	90aa	Carry	JNC	80aa	No Carry
JZ	91aa	Zero	JNZ	81aa	No Zero
JN	92aa	Negative	JNN	82aa	No Negative
JP	93aa	Positive	JNP	83aa	No Positive
JO	94aa	Overflow	JNO	84aa	No Overflow
JIR	95aa	Interrupt Request	JNIR	85aa	No Interrupt Request
JIE	96aa	Interrupt Enable	JNIE	86aa	No Interrupt Enable
JSE	97aa	Stack Error	JNSE	87aa	No Stack Error
JGE	98aa	General Error	JNGE	88aa	No General Error
JMC	99aa	Multi-Class Firing	JNMC	89aa	No Multi-Class Firing
JFW	9Aaa	Flash Write	JNFW	8Aaa	No Flash Write
JM1	9Baa	MURAM 1 Ready	JNM1	8Baa	No MURAM 1 Ready
JM2	9Caa	MURAM 2 Ready	JNM2	8Caa	No MURAM 2 Ready
JDC	9Daa	PADCUs Busy	JNDC	8Daa	No PADCUs Busy

6.5.1.1.9.J[***]I Jump on Condition to Immediate Address

Assembler Syntax: J[***]I *address*

Opcode: See table below
Argument: Address
Words: 2
Clocks: 6
Flags Affected: None

Description: Jumps to the address specified in the instruction. The jump can be to any place within the PGFLASH.

Example: `JNOI 2B11h`
Equivalent C code: `if (! Overflow) PC = 0x2B11;`

mnemonic	*Opcode*	*condition of jump*	*mnemonic*	*opcode*	*condition of jump*
JMPI	DF61	Unconditional			
JCI	D061	Carry	JNCI	C061	No Carry
JZI	D161	Zero	JNZI	C161	No Zero
JNI	D261	Negative	JNNI	C261	No Negative
JPI	D361	Positive	JNPI	C361	No Positive
JOI	D461	Overflow	JNOI	C461	No Overflow
JIRI	D561	Interrupt Request	JNIRI	C561	No Interrupt Request
JIEI	D661	Interrupt Enable	JNIEI	C661	No Interrupt Enable
JSEI	D761	Stack Error	JNSEI	C761	No Stack Error
JGEI	D861	General Error	JNGEI	C861	No General Error
JMCI	D961	Multi-Class Firing	JNMCI	C961	No Multi-Class Firing
JFWI	DA61	Flash Write	JNFWI	CA61	No Flash Write
JM1I	DB61	MURAM 1 Ready	JNM1I	CB61	No MURAM 1 Ready
JM2I	DC61	MURAM 2 Ready	JNM2I	CC61	No MURAM 2 Ready
JDCI	DD61	PADCUs Busy	JNDCI	CD61	No PADCUs Busy

6.5.1.1.10.J[***]IR Jump on Condition to Immediate Relative

Assembler Syntax: J[***]IR *offset*

Opcode: See table below
Argument: Offset
Words: 2
Clocks: 6
Flags Affected: Carry, Zero, Negative, Positive, Overflow

Description: Performs a relative jump to the address specified in the instruction; the address specified will be added to the PC. The jump can be to any place within the PGFLASH.

Example: `JSEIR 1234h`
Equivalent C code: `if (Stack Error) PC += 0x1234;`

mnemonic	*opcode*	*condition of jump*	*mnemonic*	*opcode*	*condition of jump*
JMPIR	DF61	Unconditional			
JCIR	D0C1	Carry	JNCIR	C0C1	No Carry
JZIR	D1C1	Zero	JNZIR	C1C1	No Zero
JNRI	D2C1	Negative	JNNIR	C2C1	No Negative
JPIR	D3C1	Positive	JNPIR	C3C1	No Positive
JOIR	D4C1	Overflow	JNOIR	C4C1	No Overflow
JIRIR	D5C1	Interrupt Request	JNIRIR	C5C1	No Interrupt Request
JIEIR	D6C1	Interrupt Enable	JNIEIR	C6C1	No Interrupt Enable
JSEIR	D7C1	Stack Error	JNSEIR	C7C1	No Stack Error
JGEIR	D8C1	General Error	JNGEIR	C8C1	No General Error
JMCIR	D9C1	Multi-Class Firing	JNMCIR	C9C1	No Multi-Class Firing
JFWIR	DAC1	Flash Write	JNFWIR	CAC1	No Flash Write
JM1IR	DBC1	MURAM 1 Ready	JNM1IR	CBC1	No MURAM 1 Ready
JM2IR	DCC1	MURAM 2 Ready	JNM2IR	CCC1	No MURAM 2 Ready
JDCIR	DDC1	PADCUs Busy	JNDCIR	CDC1	No PADCUs Busy

6.5.1.1.11.J[***]R Jump on Condition to Register

Assembler Syntax: J[***]R *register*

Opcode: See table below
Argument: R0, R1, R2, R3, Zero, or One
Words: 1
Clocks: 5
Flags Affected: None

Description: Sets the program counter to the address contained in one of the following registers: R0, R1, R2, R3, Zero, One.

Example: `JNM1R R0`
Equivalent C code: `if (No MURAM 1 Ready) PC = R0;`

mnemonic	*opcode*	*condition of jump*	*mnemonic*	*opcode*	*condition of jump*
JMPR	BF6r	Unconditional			
JCR	B06r	Carry	JNCR	A06r	No Carry
JZR	B16r	Zero	JNZR	A16r	No Zero
JNR	B26r	Negative	JNNR	A26r	No Negative
JPR	B36r	Positive	JNPR	A36r	No Positive
JOR	B46r	Overflow	JNOR	A46r	No Overflow
JIRR	B56r	Interrupt Request	JNIRR	A56r	No Interrupt Request
JIER	B66r	Interrupt Enable	JNIER	A66r	No Interrupt Enable
JSER	B76r	Stack Error	JNSER	A76r	No Stack Error
JGER	B86r	General Error	JNGER	A86r	No General Error
JMCR	B96r	Multi-Class Firing	JNMCR	A96r	No Multi-Class Firing
JFWR	BA6r	Flash Write	JNFWR	AA6r	No Flash Write
JM1R	BB6r	MURAM 1 Ready	JNM1R	AB6r	No MURAM 1 Ready
JM2R	BC6r	MURAM 2 Ready	JNM2R	AC6r	No MURAM 2 Ready
JDCR	BD6r	PADCUs Busy	JNDCR	AD6r	No PADCUs Busy

6.5.1.1.12.J[***]RR Relative — Jump on Condition to Register

Assembler Syntax: J[***]RR *register*

Opcode: See table below
Argument: R0, R1, R2, R3, Zero, One
Words: 1
Clocks: 5
Flags Affected: Carry, Zero, Negative, Positive, Overflow

Description: Performs a relative jump to the address contained in a register - that is, the contents of the register will be added to the PC. The register can be R0, R1, R2, R3, Zero, One.

Example:
```
JIERR R0
```
Equivalent C code:
```
if (Interrupt Enable) PC += R0;
```

mnemonic	*opcode*	*condition of jump*	*mnemonic*	*opcode*	*condition of jump*
JMPRR	BFCr	Unconditional			
JCRR	B0Cr	Carry	JNCRR	A0Cr	No Carry
JZRR	B1Cr	Zero	JNZRR	A1Cr	No Zero
JNRR	B2Cr	Negative	JNNRR	A2Cr	No Negative
JPRR	B3Cr	Positive	JNPRR	A3Cr	No Positive
JORR	B4Cr	Overflow	JNORR	A4Cr	No Overflow
JIRRR	B5Cr	Interrupt Request	JNIRRR	A5Cr	No Interrupt Request
JIERR	B6Cr	Interrupt Enable	JNIERR	A6Cr	No Interrupt Enable
JSERR	B7Cr	Stack Error	JNSERR	A7Cr	No Stack Error
JGERR	B8Cr	General Error	JNGERR	A8Cr	No General Error
JMCRR	B9Cr	Multi-Class Firing	JNMCRR	A9Cr	No Multi-Class Firing
JFWRR	BACr	Flash Write	JNFWRR	AACr	No Flash Write
JM1RR	BBCr	MURAM 1 Ready	JNM1RR	ABCr	No MURAM 1 Ready
JM2RR	BCCr	MURAM 2 Ready	JNM2RR	ACCr	No MURAM 2 Ready
JDCRR	BDCr	PADCUs Busy	JNDCRR	ADCr	No PADCUs Busy

6.5.1.1.13.JS Jump to Subroutine

Assembler Syntax: JS *subroutine*

Opcode: E0aa
Argument: Subroutine name or memory location
Words: 1
Clocks: 6
Flags Affected: None

Description: This is the basic subroutine call mechanism. The return address (in this case, the current program counter) is pushed onto the stack, then the jump to the subroutine is executed. After completion of the subroutine, the PC is taken from the stack. The jump can be up to ±127 words. Use JSI for longer jumps.

Example:

```
JS CheckForErrors
```

Equivalent C code:

```
Stack[++SP] = PC;
PC += (offset to CheckForErrors);
```

6.5.1.1.14.JSI Jump to Subroutine Immediate

Assembler Syntax:	JSI *subroutine*
Opcode:	E860
Argument:	Subroutine or memory location
Words:	2
Clocks:	8
Flags Affected:	None
Description:	The PC is pushed onto the stack, then the jump is executed. The immediate address will be loaded into the PC. The subroutine may be anywhere within the PGFLASH.
Example:	`JSI CheckMathUnit`
Equivalent C code:	`Stack[++SP] = PC; PC = &CheckPathUnit;`

6.5.1.1.15.JSIR Jump to Subroutine Immediate Relative

Assembler Syntax:	JSIR *offset*
Opcode:	E8C0
Argument:	Address offset
Words:	2
Clocks:	8
Flags Affected:	Carry, Zero, Negative, Positive, Overflow
Description:	The PC is pushed onto the stack, then the jump is executed. The immediate offset will be added to the PC. The subroutine may be anywhere within the PGFLASH.
Example:	`JSIR 1234h`
Equivalent C code:	`Stack[++SP] = PC; PC += 0x1234;`

6.5.1.1.16.JSR Jump to Subroutine Register

Assembler Syntax:	JSR *register*
Opcode:	E46r
Argument:	R0, R1, R2, R3, Zero, One
Words:	1
Clocks:	6
Flags Affected:	None
Description:	The PC is pushed onto the stack, then the jump is executed. The named register contains the address of the subroutine.
Example:	`JSR R0`
Equivalent C code:	`Stack[++SP] = PC; PC = R0;`

6.5.1.1.17.JSRR Jump to Subroutine Register Relative

Assembler Syntax:	JSRR *register*
Opcode:	E4Cr
Argument:	R0, R1, R2, R3, Zero, or One
Words:	1
Clocks:	6
Flags Affected:	Carry, Zero, Negative, Positive, Overflow
Description:	The PC is pushed onto the stack, then the jump is executed. The contents of the named register will be added to the PC.
Example:	`JSRR R1`
Equivalent C code:	`Stack[++SP] = PC; PC += R1;`

6.5.1.1.18.LDI Load Immediate

Assembler Syntax:	LDI *data, register*
Opcode:	0r14
Arguments:	[data] hexadecimal value or symbol register R0, R1, R2, R3, Zero, One, DS1, DS2, or SP
Words:	2
Clocks:	4
Flags Affected:	None
Description:	Copy immediate data to register
Example:	`LDI 1234h, R0`
Equivalent C code:	`R0 = 0x1234;`

6.5.1.1.19.MOV Move Register to Register

Assembler Syntax:	MOV *source register, destination register*
Opcode:	12rr
Arguments:	R0, R1, R2, R3, Zero, One, DS1, DS2, or SP
Words:	1
Clocks:	2
Flags Affected:	None
Description:	Copy register to register.
Example:	`MOV R0, R1`
Equivalent C code:	`R1 = R0;`

6.5.1.1.20.NOOP No Operation

Assembler Syntax:	NOOP
Opcode:	0000
Argument:	None
Words:	1
Clocks:	2
Flags Affected:	None
Description:	The NOOPinstruction performs no operation.
Example:	`NOOP`
Equivalent C code:	`None`

6.5.1.1.21.NOT Logical Negation

Assembler Syntax:	NOT *register*
Opcode:	26r-
Argument:	R0, R1, R2, R3, Zero, or One
Words:	1
Clocks:	2
Flags Affected:	Carry, Zero, Negative, Positive, Overflow
Description:	Negate a register.
Example:	`NOT R0`
Equivalent C code:	`R0 = !R0;`

6.5.1.1.22.OR Logical OR

Assembler Syntax:	OR *register, register*
Opcode:	35rr
Argument:	R0, R1, R2, R3, Zero, or One
Words:	1
Clocks:	2
Flags Affected:	Carry, Zero, Negative, Positive, Overflow
Description:	Or two registers. Place the results in the second register.
Example:	`OR R0, R1`
Equivalent C code:	`R1 |= R0;`

6.5.1.1.23.POP Pop Register off Stack

Assembler Syntax:	POP *register*
Opcode:	0r04
Argument:	Register R0, R1, R2, R3, Zero, One, or SP
Words:	1
Clocks:	2
Flags Affected:	None
Description:	Pop a register off of the stack.
Example:	`POP R0`
Equivalent C code:	`R0 = STACK[SP--];`

6.5.1.1.24.PUSH Push Register onto Stack

Assembler Syntax:	PUSH *register*
Opcode:	0r02
Argument:	Register R0, R1, R2, R3, Zero, One, or SP
Words:	1
Clocks:	2
Flags Affected:	None
Description:	Push a register onto the stack.
Example:	`PUSH R0`
Equivalent C code:	`STACK[++SP] = R0;`

6.5.1.1.25.RD[***] Read Indexed Base Register

Assembler Syntax: RD[***] *data*
RD[***] *register*

Opcode: See table below
Argument: [data] hexadecimal value or symbol
register R0, R1, R2, R3, Zero, or One
Words: 1
Clocks: 4
Flags Affected: Carry, Zero, Negative, Positive, Overflow

Description: Copy from memory address indexed from base register to register. The base address is stored in either DS1 or DS2. The offset is 8-bit immediate data or stored in one of R0-R3. The offset is ORed with the base to generate the memory address. Available instructions are shown below. The intended usage has the low order byte of the base address equal to 0.

Example 1:
```
RD1 115
```
Equivalent C code:
```
R0 = *(DS1 | 115);
```

Example 2:
```
RD2RI R0, R1
```
Equivalent C code:
```
R1 = *(DS2 | R0++);
```

mnemonic	*Opcode*	*description*
RD1	42aa	Read indexed base register using DS1 as base, 8-bit immediate as offset. Store in R0.
RD2	41aa	Read indexed base register using DS2 as base, 8-bit immediate as offset. Store in R0.
RD1R	52sr	Read indexed base register using DS1 as base, named register for offset.
RD2R	51sr	Read indexed base register using DS2 as base, named register for offset.
RD1RI	56sr	Read indexed base register using DS1 as base, named register for offset, post-increment offset.
RD2RI	55sr	Read indexed base register using DS2 as base, named register for offset, post-increment offset.
RD1RD	5Asr	Read indexed base register using DS1 as base, named register for offset, post-decrement offset.
RD2RD	59sr	Read indexed base register using DS2 as base, named register for offset, post-decrement offset.

6.5.1.1.26.RDFLG Read Flags

Assembler Syntax:	RDFLG *register*
Opcode:	10Bs
Argument:	R0, R1, R2, or R3
Words:	1
Clocks:	2
Flags Affected:	None
Description:	Copy the controller status word into a register (R0-R3).
Example:	`RDFLG R0`
Equivalent C code:	`R0 = CSW;`

6.5.1.1.27.RDI Read Immediate

Assembler Syntax:	RDI *data, register*
Opcode:	430r
Argument:	[data] hexadecimal value or symbol register R0, R1, R2, R3, Zero, One, DS1, DS2, or SP
Words:	2
Clocks:	6
Flags Affected:	None
Description:	Copy memory to register.
Example:	`RDI 1234h, R0`
Equivalent C code:	`R0 = *(0x1234);`

6.5.1.1.28.RDR Read Indirect Register

Assembler Syntax:	RDR *memory location, register*
Opcode:	50sr
Arguments:	[memory location] hexadecimal value or symbol register R0, R1, R2, R3, Zero, One, or SP
Words:	1
Clocks:	4
Flags Affected:	None
Description:	Copy memory to register.
Example:	`RDR R0, R1`
Equivalent C code:	`R1 = *R0;`

6.5.1.1.29.RETS Return from Subroutine

Assembler Syntax:	RETS
Opcode:	0706
Argument:	None
Words:	1
Clocks:	4
Flags Affected:	None
Description:	The return address is popped off the stack, then the jump is made. The return address may be up to 65,536 words away.
Example:	`RETS`
Equivalent C code:	`PC = Stack[SP--] + 1;`

6.5.1.1.30.ROTL/ROTR Rotate Register

Assembler Syntax:	ROTL *register* ROTR *register*
Opcode:	ROTL = 2Fr-, ROTR = 2Er-
Argument:	R0, R1, R2, R3, Zero, or One
Words:	1
Clocks:	2
Flags Affected:	Carry, Zero, Negative, Positive, Overflow
Description:	ROTR: rotate the register right, LSB goes to MSB. ROTL: rotate the register left, MSB goes to LSB.
Example:	`ROTR R0`
Equivalent C code:	`lsb = R0 & 1; R0 >>=1; R0 &= 0x7FFF;` `if ( lsb ) R0 |= 0x8000;`

6.5.1.1.31.SFLG[**] Set Flag

Assembler Syntax: SFLG[**]

Opcode: See table below
Argument: None
Words: 1
Clocks: 2
Flags Affected: Named flag

Description: Sets the specified flag to 1

Example: `SFLGC`
Equivalent C code: `CSW |= 0x0001; /* set carry flag */`

mnemonic	*Opcode*	*description*
SFLGC	0010	Set Carry Flag
SFLGZ	0110	Set Zero Flag
SFLGN	0210	Set Negative Flag
SFLGP	0310	Set Positive Flag
SFLGO	0410	Set Overflow Flag
SFLGIR	0510	Set Interrupt Request Flag
SFLGIE	0610	Set Interrupt Enable Flag
SFLGSE	0710	Set Stack Error Flag
SFLGGE	0810	Set General Error Flag
SFLGMC	0910	Set Multi-Class Firing Flag
SFLGFW	0A10	Set Flash Write Flag
SFLGM1	0B10	Set MURAM 1 Ready Flag
SFLGM2	0C10	Set MURAM 2 Ready Flag
SFLGDC	0D10	Set PADCUs Busy Flag

6.5.1.1.32.SHL/SHR Shift Register

Assembler Syntax:	SHL *register* SHR *register*
Opcode:	SHL = 2Dr-, SHR = 2Cr-
Argument:	R0, R1, R2, R3, Zero, or One
Words:	1
Clocks:	2
Flags Affected:	Carry, Zero, Negative, Positive, Overflow
Description:	SHR: shift the register right, LSB goes to carry bit, MSB = 0. SHL: shift the register left, MSB goes to carry bit, LSB = 0.
Example:	`SHR R0`
Equivalent C code:	`R0 >>= 1;`

6.5.1.1.33.SUB Subtract

Assembler Syntax:	SUB *subtrahend, minuend*
Opcode:	29rr
Argument:	R0, R1, R2, R3, Zero, One
Words:	1
Clocks:	2
Flags Affected:	Carry, zero, negative, positive, overflow
Description:	Subtract the contents of the first-named register from the contents of the second-named register and place the remainder in the second-named register.
Example:	`SUB R3, R0`
Equivalent C code:	`R0 -= R3;`

6.5.1.1.34.WAIT[**] Wait for Flag

Assembler Syntax: WAIT[**]

Opcode: See table below
Argument: None
Words: 1
Clocks: as many as it takes
Flags Affected: None

Description: This set of instructions waits for a particular flag to become 1. Normally, you would only use the instructions WAITIR, WAITFW, WAITMU.

Example: `WAITIR`
Equivalent C code:

```
while( (CSW & 0x0020) != 1 ); /* wait for
interrupt request */
```

mnemonic	*Opcode*	*description*
WAIT	050A	Equivalent to WAITIR
WAITC	000A	Wait For Carry Flag
WAITZ	010A	Wait For Zero Flag
WAITN	020A	Wait For Negative Flag
WAITP	030A	Wait For Positive Flag
WAITO	040A	Wait For Overflow Flag
WAITIR	050A	Wait For Interrupt Request Flag
WAITIE	060A	Wait For Interrupt Enable Flag
WAITSE	070A	Wait For Stack Error Flag
WAITGE	080A	Wait For General Error Flag
WAITMC	090A	Wait For Multi-Class Firing Flag
WAITFW	0A0A	Wait For Flash Write Flag
WAITM1	0B0A	Wait For MURAM 1 Ready Flag
WAITM2	0C0A	Wait For MURAM 2 Ready Flag
WAITDC	0D0A	Wait For PADCUs Busy Flag

6.5.1.1.35.WR[***] Write Indexed Base Register

Assembler Syntax: WR[***] *data* or WR[***] *register*

Opcode: See table below
Argument: [data] hexadecimal value or symbol
register R0, R1, R2, R3
Words: 1
Clocks: 2
Flags Affected: Carry, Zero, Negative, Positive, Overflow

Description: Copy from register to memory address indexed from base register. The base address is stored in either DS1 or DS2. The offset is 8-bit immediate data or stored in one of R0-R3. The offset is ORed with the base to generate the memory address. Available instructions are shown below.

Example 1:
```
WR2 165
```
Equivalent C code:
```
*(DS2 | 165) = R0;
```

Example 2:
```
WR1RI R2, R1
```
Equivalent C code:
```
*(DS2 | R1++) = R2;
```

mnemonic	*Opcode*	*description*
WR1	62aa	Write indexed base register using DS1 as base, 8-bit immediate as offset. Source in R0.
WR2	61aa	Write indexed base register using DS2 as base, 8-bit immediate as offset. Source in R0.
WR1R	72sr	Write indexed base register using DS1 as base, named register for offset.
WR2R	71sr	Write indexed base register using DS2 as base, named register for offset.
WR1RI	76sr	Write indexed base register using DS1 as base, named register for offset, post-increment offset.
WR2RI	75sr	Write indexed base register using DS2 as base, named register for offset, post-increment offset.
WR1RD	7Asr	Write indexed base register using DS1 as base, named register for offset, post-decrement offset.
WR2RD	79sr	Write indexed base register using DS2 as base, named register for offset, post-decrement offset.

6.5.1.1.36.WRFLG Write Flags

Assembler Syntax:	WRFLG register
Opcode:	10sB
Argument:	R0, R1, R2, or R3
Words:	1
Clocks:	2
Flags Affected:	All
Description:	Copy a register (R0-R3) into the controller status word.
Example:	`WDFLG R2`
Equivalent C code:	`CSW = R2;`

6.5.1.1.37.WRI Write Immediate

Assembler Syntax:	WRI *register, memory location*
Opcode:	630r
Argument:	register R0, R1, R2, R3, Zero, One, or SP [memory location] hexadecimal address or symbol
Words:	2
Clocks:	4
Flags Affected:	None
Description:	Copy register to memory.
Example:	`WRI R0, 1234h`
Equivalent C code:	`*(0x1234) = R0;`

6.5.1.1.38.WRR Write Indirect Register

Assembler Syntax:	WRR *source, destination*
Opcode:	70sr
Argument:	[source] R0, R1, R2, R3, Zero, One [destination] R0, R1, R2, R3
Words:	1
Clocks:	2
Flags Affected:	None
Description:	Copy register to memory. Destination address must be in R0 - R3.
Example:	`WRR R0, R1`
Equivalent C code:	`*R1 = R0;`

6.5.1.1.39.WRRI Write Indirect Register Immediate

Assembler Syntax:	WRRI *data, register*
Opcode:	73s0
Argument:	[data] hexadecimal value or symbol [register] register containing memory address
Words:	2
Clocks:	4
Flags Affected:	None
Description:	Copy immediate data to memory indirect.
Example:	`WRRI 1234h, R1`
Equivalent C code:	`*R1 = 0x1234;`

6.5.1.1.40.XOR Logical Exclusive OR

Assembler Syntax:	XOR register register
Opcode:	37rr
Argument:	R0, R1, R2, R3, Zero, or One
Words:	1
Clocks:	2
Flags Affected:	Carry, Zero, Negative, Positive, Overflow
Description:	XOR two registers. Place the results in the second register.
Example:	`XOR R0, R1`
Equivalent C code:	`R1 ^= R0;`

6.6. Flags Cross Reference

Table 6-3 lists all the instructions with the mnemonics, opcode, English description and the flags affected by each instruction. The flag abbreviations are:

C	=	Carry	SE	=	Stack Error
Z	=	Zero	GE	=	General Error
N	=	Negative	MC	=	Multi-Class Firing
P	=	Positive	FW	=	Flash Write
O	=	Overflow	M1	=	MURAM 1 Ready
IR	=	Interrupt Request	M2	=	MURAM 2 Ready
IE	=	Interrupt Enable	DC	=	PADCUs Busy

Table 6-3. Flags Cross-Reference

Mnemonic	Opcode	Description	C	Z	N	P	O	IR	IE	SE	GE	MC	FW	M1	M2	DC
ADC	33rr	Add with Carry	■	■	■	■	■									
ADD	31rr	Add	■	■	■	■	■									
AND	34rr	Logical AND	■	■	■	■	■									
CFLGC	0012	Clear Carry Flag	■													
CFLGDC	0D12	Clear PADCUs Busy Flag														■
CFLGFW	0A12	Clear Flash Write Flag											■			
CFLGGE	0812	Clear General Error Flag									■					
CFLGIE	0612	Clear Interrupt Enable Flag							■							
CFLGIR	0512	Clear Interrupt Request Flag						■								
CFLGM1	0B12	Clear MURAM 1 Ready Flag												■		
CFLGM2	0C12	Clear MURAM 2 Ready Flag													■	
CFLGMC	0912	Clear Multi-Class Firing Flag										■				
CFLGN	0212	Clear Negative Flag			■											
CFLGO	0412	Clear Overflow Flag					■									
CFLGP	0312	Clear Positive Flag				■										
CFLGSE	0712	Clear Stack Error Flag								■						
CFLGZ	0112	Clear Zero Flag		■												
CMP	39rr	Compare	■	■	■	■	■									
DEC	28r-	Decrement	■	■	■	■	■									
INC	32-r	Increment	■	■	■	■	■									
JC	90aa	Jump on Carry	■	■	■	■	■									
JCI	D061	Jump on Carry to Immediate Address														
JCIR	D0C1	Jump on Carry to Immediate Relative	■	■	■	■	■									
JCR	B06r	Jump on Carry to Register														
JCRR	B0Cr	Jump on Carry to Register Relative	■	■	■	■	■									
JDC	9Daa	Jump on PADCUs Busy	■	■	■	■	■									

Mnemonic	Opcode	Description	C	Z	N	P	O	IR	IE	SE	GE	MC	FW	M1	M2	DC
JDCI	DD61	Jump on PADCUs Busy to Immediate Address														
JDCIR	DDC1	Jump on PADCUs Busy to Immediate Relative	■	■	■	■	■									
JDCR	BD6r	Jump on PADCUs Busy to Register														
JDCRR	BDCr	Jump on PADCUs Busy to Register Relative	■	■	■	■	■									
JFW	9Aaa	Jump on Flash Write	■	■	■	■	■									
JFWI	DA61	Jump on Flash Write to Immediate Address														
JFWIR	DAC1	Jump on Flash Write to Immediate Relative	■	■	■	■	■									
JFWR	BA6r	Jump on Flash Write to Register														
JFWRR	BACr	Jump on Flash Write to Register Relative	■	■	■	■	■									
JGE	98aa	Jump on General Error	■	■	■	■	■									
JGEI	D861	Jump on General Error to Immediate Address														
JGEIR	D8C1	Jump on General Error to Immediate Relative	■	■	■	■	■									
JGER	B86r	Jump on General Error to Register														
JGERR	B8Cr	Jump on General Error to Register Relative	■	■	■	■	■									
JIE	96aa	Jump on Interrupt Enable	■	■	■	■	■									
JIEI	D661	Jump on Interrupt Enable to Immediate Address														
JIEIR	D6C1	Jump on Interrupt Enable to Immediate Relative	■	■	■	■	■									
JIER	B66r	Jump on Interrupt Enable to Register														
JIERR	B6Cr	Jump on Interrupt Enable to Register Relative	■	■	■	■	■									
JIR	95aa	Jump on Interrupt Request	■	■	■	■	■									
JIRI	D561	Jump on Interrupt Request to Immediate Address														
JIRIR	D5C1	Jump on Interrupt Request to Immediate Relative	■	■	■	■	■									
JIRR	B56r	Jump on Interrupt Request to Register														

Mnemonic	Opcode	Description	C	Z	N	P	O	IR	IE	SE	GE	MC	FW	M1	M2	DC
JIRRR	B5Cr	Jump on Interrupt Request to Register Relative	■	■	■	■	■									
JM1	9Baa	Jump on MURAM 1 Ready	■	■	■	■	■									
JM1I	DB61	Jump on MURAM 1 Ready to Immediate Address														
JM1IR	DBC1	Jump on MURAM 1 Ready to Immediate Relative	■	■	■	■	■									
JM1R	BB6r	Jump on MURAM 1 Ready to Register														
JM1RR	BBCr	Jump on MURAM 1 Ready to Register Relative	■	■	■	■	■									
JM2	9Caa	Jump on MURAM 2 Ready	■	■	■	■	■									
JM2I	DC61	Jump on MURAM 2 Ready to Immediate Address														
JM2IR	DCC1	Jump on MURAM 2 Ready to Immediate Relative	■	■	■	■	■									
JM2R	BC6r	Jump on MURAM 2 Ready to Register														
JM2RR	BCCr	Jump on MURAM 2 Ready to Register Relative	■	■	■	■	■									
JMC	99aa	Jump on Multi-Class Firing	■	■	■	■	■									
JMCI	D961	Jump on Multi-Class Firing to Immediate Address														
JMCIR	D9C1	Jump on Multi-Class Firing to Immediate Relative	■	■	■	■	■									
JMCR	B96r	Jump on Multi-Class Firing to Register														
JMCRR	B9Cr	Jump on Multi-Class Firing to Register Relative	■	■	■	■	■									
JMP	9Faa	Jump Unconditional	■	■	■	■	■									
JMPI	DF61	Jump Unconditional to Immediate Address														
JMPIR	DFC1	Jump Unconditional to Immediate Relative	■	■	■	■	■									
JMPR	BF6r	Jump Unconditional to Register														
JMPRR	BFCr	Jump Unconditional to Register Relative	■	■	■	■	■									
JN	92AA	Jump on Negative	■	■	■	■	■									
JNC	80aa	Jump on No Carry	■	■	■	■	■									
JNCI	C061	Jump on No Carry to Immediate Address														

Mnemonic	Opcode	Description	C	Z	N	P	O	IR	IE	SE	GE	MC	FW	M1	M2	DC
JNCIR	C0C1	Jump on No Carry to Immediate Relative	■	■	■	■	■									
JNCR	A06r	Jump on No Carry to Register														
JNCRR	A0Cr	Jump on No Carry to Register Relative	■	■	■	■	■									
JNDC	8Daa	Jump on No PADCUs Busy	■	■	■	■	■									
JNDCI	CD61	Jump on No PADCUs Busy to Immediate Address														
JNDCIR	CDC1	Jump on No PADCUs Busy to Immediate Relative	■	■	■	■	■									
JNDCR	AD6r	Jump on No PADCUs Busy to Register														
JNDCRR	ADCr	Jump on No PADCUs Busy to Register Relative	■	■	■	■	■									
JNFW	8Aaa	Jump on No Flash Write	■	■	■	■	■									
JNFWI	CA61	Jump on No Flash Write to Immediate Address														
JNFWIR	CAC1	Jump on No Flash Write to Immediate Relative	■	■	■	■	■									
JNFWR	AA6r	Jump on No Flash Write to Register														
JNFWRR	AACr	Jump on No Flash Write to Register Relative	■	■	■	■	■									
JNGE	88aa	Jump on No General Error	■	■	■	■	■									
JNGEI	C861	Jump on No General Error to Immediate Address														
JNGEIR	C8C1	Jump on No General Error to Immediate Relative	■	■	■	■	■									
JNGER	A86r	Jump on No General Error to Register														
JNGERR	A8Cr	Jump on No General Error to Register Relative	■	■	■	■	■									
JNI	D261	Jump on Negative to Immediate Address														
JNIE	86aa	Jump on No Interrupt Enable	■	■	■	■	■									
JNIEI	C661	Jump on No Interrupt Enable to Immediate Address														
JNIEIR	C6C1	Jump on No Interrupt Enable to Immediate Relative	■	■	■	■	■									
JNIER	A66r	Jump on No Interrupt Enable to Register														

Mnemonic	Opcode	Description	C	Z	N	P	O	IR	IE	SE	GE	MC	FW	M1	M2	DC
JNIERR	A6Cr	Jump on No Interrupt Enable to Register Relative	■	■	■	■	■									
JNIR	85aa	Jump on No Interrupt Request	■	■	■	■	■									
JNIRI	C561	Jump on No Interrupt Request to Immediate Address														
JNIRIR	C5C1	Jump on No Interrupt Request to Immediate Relative	■	■	■	■	■									
JNIRR	A56r	Jump on No Interrupt Request to Register														
JNIRRR	A5Cr	Jump on No Interrupt Request to Register Relative	■	■	■	■	■									
JNM1	8Baa	Jump on No MURAM 1 Ready	■	■	■	■	■									
JNM1I	CB61	Jump on No MURAM 1 Ready to Immediate Address														
JNM1IR	CBC1	Jump on No MURAM 1 Ready to Immediate Relative	■	■	■	■	■									
JNM1R	AB6r	Jump on No MURAM 1 Ready to Register														
JNM1RR	ABCr	Jump on No MURAM 1 Ready to Register Relative	■	■	■	■	■									
JNM2	8Caa	Jump on No MURAM 2 Ready	■	■	■	■	■									
JNM2I	CC61	Jump on No MURAM 2 Ready to Immediate Address														
JNM2IR	CCC1	Jump on No MURAM 2 Ready to Immediate Relative	■	■	■	■	■									
JNM2R	AC6r	Jump on No MURAM 2 Ready to Register														
JNM2RR	ACCr	Jump on No MURAM 2 Ready to Register Relative	■	■	■	■	■									
JNMC	89aa	Jump on No Multi-Class Firing	■	■	■	■	■									
JNMCI	C961	Jump on No Multi-Class Firing to Immediate Address														
JNMCIR	C9C1	Jump on No Multi-Class Firing to Immediate Relative	■	■	■	■	■									
JNMCR	A96r	Jump on No Multi-Class Firing to Register														
JNMCRR	A9Cr	Jump on No Multi-Class Firing to Register Relative	■	■	■	■	■									
JNN	82aa	Jump on No Negative	■	■	■	■	■									
JNNI	C261	Jump on No Negative to Immediate Address														

Mnemonic	Opcode	Description	C	Z	N	P	O	IR	IE	SE	GE	MC	FW	M1	M2	DC
JNNIR	C2C1	Jump on No Negative to Immediate Relative	■	■	■	■	■									
JNNR	A26r	Jump on No Negative to Register														
JNNRR	A2Cr	Jump on No Negative to Register Relative	■	■	■	■	■									
JNO	84aa	Jump on No Overflow	■	■	■	■	■									
JNOI	C461	Jump on No Overflow to Immediate Address														
JNOIR	C4C1	Jump on No Overflow to Immediate Relative	■	■	■	■	■									
JNOR	A46r	Jump on No Overflow to Register														
JNORR	A4Cr	Jump on No Overflow to Register Relative	■	■	■	■	■									
JNP	83aa	Jump on No Positive	■	■	■	■	■									
JNPI	C361	Jump on No Positive to Immediate Address														
JNPIR	C3C1	Jump on No Positive to Immediate Relative	■	■	■	■	■									
JNPR	A36r	Jump on No Positive to Register														
JNPRR	A3Cr	Jump on No Positive to Register Relative	■	■	■	■	■									
JNR	B26R	Jump on Negative to Register														
JNRI	D2C1	Jump on Negative to Immediate Relative	■	■	■	■	■									
JNRR	B2Cr	Jump on Negative to Register Relative	■	■	■	■	■									
JNSE	87aa	Jump on No Stack Error	■	■	■	■	■									
JNSEI	C761	Jump on No Stack Error to Immediate Address														
JNSEIR	C7C1	Jump on No Stack Error to Immediate Relative	■	■	■	■	■									
JNSER	A76r	Jump on No Stack Error to Register														
JNSERR	A7Cr	Jump on No Stack Error to Register Relative	■	■	■	■	■									
JNZ	81aa	Jump on No Zero	■	■	■	■	■									
JNZI	C161	Jump on No Zero to Immediate Address														
JNZIR	C1C1	Jump on No Zero to Immediate Relative	■	■	■	■	■									
JNZR	A16r	Jump on No Zero to Register														

Mnemonic	Opcode	Description	C	Z	N	P	O	IR	IE	SE	GE	MC	FW	M1	M2	DC
JNZRR	A1Cr	Jump on No Zero to Register Relative	■	■	■	■	■									
JO	94aa	Jump on Overflow	■	■	■	■	■									
JOI	D461	Jump on Overflow to Immediate Address														
JOIR	D4C1	Jump on Overflow to Immediate Relative	■	■	■	■	■									
JOR	B46r	Jump on Overflow to Register														
JORR	B4Cr	Jump on Overflow to Register Relative	■	■	■	■	■									
JP	93aa	Jump on Positive	■	■	■	■	■									
JPI	D361	Jump on Positive to Immediate Address														
JPIR	D3C1	Jump on Positive to Immediate Relative	■	■	■	■	■									
JPR	B36r	Jump on Positive to Register														
JPRR	B3Cr	Jump on Positive to Register Relative	■	■	■	■	■									
JS	E0aa	Jump to Subroutine														
JSE	97aa	Jump on Stack Error	■	■	■	■	■									
JSEI	D761	Jump on Stack Error to Immediate Address														
JSEIR	D7C1	Jump on Stack Error to Immediate Relative	■	■	■	■	■									
JSER	B76r	Jump on Stack Error to Register														
JSERR	B7Cr	Jump on Stack Error to Register Relative	■	■	■	■	■									
JSI	E860	Jump to Subroutine Immediate														
JSIR	E8C0	Jump to Subroutine Immediate Relative	■	■	■	■	■									
JSR	E46r	Jump to Subroutine Register														
JSRR	E4Cr	Jump to Subroutine Register Relative	■	■	■	■	■									
JZ	91aa	Jump on Zero	■	■	■	■	■									
JZI	D161	Jump on Zero to Immediate Address														
JZIR	D1C1	Jump on Zero to Immediate Relative	■	■	■	■	■									
JZR	B16r	Jump on Zero to Register														
JZRR	B1Cr	Jump on Zero to Register Relative	■	■	■	■	■									
LDI	0r14	Load Immediate														

Mnemonic	Opcode	Description	C	Z	N	P	O	IR	IE	SE	GE	MC	FW	M1	M2	DC
MOV	12rr	Move Register to Register														
NOOP	0000	No Operation														
NOT	26r-	Logical Negation	■	■	■	■	■									
OR	35rr	Logical OR	■	■	■	■	■									
POP	0r04	Pop Register off Stack														
PUSH	0r02	Push Register onto Stack														
RD1	42dd	Read Indexed Base Register 1 Immediate	■	■	■	■	■									
RD1R	52rr	Read Indexed Base Register 1 Register	■	■	■	■	■									
RD1RD	5Arr	Read-Post-Decrement Indexed Base Register 1 Register	■	■	■	■	■									
RD1RI	56rr	Read-Post-Increment Indexed Base Register 1 Register	■	■	■	■	■									
RD2	41dd	Read Indexed Base Register 2 Immediate	■	■	■	■	■									
RD2R	51rr	Read Indexed Base Register 2 Register	■	■	■	■	■									
RD2RD	59rr	Read-Post-Decrement Indexed Base Register 2 Register	■	■	■	■	■									
RD2RI	55rr	Read-Post-Increment Indexed Base Register 2 Register	■	■	■	■	■									
RDFLG	10Br	Read Flags														
RDI	430r	Read Immediate														
RDR	50rr	Read Indirect Register														
RETS	0706	Return from Subroutine														
ROTL	2Fr-	Left Rotate Register	■	■	■	■	■									
ROTR	2Er-	Right Rotate Register	■	■	■	■	■									
SFLGC	0010	Set Carry Flag	■													
SFLGDC	0D10	Set PADCUs Busy Flag														■
SFLGFW	0A10	Set Flash Write Flag											■			
SFLGGE	0810	Set General Error Flag									■					
SFLGIE	0610	Set Interrupt Enable Flag							■							
SFLGIR	0510	Set Interrupt Request Flag						■								
SFLGM1	0B10	Set MURAM 1 Ready Flag												■		
SFLGM2	0C10	Set MURAM 2 Ready Flag													■	
SFLGMC	0910	Set Multi-Class Firing Flag										■				
SFLGN	0210	Set Negative Flag			■											
SFLGO	0410	Set Overflow Flag					■									
SFLGP	0310	Set Positive Flag				■										

Mnemonic	Opcode	Description	C	Z	N	P	O	IR	IE	SE	GE	MC	FW	M1	M2	DC
SFLGSE	0710	Set Stack Error Flag								■						
SFLGZ	0110	Set Zero Flag														
SHL	2Dr-	Left Shift Register	■	■	■	■	■									
SHR	2Cr-	Right Shift Register	■	■	■	■	■									
SUB	29rr	Subtract	■	■	■	■	■									
WAIT	050A	Wait for Interrupt Request Flag														
WAITC	000A	Wait for Carry Flag														
WAITDC	0D0A	Wait for PADCUs Busy Flag														
WAITFW	0A0A	Wait for Flash Write Flag														
WAITGE	080A	Wait for General Error Flag														
WAITIE	060A	Wait for Interrupt Enable Flag														
WAITM1	0B0A	Wait for MURAM 1 Ready Flag														
WAITM2	0C0A	Wait for MURAM 2 Ready Flag														
WAITMC	090A	Wait for Multi-Class Firing Flag														
WAITN	020A	Wait for Negative Flag														
WAITO	040A	Wait for Overflow Flag														
WAITP	030A	Wait for Positive Flag														
WAITSE	070A	Wait for Stack Error Flag														
WAITZ	010A	Wait for Zero Flag														
WR1	62dd	Write Indexed Base Register 1 Immediate	■	■	■	■	■									
WR1R	72rr	Write Indexed Base Register 1 Register	■	■	■	■	■									
WR1RD	7Arr	Write Post-Decrement Indexed Base Register 1 Register-	■	■	■	■	■									
WR1RI	76rr	Write Post-Increment Indexed Base Register 1 Register-	■	■	■	■	■									
WR2	61dd	Write-Indexed Base Register 2 Immediate	■	■	■	■	■									
WR2R	71rr	Write-Indexed Base Register 2 Register	■	■	■	■	■									
WR2RD	79rr	Write Post-Decrement Indexed Base Register 2 Register-	■	■	■	■	■									
WR2RI	75rr	Write Post-Increment Indexed Base Register 2 Register-	■	■	■	■	■									
WRFLG	10rB	Write Flags	■	■	■	■	■	■	■	■	■	■	■	■	■	■
WRI	630r	Write Immediate														
WRR	70rr	Write Indirect Register														
WRRI	73r0	Write Indirect Register Immediate														
XOR	37rr	Logical Exclusive OR	■	■	■	■	■									

6.7. Interrupt Handling

The Ni1000 on-chip microcontroller has a single interrupt vector at address F000h, which is the beginning of the microcontroller program memory PGFLASH. The program counter PC is initialized to 1 instead of 0 for this reason.

The microcontroller interrupt request (IR) flag, which is CSW[5], is set when one of the following occurs:

- The host writes to the CMR register.
- The host writes to the IIR register.
- The host asserts the MCINT# pin.
- The host asserts the ERROR# pin.
- The microcontroller clears the General Error flag (CSW[8]) after setting it.

Note that the value written into CMR or IIR does not matter, any write to these registers causes IR to be set (except, of course, if CMR[15] is set, which resets the chip). For host interrupts of the micrcontroller, IIR[2:3] can be used to communicate the type of interrupt to the interrupt handler.

There is no acknowledge pin for the microcontroller interrupts. The host can determine that the microcontroller has acknowledged the interrupt request in several ways; see the example given at the end of this section.

When the interrupt is requested, the IR flag is set immediately. IR is cleared when the microcontroller software acknowledges the interrupt. If multiple interrupts occur before the microcontroller software acknowledges the first one, only one interrupt will occur.

Interrupt request is serviced when the interrupt enable (IE) flag is set. Service is provided in the following steps:

- Clear IE flag.
- Read interrupt service routine (ISR) entry from the first location of PGFLASH, at address F000h.
- Jump to subroutine ISR.

Clearing the IR flag and setting the IE flag should be done by software.

A service request by the microcontroller to the host is indicated by the SRQ# pin. It is asserted when the microcontroller writes to the XIR register. SRQ# is deasserted when the host asserts the IACK# pin. An outstanding service request is indicated by HS2[2].

The following is an example of the interrupt service routine (ISR), containing a command interpreter, hardware interrupt handler, and other service request. An example of the host service request is also provided. The assumptions are:

If the IR flag is set by multiple causes or multiple times service is provided, only the one of the highest priority will be serviced by the microcontroller. The priority order is:

- The host asserts the MCINT# pin or writes a 1 to IIR[0].

- The host asserts the ERROR# pin or the host writes a 1 to IIR[1].
- The host writes a 1 to IIR[2] or IIR[3].
- The host writes to CMR[0:14].
- The host writes a 0 to CMR[15] to reset the chip.

Since there is no acknowledge pin, the following example uses the host service request to gain acknowledge. When the interrupt request is sent by the host writing to the CMR register, service from the microcontroller is acknowledged through the general-purpose I/O registers. The following example does not handle any previous outstanding service requests and the actual implementation of the service routines is not provided; it is for illustration purpose only. It must not be used as a standard interrupt service routine. The user must implement his/her own routines to best satisfy specific application needs.

```
isrexam:          ; The physical address is F000h, which is the
                  ; first location in PGFLASH.

   cflgir         ; Clear the IR flag as soon as possible
                  ; to avoid missing new interrupts.

   push r0        ; Save registers.
   push r1
   push r2

   rdi 38h, r0    ; Read IIR.

   ldi 0fh, r1    ; Get mask for interrupt request other
                  ; than that by writing CMR.
   and r0, r1
   jnz @isr1

   ldi 8000h, r1 ; Get mask for IR by writing CMR to
                  ; screening the IR by host clearing IIR.
   cmp r0, r1
   jnz @isrret    ; No service required for this IR.

                  ; Now IR by writing CMR.
   jsi cmrintr    ; Call Command Interpreter ---
                  ; interpreter for commands written in CMR.
   jmp @isrret

@isr1:            ; Provide acknowledge to host for IR
                  ; other than by writing CMR. If you
                  ; want to do the same for IR by writing
                  ; CMR, move this part up and reorganize
                  ; the code.

   ldi 4, r1      ; Mask for SRQ in HS2.

@isrw1:
   rdi 28h, r2    ; Read HS2.
   cmp r1, r2
   jnz @isrw2
                  ;
```

```
                    ; Put script here to notify SRQ
                    ; outstanding to host or external HW to
                    ; notify that chip is waiting for
                    ; clearing SRQ.
                    ;
   jmp @isrw1

@isrw2:
   ldi 1111h,r1     ; Assume the host service request vector 1111h
                    ; is assigned for the microcontroller
                    ; interrupt acknowledge.
   wri r1, XIR      ; Now the SRQ# pin is be asserted. The
                    ; host should clear IIR with this
                    ; request.
   mov one, r1      ; Get mask for interrupt request from
                    ; MCINT#.
   and r0, r1       ;
   jz @isr2

                    ; IR is from MCINT# pin or host
                    ; a 1 to IIR.
   jsi hwirsvc      ; Call service routine.
   jmp @isr4

@isr2:
   ldi 2, r1        ; Get mask for interrupt request from
                    ; ERROR# or the microcontroller Error
                    ; flag.
   and r0, r1
   jz @isr3

                    ; IR is from ERROR# or the
                    ; microcontroller error flag.
   jsi erirsvc      ; Call service routine.
   jmp @isr4

@isr3:              ; IR is by the host writing IIR.
   jsi hsirsvc      ; Call service routine.

@isr4:              ; Clear IR if by the host clearing IIR.
                    ; This can be removed.
   rdi 38h, r0
   cmp r0, rz
   jnz @isrret
   cflgir

@isrret:            ; Return processing.
   pop r2           ; Restore registers.
   pop r1
   pop r0
   sflgie           ; Enable interrupt.
   rets
```

7. MICROCONTROLLER SOFTWARE

7.1. OVERVIEW

This chapter is a functional description of the design and implementation of the Ni1000 microcontroller software (also called microcode). Additionally, it describes conventions used in the design and should assist experienced assembly language programmers in writing their own functions to add to the command set. If these conventions are maintained, new functions can be quickly installed and debugged, without destabilizing the existing code.

7.1.1. Code Design Goals

The Ni1000 microcontroller software is designed to be efficient, modular and compact, without excess complexity. It has a deterministic response with no uninterruptable loops. There is a multi-chip protocol for increased prototype space expansion. Redundant code, which is normally common with in-line functions, is minimized. Protocol rules will be established by which the microcontroller will abide. Assuming that the host software also abides by these rules, the protocol will ensure reliable operation without hang states or I/O overruns. (See **Protocol Rules**).

7.1.2. Key Functions

The microcontroller software is commonly in an idle state, continually monitoring the status interfaces and the state of the chip. A soft command jump table is provided along with learning algorithms and interrupt service routines to and from the host controller. An expandable set of commands is supported, utilizing the I/O Register File, IRAM, and ORAM to send and receive data. The system configuration can be either single or multi-chip, the latter requiring assistance from an external host. Error management is provided for both internal and external events.

7.1.3. Code Structure

The code has a small real-time process monitor which is active in the idle state. Communication between interrupts and monitor code is accomplished via mailboxes and a command queue. Support is provided for queuing of multiple commands from the host and for multiple responses and error messages to the host.

7.1.4. Learning Code

The Ni1000 microcontroller software implements compact versions of learning code for RCE/PRCE and PNN.

7.1.5. Principal External Interfaces

There are three principal external interfaces: a microcontroller interrupt, the I/O register file, and the interface to IRAM and ORAM.

7.1.6. Error Processing

The microcontroller software identifies invalid commands and internal errors and notifies the host controller.

7.1.7. Program Flow

When the host removes the Ni1000 microcontroller from the reset state by writing 0 to bit 15 of CMR, the microcontroller begins running the initialization routine starting at location F001H. This routine initializes PPRAM, the Used flags in the PADCU, the command jump table, the command queue, the host message queue, and some status registers in GRAM. After this the microcode enters a loop in which it monitors the command queue, the host message queue, pending I/O operations, and some mailbox locations in GRAM.

When a command is written into CMR by the host controller, the microcontroller is interrupted. The interrupt service routine puts the command on the command queue, along with any associated input parameters. The monitor loop takes the command from the queue, references the command jump table, and calls the appropriate function.

When execution of a command is complete, the microcode puts a response message, including an error code if applicable, on the host message queue. As soon as the previous service request to the host has been acknowledged, the monitor loop takes the message from the queue and writes it to the XIR register, causing a new service request.

7.1.8. Ni1000 Microcontroller Code Function Blocks

INITIALIZATION
- Initialize Command Queue
- Initialize Host Message Queue
- Initialize PPRAM
- Initialize PADCU Flags
- Load Command Jump Table
- Initialize Status Registers
- Goto EXEC

EXEC
- Test For Pending Errors
- Test Error MBOX
- Test I/O Pending MBOX
- Test Command MBOX
- Else Idle & Test

LEARNING
- Initialize PPRAM
- Call LEARN Function
- Return

LEARN BEGIN	LEARN EPOCH	LEARN VECTOR

FLASH MEMORY MANAGEMENT
- Call Function
- Return

ERASE	PROGRAM	VERIFY

STANDARD COMMANDS
- Call Function
- Clear MBOX
- Return

CLASSIFY

MANUAL COMMANDS
- Call Function
- Set I/O Pending
- Clear MBOX
- Return

RAM FUNCTIONS	I/O FUNCTIONS

STANDARD I/O
- Send/Receive Learning Data
- Send/Receive Multi-Chip Data
- Send/Receive Error Data
- Set I/O Pending
- Set I/O Done
- Clear MBOX
- Return

INTERRUPT
- Disable Interrupts
- Save Context
- Process Interrupt
- Set MBOX
- Restore Context
- Enable Interrupts
- Return

ERROR
- Set Error Status
- Set MBOX
- Freeze
- Return

7.2. MEMORY UTILIZATION

7.2.1. Constants and Variables

Two words identifying the microcode type and revision number are locked in PGFLASH at hexadecimal addresses F004 and F005, respectively. They can also be found in the microcode object code file in the fifth and sixth words of the code segment. For more detail, see **Configuration Table Data,** below.

GRAM is a block of 256 memory locations mapped at hexadecimal addresses 1000 through 10FF on the Ni1000. It is maintained by the microcontroller, but can also be accessed by the host (see **Important GRAM Locations**, below). Constants and variables used by microcontroller software are loaded from instruction memory (PGFLASH) into GRAM.

7.2.2. Important GRAM Locations

Several locations in GRAM are of particular interest to the host, because they hold certain parameters that describe a network on the Ni1000. Sometimes the host must write to some of these locations, using the RAMWRITE command, when performing certain high level functions, such as loading networks into the chip. It is not necessary for the host to write to these locations for normal learning, or for restoring networks with the RESTORE command.

1. **NEXT_PT** is locked at 105Ch. Microcode sees this as the number of the column after the last one being used. When the host loads a network (PA and/or PPRAM) into the Ni1000, this parameter must be set equal to the number of prototypes, plus the number of disabled columns and columns reserved for 10-bit data that are encountered and skipped over in the process of loading PA and/or PPRAM.

2. **PT_DIM** is locked at 105Dh. This is the number of features in the network. When the host loads a network (PA and/or PPRAM) into the Ni1000, this parameter must be set equal to the number of features.

3. **LearnParadigm** is locked at 105Eh. This holds the paradigm used in training the network on the Ni1000. When the host loads a network, it should set this to a value that designates that network's paradigm. The following values are currently assigned: 0 for no network, 1 for RCE/PRCE, and 2 for PNN.

4. **SectorNumber** is locked at 1062h. Bits 5-3 represent the starting PA column number divided by 128 and bits 2-0 represent the starting PA row number divided by 32. The host writes to this location when loading a network, or switching to any specific network in the Ni1000 without using the RESTORE command.

5. **SmoothingBias** is locked at 1063h. This must be loaded with the smoothing factor exponent offset. This is important, because it is part of the smoothing factor, so changing this value changes the network itself.

6. **Param_A** through **Param_F** are locked at 1064h - 1069h. These are currently not defined, but are reserved for any additional parameters that may be needed if learning and/or classification paradigms other than RCE, PRCE, and PNN are implemented on the Ni1000 in the future.

7. **DIAG_MASK** is locked at 106Ah. It should normally be left at its default value of 0. Only one bit is currently defined. Bit 0, when set, enables the host to write anything it wants into all bits of a PPRAM entry, including the PA Usage field and the Used and Disable bits, even if the data is inconsistent or does not make sense. This is provided for debugging capability, and should never be set for normal usage. Other bits may be defined in future revisions of the Ni1000 microcontroller software.

7.2.3. Command Jump Table

During microcode initialization, the command jump table is loaded by microcode into the first 64 locations of GRAM, starting at location 1000H. Each entry in the command jump table contains the PGFLASH address of a microcode routine in the 12 least significant bits. The first 32 entries are reserved for entry points to routines that process commands from the host, and the last 32 entries are used for other functions. For each of the first 32 entries, the four most significant bits indicate how many of the OP registers (OP0 - OP5) are used for input parameters to the command.

7.2.4. Command Queue

The command queue occupies the next 16 locations of GRAM, starting at location 1040H. Commands from the host are taken from CMR by the interrupt routine and added to the command queue, along with any associated parameters from the OP registers. Frames on the command queue are of varying length, since the number of parameters varies from one command to the next. The command queue is circular.

7.2.5. Host Message Queue

The host message queue occupies the next 8 locations of GRAM, starting at location 1050H. Command completion messages or error messages are always added to this queue. When the monitor loop detects that this queue is not empty, it checks to see if the host has acknowledged the previous service request. If so, the next message is removed from the queue and written to XIR, causing a service request. The host message queue is circular. If it becomes full and the microcode has another message to add to it, the microcode disables interrupts and waits until the host acknowledges a service request, freeing up space on the queue for the new message.

7.2.6. Instruction Memory

Code space available is 4K 16-bit words and is mapped at hexadecimal F000.

7.2.7. Interrupt Vector

The chip internal interrupt vector is the PGFLASH address of the microcode interrupt service routine, relative to hexadecimal F000. It is loaded into PGFLASH location 000h.

7.2.8. Initialization Code Entry Point

The microcode begins execution at PGFLASH address 001h when the host removes it from the reset state by writing 0 to bit 15 of CMR.

7.2.9. Microcontroller Stack

The microcontroller software uses the microcontroller stack for storage of data and return addresses. This stack has 64 16-bit locations. The stack pointer is incremented before data is pushed, so location 0 is not usable.

7.2.10. Configuration Table Data

The following configuration information can be accessed via the READCONFIG command.

Microcode ID (16 bits): This number identifies the type of microcode residing in PGFLASH. ID number 0000 has been assigned to the standard microcode, which is defined in this document. (There are some differences between the Ni1000 emulator and the actual hardware that make it necessary to have two separate versions of microcode. The equivalent version of microcode that is intended only for emulation has an ID of 0001.) Other ID numbers may be defined later if the 32 available opcodes are redefined for other command sets.

Microcode Revision Number (16 bits): Bits 15-12 = 0000; bits 11-8 = major revision; bits 7-4 = minor revision; and bits 3-0 = fix revision. For revision 2.0, the revision number is hexadecimal 0200.

Learning Paradigm: This identifies the paradigm associated with the neural network that is currently active on the chip. It is set during learning via the LEARNBEGIN command. As an alternative, the host can set the learning paradigm by writing to GRAM location 105E hex. This may be done in cases where the network is loaded from external files, instead of being learned on-chip. The following values are currently assigned: 0 for no network, 1 for RCE/PRCE, and 2 for PNN.

Number of Classes: This is actually the highest class number + 1. However, results are reported for all classes up to this number; so it is the number of classes active on the chip.

Number of Prototypes: The number of prototypes in the active network. Unused and disabled prototypes are not included.

Number of Features: The number of features currently being used in the active network.

Maximum Radius: The maximum radius value (a parameter associated with RCE/PRCE learning) that was used in training the currently active network.

Minimum Radius: The minimum radius value (a parameter associated with RCE/PRCE learning) that was used in training the currently active network.

Sector Number: The location of the currently active network. This number ranges from 0 to 64. Bits 0-2 specify the starting row number in PA divided by 32, and bits 3-5 specify the starting column number in PA and PPRAM divided by 128.

Smoothing factor exponent offset: The current setting for the offset that will be added to the normal smoothing factor exponent bias of -13 during probabilistic classification.

Param_A through **Param_F**: These are currently not defined, but are reserved for any additional parameters that may be needed if learning and/or classification paradigms other than RCE, PRCE, and PNN are implemented on the Ni1000 in the future.

7.3. PROGRAMMING CONVENTIONS

7.3.1. Register Use

Existing microcode subroutines do not always preserve the values of DS1, DS2, and R0 - R3. This means that for many subroutines, the calling routine is responsible for saving and restoring any registers it is using. This can be done by pushing them onto the microcontroller stack before calling the subroutine, and popping them after returning.

7.4. ERROR SUPPORT

Asynchronous errors have priority over synchronous errors. Currently the only asynchronous error is caused by the host controller or by another chip, via the ERROR# pin. The ERROR# pin causes the microcontroller to "freeze", until the error has been reset by the host controller. All other errors are communicated to the host controller via the XIR register in the order in which they occur.

7.4.1. Error Codes

When an error occurs, the microcontroller informs the host by writing a nonzero error code to XIR[14:8]. This allows for up to 127 different error codes. The following error codes (shown in hexadecimal) are currently assigned:

01: UNIMPLEMENTED
The command opcode issued by the host to the microcontroller through CMR is not implemented.

02: COMMAND_QFULL
The command queue is full and the microcontroller cannot accept the last command sent by the host.

03: STACK_ERROR
This error indicates that the microcontroller stack has overflowed or underflowed. This can only be caused by a microcode bug. There is no guarantee that this error message will ever actually be issued if a stack error occurs. A more reliable way for the host to detect a stack error is to check HS1 bit 7, which reflects the state of the microcontroller's Stack Error flag.

04: CHIP_FULL
The prototype array is full and the LEARNBEGIN command was unable to commit the current prototype.

05: BAD_RCEB_VALUE
The RCEB bit in the CRA register is 0, and it must be 1 for the current command. This error is for commands that load data into ORAM under microcode control (e.g., READCONFIG or COLUMNREAD). According to the protocol for this microcode, only the host should write to CRA.

06: COUNT_TOO_BIG
The count parameter is out of bounds for the current command.

07: RANGE_TOO_BIG
The combination of the count parameter with the starting location parameter causes an out of range condition for the current command.

08: PPRAM_DIS_ERROR
The host attempted, via RAMWRITE or PPRAMWRITE, to change the Disable bit for a column in PPRAM.

09: PPRAM_10BIT_ERROR
The host attempted, via RAMWRITE or PPRAMWRITE, to set the Used bit in PPRAM for a column that is reserved for 10-bit data and thus cannot be used for prototypes.

0A: PARAM_ERROR
A microcontroller routine had a parameter passed to it that is out of range. A common cause for this error is attempting to classify without initializing the network. CLASSMODE requires that two parameters in GRAM be initialized. The number of features must be at hexadecimal address 105D. The number of the column after the last one used by the network must be at hexadecimal address 105C. (Normally this is the same as the number of prototypes, unless there are disabled or unused columns in the range of columns used by the network.) These locations are loaded during normal learning. They can also be loaded via RAMWRITE.

0B: DIS_COLUMN_ERROR
The host attempted to program or erase a disabled column in the prototype array.

0C: ERASE_ERROR
An attempt to erase a column in the prototype array has failed. The error occurred on the erase operation itself, after both columns in the block were successfully programmed to all ones. When this error occurs, the microcode disables both columns in the block that failed to erase.

0D: BAD_COLUMN_ERROR
An attempt to program data into a column in the prototype array has failed. The error occurred on the programming operation itself, after the erase operation completed successfully (if it was necessary). When this error occurs, the microcode disables both columns in the block that could not be programmed.

0E: PROTOTYPE_LOST
A prototype has been lost from the network: an attempt by LEARNBEGIN to commit a new prototype failed because the column went bad. The other column in the same block did not contain a prototype. When this error occurs, the microcode disables both columns in the block that failed.

0F: BOTH_LOST
Two prototypes have been lost from the network: an attempt by LEARNBEGIN to commit a new prototype failed because the column went bad. The other column in the same block contained a prototype, which was also lost. When this error occurs, the microcode disables both columns in the block that caused the error.

10: PPRAM_ERROR
An attempt was made, via the PPRAMWRITE command, to set both the Used and Disable bits in a PPRAM entry. The microcontroller does not allow this.

11: RESTRICTED_ERROR
An attempt was made to alter a location that is protected by the microcode. Currently the only locations that are protected by microcode are the command queue, the host message queue, and their pointers. These locations are in GRAM, mapped at hexadecimal addresses 1040 through 105B.

12: UNDEFINED_INT
An undefined interrupt was taken. This means a value greater than 31 was written to CMR, or a value of 8 or C was written to IIR.

13: NOT_ERASED
An attempt was made to write data in 10-bit mode into the prototype array with the COLUMNWRITE command, but the area of PA being written to is not erased. In 10-bit mode, the host must erase the block and restore any other data in the block.

14: PGM1_ERROR
While preparing to erase a block in the prototype array, the microcontroller was not able to program the first column in the block to all 1's. In this context, the first column is the target column for COLUMNERASE or COLUMNWRITE (whether in the high or low half of PA), or the lower numbered column for BLOCKERASE.

15: PGM2_ERROR
While preparing to erase a block in the prototype array, the microcontroller was not able to program the second column in the block to all 1's. In this context, the second column is the other column that shares a block with the target column for COLUMNERASE or COLUMNWRITE (whether in the high or low half of PA), or the higher numbered column for BLOCKERASE.

16: BAD_PARADIGM
The LEARNVECTOR command returns this error if the learning paradigm at hexadecimal address 105E in GRAM is either 0 or some undefined value. The learning paradigm is normally set by LEARNBEGIN. Defined values are: 0 for no network, 1 for RCE/PRCE, and 2 for PNN.

17: NO_PACT
The RESTORE command returns this error if the PA Configuration Table does not exist.

18: NO_SUCH_NET
The RESTORE command returns this error if the specified network is not listed in the PA Configuration Table.

19: NOT_BACKED_UP
The RESTORE command returns this error if the PA Configuration Table indicates that the specified network exists, but is not backed up in the prototype array.

1A: PACT_CORRUPTED
The RESTORE command returns this error if an inconsistency is found in the PA Configuration Table.

1B: BAD_BIT_MODE
Using COLUMNWRITE, the host attempted to program in 5-bit mode a column that is reserved for 10-bit mode data, or to program in 10-bit mode a column in a block in which either column contains (5-bit mode) prototype data.

7.5. DEBUGGING SUPPORT

Currently the only debugging mechanism is the interrupt caused by the MCINT# pin. This interrupt takes a snapshot of microcode activity by making some status information visible in some of the I/O registers. Bits 0-4 of HS1 get the value they had when the interrupt occurred. OP0 - OP3 get the contents of R0 - R3 at the time the interrupt occurred. OP4 gets the value of the stack pointer before the interrupt, and OP5 gets the return address that was pushed on the microcontroller stack when the interrupt occurred. To make the microcontroller go back to what it was doing, the host should cause another interrupt, preferably by writing 0 to IIR.

7.6. HOST TO CHIP COMMUNICATION

7.6.1. Microcontroller Initialization

After resetting the Ni1000, the host writes 0 to bit 15 of CMR. This starts the microcode running. The microcode keeps interrupts disabled until it has finished running the initialization routine. As soon as interrupts are enabled, as indicated by bit 6 of HS1, the microcontroller is ready to accept commands from the host. Note that the microcontroller does not raise a service request when initialization is complete.

7.6.2. Microcontroller Interrupts

There are four ways for the host to interrupt the microcontroller: by asserting the ERROR# pin, by asserting the MCINT# pin, by writing directly to the IIR register, and by writing to the CMR register. In any of these cases, the microcontroller is interrupted immediately if interrupts are enabled; otherwise, the interrupt is deferred until the microcontroller enables interrupts. The interrupt forces the microcontroller to go to the interrupt service routine, the PGFLASH address of which is stored in the first location of PGFLASH. The microcontroller saves some state and then examines the contents of IIR to determine the cause of the interrupt.

When the ERROR# pin is asserted, it causes bit 1 of IIR to be set to 1. The microcode interrupt handler goes into a tight loop, watching the ERROR# pin. As soon as the ERROR# pin is no longer asserted, the microcontroller goes back to what it was doing when the interrupt occurred.

When the MCINT# pin is asserted, it causes bit 0 of IIR to be set to 1. The microcode interrupt handler restores bits 0-4 of the flags (in CSW and HS1) to the value they had when the interrupt occurred. OP0 - OP3 get the contents of R0 - R3 at the time the interrupt occurred. OP4 gets the value of the stack pointer before the interrupt, and OP5 gets the return address that was pushed on the microcontroller stack when the interrupt occurred. The microcontroller waits in a tight loop until another interrupt condition is asserted, then it restores OP0 - OP5 and allows the next interrupt to occur. The host should use the "null interrupt" (i.e., write 0 to IIR) to break the microcontroller out of this loop; then the microcontroller will go back to what it was doing when the interrupt occurred.

The IIR register identifies the cause of an interrupt to the microcontroller. The host can cause an interrupt by writing directly to IIR. Only bit 15 and bits 3-0 are implemented. Bits 15, 1, and 0 have special hardware significance. The host normally should not write anything but 0's to these three bits (although it is possible to fool the microcontroller into taking an invalid CMR, MCINT#, or ERROR# interrupt). That leaves bits 3-2 available to encode other special interrupts defined by the host and microcontroller software. Writing 0 to IIR causes the null interrupt, which is normally used to exit from the MCINT# interrupt. Writing 4 to IIR tells the microcontroller during multiple-chip learning that the global minimum distance is ready in OP4; the microcode sets a status bit in SSR when this interrupt occurs. The other two values, 8 and C, currently cause undefined interrupts.

When the host writes to CMR, it causes bit 15 of IIR to be set to 1. This interrupt is the normal method for issuing commands to the microcontroller. The value written to CMR is interpreted as a command opcode. The only hardware constraint on the command opcode is that bit 15 must be 0. (Writing 1 to bit 15 of CMR resets the chip.) In the current microcontroller software design, legal opcodes are 5 bits wide, ranging from 0 to 31.

To issue a command to the Ni1000, the host first writes the input parameters to the specified I/O registers, then writes the command opcode to the CMR register. The write to CMR causes bit 15 of the IIR register to be set and causes an interrupt to the Ni1000, setting the interrupt request flag (IR), which is visible to the host. If the command has an input vector associated with it, the host waits until the microcontroller code clears IR, signifying that it has taken the interrupt. Then it waits until the microcontroller code clears bit 0 of SSR, signifying that it is ready for the vector. Then after making sure IRAM is empty, it writes the input vector into IRAM. (The host can also interrupt the microcontroller by writing to IIR, or by pulling the ERROR# or MCINT# pin.)

7.6.3. I/O Register Usage

In this protocol, OP0 - OP3 are normally reserved for input parameters (PPRAMREAD being the only exception). OP5 is used for output if needed, as in the case of RAMREAD. OP4 is reserved for communicating the local and global minimum distances between the host and the Ni1000 during learning on a multi-chip network. Note that the host must be able to handle a request for global minimum distance when there is a learning command in progress on the Ni1000. The local and global minimum distances are passed through OP4, and bits in the SSR register will be used to indicate when the value in OP4 is a valid local or global minimum distance.

SSR should not be written to by the host. The following bits in SSR are currently defined.

Bit	Meaning
0	Serves a flow control. When set, indicates that the host should wait for the microcode to complete the current command.
1	Set to 1 when bit 2 is valid.
2	Set to 1 during learning if the network has changed during the current epoch.
3	Set by the LEARNVECTOR command when it commits a prototype.
4	Set to 1 when the command queue is not empty.
5	Set to 1 when the host message queue is not empty.
6	Unused.
7	Set to 1 during learning when the microcontroller has loaded the local minimum distance into OP4.
8	Set to 1 during the MCINT# interrupt.
9	Set to 1 by the LEARNVECTOR command when a count field in PPRAM overflows.
10	Set to 1 by the COLUMNREAD command when the PA column being read is disabled.
11	Set to 1 by the COLUMNREAD command when the PA column being read contains 10-bit data.
12	Set to 1 by LEARNVECTOR if an existing prototype was shrunk to accommodate a new prototype.
13	Unused.
14	Unused.
15	Set to 1 when the prototype array becomes full.

7.6.4. Host Service Requests

Upon completion of a command, the Ni1000 writes any output parameters to I/O registers or ORAM, then writes the command opcode to the XIR register. The write to XIR causes a service request to the host via the SRQ# pin. The host takes the contents of XIR and all output data, then acknowledges the service request via the IACK# pin.

A service request (SRQ) can be caused by ORAM becoming full (if the host enabled that type of SRQ via the OSR bit in CRA), or by the microcontroller writing to the XIR register. The host can identify the cause of the SRQ by looking at XIR. The value FFFFH is reserved for when ORAM becomes full. Other values are assigned by host and microcontroller software. The following format for XIR is implemented in the microcontroller code.

15	14 … 8	7 … 6	5 … 0
0	Error Code	Reserved	Command Opcode

15: Making this 0 ensures that no microcontroller-defined value will ever be FFFFH.
14-8: This is a 7-bit error code. Zero means no error. For other defined values, see **Error Codes.**
7-6: These 2 bits are reserved and must be 00.
5-0: If bit 5 is 0, then bits 4-0 contain the opcode of the command that was just completed (or the command that was executing when the error signified by bits 14-8 occurred). If bit 5 is 1, some other command or message is being sent to the host. For example, XIR will contain 20h when the chip requests the global minimum distance from the host.

7.6.5. Protocol Rules

The protocol rules are not hardware limitations. The first rule, for example, does not mean that the microcontroller is not capable of writing to CRA. It actually is capable of writing to CRA; otherwise, the rule would not be necessary. The purpose of these rules is to define the protocol that should be followed by any host driver code that communicates with *this version* of Ni1000 microcontroller software, so that hangs, overruns, and loss of data can be prevented. This protocol should also be followed by any future microcode routines that are integrated with this microcontroller software.

1. **The microcode will not write to the CRA register.** The functions controlled by CRA should be under control of the host processor. If the microcontroller was allowed to write to CRA, it could conflict with writes to CRA by the host.

2. **The host will not write to the CRB register.** The functions controlled by CRB should be under direct control of the microcode. For classification, the host would want to set or clear the OFPE bit. It must do this indirectly by passing a parameter with the CLASSMODE command.

3. **The host will not write to the SSR register.** The microcontroller modifies the contents of SSR at various times. If the host was allowed to write to SSR, it could conflict with writes to SSR by the microcontroller.

4. **The host will not acknowledge a service request until it has taken the contents of XIR and any other output parameters.** This rule in conjunction with rule 5 ensures that the chip will not overrun the host, causing service requests and output parameters to be lost.

5. **The microcontroller will check to make sure that SRQ# is not active before it writes any output parameters or writes to XIR..**

6. **The microcontroller will not clear the interrupt request flag (i.e., issue CFLGIR) until it has set the flow control bit in SSR. It will not clear the flow control bit until it has taken the incoming command from CMR and any related parameters from OP0 - OP3.** This rule in conjunction with rule 7 ensures that the host will not overrun the microcontroller, causing commands or input parameters to be lost.

7. **Before writing any input parameters to OP0 - OP3 or any command to CMR, the host will first check HS1 to make sure the interrupt request flag (IR) is not still asserted due to a prior interrupt. After IR is found to be inactive, the host will check SSR to make sure the flow control bit is clear, signifying that the microcontroller code has taken any previous commands and parameters from the I/O registers.**

8. **For any command that has an input vector associated with it, the microcontroller will not clear the interrupt request flag (i.e., issue CFLGIR) until it has set the flow control bit in SSR. It will not clear the flow control bit until it is ready to receive and process the vector.** Commands affected include INPUTLOAD, COLUMNWRITE, and LEARNVECTOR. This rule in conjunction with rule 9 ensures that the host will not write the input vector too soon.

9. **For any command that has an input vector associated with it, before loading the vector into IRAM, the host will check HS1 to make sure that the interrupt request flag (IR) is not still**

asserted, then check SSR to make sure the flow control bit is clear, signifying that the microcontroller code is ready to receive the vector. Commands affected include INPUTLOAD, COLUMNWRITE, and LEARNVECTOR. *This rule does not apply to vectors sent to IRAM for classification after the CLASSMODE command has completed.* A derivative of rules 7, 8, and 9 is that no command can be queued behind a command with an associated input vector until after the vector has been written to IRAM.

10. **For any command that uses the DIM register, the host will not change the contents of the DIM register until the command is finished.** The contents of DIM are used in determining when IRAM and ORAM are full. This rule implies that if commands are queued, no more than one command that uses the DIM register can be in the command queue at any given time, unless the same value is needed in DIM.

11. **For any command, the host will clear any input registers (OP0 - OP3) not used by that command to 0.** This rule is to ensure upward compatibility of existing driver code in the likely event of future expansion of parameter lists for any existing commands.

12. **When the PPRAMREAD command is executing on the chip, the host will not write any input parameters to OP0 - OP2 or write any new commands to CMR until the PPRAMREAD command is finished.** OP0 - OP3 are normally reserved for input parameters. No commands can be queued behind PPRAMREAD, because it uses OP0 - OP2 for output. This is actually an advantage, because the host can do a PPRAMREAD-modify-PPRAMWRITE without having to shuffle the data around.

13. **At any given time, the host should not queue more than one command using the same I/O registers for output.** That is, the host should not put two RAMREAD commands, or two PPRAMREAD commands, on the queue. Doing so will not cause an error, but output data is not queued. Therefore, the data from all but the last RAMREAD or PPRAMREAD will be lost.

7.6.6. Command Opcodes

The host issues a command to the microcontroller by writing a command opcode to the CMR register. In the current microcontroller software design, legal opcodes are 5 bits wide, ranging from 0 to 31. The following command opcodes are currently defined:

00: READCONFIG
This command causes the microcode to output chip configuration information through ORAM. Before issuing READCONFIG, the host must make sure the RCEB bit in the CRA register is set to 1. (Otherwise the BAD_RCEB_VALUE error code will be returned.) The microcode loads DIM bits 13-8 with the number of 16-bit ORAM words - 1 and leaves ORAM in auto mode so the host can access the data. The format of the configuration data in ORAM is as follows. (For more detail, see the Configuration Table Data section.)

ORAM word 0:	Microcode ID
ORAM word 1:	Microcode Revision Number
ORAM word 2:	Undefined
ORAM word 3:	Undefined
ORAM word 4:	Paradigm used to learn the currently active network
ORAM word 5:	Number of classes in the active network
ORAM word 6:	Number of prototypes in the active network
ORAM word 7:	Number of features in the active network
ORAM word 8:	Maximum Radius used in learning the active network
ORAM word 9:	Minimum Radius used in learning the active network
ORAM word 10:	Sector Number for the active network
ORAM word 11:	Smoothing factor exponent offset
ORAM words 12-17:	Six more words of configuration information that are reserved in GRAM but not currently defined

01: CLASSMODE
This command puts the Ni1000 into classification mode. Before issuing CLASSMODE, the host sets the bits in CRA to the desired values. For instance, CRA bit 1 selects either probabilistic or RCE mode. (PRCE and PNN both use probabilistic classification.) The host writes the index of the last feature (i.e., the size of the input vector minus 1) to DIM bits 7-0. For probabilistic classification, OP0 is set to zero if results are desired in internal 16-bit mode, or nonzero for IEEE-32 mode. As part of the functionality of this command, microcode scans the network and loads the highest class number (i.e., the class count minus 1) into DIM bits 13-8.

After the CLASSMODE command has completed, the host can write input vectors into IRAM and read classification results from ORAM. The Ni1000 will be taken out of classification mode and put into microcontroller mode when the host issues any of the other commands.

02: SETCLOCK
The microcontroller must know what the clock period is so that it can erase and program PA flash memory. The host must write the actual clock period in nanoseconds into OP0 and issue this command, right after initialization. Any microcontroller code that erases or programs PA flash memory will not function properly until after SETCLOCK has been issued.

03: INPUTLOAD
The host uses this command to just load a vector into IRAM and let it remain there. DIM bits 7-0 contain the number of bytes minus 1.

04: COLUMNREAD
This command is used to read part of a column from the prototype array (between 1 and 128 rows) through ORAM. Before issuing COLUMNREAD, the host must make sure that the RCEB bit in the CRA register is set to 1. (Otherwise the COLUMNREAD command will return the BAD_RCEB_VALUE error code.) OP0 has the number of the column to read, and OP1 has the first row in that column to read. Each row in the PA has 10 bits of data. PA rows used for classification are always stored in 5-bit mode (as 5 bipolar bit pairs), so the data can be represented as a 5-bit value. Data can also be examined in 10-bit mode, but only half as many rows can be read with one COLUMNREAD command. OP2 has the mode in which the data is to be output: 0 for 5-bit mode, or 1 for 10-bit mode. DIM bits 13-8 must contain the index of the last 16-bit word to be read from ORAM. In 10-bit mode, this is the number of rows to be read minus 1; in 5-bit mode, it is the number of rows to be read divided by 2, minus 1. In 10-bit mode, each row of ORAM will have one PA row in the low order 10 bits. In 5-bit mode, each byte of output in ORAM will contain one PA row in the high order 5 bits. Note that depending on the mode, either two or four COLUMNREAD commands are required for the host to read an entire column (256 rows) from the prototype array.

COLUMNREAD also provides additional information that may be of interest to the host program. If the column is disabled, COLUMNREAD sets bit 10 of SSR to 1. If the column is reserved for 10-bit data, COLUMNREAD sets bit 11 of SSR to 1. (See the **PA Usage Field** section.) A COLUMNREAD in 5-bit mode verifies that the data in each row being read from the PA consists of 5 bipolar bit pairs (01 or 10). If that is not the case for any row being read, the least significant bit is set to 1 in the output byte containing the data for that row.

05: COLUMNWRITE
This command is used to write a vector via IRAM into a column, or part of a column, in the prototype array. If only part of a column is written, the rest of the column is unchanged. DIM bits 7-0 must contain the IRAM index of the last byte (i.e., the size of the input vector minus 1). OP0 has the number of the column to be written, and OP1 has the index in the PA column of the first row to be written. Each row in the PA has 10 bits of data. PA rows used for classification are always stored in 5-bit mode (as 5 bipolar bit pairs), so the data can be input as a 5-bit value. Data can also be input in 10-bit mode. OP2 has the mode in which the data is to be input: 0 for 5-bit mode, or 1 for 10-bit mode. In 5-bit mode, each byte of the input vector written to IRAM will contain one row in the high order 5 bits. In 10-bit mode, two vectors must be written to IRAM: the first vector has the high order 5 bits for each row in the high order 5 bits, and the second vector has the low order 5 bits for each row in the high order 5 bits. If 10-bit mode is used, the PA rows being written to must already have been erased; otherwise the COLUMNWRITE command will return the NOT_ERASED error code. This command is flexible enough to write a single row, or any number of rows in any area of the column, or the whole column.

06: COLUMNERASE
This command erases one column in the prototype array. If the other column in the same block is used, its contents will not be changed (i.e., the other column in the block will be saved and restored). OP0 has the number of the column to be erased.

07: BLOCKERASE
This command erases one block (two columns) in the prototype array. The block number in OP0 is the number of the first column. For instance, if OP0 contains 0, columns 0 and 512 will be erased.

08: PPRAMREAD
This command reads all PPRAM data for the column specified in OP0. The microcontroller writes the data from PPRAM1, PPRAM2, and PPRAM3 into OP0, OP1, and OP2, respectively.

09: PPRAMWRITE
This command loads the PPRAM data for the column specified in OP3. The data for PPRAM1, PPRAM2, and PPRAM3 must be provided in OP0, OP1, and OP2, respectively.

0A: RAMREAD
This command reads the contents of a mapped memory location from the Ni1000. The address must be provided in OP0. The microcontroller writes the data into OP5. This command should not be used to access internal Ni1000 control registers, the prototype array, or PGFLASH.

0B: RAMWRITE
This command writes the data from OP1 into the mapped memory location whose address is in OP0. Extreme caution should be exercised if this command is used. Some locations are protected by microcode, and microcode will respond to any attempt by the host to write to these via the RAMWRITE command with the RESTRICTED_ERROR error code. Currently the only protected locations are the command queue, the host message queue, and their pointers. These locations are in GRAM, mapped at hexadecimal addresses 1040 through 105B.

The host should not attempt to use this command to write to internal Ni1000 control registers, the prototype array, or PGFLASH. The internal Ni1000 control registers should be written to only by microcode routines that control the transition of the chip among the various modes of operation. Attempts to switch modes by using RAMWRITE to load the internal registers individually will not be successful. The primary reason for this is that microcode puts the entire chip into microcontroller mode each time it enters the RAMWRITE command (as is the case for all commands). To put the chip into classify mode, the host must issue the CLASSMODE command. The data that gets loaded into some of the internal control registers by CLASSMODE is derived from parameters in GRAM that the host can modify via RAMWRITE. (See **Important GRAM Locations**).

0C: LEARNBEGIN
This command begins a learning session. It initializes the prototype array by marking all columns unused in PPRAM, then loads parameters that are to be used during learning. The host writes the size of the input vectors (#bytes - 1) to DIM bits 7-0. OP0 specifies the learning paradigm that is to be used. (Values currently defined are 1 for RCE/PRCE, and 2 for PNN.) OP1 must contain the smoothing factor, which is used for probabilistic classification. OP2 and OP3 contain the minimum and maximum values, respectively, for the threshold radius (the size of the influence field of a prototype, used for RCE classification). LEARNBEGIN is a general purpose command that is designed to be the first step for learning in any paradigm. For RCE/PRCE, a typical learning session looks like this:

(LEARNBEGIN, LEARNVECTOR, LEARNVECTOR, . . . , LEARNVECTOR);
(LEARNEPOCH, LEARNVECTOR, LEARNVECTOR, . . . , LEARNVECTOR);
.
.
.
(LEARNEPOCH, LEARNVECTOR, LEARNVECTOR, . . . , LEARNVECTOR, LEARNEND)

The same set of learning vectors is repeated for each epoch. After each epoch, the host looks at a flag in the SSR register. If that flag indicates that the network did not change during that epoch, learning is completed. For PNN, each learning vector is committed as is, and only one pass is taken through the learning vectors, so the SSR flag that indicates whether the network changed is ignored by the host. PNN learning looks like this:

(LEARNBEGIN, LEARNVECTOR, LEARNVECTOR, . . . , LEARNVECTOR, LEARNEND)

0D: LEARNVECTOR
This command learns from one learning vector according to the paradigm previously specified by LEARNBEGIN. The host should make sure that the OSR bit in the CRA register is set to 1 before issuing this command. The class number for the input vector is provided in OP0. The LEARNVECTOR command uses bit 3 of the SSR register to inform the host when it commits a prototype. If no prototype is committed, SSR bit 3 is reset to 0 when LEARNVECTOR completes. For multichip learning, bit 15 of OP0, when set, tells the learning algorithm not to commit the prototype. When bit 14 is set, the learning algorithm clears a region in the feature space around the input vector.

During PRCE learning with very a large learning vector set, it is conceivable that a count field in PPRAM may increment past the maximum value (65535) and wrap to 0. If such a condition were ignored, it would result in great loss of accuracy in classification. The way the microcontroller software handles this condition allows simple host programs to get reasonably accurate results in all but the rarest of cases (i.e., when a count is incremented far past the limit or when many counts overflow). It also makes it possible for more sophisticated host software to compensate for the overflow. The LEARNVECTOR command sets bit 9 in SSR when a count is incremented from FFFEh to FFFFh (i.e. 65534 to 65535). A count is not allowed to roll over, so once at FFFFh it remains there, without causing additional overflows. If the host software chooses to correct the counts, it must monitor SSR bit 9 after each LEARNVECTOR command. Every time the bit is set, the host must use PPRAMREAD to locate the offending prototype(s) by finding FFFFh in the count field. The host then uses PPRAMWRITE to set the count to 0 and proceeds with learning. Once learning has completed, the counts of the vectors that overflowed indicate the number of additional increments past FFFF. The host may then add duplicate prototypes for the prototypes that overflowed, with their counts set to FFFFh.

0E: LEARNEPOCH
This command starts a new learning epoch by clearing all count fields in PPRAM.

0F: LEARNEND
This command ends a learning session, performing whatever post-processing is implemented for learning in the paradigm previously specified in LEARNBEGIN. LEARNEND does not currently do anything, but it should be included at the end of a learning session because functionality may be added in future revisions.

10: RESTORE
This command restores the contents of PPRAM for all prototypes used by a network, along with some other information about the network, from an area in the prototype array. The sector number of the network to be restored is in OP0.

OPCODE	COMMAND	INPUTS	OUTPUTS
00	READCONFIG	none	XIR: 00000 DIM[13:8]: (# of bytes / 2) - 1 ORAM: configuration data
01	CLASSMODE	DIM[7:0]: dimension OP0: IEEE-32 conversion flag	XIR: 00001 DIM[13:8]: class count - 1
02	SETCLOCK	OP0: clock period	XIR: 00010
03	INPUTLOAD	DIM[7:0]:IRAM dimension IRAM: input vector	XIR: 00011
04	COLUMNREAD	DIM[13:8]: ORAM dimension OP0: column # OP1: first row # OP2: mode - 0 for 5-bit; 1 for 10-bit	XIR: 00100 ORAM: PA column data
05	COLUMNWRITE	DIM[7:0]: IRAM dimension OP0: column # OP1: first row # OP2: mode - 0 for 5-bit; 1 for 10-bit IRAM: input vector(s)	XIR: 00101
06	COLUMNERASE	OP0: column #	XIR: 00110
07	BLOCKERASE	OP0: block #	XIR: 00111
08	PPRAMREAD	OP0: prototype #	XIR: 01000 OP0: PPRAM1 data OP1: PPRAM2 data OP2: PPRAM3 data
09	PPRAMWRITE	OP0: PPRAM1 data OP1: PPRAM2 data OP2: PPRAM3 data OP3: prototype #	XIR: 01001
0A	RAMREAD	OP0: address	XIR: 01010 OP5: data
0B	RAMWRITE	OP0: address OP1: data	XIR: 01011

OPCODE	COMMAND	INPUTS	OUTPUTS
0C	LEARNBEGIN	DIM[7:0]: dimension OP0: learning paradigm OP1: smoothing factor OP2: Minimum Radius OP3: Maximum Radius	XIR: 01100
0D	LEARNVECTOR	OP0: class # IRAM: input vector	XIR: 01101
0E	LEARNEPOCH	none	XIR: 01110
0F	LEARNEND	none	XIR: 01111
10	RESTORE	OP0: sector number	XIR: 10000

7.7. PROTOTYPE ARRAY MANAGEMENT

This section describes some concepts related to how the Ni1000 microcontroller software manages the prototype array. These include the Bad Column Table, sectors, the PA Configuration Table, backed up networks, and the PA Usage field in PPRAM.

7.7.1. Bad Column Table

The Bad Column Table (BCT) on each Ni1000 contains information specific to that particular chip. It should be saved to a neuron file in case it is ever accidentally erased. It contains the bad column information generated during wafer sort and finalized at package sort. It is stored in the last usable column in the prototype array (usually column 1023). The other column in the BCT block (normally column 511) is not available for storage of prototypes. The host may store a backup copy of the BCT in this other column. Initialization microcode reads the BCT and marks any bad columns Disabled in PPRAM.

To avoid possible problems during repetitive programming and erasing, the BCT has data for bad blocks, not just bad columns. If one column is bad, then the other column is unusable and also considered bad. In the BCT's Bad Block Map, each block is represented by two bits, and one PA row stores the data for four blocks. The Bad Block Map occupies 128 rows, representing all 512 blocks. Each row has the following format:

Bits 9-8:	Number of bad blocks in this group of four: 00 (0), 01 (1), 10 (2), 11 (3 or 4)
Bits 7-0:	Data for four blocks (for example: 0 through 3, with 3 in the low order bits)
	10 Bad block
	01 Good block
	00, 11 Only occurs if BCT is corrupted

Rows 0-1 and 144-145 contain defined patterns that are not likely to occur by accident. These serve to identify the Bad Column Table.

The row by row format of the Bad Column Table is as follows:

0	270h -- Defined bit pattern to identify this column.
1	18Fh -- Defined bit pattern to identify this column.
2-4	Serial Number
5-6	Sorted Date
7	Reserved
8	Number of bad blocks
9-15	Reserved
16-143	Bad Block Map
144	190h -- Defined bit pattern to identify this column.
145	26Fh -- Defined bit pattern to identify this column.
146-255	Bad Column List (two entries for each bad block)

7.7.2. Sectors

The PADCU_ARR register determines the starting row and column numbers for the area of the prototype array being used by the currently active network. This effectively divides the prototype array into 64 "sectors", where each sector can potentially be the starting point for a network. Depending on the number of features and the number of prototypes, the prototype array can potentially hold features for up to 64 networks. Sectors are numbered as shown.

	0	128	256	384	512	640	768	896
0	0	8	10	18	20	28	30	38
32	1	9	11	19	21	29	31	39
64	2	A	12	1A	22	2A	32	3A
96	3	B	13	1B	23	2B	33	3B
128	4	C	14	1C	24	2C	34	3C
160	5	D	15	1D	25	2D	35	3D
192	6	E	16	1E	26	2E	36	3E
224	7	F	17	1F	27	2F	37	3F

A network can occupy more than one sector; in fact, that is normally the case. The sector number for a network is the number of the sector whose first row and column are the first row and column of the network.

The default sector number is 0, and the sector number currently must be 0 for learning. PADCU_ARR is not accessed directly by the host. To set the sector number, the host writes the number to GRAM at location 1062 hex. The microcode sets PADCU_ARR according to that GRAM location when it enters classification mode.

If there are multiple networks on the chip, switching from one network to another also involves loading PPRAM with the data for that network and restoring other parameters associated with the network (e.g., the number of features, column number of the column after the last used prototype, learning paradigm, etc.). Even if there is only one network, all of these things have to be reloaded after the chip is powered up or reset. (The features in the prototype array are nonvolatile and do not have to be reloaded.)

If there is enough space, all of the volatile information for a network can be backed up in the prototype array itself, and then the RESTORE command can be used to activate a specific network, reloading all of its volatile parameters including the sector number. The RESTORE command requires a PA Configuration Table (see next section).

7.7.3. PA Configuration Table

The Prototype Array Configuration Table (PACT) contains useful information about all networks present in the chip. The PACT is never written to by the Ni1000 microcontroller software. If it exists, the host is responsible for keeping it up to date. The PACT occupies one block (two columns) in the PA, and it will be in the last usable block before the one containing the BCT. For example, if the BCT is in column 1023 and the PACT exists, it will be in columns 510 and 1022 (provided that block 510 is not disabled).

On initialization, the microcode sets the PA usage field in each PPRAM entry according to the information in the PACT. The RESTORE command uses the PACT to locate the backed up parameters for a specific network. The format of the PACT is as follows:

7.7.3.1. High Column

0 18Fh -- Defined bit pattern to identify this column.
1 190h -- Defined bit pattern to identify this column.
2 Number of networks in the PA (0-64).
3-7 Reserved.
8-15 Boolean "sector taken" bit map -- 64 bits. A sector is "taken" if columns are used for prototypes starting with the first column in the sector. One implication is that the sector is not available for learning.

8	x	x	0	8	10	18	20	28	30	38
9	x	x	1	9	11	19	21	29	31	39
10	x	x	2	A	12	1A	22	2A	32	3A
11	x	x	3	B	13	1B	23	2B	33	3B
12	x	x	4	C	14	1C	24	2C	34	3C
13	x	x	5	D	15	1D	25	2D	35	3D
14	x	x	6	E	16	1E	26	2E	36	3E
15	x	x	7	F	17	1F	27	2F	37	3F

16-31 Reserved.
32-63 Backup Block List. A list of all blocks (except the BCT and PACT blocks) used for storage of data in 10-bit mode. Each entry in the list occupies two rows, specifying a range of blocks. The end of this list is marked by a 0, 0 entry.
First row: First block number (the number of the low column).
Second row: Number of consecutive Backup blocks (bad blocks aren't counted).
64-127 Row Usage Table. For each sector, one row at row number (64 + sector number).
Specifies only the number of rows used by 5-bit data (i.e., prototypes). This number includes Backup data in 5-bit mode (method 1), if any. Any rows used within a sector must start with the first and have no gaps.
128-255 Network Identification Table.
For each sector, two rows of information starting at row number (128 + 2 * starting sector number).
First row: 10-bit Network ID assigned by host (if 0, no network starts here).
Second row: 10-bit CRC generated by host.

7.7.3.2. Low Column

The low column contains a Network Information Table. For each sector, there are four rows of information starting at row number (4 * starting sector number). This information is not relevant unless the Network Identification Table in the high column indicates that there is a network with that sector number.

First row Bit 9 Backed up in PA? (0 = No, 1 = Yes).
Bit 8 Backup method (0 = method 0, 1 = method 1).
Bits 7-0 Row number where Backup data begins.
Second row Column number where Backup data begins.
Third row Number of features in network. (Does not include features used for method 1 backup data.)
Fourth row Number of prototypes in the network.

7.7.4. Backed Up Networks

7.7.4.1. Backup Header

If the host system chooses, and if there is enough space, volatile information associated with networks can be backed up in unused areas of the prototype array. PA columns are dedicated to Backup storage a block (2 columns) at a time. It will not be legal to use one column for 5-bit data and use the other column in the same block for 10-bit data. If more than one column is used for backup data for any one network, the last row of each Backup column (excluding the last) will point to the next column where the data is continued.

For each network that is backed up in the PA, the Backup information begins with 16 rows of data describing the network as a whole. (The host can get this information for the active network via the READCONFIG command.) This 16-row block of data is referred to as the Backup Header. When stored in PA, the information starts at the row and column number indicated in the PA Configuration Table.

row 1:	Learning Paradigm (10 bits)
row 2:	Maximum Radius[12:10] | Minimum Radius[12:10] | Smoothing Factor Exponent Offset (4 bits)
row 3:	Maximum Radius[9:0]
row 4:	Minimum Radius[9:0]
row 5:	Parameter A[15:8] | 00
row 6:	00 | Parameter A[7:0]
row 7:	Parameter B[15:8] | 00
row 7:	00 | Parameter B[7:0]
row 9:	Parameter C[15:8] | 00
row 10:	00 | Parameter C[7:0]
row 11:	Parameter D[15:8] | 00
row 12:	00 | Parameter D[7:0]
row 13:	Parameter E[15:8] | 00
row 14:	00 | Parameter E[7:0]
row 15:	Parameter F[15:8] | 00
row 16:	00 | Parameter F[7:0]

7.7.4.2. PPRAM Format

In addition to the Backup Header, the contents of the PPRAM entries for the prototypes in the network are backed up in the PA. In PPRAM, the data has the following format:

PPRAM3[15:0]:	Count (unsigned, 16 bits)
PPRAM2[15:14]:	PA Usage field
	00: unused column
	01: column contains data in 5-bit format
	10: column reserved for 10-bit format data
	11: reserved encoding
PPRAM2[13]:	Disable bit, set to 1 for bad (unusable) columns
PPRAM2[12:0]:	Threshold Radius (13 bits)
PPRAM1[15:8]:	Smoothing Factor (8 bits)
PPRAM1[7]:	Used bit
PPRAM1[6]:	Probabilistic bit
PPRAM1[5:0]:	Class number (6 bits)

7.7.4.3. Backed Up PPRAM Data

When PPRAM data is backed up in the prototype array, only 45 bits of each 48-bit entry are saved. The PA Usage field and the Disable bit are not saved, since the information they contain can be inferred from the fact that the entry is backed up. (It would not be necessary to save the Used bit either, but it is saved anyway for simplicity's sake. The microcontroller will force the Used bit to 1 when the network is restored, regardless of what was saved.) There are two methods for backing up the PPRAM data. The method used is indicated in the PA Configuration Table.

7.7.4.3.1. Method 0

Store each PPRAM entry in 10-bit mode in 4.5 rows of a 10-bit PA column. The PPRAM entries are saved in consecutive order starting immediately after the Backup Header. Only PPRAM entries for the prototypes are saved; i.e., there are no gaps in the storage for disabled columns or 10-bit columns. The PPRAM data for two prototypes fills up 9 rows of a PA column, as follows:

row 1:	prototype 1 PPRAM3[9:0]
row 2:	prototype 1 PPRAM3[10] | PPRAM2[12:4]
row 3:	prototype 1 PPRAM1[15:10] | PPRAM2[3:0]
row 4:	prototype 1 PPRAM1[9:0]
row 5:	prototype 1 PPRAM3[15:11] | prototype 2 PPRAM3[15:11]
row 6:	prototype 2 PPRAM3[9:0]
row 7:	prototype 2 PPRAM3[10] | PPRAM2[12:4]
row 8:	prototype 2 PPRAM1[15:10] | PPRAM2[3:0]
row 9:	prototype 2 PPRAM1[9:0]

Pairs of PPRAM entries are saved in this format until the backup column has less than 10 empty rows left. Then the column number of the next column, where the PPRAM data will be continued starting in row 0, gets written into row 255. It is okay if a few rows are left unused between the last PPRAM pair in a backup column and the pointer to the next backup column, as long as it is less than 9 unused rows. A pair of PPRAM entries will not straddle a column boundary, even if there are more than 4 but less than 9 rows available at the end of a column. There will never be a single backed up PPRAM entry by itself, unless the network has an odd number of prototypes and this is the last one.

Assuming that the Backup information for a network begins at row 0 of a backup column (it doesn't have to), the first Backup column can hold the Backup Header and up to 52 PPRAM entries, and each backup column after that can hold up to 56 PPRAM entries.

7.7.4.3.2. Method 1

Store each PPRAM entry in 5-bit mode in 9 rows of its own PA column, immediately following the last programmed weight for this network. This method might be preferred if there are some unused rows after the last weight in a network, especially if you can fill in 9 more rows without crossing a sector boundary. It

saves columns, because only the 16 rows of Backup Header information need to be stored for this network in a dedicated backup column. Use the following format:

row 1: PPRAM3[15:11]
row 2: PPRAM3[10:6]
row 3: PPRAM3[5:1]
row 4: PPRAM2[12:9] | PPRAM3[0]
row 5: PPRAM2[8:4]
row 6: PPRAM1[15] | PPRAM2[3:0]
row 7: PPRAM1[14:10]
row 8: PPRAM1[9:5]
row 9: PPRAM1[4:0]

7.7.5. PA Usage Field

In the Ni1000 microcontroller software, bits 15-14 of PPRAM2 are defined as the PA Usage field. They do not affect the hardware. This field indicates whether the corresponding PA column contains any meaningful data:

00	This column is currently not used for anything.
01	At least one row in this column has been programmed with 5-bit data.
10	The host has reserved this column for data in 10-bit mode.
11	Reserved.

The host is not permitted to alter the PA Usage field directly. Microcontroller software ignores the corresponding bits in the input data for PPRAMWRITE, and for RAMWRITE when the target address is in the PPRAM2 address range. (The host can override this restriction by setting bit 0 of the diagnostic word in GRAM at address 106Ah, but that is not recommended.) Revision 2.0 microcontroller software maintains the PA Usage field as follows:

1. PA columns must be reserved for 10-bit data a block (2 columns) at a time. 10-bit and 5-bit data are not permitted in the same block of PA.

2. Initialization microcode sets the PA Usage field to 10-bit (10) for both columns in the BCT and PACT blocks, and for both columns in each block that is reserved for 10-bit data in the Backup Block List in the PACT.

3. Initialization microcode sets the PA Usage field to 5-bit (01) for each column that contains prototype features for any network. This information is also derived from the PACT.

4. Initialization microcode sets the PA Usage fields for all other columns to unused (00).

5. LEARNBEGIN clears the PA Usage fields of *all columns on the chip*, except 10-bit (10) and disabled columns.

6. LEARNVECTOR sets the PA Usage field of the column indicated by NEXT_PT to 5-bit (01). If the column in NEXT_PT is 10-bit (10) or disabled, it is skipped. Successive columns will continue to be skipped until one is found which is 5-bit (01) or unused (00), up to the end of the PA. If no such column is found, a CHIP_FULL error is returned.

7. A 5-bit COLUMNWRITE sets the PA Usage field of the indicated column to 5-bit (01), provided the column is currently marked unused (00) or 5-bit (01). The other column in the block is not affected. The write is not performed and generates an error if the column is marked 10-bit (error BAD_BIT_MODE), or disabled (error DIS_COLUMN_ERROR).

8. A 10-bit COLUMNWRITE sets the PA Usage fields of *both* columns in the block to 10-bit (10). The write is not performed and generates an error if *either* column in the block is 5-bit (01) or disabled. Furthermore, the target column must be in an erased condition (i.e. all 0's) before issuing this command, otherwise NOT_ERASED is returned.

9. BLOCKERASE sets the PA Usage fields of both columns in the block to unused (00), regardless of current usage. An attempt to erase a disabled block generates an error (DIS_COLUMN_ERROR).

10. A COLUMNERASE of a 5-bit column sets its PA Usage field to unused (00), unless the column is disabled (an error is returned if so). The PA Usage field of the other column in the block is unaffected.

11. A COLUMNERASE of a 10-bit column does not affect the PA Usage field. Both columns in the block remain 10-bit. Unlike 5-bit COLUMNERASE, all bits of the other column in the block are preserved (see note below).

7.7.5.1. Cautionary Note

Revision 2.0 of the microcontroller software assumes all 256 rows of a column marked 5-bit or 10-bit contain valid data. Therefore, whenever IRAM is used to save and restore a column or parts of a column,

any rows that were erased (000h) will become 155 hex. This value still translates to a zero in 5-bit mode, but increases the amount of subsequent data-handling for the microcode. The following operations can give rise to this situation:

1. Writing to a 5-bit column. If a subset of the column is written, any 10-bit zeroes (000h) in the rest of the column become 5-bit zeroes (155h). If the other column in the block is marked 5-bit, any 10-bit zeroes (000h) in it will become 5-bit zeroes (155h).

2. Erasing a 5-bit column. If the other column in the block is marked 5-bit, any 10-bit zeroes (000h) in it will change to 5-bit zeroes (155h).

7.8. ADDING CUSTOMIZED MICROCODE

Nestor's standard microcode is, to some extent, modular. This makes it possible to choose subsets of the implemented commands, opening up space in PGFLASH for customized functions. The microcode source currently resides in one required file and four optional files:

- **NIUCODE** – This is the main file. It contains the initialization code, the dispatcher loop, and the following commands: READCONFIG, CLASSMODE, INPUTLOAD, COLUMNREAD, PPRAMREAD, PPRAMWRITE, RAMREAD, and RAMWRITE.

- **PAPROG** – The SETCLOCK command and all of the low level routines that are needed for programming or erasing the prototype array. ***This file must be included if either LEARNRBF or CLMNWR is included.***

- **LEARNRBF** – LEARNBEGIN, LEARNVECTOR, LEARNEPOCH, and LEARNEND commands.

- **CLMNWR** – COLUMNWRITE, COLUMNERASE, and BLOCKERASE commands.

- **RESTORE** – The RESTORE command.

8. GLOSSARY

Bayes Rule—A statistical approach used in pattern classification when overlap exists among the fields of influence of prototype classes. The mathematical form of the *Bayes* rule is: $P(\omega|x) = p(x|\omega)\ P(\omega)\ /\ p(x)$, where the *a priori* probability $P(\omega)$ and the conditional probability density $p(x|\omega)$ are known. In the context of the Ni1000 Accelerator, ω is a classification class, and x is an input pattern. *Bayes* rule shows how the input pattern x changes the *a priori* probability $P(\omega)$ to the *a posterior* probability $P(\omega|x)$. Under this rule, input patterns are assigned to the class with the highest *PD*, $P(\omega|x)$.

City Block Distance—Also called "Manhattan Distance", or "L1 Norm" in mathematical terms, a measure of the distance between an input vector and a prototype vector. Absolute differences between corresponding components of the two vectors are summed to form this distance. "City Block" refers to the orthogonality in the distance computing. It is used in the Ni1000 Recognition Accelerator to simplify the implementation.

Class—A category into which prototypes are grouped and input patterns are classified. The Ni1000 Recognition Accelerator supports up to 64 classes.

Class Firing—If at least one prototype of a class fires (see *prototype firing*), the class is said to fire. Each prototype has a class ID number to indicate the class to which it belongs.

DCU—One of the distance calculation units. There are 512 DCUs in the Ni1000 Accelerator.

Deterministic Radial Basis Function—A radial basis function that does not overlap with an RBF of another class so that classifications done with it are deterministic. Compare *Probabilistic Radial Basis Function.*

Deterministic RBF—See *Deterministic Radial Basis Function.*

Dimension—The number of components in the input and prototype vectors. The Ni1000 Recognition Accelerator supports vectors with up to 222 components (or dimensions).

Feature Space—The complete range of possible patterns of input data. A point in a multidimensional feature space corresponds to an input vector. Classification defines regions within the feature space. The Ni1000 Accelerator supports feature space of up to 222 dimensions and combines small units of feature space into complex regions representing *classes.*

Field of Influence—A region in the *feature space* associated with each prototype, subsequently with each class, and indicated by the value of the *threshold distance.* An input pattern is within the field of influence of a class if its *city block distance* to one of the prototype vectors in the class is less than the threshold distance of that prototype.

Flash EPROM—Electrically erasable and programmable read-only memory. Unlike conventional EPROM which requires physical removal from the computer and UV exposure, erasing and programming on the flash memory may be done in-system by applying high electrical voltage to the memory cells. The contents of the flash memory are preserved after power-down. The Ni1000 Recognition Accelerator uses flash memory to store microcontroller program (*PGFLASH*) and prototype array (PA).

GRAM—General-purpose random-access memory, used by the on-chip microcontroller of the Ni1000 Accelerator.

Incremental Learning—Addition of new prototypes to the existing base and/or adjustment of parameters. This is the normal learning mode on the chip, assuming initialization has occurred, at least one prototype is stored, and some training had been done.

Influence Field—Same as *field of influence.*

IRAM—The on-chip input buffer random access memory.

Lambda (λ)—Same as *threshold distance.*

Learning—Also called training or adaptation, a process of selectively choosing a set of prototypical vectors from the training data and adjusting appropriate parameters associated with those vectors. Ideally, the prototypical vectors chosen are best indicators of output based on input. The parameters associated with each prototypical vector include the threshold distance for RCE, the amplitude constant and decay constant of the exponential distribution function for PRCE.

Manhattan Distance—See *city block distance.*

MC—The on-chip microcontroller.

Minimum Threshold Distance—A prescribed global constant, intended to limit the number of prototypes selected in the learning process by forcing each prototype to have a minimum region of influence in the pattern space.

MU—The on-chip mathematical unit.

MURAM—The on-chip mathematical unit RAM. There are two MURAMs.

Neural Network Classifier—An implementation of an algorithm that accepts input patterns and outputs classification information. The Ni1000 Recognition Accelerator is such a classifier that uses the *RCE* and/or *PRCE* algorithms.

Neuron—In biology, a cell in the brain that produces an output signal in response to multiple input signals. In the Ni1000 Recognition Accelerator, it is a structure that computes the *city block distance* between an input vector and a pre-stored prototype vector. For *RCE,* this distance is then compared with the radial threshold distance associated with the prototype vector to produce a binary-valued output. For *PRCE,* this distance is used to compute a floating-point number which enters the *PDF* calculation.

ORAM—The on-chip output buffer random access memory.

PA—The on-chip prototype array.

PADCU—*PA* and *DCU*s.

Parzen-Windows—A technique to compute probability density functions. It assumes that within a small region of the feature space, the density function does not vary appreciably, and the probability that a pattern of class C falls within the region is simply the number of vectors in class C in the region, Kc, divided by the total number of vectors in the feature space. In this technique, $Kc = \Sigma\ \phi(p0 - p(k))$, where the sum is over all patterns p(k) in class C, and p0 is the center of the region. The window function $\phi(f - f(k)) = 1$, if $|f - f(k)|$ is less than a threshold value, and 0 otherwise. The threshold value defines the region of estimation.

Pattern—An input vector to be classified. The Ni1000 Recognition Accelerator can accept input vectors with up to 222 dimensions, each of which has a 5-bit resolution.

PDF—See *probability density function.*

PGFLASH—Flash memory used in the Ni1000 Accelerator to store the program for the microcontroller. Default microcontroller program supports *RCE* or *PRCE,* and *PNN* algorithm. PGFLASH is erasable and programmable. See *Flash EPROM.*

PNN—See *probabilistic neural network.*

PPRAM—The on-chip prototype parameter random-access memory. There are three PPRAMs to store, for each prototype vector, its class ID, decay constant, receptive field radius, count, and flags.

PRCE—See *probabilistic Restricted Coulomb Energy.*

Probabilistic Neural Network—A pattern-recognition algorithm that classifies patterns using probability distribution and *Bayes rule.* All training patterns are stored and used to estimate the probability density functions. Learning is rapid (one pass through the training set) and the continuous-valued (Gaussian) estimators perform spatial averaging, resulting in improved PDF estimates in regions of low sample density. A drawback is that memory usage is inefficient so that large data sets require large networks.

Probabilistic Radial Basis Function—A radial basis function that overlaps with an RBF of another class so that classifications done with it are probabilistic. Compare *Deterministic Radial Basis Function.*

Probabilistic RBF—See *Probabilistic Radial Basis Function.*

Probabilistic Restricted Coulomb Energy—A pattern-recognition algorithm that combines *RCE* and *PDF* estimation. The Ni1000 Accelerator computes simultaneously the RCE (firing class IDs) and PRCE (PDFs) results, and the user can select to output either or both. The classification decision is made using *Bayes rule* in the case of multiple firing classes.

Probability Density Function—A specification of the dependence of prototype-classes distribution on an input pattern, used with *Bayes rule* to determine the class to which the input pattern belongs. Functional forms can vary, and are chosen to be the sum of decaying exponentials in the Ni1000 Accelerator.

Prototype—A stored vector that serves to represent the typical features of the patterns to be classified. The Ni1000 Recognition Accelerator supports up to 1000 such stored vectors of 222 *dimension*s each, and up to 8000 stored vectors of 26 dimensions or less. These vectors are selected from the training data set during the *learning* process, and are grouped into up to 64 *class*es.

Prototype Firing—An indication of a match between an input vector and a stored prototype. Firing occurs when the input vector is within the *influence field* of the prototype.

Radial Basis Function—A radially symmetric function that has a maximum at some point in its input space and that falls off to zero rapidly at large distances from that point. The region around the symmetry point can be thought of as an influence (or receptive) field since input vectors which fall near this point will result in non-zero response from the RBF logic. The RBF used in the Ni1000 Accelerator is a decaying exponential function.

RBF—See *radial basis function.*

RCE—See *Restricted Coulomb Energy.*

Restricted Coulomb Energy—A pattern-recognition algorithm that is supported by the Ni1000 Accelerator. Training is supervised and selective, in the sense that not all training vectors are committed as prototypes. Several passes of the entire training set may be required. Classification is done using a set of *RBF*s for the prototype vectors. The advantage is that *RCE* may not require as large a network as *PNN.*

Threshold Radius—A parameter associated with each prototype that defines the prototype's field of influence. It is determined during the learning process. When the *city block distance* of an input vector to the prototype is less than the threshold radius, a match is found and the prototype fires. See also *field of influence*, and *minimum threshold distance.*

9. INDEX

+

+12 Volt Programming Supply **3-37**
+5 Volt Memory Supply **3-37**
+5 Volt Supply **3-37**

6

64/32# 3-36
64-Bit or 32-Bit Data Bus **3-36**

A

A[0:15] 3-35
Abus 4-1
Access Modes 4-3, 5-1
Active-Low Signal Names x
Address Bus 4-3
Address Strobe **3-35**
ADS# 3-35
AIO 4-3
Alignment 5-16
Assembler 6-1

B

Bad Column Table 5-44
Bayes Decision Theory 2-7
BERR# 3-36
BLAST# 3-35
BRDY# 3-36
Burst Last **3-35**
Burst Ready **3-36**
Bus Cycles 4-5
  Burst 4-5, 4-6, 4-9, 4-11, 4-12
  Definition and Control Signals 4-7
  I/O-Register Read or Write 4-6, 4-8
  IRAM Burst Write 4-11, 4-12
  IRAM Non-burst Write 4-9, 4-10
  Non-burst 4-5, 4-6, 4-8, 4-9, 4-10
  ORAM Burst Read 4-12, 4-13
  ORAM Non-burst Read 4-11
  PGFLASH Read or Write 4-8, 4-9
  Reset 4-13, 4-14
Bus Error **3-36**, 4-7
Bus Interface 3-15
Bus Operations 4-1
Buses 3-24, 4-1, 4-2
  ABUS *3-24*
  AIO 4-3
  Data I/O *4-1*
  DBUS *3-24*
  DIO 4-1, 4-3
  Internal Address *4-1*
  Internal Data *4-1*
  PABUS *3-24*, *4-1*, 4-4
  PDBUS *3-24*, 4-1, 4-4, 4-5
  Program Address *4-1*
  Program Data *4-1*

C

Chip Select **3-35**
City-block distance 2-5
City-block distances *3-5*
Class 2-1
Classification 2-4, 5-64
Classification Timing 3-32
Classifier 3-1, 3-3
CLK 3-35
Clock **3-35**
CMR 5-4
CMR register 4-1, 4-5, 4-13
Computational Precision 3-22
Control and Status Registers **5-45**
CRA 5-11
CRB 5-12
CS# 3-35

D

D[0:63] 3-35
Data **3-35**
Data Bus 4-3
Data I/O Bus *4-1*
Dbus 4-1
DIM 5-4
DIO 4-3
DIO Bus 4-1
Distance Calculation Units 3-1, 3-4

E

Error **3-36**
ERROR# 3-36
Errors 3-32
Exponential decay 2-7

F

Feature space 2-3
Feature vector 2-1
Features 1-1

Field of influence 2-4
Floating Point Format
16-Bit Internal 3-23
32-Bit IEEE 3-24
Floating-point format 3-2, *3-16*
16-bit internal 5-21
32-bit IEEE 5-21
Format
RCE Classification Results 5-20

G

General-purpose RAM 3-2
GRAM 5-25, **5-36**
Ground **3-37**

H

Hardware Architecture 3-1
Hardware Setting Registers **5-46**
Hardware-Controlled Access Modes 4-1, 4-3
HS1 5-6
HS2 5-7

I

I/O Registers *3-15*, *3-17*, 5-3, 5-25
CMR 5-4
CRA 5-11
CRB 5-12
DIM 5-4, 5-5
HS1 ..... 5-6, 5-7
HS2 ..... 5-7, 5-8
IDR 5-5
IIR 5-10
OP 5-13
SSR 5-5, 5-6
XIR 5-9, 5-10
IACK# 3-36
Identification Register 5-5
IDR 5-5
IIR 5-10
Instruction Sequences
Classification 5-64
PGFLASH Programming **5-33**
PPRAM Access 5-59
Prototype Array Access 5-51
Internal Address Bus *4-1*
Internal Address Map 5-24
Overview 3-24
Internal Data Bus *4-1*
Interrupt 5-37, 6-57
Interrupt Acknowledge **3-36**
Interrupts 3-31
IRAM 3-2, 3-4, *3-15*, *3-18*, 5-14, 5-26
Access Addresses 5-14
Address assignment 5-15
Pre-Write Latch 3-19, 5-15
Software-controlled modes 5-14
Write by the Host 5-17

L

Learning 2-4, 5-67
Little-Endian Convention x

M

Manhattan distance 2-5
Math Unit 3-1, *3-4*, 3-10
Pipeline 3-11
Math Unit RAMs *3-4*, 3-13
MC# 3-36
MCINT# 3-36
Memory and Register Address Map 5-25
Microcontroller 3-1, 3-24, 3-36
Addressing Modes 6-2
Architecture 3-24
Flags 3-28, 6-1, 6-2, 6-5, 6-48
Instruction Set 3-28, 6-7
Interrupt 5-37, 6-57
Program Memory 3-29
Registers 3-27, 6-1
Microcontroller flags: 6-48
Microcontroller Instructions 3-28
Conditional Jumps 6-4
Data Transfer Operations. 6-6
Flag Operations 6-5
Flags Cross Reference 6-48
Flags Cross-Reference 6-48
Math and Logical Operations 6-7
Stack Operations 6-5
Subroutine Calls 6-5
Summary 6-4
Microcontroller Instructions: 6-7
Microcontroller Interrupt **3-36**
Modes
Access 4-3
Hardware-Controlled Access 4-3
NORMAL **4-1**, 4-3, 4-4, 4-5, 4-6, 4-9, 4-10
PG **4-1**, 4-3, 4-4, 4-5, 4-8
RESET **4-1**, 4-5
MULTCHIP# 3-36
Multichip Add-In Board 1-3
Multi-Chip Operation **3-36**
Multichip systems 3-32
MURAMs *3-4*, 3-13, 5-27, 5-60
Class List MURAMs *3-10*, 3-13, 3-15, 5-61
Class-List MURAM 5-62
Flag MURAM *3-10*, 3-13, 5-61
MURAM_CR Register 5-64
Probability MURAMs3-10, 3-13, 3-15, 5-61, 5-63

registers 5-27
Software-controlled mode 5-63

N

NORMAL **4-1**, 4-3, 4-4, 4-5, 4-6, 4-9, 4-10
NORMAL mode 5-1
Notation x

O

OP 5-13
ORAM 3-2, *3-4*, 3-13, *3-16*, 3-21, 5-18, 5-26
Access Addresses 5-18
Bit assignment 5-19
Output Possibilities 5-20
Pre-Write Latch 3-22, 5-19
Read and Write by the Microcontroller 5-21
Read by the Host 5-22
Retrieving Both Class and Probabilistic Data by the Host 5-23
Software-controlled modes 5-18

P

PAbus 4-1, 4-4
PADCU Registers 5-26
Pattern Recognition 2-1
PDbus 4-1, 4-4, 4-5
PDF 2-1
PG **4-1**, 4-3, 4-4, 4-5, 4-8
PG mode 5-1
PGFLASH. 3-1, 3-29, **4-1**, 4-4, 4-5, 4-8, 4-13, 5-28
Erase **5-34**
Erase Verify **5-34**
PGF_ADR Register 5-29
PGF_CR1 Register 5-30
PGF_CR2 Register **5-32**
PGF_DR Register 5-30
PGF_SR Register 5-33
Program **5-35**
Program Verify **5-36**
Read **5-34**
Registers 3-29, 5-27
Software-controlled mode 5-28
Standby **5-33**
Pipeline 3-10, 3-33
PNN 1-1, 2-5
PPRAMs 3-7, 5-27, 5-55
Read and Write by the Microcontroller 5-59
registers 5-27, 5-58
Software-controlled mode 5-57
Used flag 5-57
Word Format 3-8, 5-56
PRCE 2-4, 2-6
Principles Of Operation 2-1
Probabilistic Neural Networks 1-1, 2-5
Probabilistic RCE 2-4, 2-6
Probability Density Function 2-1, 2-7
Program Address Bus *4-1*
Program Data Bus *4-1*
programming 6-1
Prototype Array 3-1, 3-4, *3-5*, 5-28
Access 5-42
ARR Register 5-48
AUX Register 5-47
Bad Column Table **5-43**, 5-44
CSA Register 5-46
CSB Register 5-46
DCU_DIM Register 5-51
Erase by the Microcontroller 5-54
MODE Register 5-47
NCA Register 5-51
NCB Register 5-51
Programming 3-7
Programming by the Microcontroller 5-53
Read by the Microcontroller 5-52
Segmentation 5-49
Software-controlled modes 5-43
Prototype Parameter RAM *3-4*, *3-7*

R

Radial Basis Function 2-4
RBF 2-4
RCE 2-4
RDY# 3-35
Registers
I/O Registers 5-3
MURAM register 5-64
PADCU Registers **5-45**, **5-46**, 5-47, 5-50
PGFLASH registers 5-29
PPRAM registers 5-58
Reilly-Cooper-Elbaum algorithm 2-4, 2-5
Reset 3-30, 3-37, **4-1**, 4-3, 4-5
RESET mode 5-1
RESET# 3-37

S

Service Request **3-36**
Signal Descriptions 3-35
SRQ# 3-36
SSR 5-5
System-Level Architecture 3-32

T

Threshold radius 2-5
Timer 3-30, 5-25, **5-36**, **5-37**
Training set 2-1, 2-4

V

V_{cc} 3-37
V_{cx} 3-37
V_{pp} 3-37
V_{ss} 3-37

W

W/R# 3-35
Write or Read **3-35**

X

XIR 5-9