

Copyright © 1997, by the author(s).  
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

**SPEECH RECOGNITION FOR PORTABLE  
MULTIMEDIA TERMINALS**

by

Andrew Joseph Burstein

Memorandum No. UCB/ERL M97/14

21 February 1997

**SPEECH RECOGNITION FOR PORTABLE  
MULTIMEDIA TERMINALS**

Copyright © 1996

by

Andrew Joseph Burstein

Memorandum No. UCB/ERL M97/14

21 February 1997

**ELECTRONICS RESEARCH LABORATORY**

College of Engineering  
University of California, Berkeley  
94720

Abstract

# Speech Recognition for Portable Multimedia Terminals

by

Andrew Joseph Burstein

Doctor of Philosophy in Engineering – Electrical Engineering and Computer Science

University of California at Berkeley

Professor Robert W. Brodersen, Chair

Portable multimedia terminals will require speech recognition for users to perform command and control operations. The speech recognition may either be implemented in software in computers on a high speed network, or in low-power hardware in the terminal itself. To facilitate the writing of applications using speech recognition, a new extension to the tcl language, along with a toolkit of classes, widgets, and applications, provide an easy to program API and a consistent user interface. This thesis also presents the architecture for low-power hardware to perform speech recognition and describes a custom CMOS IC that implements an important part of the speech recognition algorithm.

Robert W. Brodersen

Chairman of Committee

*To my parents,  
Myra and Albert Burstein.*

---

# TABLE OF CONTENTS

---

<b>Table of Contents</b> .....	iv
<b>List of Figures</b> .....	vi
<b>List of Tables</b> .....	ix
<b>Acknowledgments</b> .....	x
<b>CHAPTER 1. Introduction</b> .....	1
1.1 InfoPad and Hand-Held Devices .....	1
1.2 Speech Recognition in InfoPad .....	3
<b>CHAPTER 2. Algorithm</b> .....	7
2.1 Goals .....	7
2.2 RASTA-PLP .....	8
2.3 Hidden Markov Model .....	11
2.4 Multi-Layered Perceptron .....	17
2.5 Clustered Grammar .....	24
2.6 Recognition Results .....	28
2.7 Summary .....	30
<b>CHAPTER 3. Software:</b>	
<b>User Interface and Applications</b> .....	31
3.1 Goals .....	31
3.2 Widgets in the SpRcg Toolkit .....	32
3.3 Speech Controlled Applications .....	36
3.4 Combining Handwriting and Speech Recognition .....	43

<b>CHAPTER 4.</b>	<b>Software:</b>	
	<b>Programmer's Interface</b> .....	46
4.1	Overview & Goals .....	46
4.2	Writing Applications for InfoPad .....	48
4.3	Low Level Implementation .....	50
4.4	High Level Objects and Procedures: spRcg_gramDict .....	54
4.5	Mega-Widgets .....	56
4.6	GramCracker: A Tool to Build Vocabularies and Grammars ..	62
4.7	Design Example .....	66
<b>CHAPTER 5.</b>	<b>Low Power Memory Design</b> .....	77
5.1	Overview .....	77
5.2	Sram .....	78
5.3	How to use the Memories .....	99
<b>CHAPTER 6.</b>	<b>Low-Power Speech Recognition Architecture</b> .....	103
6.1	Goals .....	103
6.2	Previous Work .....	103
6.3	Next Generation: Low Power .....	104
6.4	Front End .....	108
6.5	MLP .....	109
6.6	Viterbi .....	110
<b>CHAPTER 7.</b>	<b>Low-Power Multi-Layer Perceptron Design</b> .....	121
7.1	Architecture .....	121
7.2	Summary of Low Power Techniques .....	128
7.3	Selecting Bit Sizes .....	130
7.4	Interface .....	130
7.5	Results .....	132
<b>CHAPTER 8.</b>	<b>Conclusion</b> .....	136
8.1	Contribution .....	136
8.2	Discussion & Future Work .....	136
<b>Bibliography</b> .....		139
<b>Appendix: Directories</b> .....		142

---

# LIST OF FIGURES

---

<b>CHAPTER 1.</b>	1
Figure 1-1. Overview of InfoPad.	2
Figure 1-2. Difficulty of various speech recognizer types.	4
<b>CHAPTER 2.</b>	7
Figure 2-1. Choice of components in the SpRcg system.	8
Figure 2-2. The role of front end signal processing.	9
Figure 2-3. Block diagram of RASTA-PLP.	10
Figure 2-4. Hierarchy in a hidden Markov model.	12
Figure 2-5. 10 state hmm for the word "oh."	15
Figure 2-6. Multi-layered perceptron to compute phoneme probabilities.	19
Figure 2-7. Hierarchy in the HMM.	25
Figure 2-8. Clustered grammar reduces complexity.	25
Figure 2-9. Example of a grammar cluster.	27
<b>CHAPTER 3.</b>	31
Figure 3-1. spRcg_NBest widget.	32
Figure 3-2. SpRcg_controller widget.	33
Figure 3-3. spRcg_adapter widget.	35
Figure 3-4. Grammar cascade.	36
Figure 3-5. Web browser main display.	37
Figure 3-6. Web browser control panels.	41
Figure 3-7. Speech controlled ic layout editor.	44
<b>CHAPTER 4.</b>	46
Figure 4-1. Design trade-offs.	47
Figure 4-2. Architecture of speech recognition software.	48



Figure 4-3.	Process for adding speech recognition to a tcl/tk application. . . . .	49
Figure 4-4.	Output of MLP for an utterance of “reload.” . . . . .	63
Figure 4-5.	GramCracker displaying grammar for a WWW browser. . . . .	64
Figure 4-6.	Grammar cluster dialog. . . . .	65
Figure 4-7.	Performing phonetic recognition in GramCracker. . . . .	66
Figure 4-8.	GramCracker using the full vocabulary. . . . .	67
Figure 4-9.	Demo grammar. . . . .	68
Figure 4-10.	Transcribed pronunciation of “square.” . . . . .	68
Figure 4-11.	“Square” with optional starting and ending silence. . . . .	69
Figure 4-12.	“Square” with optional “s.” . . . . .	69
Figure 4-13.	Listing of demo grammar and dictionary file. . . . .	70
Figure 4-14.	Creating the speech objects and widgets. . . . .	71
Figure 4-15.	Controlling recognition. . . . .	72
Figure 4-16.	Parsing the recognized sentences. . . . .	74
Figure 4-17.	The rest of the application source code. . . . .	75
Figure 4-18.	The finished demo application. . . . .	76
<b>CHAPTER 5.</b>	. . . . .	<b>77</b>
Figure 5-1.	SRAM memory cell schematic. . . . .	80
Figure 5-2.	SRAM senseamp and write circuit diagram. . . . .	81
Figure 5-3.	SRAM dummy word driver. . . . .	83
Figure 5-4.	SRAM self-timing circuitry. . . . .	84
Figure 5-5.	SRAM control circuits. . . . .	86
Figure 5-6.	Address latch and driver. . . . .	87
Figure 5-7.	Column select latch and driver. . . . .	88
Figure 5-8.	Tree-structured sram address decoder and row drivers. . . . .	89
Figure 5-9.	SRAM cell layout. . . . .	91
Figure 5-10.	SRAM block layout. . . . .	92
Figure 5-11.	Layout of 3 kWord x 32 bit sram integrated circuit. . . . .	93
Figure 5-12.	ROM address latch. . . . .	95
Figure 5-13.	ROM address decoder and wordline driver. . . . .	95
Figure 5-14.	ROM senseamp. . . . .	96
Figure 5-15.	ROM control circuits. . . . .	99
Figure 5-16.	Layout of 1024 word x 16 bit ROM block. . . . .	101
Figure 5-17.	Sample parameters to synthesize a ROM. . . . .	102
Figure 5-18.	Example irsim commands to simulate ROM. . . . .	102
<b>CHAPTER 6.</b>	. . . . .	<b>103</b>
Figure 6-1.	Previous UCB speech recognition hardware system. . . . .	104
Figure 6-2.	Comparison of hardware partitions. . . . .	105

Figure 6-3.	Low-power system architecture. . . . .	107
Figure 6-4.	SORASTA IC layout. . . . .	108
Figure 6-5.	Sorasta chip algorithm. . . . .	109
Figure 6-6.	Viterbi chip architecture and data flow. . . . .	110
Figure 6-7.	Dictionary subsystem memories. . . . .	112
Figure 6-8.	Phone subsystem memories. . . . .	113
Figure 6-9.	Grammar subsystem memories. . . . .	115
Figure 6-10.	Backtrace subsystem memories. . . . .	117
<b>CHAPTER 7.</b>	. . . . .	<b>121</b>
Figure 7-1.	MLP chip architecture. . . . .	122
Figure 7-2.	Datapath for MLP layers. . . . .	124
Figure 7-3.	State transition diagram of layer control. . . . .	125
Figure 7-4.	Input buffer control state diagram. . . . .	127
Figure 7-5.	Input buffer datapath. . . . .	128
Figure 7-6.	MLP chip layout. . . . .	134
Figure 7-7.	MLP chip bonding diagram. . . . .	135
<b>CHAPTER 8.</b>	. . . . .	<b>136</b>
Figure 8-1.	Partitioning recognition across the network. . . . .	138

---

## LIST OF TABLES

---

Table 2-1.	N-best digit recognition. . . . .	16
Table 2-2.	Breakdown of digit recognizer errors. . . . .	17
Table 2-3.	MLP output scoring by phoneme. . . . .	22
Table 2-4.	MLP Nth best results. . . . .	23
Table 2-5.	N-best digit recognition using spRcg. . . . .	29
Table 2-6.	SpRcg user test results. . . . .	29
Table 5-1.	Voltage scaling a 128 word 12 bit 0.6 $\mu\text{m}$ SRAM. . . . .	94
Table 5-2.	SRAM speed and energy vs. size at 1.5V. . . . .	94
Table 5-3.	ROM speed & energy vs. size and voltage. . . . .	100
Table 6-1.	Comparison of memory sizes. . . . .	106
Table 6-2.	Memory sizes for various viterbi configurations. . . . .	119
Table 7-1.	MLP ROM sizes. . . . .	129
Table 7-2.	MLP RAM Sizes. . . . .	129
Table 7-3.	MLP chip outputs. . . . .	130
Table 7-4.	MLP chip inputs. . . . .	131
Table 7-5.	MLP chip power supplies. . . . .	132
Table 7-6.	Parameter input values. . . . .	132
Table 7-7.	MLP chip power consumption. . . . .	133

---

## ACKNOWLEDGMENTS

---

*First and foremost I wish to thank my family. My parents Myra & Albert have given me unconditional love and support throughout my life. They deserve the credit for everything I have accomplished. My brother and sister-in-law Jeff & Cheryl always helped me to get away from it all on vacations, and lately provided a welcome distraction with my new nephew Gregory. I also wish to thank my grandparents, Selma & David Sole and Pauline & Aaron Burstein for never questioning how I could be 2 years away from graduation for 3 straight years. My only regret is my grandfather Aaron could not be with us at my graduation. My cousin (and friend) Beth Eilers has helped me survive this last, and hardest, year and a half. I am indebted to you all.*

*Next I wish to thank my many friends among my fellow graduate students. By far the best part of being at Berkeley was working and playing with this (by and large:-) great group of people. Among the many are Eric Boskin, who showed me it is possible to graduate. Sam Sheng, who didn't, but did help me out in just about every other way. Tony Stratakos and Dave Lidsky, who made me feel like a good teacher. Tom Burd, who expanded my vocabulary. The gang that's still haunting 550 Cory Hall — Jennie Chen, Shankar Narayanaswamy, Ian O'Donnell, Arthur Abnous, Craig Teuscher, Jeff Gilbert, Renu Mehra, Lisa Guerra, and Ole Bentz — and those who got away — Jane Sun, John Barry, Tony Stölzle, Monte Mar, Lars Thon, Sonia Sachs, and Anantha Chandrakasan to name a few — all made it worth while.*

*Several members of the faculty and staff have helped me greatly. Professor Jan Rabaey helped me to learn a great deal while TAing 141 and 241, and let me teach some of the lectures. Professor Nelson Morgan helped me with my speech recognition research and generously agreed to be on my Quals and Thesis committees. Likewise, Professor Charles Stone donated his time for my Quals and Thesis committees. Kevin Zimmerman, our indefatigable system administrator has somehow managed to keep our computers running without losing his sanity or sense of humor (as far as I can tell).*

*Finally, I wish to thank my research advisor, Robert Brodersen. He has provided me with the best support and resources that a grad student could ever hope for. What's more, he has taught me the importance of having and communicating a vision, a lesson I will use for the rest of my life.*

---

# 1 INTRODUCTION

---

This thesis describes the application of speech recognition to a new environment: hand-held, multi-media, wireless computing. This chapter will first describe the InfoPad project, a large research project that is exploring this new environment. Then, this chapter describes the need for speech recognition in hand-held terminals.

## 1.1 InfoPad and Hand-Held Devices

The recent explosive growth of the Internet and the World Wide Web (WWW) have shown that the ability to find and access information is at least as important as the ability to store information. The ability to create local copies of information on one's own hard disk is becoming increasingly irrelevant; the network itself is now the storage device. It makes more sense to retrieve the most up to date information from the source on demand, rather than store local copies that rapidly become obsolete and that fill expensive storage devices.

In addition to using the network to store data, it also makes sense to treat the network as a computational resource, rather than adding increasingly powerful processors and peripherals to portable computers. Attaching high speed computers to the network instead of placing them in portable devices has two advantages. The first is a reduction in power consumption and size. Portable devices are powered by batteries, so their hardware design must place as much emphasis on low power and low weight as it does on high computational throughput, leading to inevitable trade-offs. Non-portable hardware, in contrast, has much less severe penalties for power consumption and weight. Inevitably, non-portable hardware will be cheaper than portable hardware of the same computational power.

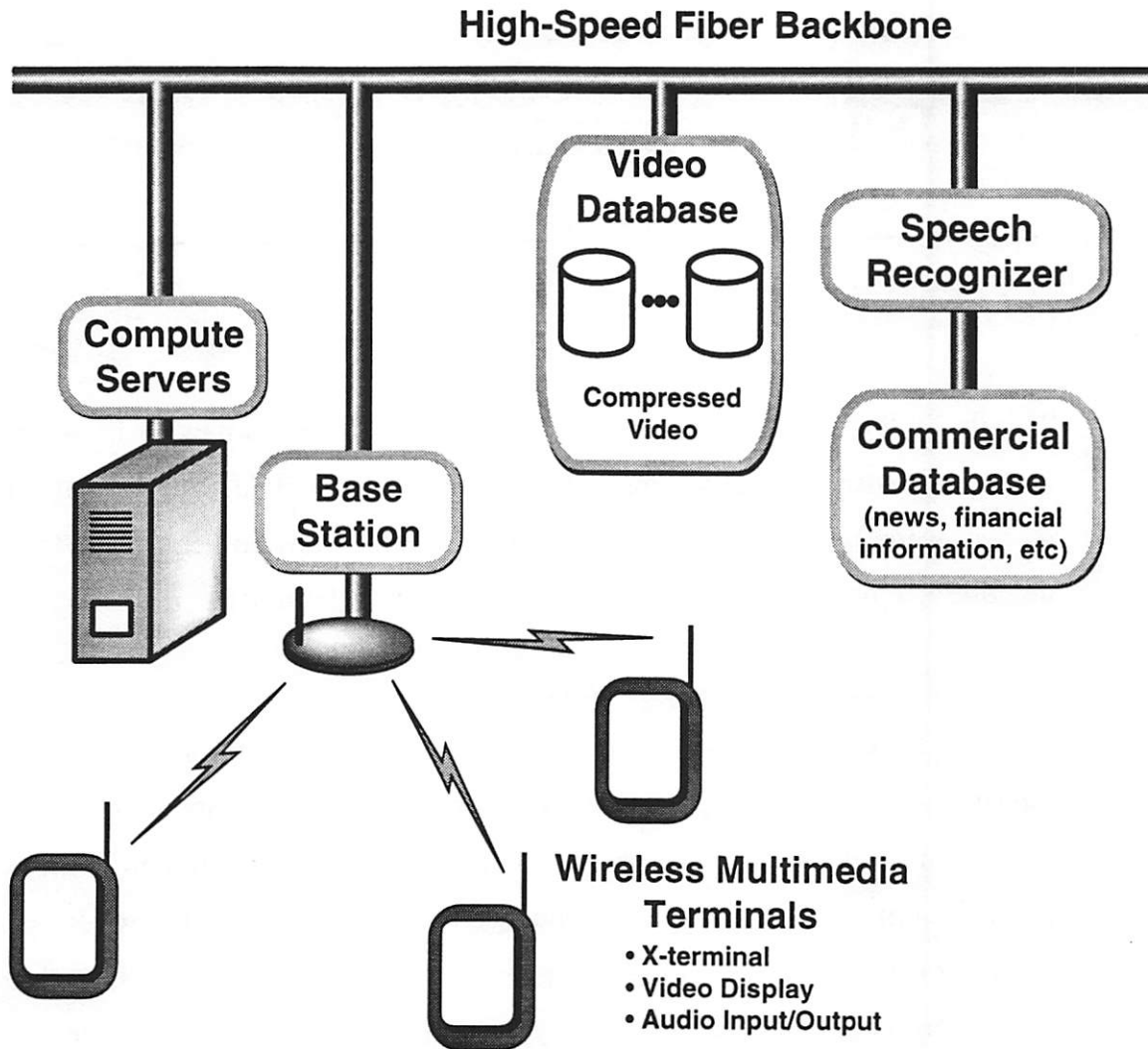


Figure 1-1. Overview of InfoPad.

The second advantage of attaching processing power to the network is availability. A processor in a laptop is available only to that laptop's user; further, it is the only processor available to that user. Most of the time, the user is not performing complex operations, so that processor is sitting idle, although it still adds to the weight, cost, and power consumption of the laptop. When the user does want to perform computations, s/he usually wants as much throughput as possible. On the other hand, if the user performs computations on a workstation on the network, other users can utilize that workstation's processor when it would be otherwise idle. Even better, by using the techniques demonstrated by the Network of Workstations (NOW) project [18] a user can utilize the power of many workstations

during those times s/he needs high computational throughput. Overall, leaving computational and storage assets on the network allows for the most economical design of hardware as well as the most cost effective sharing of resources.

With most or all storage and computational tasks left on the network, the portable device is reduced to an input and output device; connectivity and easy-to-use interfaces become paramount. The InfoPad system fulfills this role; it provides audio, text/graphics, and full motion video output; its input data types are pen and audio. InfoPad also has excellent network connectivity: pico-cell radio links provide 2 Mbit/second downlink bandwidth and 64kbit/second uplink bandwidth. See Figure 1-1 for an overview of InfoPad.

The InfoPad's I/O capabilities combined with its radio connectivity to network storage and computation make it seem to have the capabilities of a portable workstation with one interesting exception: it has no keyboard. The main reason for excluding a keyboard is size: the ultimate form factor for infopad will be tablet size, wafer thin, and entirely covered by the display. Another reason for excluding a keyboard is ease of use. Although many people use keyboards every day, most people are not prolific at typing; furthermore, an increasing number of those that are prolific are succumbing to repetitive stress injuries. In contrast, speaking and writing are nearly universal skills. A portable terminal with pen and speech input would be an ideal combination of small size and ease of use.

## **1.2 Speech Recognition in InfoPad**

### **1.2.1 Software**

Replacing a keyboard with a speech recognizer is a difficult task. Despite steady advances in the state of the art of speech recognition, speech recognizers are still nowhere near as skillful as humans are at transcribing spoken words into written text. In order to improve recognition accuracy, it is helpful to relax one or more of the constraints on speech recognition as shown in Figure 1-2. The first constraint is to use isolated words, forcing speakers to pause between words, as opposed to continuous speech, in which words may be slurred together. The second constraint is speaker dependent recognition, which requires extensive training on every new speaker, as opposed to speaker independent recognition, which requires no new training for new users. In between these two extremes are speaker adapt-



able systems, which will work acceptably well without training on new users, but which will perform even better with more training. The third constraint is the size of the vocabulary, which measures the number of words the recognizer can understand at any given time. The fourth and final constraint is the perplexity of the grammar, a measure of how many words can follow another word in a sentence.

This thesis describes the implementation of a speech recognition system, called spRcg, for

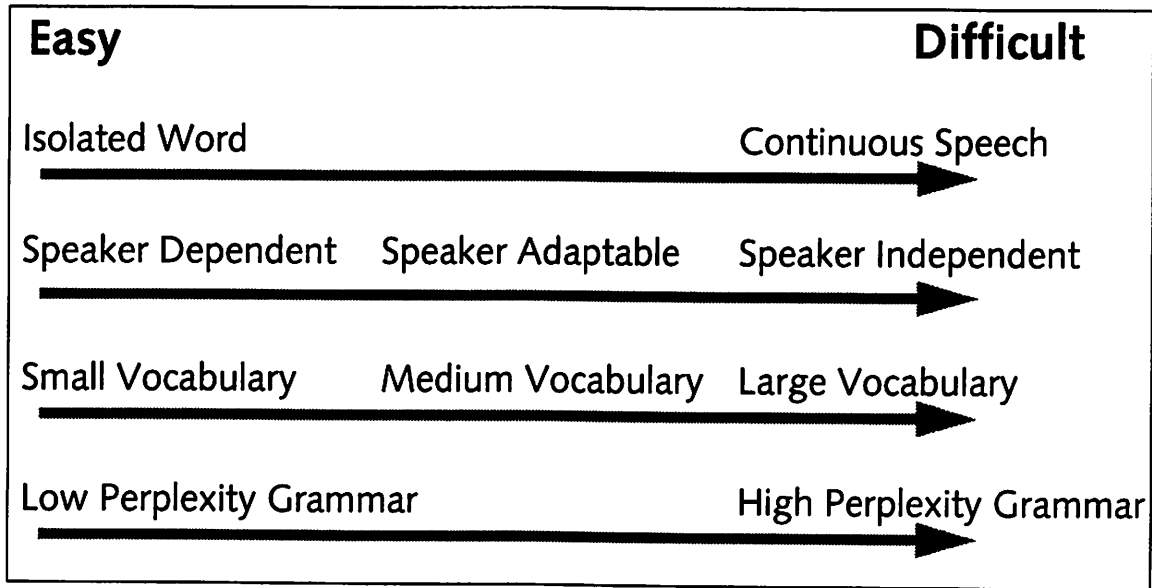


Figure 1-2. Difficulty of various speech recognizer types.

use with portable terminals such as the InfoPad. In terms of Figure 1-2, the recognizer accepts continuous speech, is speaker independent (with speaker adaptable capabilities), uses small to medium vocabulary, and is optimized for constrained (low perplexity) grammars.

Speech recognizers output whole words or, in the case of continuous speech recognizers, whole sentences; keyboards output individual ascii characters. Given their differences, it is to be expected that programs can not simply swap out keyboard input and swap in speech recognition input. Furthermore, in order to make the recognition task as easy as possible on the vocabulary size constraint and the grammar perplexity constraint, it is desirable to pass information from the application back to the speech recognizer. For

these reasons, this researching includes the development of programming interfaces and toolkits to allow programmers to include speech recognition into their applications with minimal of effort.

Using speech recognition (along with handwriting recognition) in applications represents a paradigm shift from the window, icon, mouse, pointer (WIMP) strategy. As part of the research effort into developing a new user interface for portable terminals, the speech recognition software toolkit includes methods that allow users to control the speech recognizer and to correct recognition errors. Furthermore, as a demonstration of this new software, this research includes an application program, a speech recognition controlled World Wide Web browser.

### **1.2.2 Hardware**

One major contribution of the InfoPad project has been the advancement of low power integrated circuit design techniques. Although the InfoPad model calls for performing the majority of all computations on the network, there must always be some computations performed locally on the pad itself. InfoPad performs such tasks as multiplexing and demultiplexing packets, decoding vector quantized video, and updating frame buffers on full-custom, low-power CMOS integrated circuits. In fact, the low power design techniques that were applied to these chips were so successful that they consume a total of less than 5 mW, a trivial portion of the InfoPad's power consumption. This lower power design was so successful, that future versions of InfoPad will be able to have increased computational throughput while still maintaining extremely low power consumption.

In keeping with the low power design research of the InfoPad project, this thesis also describes work done to design a low power, custom CMOS IC implementation of InfoPad's speech recognition algorithm.

Performing the recognition on the network is a practical solution, but there are also advantages to performing recognition in the portable device itself. The first advantage is bandwidth. Transmitting good quality speech to the network requires an uplink bandwidth of about 64 kbit/second. In contrast, transmitting pen data requires only about 2 kbit/second, so by far the majority of the uplink bandwidth is consumed by audio. If speech recognition

was performed on the InfoPad terminal, then the uplink would only need to transmit an ascii representation of the recognized words, needing only a few hundred bits/second. This reduction in uplink bandwidth can reduce the needed complexity and power consumption of the InfoPad's radio, and could also allow operation in regions with only low bandwidth radio links.

The second advantage is autonomy. If a future InfoPad has enough local computation to perform useful applications locally, as seems likely given current research into low power microprocessors [2][6], then an InfoPad could be used even when it is outside of an area that allows radio connections to the network. In these circumstances, the pad will need to perform speech recognition in order to be useful. However, general purpose microprocessors are not optimized for the tasks involved in speech recognition, so the lowest cost and lowest power implementation of speech recognition would be on custom hardware. One could also imagine the usefulness of a low power speech recognition chipset in devices other than InfoPad.

In addition to making a contribution to InfoPad and to speech recognition, the research described in this thesis also made contributions to the work in low power CMOS design. In particular, this thesis describes low power memory design techniques that are needed for low power speech recognition hardware, but which are also generally useful, having been incorporated in many of the InfoPad's custom chips.

---

## 2 ALGORITHM

---

### 2.1 Goals

Many researchers are working to improve the recognition accuracy of speech recognition algorithms. The goals of this research is to use these algorithms wherever possible and to adapt them to the software and or hardware requirements of portable recognition wherever necessary.

Throughout the speech recognition research community, there are algorithms that are commonly used. For example, the Viterbi algorithm on hidden Markov models is widely used. This research seeks to use these “state of the art” algorithms for two reasons. First, because using them produces good results. Second, to demonstrate that the hardware, programming, and user interface techniques developed in the course of this work are applicable to widely used algorithms.

There is not a consensus in the speech recognition community on the relative merits of competing algorithms or variations on algorithms. For example, there are several popular front end signal processing algorithms, including RASTA [8][9] and LPC based cepstral coefficients. Variations on hidden Markov models abound, especially in regard to topologies and to classes of output probability distributions. Generally, when these variations exist, they each provide roughly equivalent recognition accuracy, or their relative merits have not yet been decided.

Usually, “relative merits” refers to recognition accuracy, or perhaps efficient implementation on general purpose hardware. Implementing speech recognition in low power hard-

ware has a different set of criteria, so clear winners may be obvious in this context among otherwise equivalent algorithms.

Figure 2-1 shows the components in a typical speech recognition system, as well as the specific choices of algorithm used in the spRcg speech recognition system. The following sec-

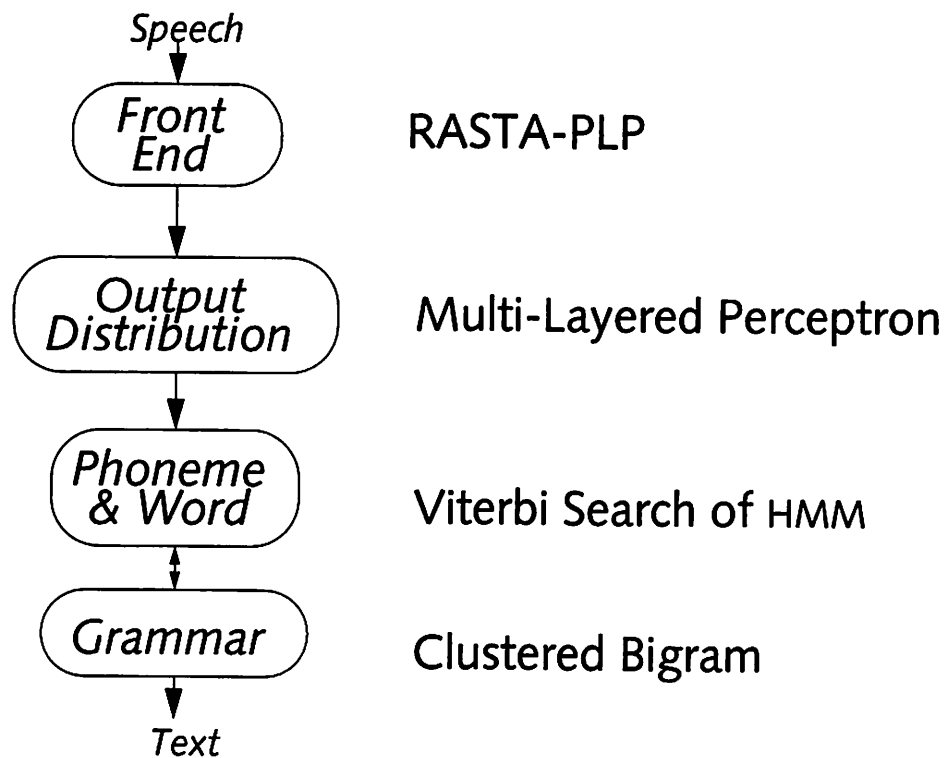


Figure 2-1. Choice of components in the SpRcg system.

tions describe each individual component.

## 2.2 RASTA-PLP

Speech recognizers use front end signal processing to reduce input data rates and to extract the parts of the audio signal that contain phonologically important information. The spRcg speech recognizer uses the RASTA-PLP [8][9] algorithm to perform front end signal processing

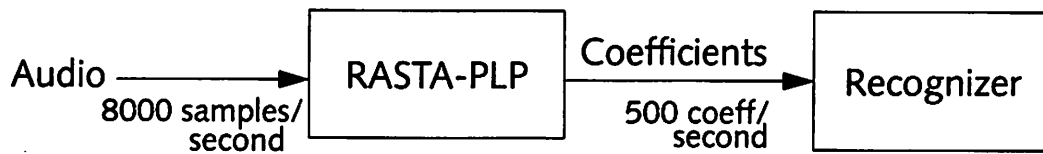


Figure 2-2. The role of front end signal processing.

The RASTA-PLP algorithm is based on the Perceptual Linear Predictive (PLP) algorithm, which applies psychoacoustic principles to an all-pole model of the auditory spectrum. The RASTA technique enhances the usefulness of PLP for speech recognition by removing the parts of the signal that are contributed by the acoustic channel - the microphone and room acoustics. These components of the signal contribute constant or slowly varying parts of each band, which are removed by the band pass filtering stage, as shown in Figure 2-3. Only the time varying part of the signal - which presumably contains the important parts of the speech signal, are left.

RASTA-PLP's advantages for the spRcg speech recognizer are twofold. First, the RASTA technique provides robust recognition even in varying acoustic environments. Since InfoPad is portable, it records speech in a variety of environments, with little control over the relative placement of the microphone. Furthermore, it is helpful to be able to train on standard databases, which contain speech recording under ideal circumstances and with expensive (and carefully placed) microphones, and then be able to recognize speech in real world conditions.

The second advantage is coding efficiency. RASTA-PLP can achieve equivalent recognition accuracy using fewer coefficients than can other common algorithms such as cepstral analysis. Where cepstral analysis of speech sampled at 8 kHz typically uses 8 coefficients, RASTA-PLP can perform well with 5. With fewer coefficients to process, the output probability computation (in the case of spRcg, the output probability multi-layered percep-

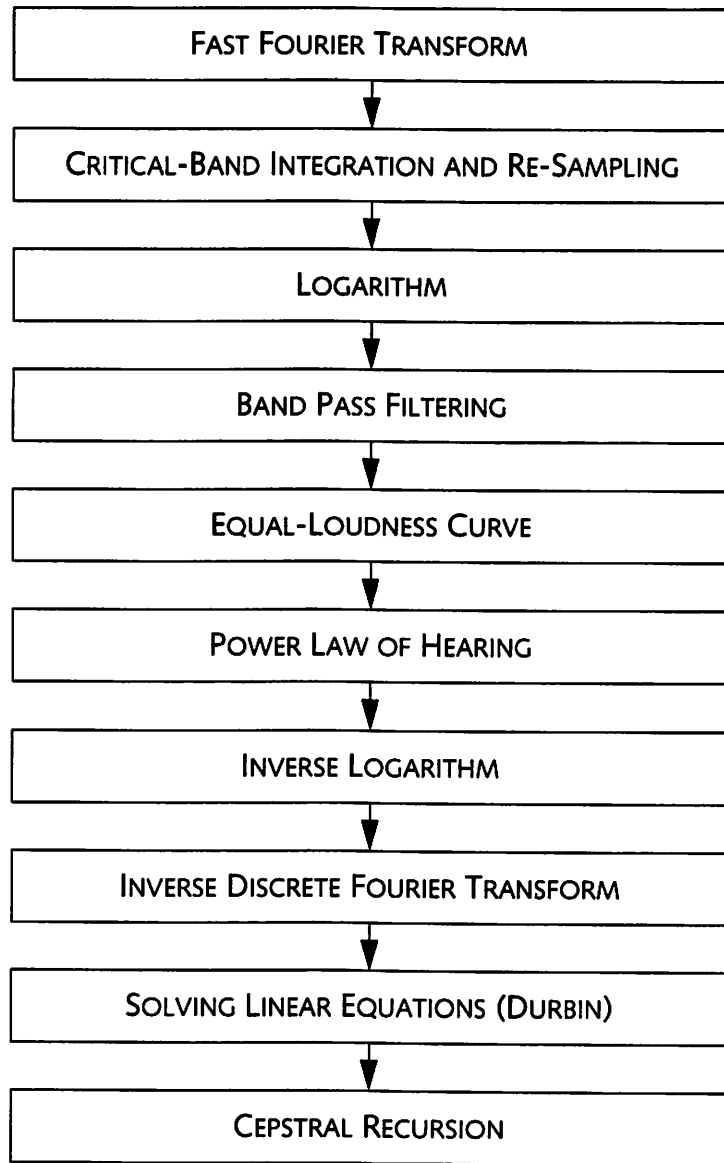


Figure 2-3. Block diagram of RASTA-PLP.

tron) needs to perform fewer computation and fewer memory accesses, thereby saving power consumption in special purpose hardware.

## 2.3 Hidden Markov Model

spRcg's speech recognizer is based on the hidden Markov model. Several excellent references describe hidden Markov models and their uses in speech recognition [11][12][19][26]. This section provides a brief overview, then presents the HMM used by the previous generation Berkeley speech recognition hardware as an example. The HMM in spRcg's speech recognizer is described later in this chapter.

### 2.3.1 Algorithm

The hidden Markov model (HMM) describes a model for the process of generating speech; the Viterbi algorithm is used to recognize speech by finding the most likely path through an HMM for any given utterance.

An HMM is a graph, containing states connected by edges, that represents spoken language. The act of speaking is modeled by the process of traversing this graph, from one state to another, according to a random process determined by probabilities associated with the edges in the graph. The actual sounds of speech, or at least some coefficients derived from these sounds, are randomly generated according to probability distribution functions that are attached to the presently occupied states. The Markov model is "hidden" because the outside world cannot see which state is currently occupied; the outside world only sees the coefficients emitted by the HMM. The task of the Viterbi algorithm is to infer the most likely sequence of states through the HMM, given a particular HMM and given a particular sequence of coefficients

An HMM is a model of a spoken language, or of part of a spoken language. The knowledge incorporated into an HMM can be described at three levels: phonetic, word, and grammar (Figure 2-4). At the lowest level, the phonetic level, the HMM describes, in a statistical fashion, how the individual speech sounds of a language relate to the coefficients used to encode their acoustic expression. Phonemes are linguistic categories used to describe all of the sounds in a language. Different languages have different phonemes; even the phonemes in a particular language are subject to interpretation. For example, different systems use slightly different lists to describe the phonemes in English. The same phoneme



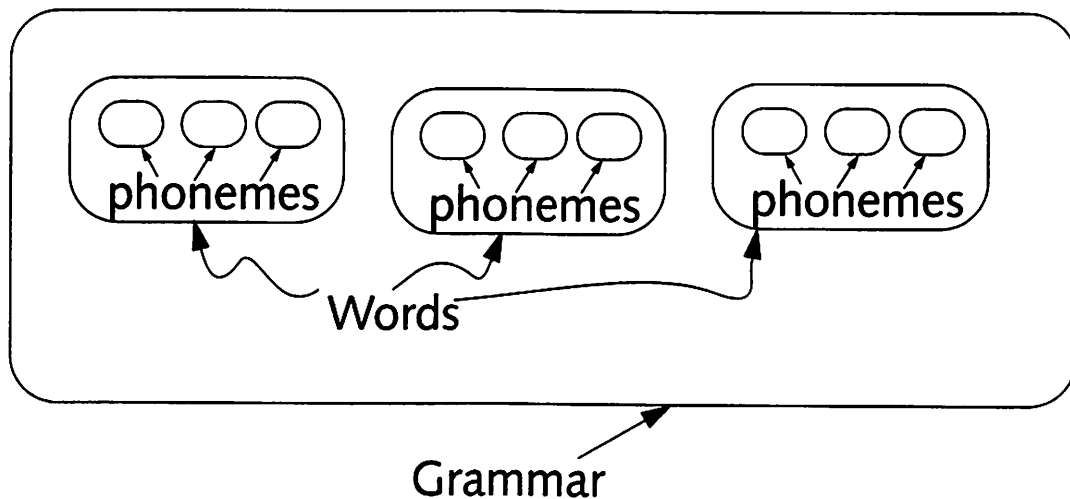


Figure 2-4. Hierarchy in a hidden Markov model.

can sound differently depending on such variable as where it occurs in a word, what are the surrounding phonemes, the speaker's accent, and the speaker's intonation.

Phoneme models are themselves small HMMs. They can be connected together with edges to assemble larger HMMs that model words. In turn, the words can be connected by edges to form large HMMs that model the entire spoken language (or at least that part of it that the recognizer can recognize).

Speech recognition systems often distinguish phonemes on a different basis than do linguists. Meaning is less important; acoustics are more important. For example, coarticulation, the influence of one phoneme on a neighboring phoneme's pronunciation, can have a major effect on the acoustics of a phoneme. Speech recognizers often use multiple models to represent the same phoneme depending on its neighbors. Biphones are models of phonemes depending on one of their neighbors; left biphones are identified by the preceding phoneme, right biphones, the following phoneme. A triphone is a model of a phoneme with a particular phoneme preceding it and a particular phoneme following it. Different speech recognition systems are often distinguished by their choice of which phonemes, biphones, and triphones they use, in addition to the topology of the HMM used to model individual phonemes[12].

In terms of recognition accuracy, there is a trade-off between using a larger number of more specific phonemes and a smaller number of more general phonemes. If properly trained, more specific phonemes can provide greater recognition accuracy. On the down side, the greater the number of phoneme models, the less training data there is for each phoneme, until ultimately some of the phoneme models would be under-trained.

An alternative approach to using phonemes as the basic building blocks of the HMM is to use words as the basic building block. The trade-off is one of flexibility versus accuracy. A phoneme based recognizer is more flexible because once the phonemes are trained, models of new words can be assembled from existing phonemes without requiring any new training. Word models can have higher recognition accuracy because they implicitly train all of the phonemes in their proper context. However, adding new word models to a recognizer requires new training (and worse, gathering new training data). Furthermore, word models are harder to train than phoneme models because they typically have a smaller training set. SpRcg's speech recognizer uses phoneme models so that applications can create new words without requiring training.

### **2.3.2 HMM for Berkeley Hardware**

This section describes a speaker independent, continuous word digit recognizer that was developed to produce models for the previous generation speech recognition hardware [13][24][25]. It serves as a case study and illustrates one implementation of a speech recognizer, (although it is somewhat different from the one currently used in spRcg). The main distinguishing features of this recognizer were its use of word models and its use of discrete output probability distribution functions.

It is important to examine this previous generation speech recognition system because its implementation uncovered the bottlenecks inherent in speech recognition hardware. Examining this system lead to the software choices described in this chapter and the hardware choices described in Chapter 6.

### **2.3.2.1 Data and Preprocessing**

Training and test data came from the TIDIGITs[28] speaker independent, connected digit corpus, which contains samples of speaker saying sentences containing between one and seven digits. Both male and female speakers were used for training and testing, but no juvenile speakers were used.

Speech data was subsampled from 20kHz to 16kHz, then processed with a mel-frequency weighted cepstral analysis, which weights frequency bands by their psychoacoustic importance. The resulting cepstral coefficients were vector quantized (VQ)[7] because the HMM used discrete output probability distribution functions. Therefore, a clustering algorithm used the training data to produce four codebooks: power, power derivative, cepstrals, and cepstral derivatives. The two cepstral codebooks have 256 entries; the power codebooks, 16.

The VQ codebooks were generated by a clustering algorithm as follows. Each vector was normalized so that all of its dimensions had the same standard deviation, so that each dimension was treated as equally important when performing vector quantization. Next, an initial codebook was selected by randomly choosing vectors from the test data. The rest of the test data was categorized according to their closest entries in the codebook; all of the data for a particular entry formed a “cluster.” After all the data was categorized, the geometric center of each cluster formed the entries in a new codebook. This clustering process was repeated iteratively until the net distortion, the sum of the euclidean distances from the vectors to their nearest codeword, stopped improving.

### **2.3.2.2 Models**

The vocabulary contains twelve words: the digits “one” through “nine,” as well as separate models for “oh” and “zero” (even though they represent the same digit) and an explicit model for pauses between words..(The pause model was helpful because the transcription of the data often had explicit pauses. The word models also had built in transitions, as described below, to help with brief silences between words that were not transcribed as pauses.)

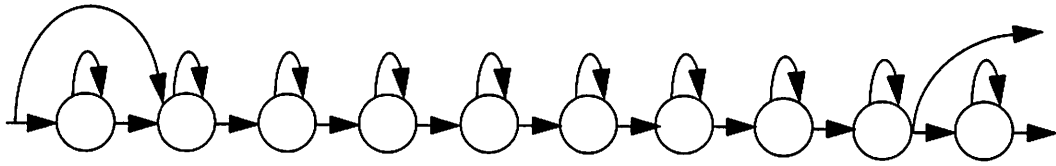


Figure 2-5. 10 state HMM for the word "oh."

Figure 2-5 shows the topology of the HMM of the word "oh." The states in the middle of the model have transitions to themselves and to the state immediately following. The first and last states are intended to model silence that may or may not occur between spoken words, so there are additional transitions—to the second state and from the second to last state—to bypass the first and last states (in case there is no silence). The topologies of the other digits' models are similar, except they have more states in the middle because the words are longer and contain more sounds. All the other digits' models have 15 states, except for "zero" and "seven" which have 20 states. The explicit silence model has two states.

Each state has four output probability distribution functions, one each for the cepstrals, cepstral derivative, power, and power derivative vectors. The first two contain 256 entries; the latter two, 16 entries. Since each quantized vector has its own output PDF, each vector is modeled as being independent of the others.

### 2.3.2.3 Training and Recognition

The models were iteratively trained using the forward backward algorithm[19] until their recognition results on the test set showed no improvement. When training on a particular sentence, the word models are assembled in the same sequence as they were spoken in the sentence. Thus, the training data must be labelled, but it does not need to be segmented because the forward-backward algorithm automatically aligns the models with the speech. For each training sentence, the same word models are re-assembled into a new sentence before training.

Unlike the training HMMs, the recognition HMM contains the word models in an almost fully connected configuration. Thus, a sentence may contain any number of words and any word may follow any other word, with one exception: the word “oh” may not follow the word “zero.” This rule eliminates an otherwise frequent error by the recognizer. It has little drawback because speakers generally use only one of the two versions of the digit 0 in a sentence. Otherwise, the grammar is completely uniform, with all word to word transitions probabilities identical.

Testing was performed on 28583 words contained in sentences of one to seven digits. In addition to Viterbi recognition, the models also used an N-Best algorithm to detect the 5 best sentence matches. Table 2-1 shows that using the two best sentences greatly decreases

N-Best	Number of Errors	Recognition Accuracy
1	478	.983
2	240	.992
3	191	.993
4	169	.994
5	154	.995

Table 2-1. N-best digit recognition.

the number of errors, with rapidly diminishing results with larger values of N. The number of errors is the total number of insertions, deletions, and substitutions, as determined by a dynamic programming algorithm comparing the correct answer to the recognized sentence. The model for silence is not included in the error calculations because it does not affect the meaning of the sentence.

The errors in the most likely sentence are broken down in Table 2-2. Errors are sorted by word and by type. Since substitutions involve two words, the missing word and the added word, each substitution is in the table twice: once for the missing word and once for the added word. The word “oh” caused the most problems, being responsible for almost one third of all the errors. The word “oh” is probably the most difficult to recognize in a sentence because it is so brief in duration that it is easily slurred into the pronunciation of preceding and following words.

Word	Insertions	Deletions	Substitutions (missing)	Substitutions (added)
oh	30	50	75	64
one	4	8	13	18
two	0	13	40	27
three	1	0	21	18
four	4	7	24	38
five	13	3	48	24
six	2	4	5	10
seven	0	5	13	7
eight	3	24	23	19
nine	6	5	20	64
zero	0	7	7	0
Total	63	126	289	289

Table 2-2. Breakdown of digit recognizer errors.

These digit models were converted to a format readable by the UCB speech recognition hardware and were used to test the functionality of the system.

## 2.4 Multi-Layered Perceptron

### 2.4.1 Overview

Every state in an HMM has an output probability distribution function (PDF) that maps input observations into probabilities. Instead of using conventional discrete or continuous probability functions, it is also possible to use a multi layered perceptron to perform this mapping. Using an MLP is not a radical departure - it is simply another statistical technique to compute probabilities based on a large set of training data. As a statistical technique, an MLP does have some advantages as part of a speech recognition system in general and as part of a low power system in particular.

A statistical model of speech should derive as much of its content as possible from training data. Applying outside constraints is fine as long as they are based on correct assumptions. However, constraints that are imposed to make the training and recognition task more computationally tractable will reduce the overall recognition accuracy because they prevent the models from identifying certain types of patterns in the data.

The first advantage of an MLP is its lack of a priori assumptions. Other statistical methods contain built in assumptions about the form and nature of the probability distribution, even before training begins. For example, consider a discrete probability distribution based on vector quantization. Vector quantization is based on some distortion metric that determines the relative importance of the coefficients. The choice of a distortion metric might be based on past experience or good heuristics, but inevitably it will assign importance to irrelevant data in some cases because all of the coefficients do not always have the same relative importance. Models that train on this vector quantized data will be limited in their accuracy because some of the quantization was based on unimportant data; the training process has no way to correct poor assumptions that are implicit in the distortion measure. For another example, consider a continuous probability distribution function made as a mixture of gaussians. Simply the choice of using gaussians places some constraints on the possible forms the PDF can take. Furthermore, many systems use diagonal covariance matrices (that is, assume the coefficients are independent random variables) in order to limit the parameters to a manageable number. However, speech coefficients are usually not independent of each other, so using a diagonal coefficient matrix causes the loss of valuable information—information an MLP could use.

A priori assumptions about the statistics of speech have a second drawback, one that is particularly important to low power hardware design. Namely, these assumptions force the recognizer to store a large number of redundant or irrelevant coefficients. For example, consider the PDF's included in a set of biphone models of a single phoneme in different contexts. These models should be somewhat different - which, after all, is the reason for having biphones in the first place. However, they should all have a great number of similarities since they model the same phoneme. Unfortunately, each phoneme must repeat this same information in its own PDF's, which means there must be more coefficients in total than are really needed.

For a more complete discussion of MLPs and their uses in speech recognition, see[1].

### 2.4.2 Algorithm

A Multi-Layer Perceptron (MLP) is a stateless model that produces a set of outputs based on a set of inputs. In the case of the speech recognition system, the inputs are the coefficients from a frame of speech data, as well its context, which contains the coefficients from several preceding and following frames. The outputs of the MLP correspond to  $P\langle phoneme|data \rangle$ . Because standard HMMs expect  $P\langle data|phoneme \rangle$ , the MLP output can be converted as follows:

$$P\langle data|phoneme \rangle = \frac{P\langle data \rangle}{P\langle phoneme \rangle} \cdot P\langle phoneme|data \rangle$$

(except standard HMM's PDFs usually do not use the context as part of their inputs).

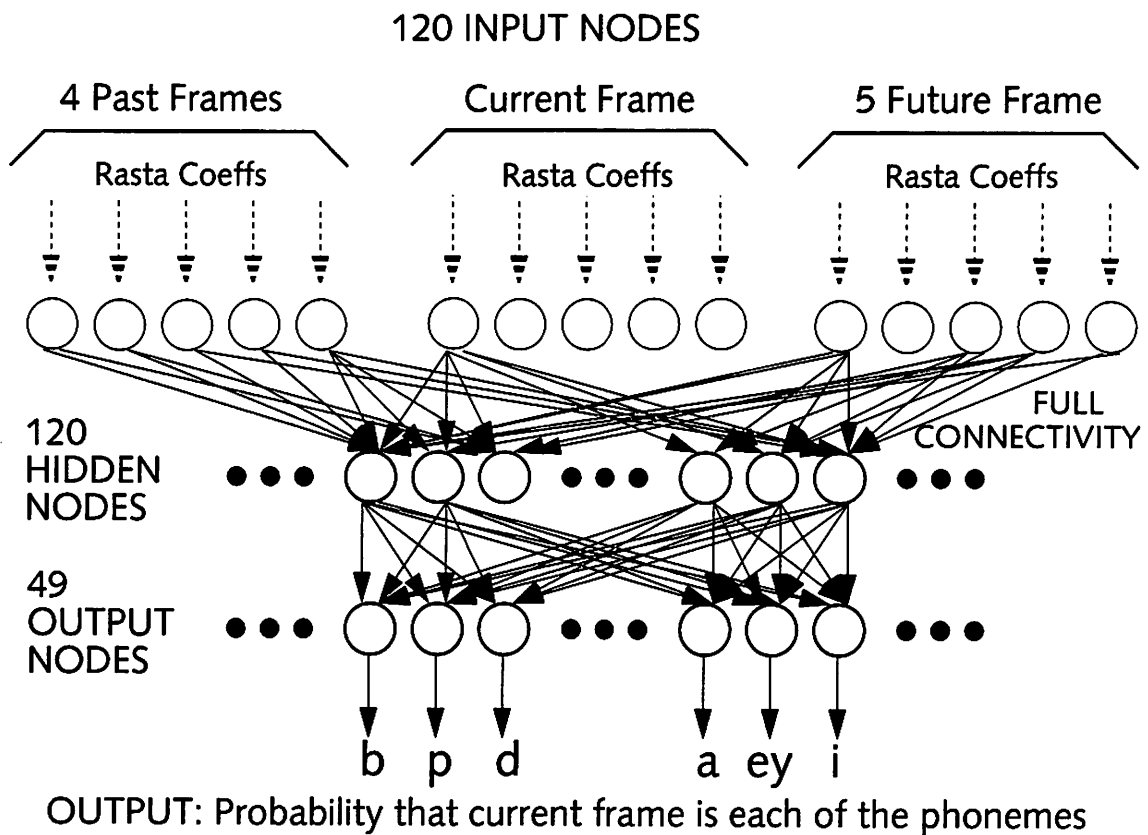


Figure 2-6. Multi-layered perceptron to compute phoneme probabilities.



The MLP contains three layers: the input layer, the hidden layer, and the output layer as shown in Figure 2-6. (By convention, some call this a three layer MLP while others would call it a two layer MLP because the input layer is really just a place holder for the input coefficients.) These inputs are organized by frame, with the earliest frame's coefficients feeding the left-most inputs; the last frame's feeding the inputs on the right. For speech sampled at 8kHz, there are 5 RASTA coefficients plus their first derivative with respect to time, as well as log power and its derivative, for a total of 12 coefficients per frame. After finishing its computation, the MLP left shifts the inputs by one frame, adding a new frame to the right and eliminated the left-most (oldest) frame. Although the MLP's input spans several frames, its output probabilities are considered to correspond only to the "current" frame, with the context frames only there to help the MLP identify the current frame. Furthermore, since the hidden and output layers of the MLP are memoryless, they do not make use of the fact that the inputs are shifted frame by frame. The phoneme probabilities could be computed in any order as long as the context is presented correctly; shifting the frames is simply a convenient way to compute the probabilities.

Each node in the hidden layer performs a multiply-accumulate function, followed by a nonlinear threshold function. The network is fully connected, meaning that every input has an edge, with its own coefficient, to every node in the hidden layer. After the input nodes have been assigned their coefficients, each hidden node multiplies the value of each input node by the coefficient on the edge connecting them, then accumulates these sums. Finally, each hidden node applies the nonlinear threshold function:

$$F(x) = \frac{1}{1 + e^{-x}}$$

After the hidden layer has finished its computation, the output layer computes the phoneme probabilities. The output layer performs the same algorithm as the hidden layer does, except its inputs are the outputs of the hidden layer. Since the range of the threshold function is (0,1), the nodes' outputs are already in the proper form to represent probabilities. Each output node corresponds to a specific phoneme. The hidden nodes, in contrast, are not assigned any fixed representation; their "meanings" are determined automatically during training.

As shown in Figure 2-6, the MLP in spRcg uses 120 input coefficients (from a 10 frame context) , 120 hidden nodes, and 49 outputs. Larger MLPs give higher recognition accuracies, but this size MLP was chosen as a compromise for ease of implementation in hardware and for computational complexity on a workstation.

### 2.4.3 Training

The MLPs create their coefficients by training on a large set of speech data that has been phonetically labeled. In brief, training is as follows: For each frame of speech, the correct answers are presented at the outputs. The MLP then alters coefficients in a way that reduces the error, the difference between the computed output and the correct answer. This training algorithm is called error back propagation (EBP).[20]

SpRcg's MLPs were trained on the TIMIT training set, which contains 462 speakers saying 8 sentences each. This database is well suited to training MLPs because its data has already been segmented and phonetically transcribed. Furthermore, it contains a large number of American English speakers, grouped by dialect region, speaking a large number of words.

After each training iteration, the MLPs were cross validated against a test set by measuring their frame-by-frame phoneme identification. During the course of training, the gain term in the EBP was multiplied by 0.93 after each iteration. Training the MLP with 120 input and 120 hidden nodes took 40 iterations.

### 2.4.4 Results

Table 2-3 shows the results of the MLP identifying phonemes frame by frame on the the TIMIT test set, which contains 168 speakers saying 8 sentences each. The first column is the name of the phoneme. The next three columns show the number of frames correctly identified, the total number of frames, and the percent correct on a frame by frame basis.

The first row of Table 2-4 shows the overall results of the MLP's frame-by-frame classification of phonemes. (When computing the score in Table 2-4, several sets of phonemes were considered equivalent, as described in [11]. Namely, ax and ah, ix and ih, aa and ao, zh and sh, en and n, el and l; and epi, vcl, cl and sil.) Usually, when the MLP's highest

Phoneme	Correct	Total	Score
b	885	1564	0.56586
d	609	2030	0.30000
dx	1279	1826	0.70044
g	438	1323	0.33107
p	2324	4217	0.55110
t	3387	6704	0.50522
k	3742	6250	0.59872
q	2660	5852	0.45455
cl	18934	24970	0.75827
vcl	8155	14252	0.57220
jh	1001	1880	0.53245
ch	1275	2249	0.56692
s	14400	24876	0.57887
sh	3592	5579	0.64384
z	4898	10629	0.46081
zh	58	636	0.09119
f	4737	9399	0.50399
th	106	2255	0.04700
v	1114	4229	0.26342
dh	749	3332	0.22479
m	4141	9106	0.45476
n	5641	12712	0.44375
ng	951	2474	0.38440
en	528	1656	0.31884
l	4117	11511	0.35766
r	4845	10811	0.44815
w	3483	5856	0.59477
y	1034	2132	0.48499
hh	1755	3824	0.45894

Table 2-3. MLP output scoring by phoneme.

Phoneme	Correct	Total	Score
el	1584	3171	0.49953
iy	12620	16735	0.75411
ih	3602	11213	0.32123
eh	2642	11272	0.23439
ey	5397	10604	0.50896
ae	4748	10458	0.45401
aa	2421	10691	0.22645
aw	1435	3750	0.38267
ay	7424	10982	0.67602
ah	2210	7756	0.28494
ao	4268	9257	0.46106
oy	667	2242	0.29750
ow	2416	7859	0.30742
uh	198	1655	0.11964
uw	2201	5564	0.39558
er	8848	16349	0.54120
ax	2318	6880	0.33692
ix	4123	13035	0.31630
sil	40657	47168	0.86196
epi	767	1387	0.55299

Table 2-3. MLP output scoring by phoneme.

Nth Best	Fraction Correct
1	0.56225
2	0.71125
3	0.78605
4	0.83243
5	0.86457

Table 2-4. MLP Nth best results.

probability output was not the correct phoneme, the correct output was still one of the few highest outputs. The subsequent rows show how often the correct phoneme was among the N highest outputs, for different values of N.

## 2.5 Clustered Grammar

### 2.5.1 Grammar Bottleneck

As described in Chapter 5 and [3][4][5], the key to designing low-low power hardware is the ability to break up an algorithm into small, local, slowly running, parallel elements. Within an HMM, the grammar poses the biggest bottleneck while trying to parallelize the Viterbi search algorithm. Ideally, the task of recognizing speech with a large vocabulary model should be split over many individual processors, each searching through a subset of the overall vocabulary. Each of these processors would have modest speed requirements, and hence could operate at low voltages. Furthermore, each processor could have its own low power memories on-chip, greatly reducing interconnect capacitance, and hence power consumption. When performing a viterbi search within a phoneme or within a word model, all transitions are to a well defined set of nearby states, so the algorithm can easily be split.

However, the problem occurs at the grammar level, i.e. transitions between words. One of the most common statistical grammars, the bigram grammar, uses transitions between every pair of words. Thus, for a vocabulary of  $N$  words, there are  $N^2$  transitions to be processed. For large vocabularies, this grammar processing will be the bottleneck. For example, in the UCB hardware system (see Section 2.3.2 and Section 6.2), grammar processing was the greatest bottleneck.

Word-pair grammars, which list all possible successors for a given word, can have a much lower number of edges. Thus, they are more suitable for a low power system. However, word-pair grammars are not ideal because they can have irregular structures. Since any word can still potentially connect to any other word, and to any number of other words, hardware devoted to grammar processing must retain a great deal of flexibility. In hardware design, flexibility means added size and complexity - hence it means added power consumption.

### 2.5.2 Breaking the Bottleneck

The spRcg speech recognition system uses a clustered grammar, which can be considered a transformed version of the word-pair grammar. The clustered grammar gets its name

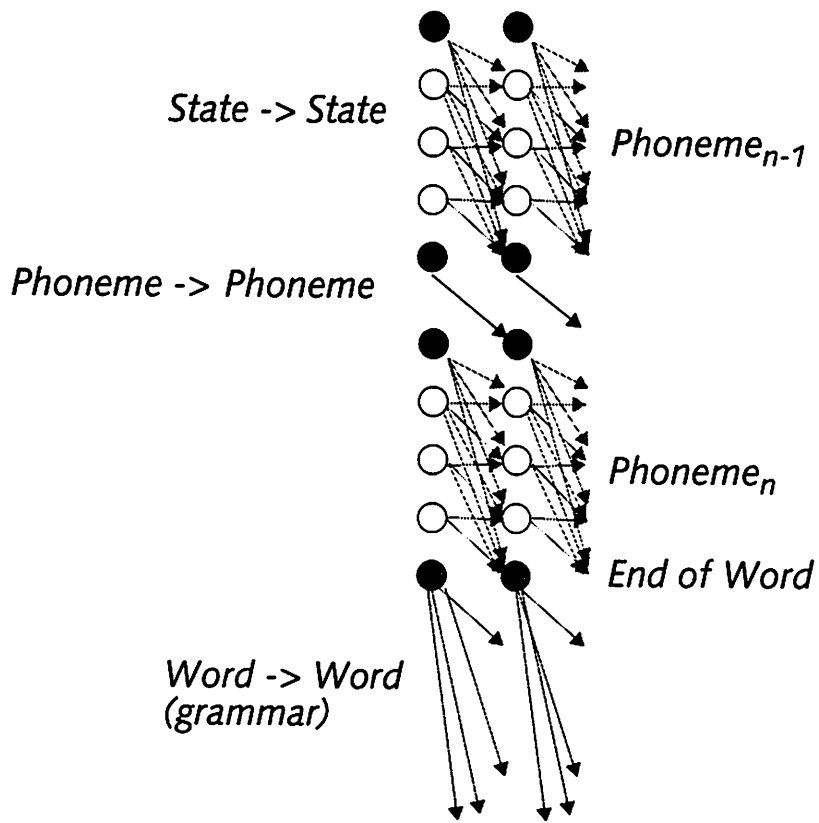


Figure 2-7. Hierarchy in the HMM.

because it requires each word in the vocabulary to be a member of one or more clusters. Grammar transitions occur between clusters, not between words.

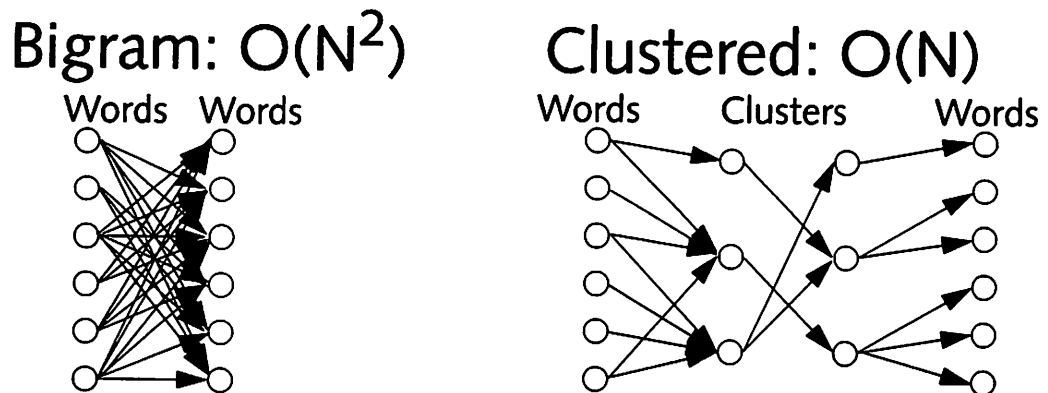


Figure 2-8. Clustered grammar reduces complexity.

As shown in Figure 2-8, a bigram grammar can be transformed into a clustered grammar, greatly reducing the number of edges and hence reducing both the computational complexity and the amount of memory needed. Furthermore, since the bigram and clustered grammars are topologically equivalent, clustering the grammar causes no loss in recognition accuracy.

Clusters group words that are used interchangeably in the grammar. At the most general level, a cluster might represent a part of speech - noun, verb, adjective, etc. More specifically, words in a cluster will probably share more semantic similarities. For example, Figure 2-9 shows two clusters used in a vocabulary for an integrated circuit layout editing program. The cluster on the right contains various layers of an integrated circuit; on the left, possible actions that can be performed on layers.

Clustering does not impose a severe restriction on grammars. In fact, for the intended use of the spRcg speech recognizer—command and control of applications—clusters are a natural and easy way to create grammars. Languages for interacting with computer applications, as compared with natural languages - have highly regular grammars. First of all, programmers tend to create regular languages because they are easy to parse. Secondly, many speech commands to programs will duplicate menu functions, which are themselves regular and hierarchical.

### **2.5.3 Programming With Clusters**

Furthermore, a clustered grammar fits well into an object oriented programming scheme. (See Chapter 4 for more on programming with spRcg.) One of the key attributes of object oriented programming is inheritance: when a new class is created, it can inherit properties and functions from one or more parent classes. The advantage of inheritance are extensibility - new classes can be added without having to rewrite a lot of existing code - and simplicity - by inheriting properties and data, new classes don't need to duplicate existing code.

For example, consider a programmer who must add capabilities for a third layer of metal to an integrated circuit layout editor. In an object oriented program, much of the work can be done by inheritance, since most of the properties of metal 3 are similar to the prop-

erties of other layers, in particular to the metal 2 layer. Thus, hypothetically, the program might have a Metal3 class that is a descendant of the EditableLayer class and the WiringLayer class.

What does this have to do with speech recognition? For a speech controlled program, the programmer must also add the words “metal three” to the appropriate places in the vocabulary. Since the programmer just added the word to the program, s/he probably doesn’t have a large sample of training data to determine the proper bigram probabilities. Thus, using a bigram grammar is often not practical for applications programmers. Rather, the ideal situation is for the new word to be placed automatically into the proper grammar clusters as determined by its inheritance. For example, “metal three” would be inserted into the cluster shown in Figure 2-9 because of its inheritance from the EditableLayer class, and presumably would be inserted into other clusters appropriate to a member of the WiringLayer class.

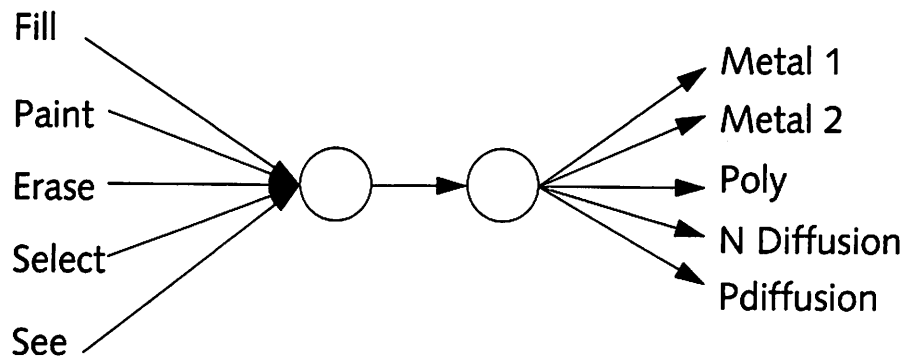


Figure 2-9. Example of a grammar cluster.

Using this paradigm, grammar clusters correspond to object oriented classes. The words in a cluster can be children of that class. Alternatively they can be class member functions (such as fill, paint, erase, etc.).

Furthermore, by using instances of an object as members of a grammar cluster, the program can easily change its grammar while it is running to reflect new data. For an example



of this technique, see the discussion of adding bookmarks to a world wide web browser application.

The grammar clusters themselves are not states in the HMM: they do not have output probabilities and there is no delay in traversing them. Rather, they are abstract place-holders like the start and end nodes in the word models.

Each cluster is instantiated twice in the HMM, once in the first layer, once in the second layer. Clusters in the first layer collect transitions from their member words, keeping the transition with the largest probability (in accordance with the Viterbi algorithm). Only after the frame has been processed, are the transitions computed from clusters in the first layer to clusters in the second layer, again according to the Viterbi algorithm. During the next frame, when a word starts, it receives its starting probabilities as stored in its clusters in the second layer. Thus, the first layer of cluster instances collect word ending transitions from the current frame, while the second layer of clusters store word ending transitions from the previous frame.

## 2.6 Recognition Results

Table 2-5 shows recognition results of the spRcg system on the TIDIGITs database using most likely pronunciation models. The higher error rates compared with Table 2-1 reflect the general purpose nature of the spRcg system. Namely its use of phonemes instead of word models and the fact that the phonemes were not trained on digits. Furthermore, the spRcg system uses 8 kHz sampling, compared to 16 kHz for the system of Table 2-1. Other work [14] has shown that a hybrid speech recognition system similar to spRcg's can achieve as low as 0.9% error if it is trained on the TIDIGITs database itself.

It is also interesting to note that almost half of the errors involved the digit "nine." This suggests that a better pronunciation model for "nine" would have significantly improved the error rate. It also suggests that a pronunciation adaptation scheme, such as the one described in Section 3.2.3, could produce large improvements by allowing the user to correct a few problem words.

	N-Best	Number of Errors	Recognition Accuracy
Male	1	2092	0.85
	2	1851	0.87
	3	1791	0.87
	4	1462	0.90
Female	1	4038	0.72
	2	3814	0.74
	3	3765	0.74
	4	3333	0.77

Table 2-5. N-best digit recognition using spRcg.

The difference in recognition accuracy between men and women is hard to understand because the MLP was trained on both male and female speakers.

Table 2-6 shows percent error rates from a user test of three males and one female speaking typical command to navigate the World Wide Web browser described in Section 3.3.1. The subjects first performed the test, which consisted of 38 sentences, using the standard pronunciations, then repeated the test after customizing the pronunciations by speaking each word in the vocabulary once (using the widget described in Section 3.2.3). The numbers in the table reflect whether the entire command was correctly recognized.

Speaker	Nth Best Task Error Rate (Percent)							
	No customization				Customized			
	1	2	3	4	1	2	3	4
AB	16	5	3	3	21	11	5	5
SS	26	13	8	8	24	18	13	13
JC	34	29	24	18	21	18	18	18
AS	34	21	18	13	18	13	11	11
Average	28	17	13	11	19	15	12	12

Table 2-6. SpRcg user test results.

The initial pronunciations were chosen to work well with the programmer (AB), so customization did not help for that subject. For the other subjects, customizations showed varying degrees of improvement in the first recognition result. In an actual application,

the users would use the customizer to improve only those words that caused the most recognition errors, so recognition accuracy should go up with continued use.

A large proportion of the errors involved misrecognition of digits. So this test, along with the TIDIGIT test, suggests that future improvements should go towards improving digit recognition.

## **2.7 Summary**

This chapter described the components of a speech recognizer in general, and the specific choices that were made for the spRcg speech recognition system. Most of the algorithms chosen—RASTA-PLP, Multi-Layer Perceptrons, and Hidden Markov Models—were chosen because they are both commonly used algorithms and are well suited to low-power hardware implementations. However, one part of the Hidden Markov Models—the grammar—had to be modified because it was the limiting factor in parallelizing the Viterbi algorithm.

---

## **3 SOFTWARE: USER INTERFACE AND APPLICATIONS**

---

SpRcg provides a complete toolkit for incorporating speech recognition into applications programs. This chapter describes the components of spRcg's toolkit that are seen by the user, as well as applications that were written using this toolkit. The internal programming details of the toolkit are discussed in Chapter 4.

### **3.1 Goals**

#### **3.1.1 Creating a User Interface**

The spRcg system's toolkit provides a standard user interface for all applications using speech recognition. Writing the demonstration applications helped to test the implementation of the toolkit and also gave feedback on what should be changed. Writing the applications also helped to develop the programming interface to the toolkit, described in Chapter 4.

#### **3.1.2 Ease of Use**

These applications and mega-widgets represent the first generation user interface for InfoPad. As such, they provide valuable feedback from users on what constitutes a good user interface. As users work for the first time on computers without keyboards, they learn what sorts of feedback they require from applications that rely on recognition.

#### **3.1.3 Enabling and Demonstrating InfoPad**

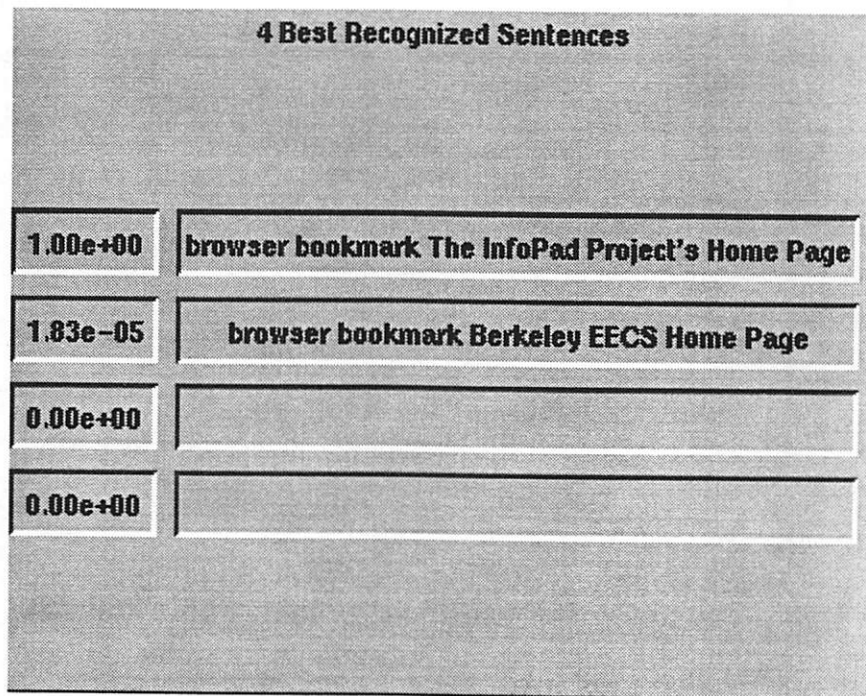
In addition to demonstrating speech recognition, these applications are intended to demonstrate and test all of InfoPads capabilities, as well as making the InfoPad a useful, user

friendly device. The World Wide Web browser in particular demonstrates the InfoPad's network connectivity and multimedia capabilities. The Web itself represents the (current) epitome of network connectivity and network based data access. The browser uses every form of InfoPad's input and output: handwriting recognition and speech recognition input, text, graphics, audio, and video output.

### 3.2 Widgets in the SpRcg Toolkit

#### 3.2.1 Answer Widget

The `spRcg_NBestDisplay`. provides feedback to the user about the speech recognizers



4 Best Recognized Sentences	
1.00e+00	browser bookmark The InfoPad Project's Home Page
1.83e-05	browser bookmark Berkeley EECS Home Page
0.00e+00	
0.00e+00	

Figure 3-1. `spRcg_NBest` widget.

NBest answers. If the recognizer never made mistakes, there would be no reason to display its output as text: the user already knows what s/he said, so the program should execute the user's command. However, if the recognizer occasionally misrecognizes or is unable to recognize a sentence, it is helpful to give the user feedback about what went wrong, as well as a way to correct the mistake. First of all, it reduces a user's frustration if s/he can see the

mistake, rather than having to infer that the recognizer made an error based on its actions. Also, some recognition results might not have a visible effect. Thus the feedback allows the user to confirm whether the program executed the right command. Even better, `spRcg_NBestDisplay`, can help the user correct recognition errors. With an NBest recognizer, if the first answer is incorrect, the correct answer is often one of the other possible sentences identified by the recognizer. If this is the case, the user can click on the correct response, causing the wrong answer to be undone, and the correct answer to be executed.

Providing textual output also helps the user adjust volume, silence, and garbage word settings. The user can quickly see if there are a lot of “not recognized” responses or if sentences are being cut off in the beginning or end and can adjust levels accordingly.

### 3.2.2 Control Panel

The `spRcg_controller` contains five sliders and two bars for controlling and displaying user-setable parameters.

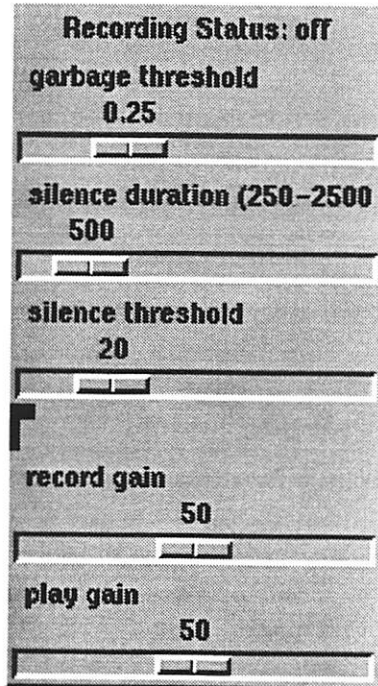


Figure 3-2. `SpRcg_controller` widget.

The garbage threshold slider determines how sensitive the recognizer is when deciding whether an utterance is not in its vocabulary. The probability of the garbage phoneme is the value of the garbage threshold times the largest probability from the MLP for the current frame.

The play gain slider controls the volume when the recorder plays back the last sentence it recorded. This has no effect on recognition, but is provided as a convenience for the user.

The remaining controls and displays are related to the recorder's silence detection, which automatically determines when a sentence has started and stopped. The bar above the record gain slider acts as a volume meter: its length represents the power of the speech being recorded. Immediately above the volume bar (and below the silence threshold slider) is the silence threshold bar. When the volume bar is longer than the silence threshold bar, the recorder thinks the user is speaking; when the volume bar is shorter, the user is considered silent. If the user is silent for longer than the silence duration, the recorder decides that the user has completed a sentence. This arrangement allows the user to adjust for varying microphones, speaker loudness, and background noise.

### 3.2.3 Customizer

SpRcg implements a speaker adaptable recognizer, which means that it works without individual user training but it performs better with some user customization. In particular, the `spRcg_adapter` widget helps the user customize the pronunciation of individual words in case the recognizer is having difficulty recognizing the way that particular user says those particular words<sup>1</sup>. The adaptation process is as follows. The user selects the word *s/he* want to adapt from the main list box (which displays all of the words in the current dictionary). Next, *s/he* presses the "record" button and speaks the word. If *s/he* desires, clicking the "Play Recording" will let the user hear that word as it was just recorded. Clicking the "Ok" button stores that word, as was pronounced in that recording, in a custom vocabulary.

---

1. This form of adaptation is different than other speaker adaptation systems that alter the parameters of the acoustic probability estimator, which in our case is the MLP.

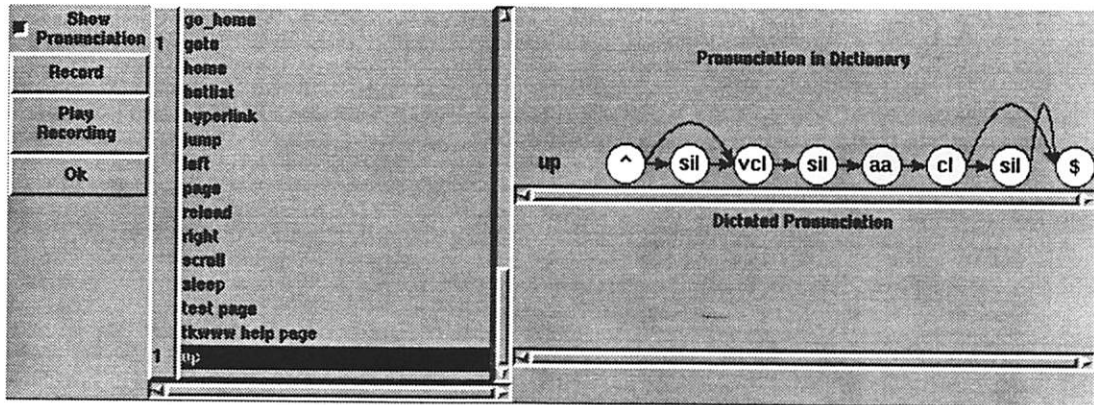


Figure 3-3. spRcg\_adapter widget.

The listbox to the left of the display of the words allows the user to see which words s/he has customized in the past, and allows her/him to selectively turn on or off customization on a word by word basis. If this listbox displays a “1” by a word, it has been customized and the customization is in use; if it displays a “o,” it has been customized but the customization is not in use (i.e. the original pronunciation is being used). Clicking on a “1” or a “o” will change it to the other state. If a word has never been customized, the space beside that word is empty.

If the user wants to compare the original pronunciation with the customized one, s/he can click the “Show Pronunciation” button, which causes a graphical display (the same as in the gramCracker program described in Section 4.6) to appear. This is somewhat of an advanced feature, since the user needs to know something about phonetics to interpret the display.

### 3.2.4 Grammar Cascade

The grammar cascade is designed to provide a “help” function for users.

One of the most difficult user interface problems with speech recognition is prompting the user on what s/he can say; in other words, telling the user all legal sentences in the vocabulary. In conventional interfaces, the user is given visual prompts: menus display all of their items and sub-menus when selected; listboxes display sets of choices such as all



the files in a directory. The grammar cascade menu is designed to provide similar visual feedback to speech recognition users.

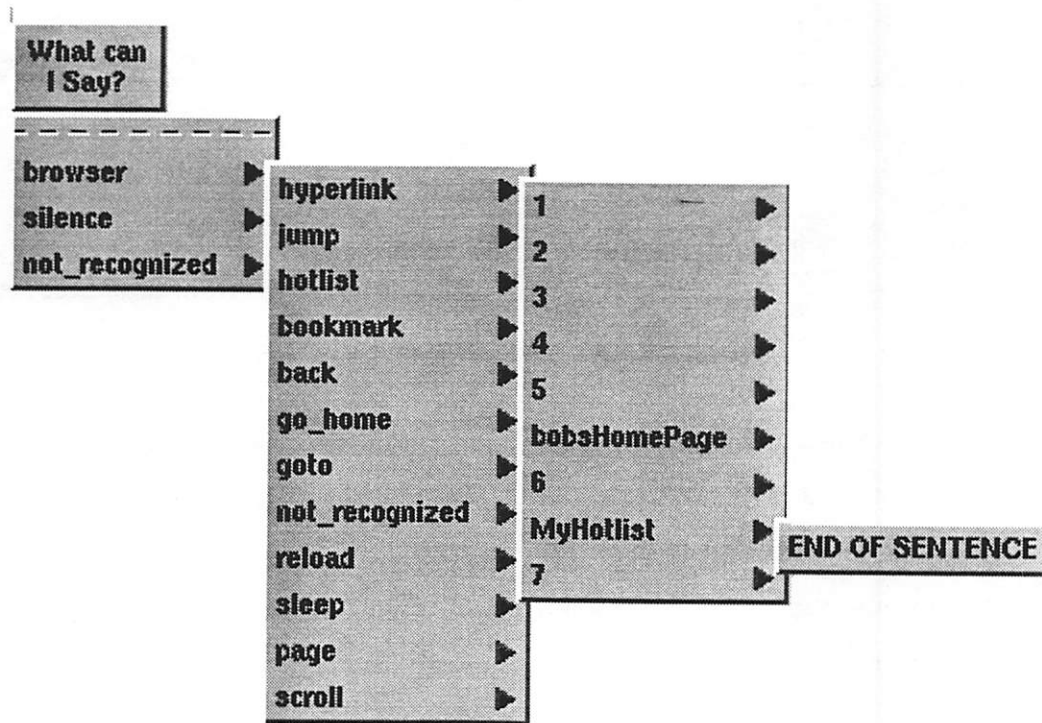


Figure 3-4. Grammar cascade.

The grammar cascade is a pull down menu with many levels of sub-menus. The first menu is a list of all of the words that are legal first words in the sentence. When one of these words is selected, it creates a sub-menu which presents the list of words that may follow that word. This process can be repeated indefinitely, thus providing a simple, visual way to explore the vocabulary space.

### 3.3 Speech Controlled Applications

#### 3.3.1 World Wide Web Browser

As a demonstration of speech recognition, it demonstrates how to use all of the high level speech recognition widgets and classes, as well as illustrating several ways to incorporate speech recognition into the operation of an application. As a demonstration of InfoPad,

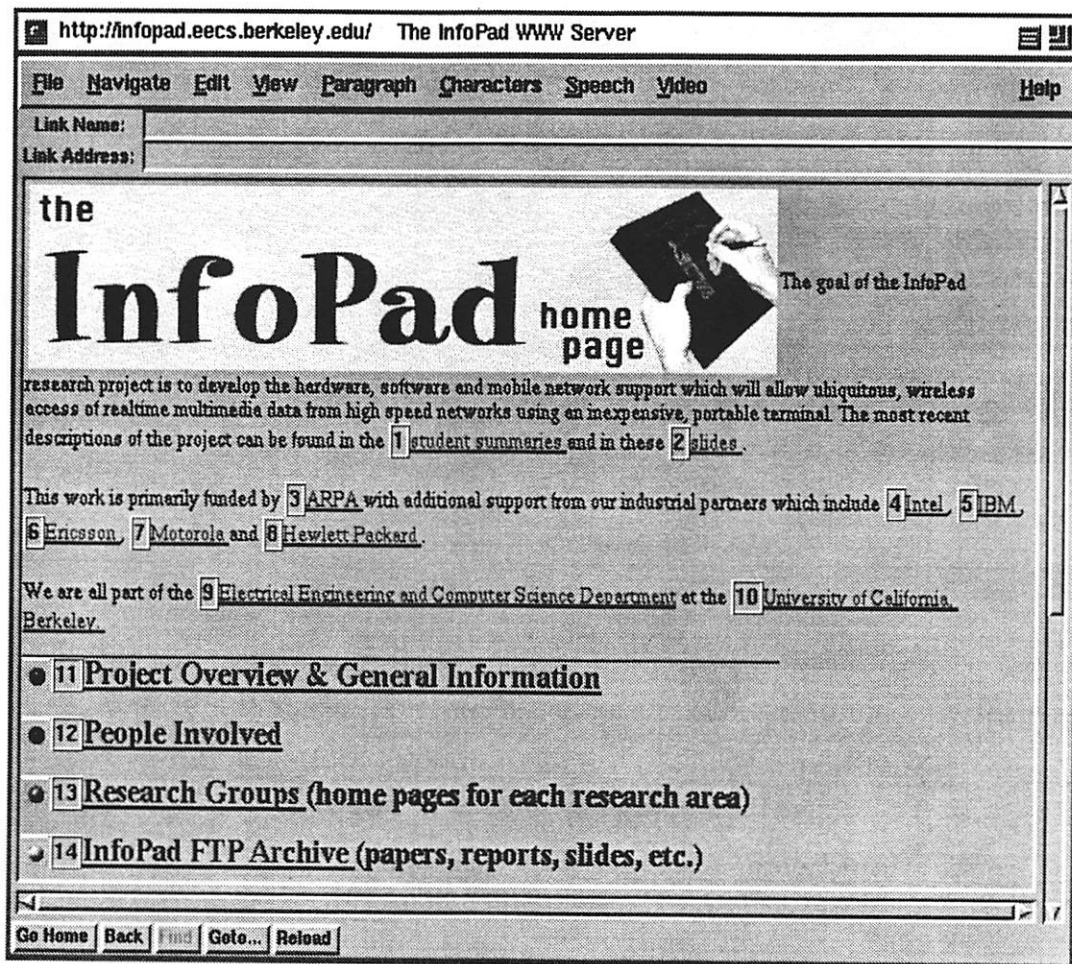


Figure 3-5. Web browser main display.

the WWW browser demonstrates the pad's multimedia input and output capabilities, as well as its excellent network connectivity.

The program, which is based on the freeware tkwww [29], has a front end written in tcl/tk, with a C back end based on the CERN WWW client library [30]. The back end handles network connections to and data transfers over the World Wide Web. It is implemented as a tcl extension, so WWW transactions can be performed as tcl commands. The front end parses and displays html files, handles user commands, and calls helper functions for certain data types retrieved over the web.

The most important enhancements added to the original tkwww is the use of speech recognition. Additional enhancements included allowing inline images while displaying html pages, altering the look and feel, updating to the latest versions of tcl (7.4) and tk (4.0), as well as fixing miscellaneous bugs.

### **3.3.1.1 Navigation & Control**

The web browser allows speech control at several degrees of sophistication. At the simplest level, the user can duplicate the action of pressing buttons or selecting menu items by speaking single word commands. For example, “back,” “go home,” and “reload” are voice commands that provide simple WWW navigation.

The more sophisticated commands provide solutions to a tough speech recognition problem: how to use speech to navigate through an ascii world. Computer files and directories, as well as WWW locations (URLs), page names, and hyperlink names are all specified by text names whose pronunciations are uncertain or difficult. Many of these identifiers contain punctuation marks, making their phonetic representation uncertain; others are dependent on capitalization, which does not show up in the pronunciation.

The problem of using speech to identify any text string is most apparent when using speech to select items in a list of WWW bookmarks. Bookmarks are convenient links to a user’s favorite sites. When viewing a Web page, the user can add its location to her/his bookmarks; later, selecting that bookmark will return the user to that page. The problem is creating a phonetic representation of the bookmarks name so the user can use speech to return to that page.

The solution is to have the user speak the name (or nickname) of that page once, allowing the recognizer to perform phonetic transcription. The web browser stores that pronunciation with the bookmark, along with its ascii name and its URL. In addition to storing the phonetic transcription, the web browser stores a recording of the user saying the name of the bookmark. The main purpose is to refresh the user’s memory if s/he forgets what s/he called that bookmark. Also, other users can use speech to navigate other peoples bookmarks after hearing their recordings.

Why are nicknames useful? Consider the titles of some pages in the author's bookmarks:

Berkeley EECS Home Page  
Sound Bytes: WWW  
TV Themes Home Page : By Patrick G. Kenny  
MovieLink 777FILM OnLine: Main Menu  
Tcl/Tk Project At Sun Microsystems Laboratories

These titles are either too long to remember how to say them exactly the same way each time, or have ambiguous pronunciations.

All of these activities can be performed from the bookmark dialog box. The main window displays the names of all the bookmarks. Buttons on the top allow recording, playback, and storing of names, as well as allowing playback of previously stored recordings. The bookmark dialog box is intended to be used when adding a new speech enabled bookmark, or when reviewing and editing old bookmarks. To navigate to one of the bookmarks, the user only has to say the word "bookmark" or "hotlist" followed by the name of the bookmark. In fact, these bookmarks show that sometimes speech recognition commands are easier to use than the equivalent menu commands.

Using speech to select hyperlinks imbedded in web pages also poses a problem. The problem is similar to the one described above, dealing with bookmarks, in that it is hard to determine their pronunciation. Many hyperlinks have long names, hard to pronounce proper names, or are icons and pictures. Another problem are hyperlinks called "here," as in "click *here* to see...."

The web browser demonstrates two techniques to deal with verbal hyperlinks. The first technique is called "verbal link numbers." When the user enables this option, the icon of a number appears next to each hyperlink. Since the pronunciation of numbers is not ambiguous, the user can simply say the command "hyperlink" or the command "jump" followed by saying the number.

Verbal link numbers highlight a few of the differences between hyperlinks and bookmarks. First, hyperlinks are in a users direct view while bookmarks are usually out of view. Thus, it is easy to provide visual cues in the former case; associating numbers with book-

marks would be less successful because it would force the user to remember the bookmarks in a particular order, when they are inherently random. Furthermore, most items in the list of bookmarks are favorite sites, which are visited often. Hence, the user will remember how to say the names of the bookmarks, just as s/he remembers the names of familiar places, people, etc. For example, someone who reads the comic strip Dilbert's web page every day will probably remember what s/he calls that sight. On the other hand, a user can encounter hundreds or thousands of hyperlinks a day, many of them seen for the first time.

The second solution to verbal hyperlinks is similar to the bookmark strategy. The author of an html page can associate speech with a hyperlink as follows. Using the web browser to edit the page, s/he can dictate a name, which is then recorded and phonetically transcribed. The web browser automatically embeds the phonetic transcription, as well as a pointer to the audio file of the author saying the name of the link, in the html file itself. This information is stored as attributes in the hyperlink. Since attributes are a standard part of html, any other web browsing software can navigate that page, although it will be oblivious to both types of speech information. When another invocation of the speech enabled web browser reads that page, it automatically reads the speech information, and adds the phonetic transcriptions to its dictionary as new words. Unfortunately, this phonetic transcription depends on the pronunciation of the original page designer, so it may not work well for other speakers.

From the user's perspective, there are three differences when viewing a web page with speech enabled hyperlinks. First, the speech enabled hyperlinks appear in a different color. Second, clicking the right mouse button on these links will download the audio file of that link, playing it on the InfoPad (or workstation). Note, however, that this only for benefit of the user's ears; the web browser does not try to recognize it because the name's pronunciation is already embedded in the html page. Third, and most importantly, the user can invoke the hyperlink by saying its name preceded by the command "hyperlink" or the command "jump."

This technique outlines a solution, or at least a compromise, for using speech in an ascii world. Essentially, names will contain two or three components. The first is the currently existing ascii identifier. The second is a phonetic identifier, composed of a standard set of phonetic symbols. The third, optional identifier would be an audio file. This triad of identifiers can ultimately be applied to files and file systems, as well as all components of the World Wide Web.

In the mean time, however, most of the WWW does not contain embedded speech recognition information. Thus, the second technique for using verbal commands to select hyperlinks does not require any modification of existing html pages; all the information is kept locally in the speech enabled web browser.

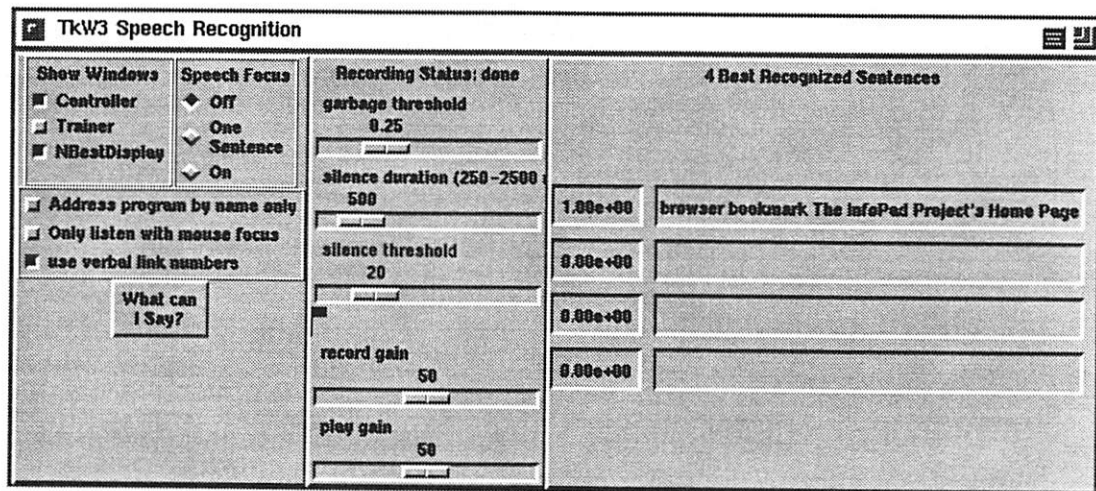


Figure 3-6. Web browser control panels.

### 3.3.1.2 Focus

The speech enabled web browser also illustrates several, user-selectable models for speech focus. Determining speech focus means answering the question, “who are you talking to?”

Focus is harder to determine speech than it is for mouse, pen, or keyboard input. First of all, button clicks and key presses are easily located in time to within a few milliseconds. A spoken sentence, on the other hand takes one or more seconds, so it is harder to associate with a particular moment in time. Second, mouse, pen, and keyboard focus usually fol-

lows the icon. Usually these events are directed to a particular window, either by moving the cursor into that window or by clicking on that window. Sometimes this requires two separate actions to input data: for example, first selecting a window, then typing into it. Other times, the action and focus occur simultaneously: for example clicking on a button combines fixing the focus and issuing a command in the same action. Another problem is that people use speech to communicate to other people, not just their computers. When a user types on the keyboard, s/he must be communicating with the computer; when s/he talks, she may or may not be talking to an application on the computer.

It is possible to have speech recognition use the same focus model, but this would eliminate some of speech's advantages. Basically, the user could click on the window to which s/he is speaking. This is fine when using a single application. However, when dealing with multiple applications or windows, this focus model eliminates some of the "hands off" advantages of speech recognition. Another problem with this focus model is it only allows focus to a single application. It should be allowable to "broadcast" a speech command. Some commands might be relevant to many applications; in other cases, it might be best for the applications to decide which one is being spoken to. In this case, multiple applications would be recognizing the speech simultaneously; whether or not they respond depends on the results of the recognition.

The speech enabled web browser has two user-controlled settings that determine its speech focus model. The first setting is controlled by a set of radio buttons that select speech focus "off," "one sentence," or "on." "Off" turns off all speech recognition. "One sentence" turns on the speech recognizer for one sentence (as determined by silence detection at the start and end of the sentence), then turns it off. This option is similar to the click for focus model, but it also specifies that the user's next utterance is meant for the application. The third option, "on," causes the web browser to continually record sentences and to execute them as commands.

The other user-controlled setting, "address browser by name," is useful in conjunction with the "on" speech focus model, together with out of vocabulary detection. Essentially, the user just has to begin any command to the web browser by its name. For example,

“browser reload.” If the user doesn’t like the name “browser,” (the default) s/he can change it by user the adapter mega-widget. This focus model is the same as a person uses when addressing one person in a group: “Fred, do this,” “Mary, do that,” etc.

Out of vocabulary detection is essential to this focus model. If the user begins any sentence with anything but the browser’s name, that sentence should be identified as being out of the vocabulary. Hence, the browser assumes the user was talking to another application (or a human, or was sneezing, etc.) and will ignore the sentence.

### **3.3.2 Magic layout editor**

The magic controller is a tcl/tk based application that provides speech recognition control of the magic [16] integrated circuit layout program. This program shows one way of integrating speech recognition into other X window applications that are not tcl/tk based without having to modify that program.

Normally, magic requires input from the keyboard and the mouse. The keyboard provides macros and commands while the mouse provides mostly placement information. Speech recognition is meant to replace the keyboard input, allowing use with just speech and the mouse.

Since magic is not a tk application, the magic controller sends it information by sending key-press events to the cursor location. The controller uses a small helper application, stringzevent to perform this task.

## **3.4 Combining Handwriting and Speech Recognition**

Since speech recognition and handwriting recognition each have their strengths and weaknesses, the two can be used synergistically to provide a better user interface.

Speech’s strengths are its ease of use and its speed. It requires less effort for a user to speak a word or sentence than to write it. A user can also speak words faster than s/he can write them. Thus, as long as the recognition accuracy is high enough, it is more convenient to speak words than to write them. In situations such as issuing commands, the application



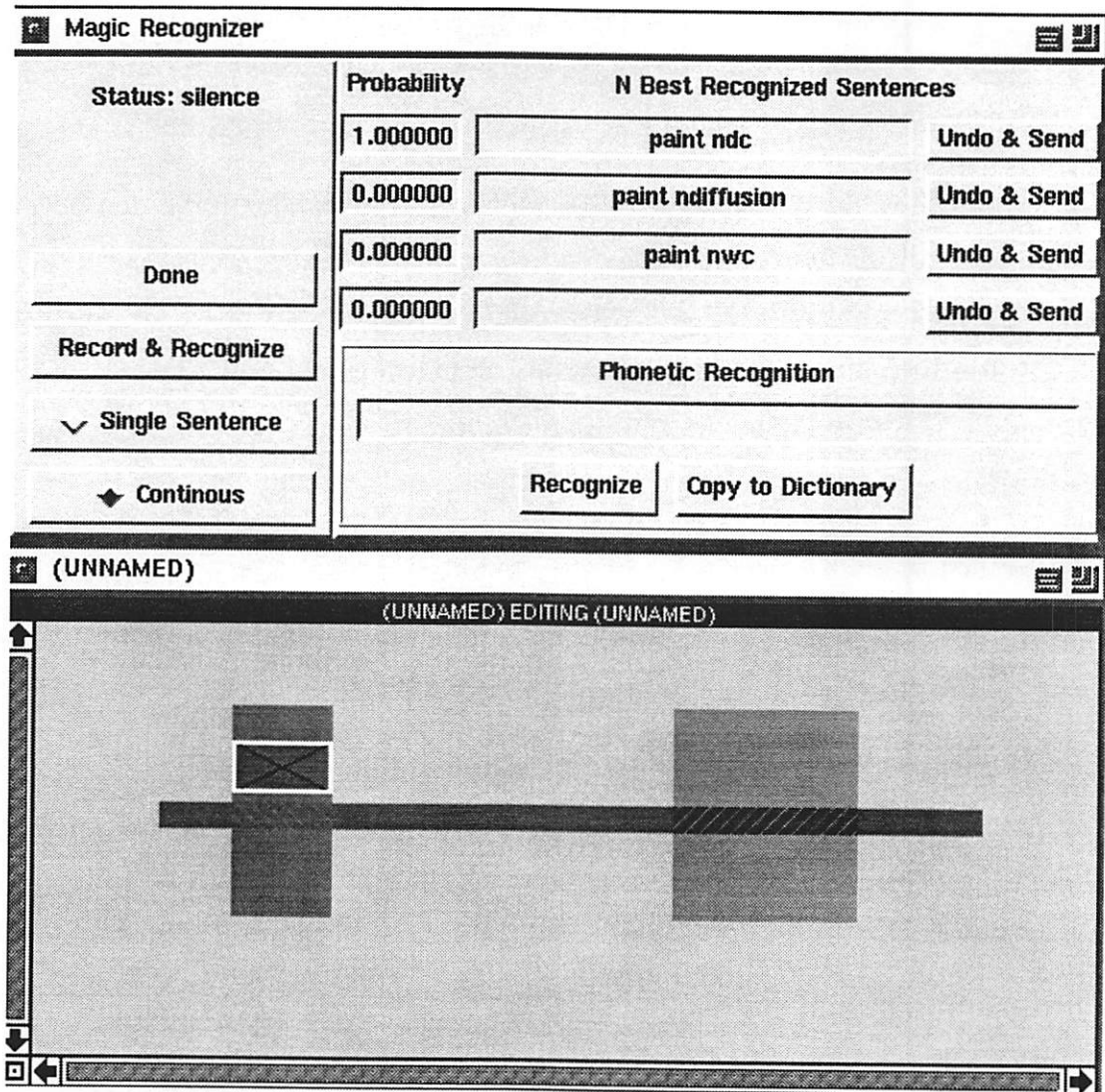


Figure 3-7. Speech controlled ic layout editor.

should be able to constrain the vocabulary and grammar sufficiently to get high enough accuracy.

When the user must enter ascii data, handwriting recognition has several advantages. The first is speed: verbally spelling a word is no longer faster than writing it, and is probably more tedious. The second advantage is accuracy: speech recognition on the "e" set - "b," "c," "d," "e," "g," "p," "t," "v," and "z" - is notoriously difficult.

Thus, pen and speech can be used simultaneously to provide a faster, more natural interface and to shore up each other's weaknesses. For example, a user could enter text with the pen, then use speech combined with gestures ("move this text over here") to edit the document. Also, the user could correct speech recognition errors by changing the spelling with a pen, or could correct pen recognition errors with verbal commands. For more discussion of using pen and speech recognition together, see [15].

Because both the spRcg speech recognizer and InfoPad's handwriting recognizer are implemented in tcl/tk, it is easy for a programmer to include both of them in the same program and to have them working together. Chapter 4 shows the programming interface that creates the objects, widgets, and applications described in this chapter.

---

## 4 SOFTWARE: PROGRAMMER'S INTERFACE

---

### 4.1 Overview & Goals

This section describes software that facilitates the inclusion of speech recognition in applications for InfoPad. The design of this software reflects a compromise between the conflicting goals of ease of programming and high recognition accuracy.

From the programmer's perspective, the ideal speech recognizer should be as far removed from the program as possible. The recognizer should act as a buffer between the speaker and the application, intercepting all audio signals, converting them to text, and then sending them as ascii to the application. The programmer would rather not deal with phonemes, vocabularies, and grammars; he/she would prefer that it be no more complicated to receive spoken data than it is to receive data from a keyboard. In short, applications programmers should not have to become speech recognition experts in order to write applications for InfoPad.

Unfortunately, the best way to achieve high speech recognition accuracy is to link tightly the speech recognizer to individual applications. If speech recognition were a solved problem, then it would indeed be possible to have a continuous speech, speaker independent, large vocabulary, natural English grammar speech recognition "server" that listened to everything the user said and passed on recognized text to the appropriate application. With the current state of the art in speech recognition, it is necessary to relax one, or preferably several, of the speech recognizer's attributes (continuous vs. discrete speech, speaker independent vs. speaker dependent, large vocabulary vs. small, or unconstrained

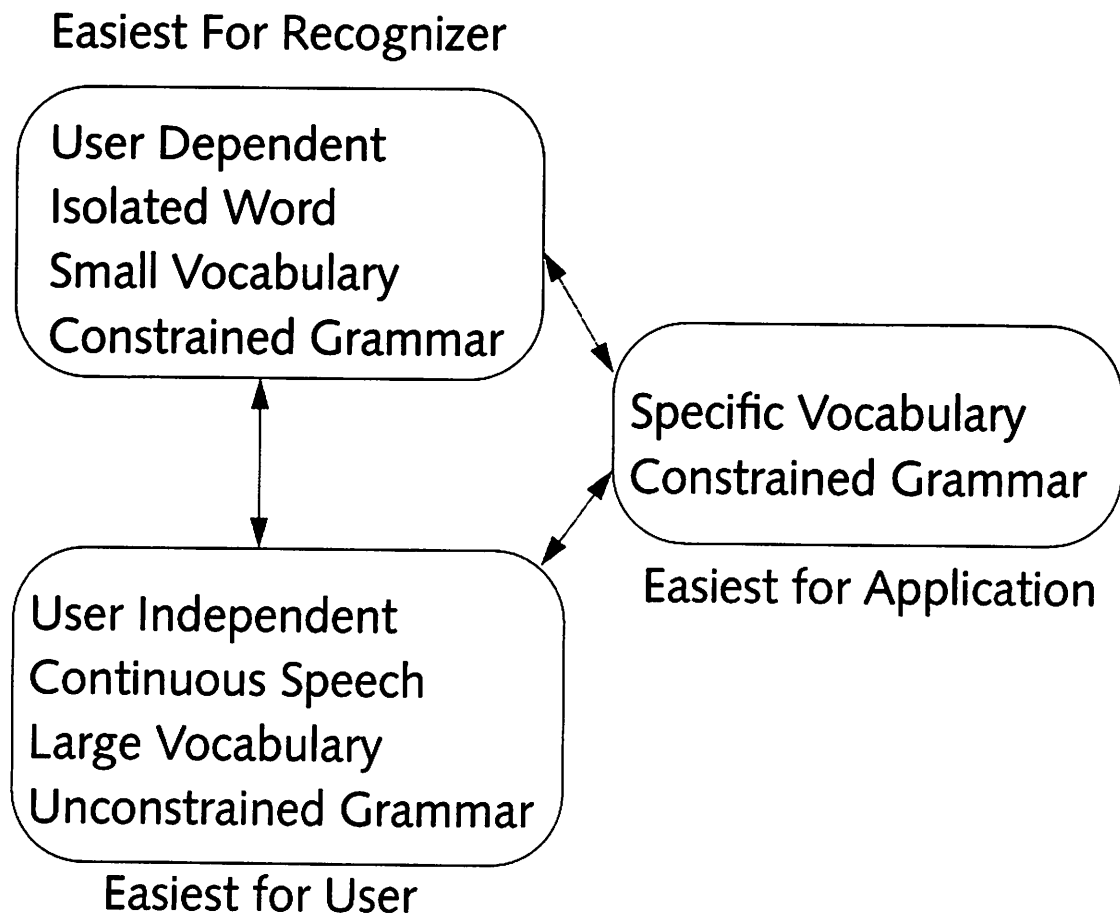


Figure 4-1. Design trade-offs .

grammar vs. constrained) to make the recognition task easier, and thereby achieve higher recognition accuracies (see Figure 4-1). In the case of the InfoPad speech recognition, the compromise was made with the size of the vocabulary and the complexity of the grammar, while maintaining speaker independence and continuous speech recognition.

The software architecture, shown in Figure 4-2, is structured to allow multiple levels of access to the recognizer; the programmer can trade off simplicity of use for greater control as she/he moves from high level interfaces to lower levels. At the highest level, the programmer can use incrTcl classes and mega-widgets. At the intermediate level are tcl commands. Finally, C++ classes form the actual implementation of the speech recognition algorithm.

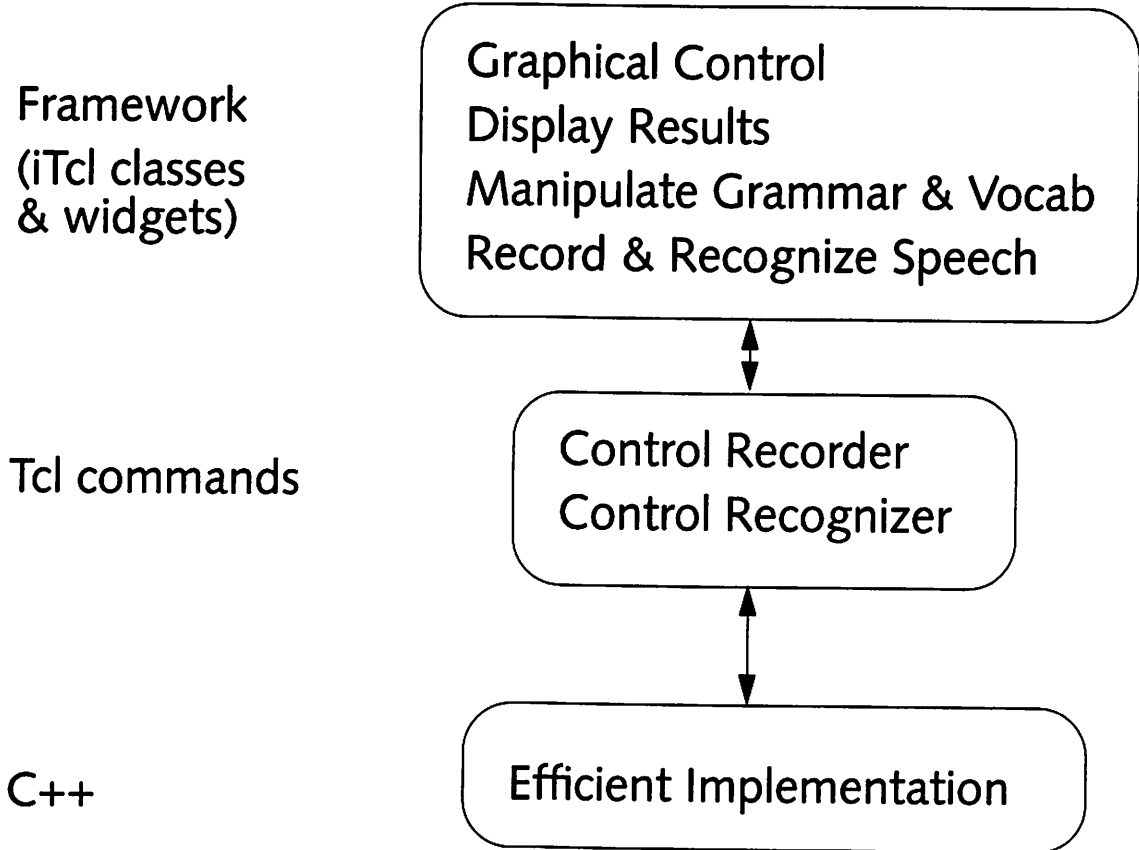


Figure 4-2. Architecture of speech recognition software.

#### 4.2 Writing Applications for InfoPad

This programming interface for speech recognition is part of a larger effort to define a user interface for InfoPad. Since InfoPad uses a standard X windowing system display, it uses standard applications and tools whenever possible, in order to maximize portability and ease of development. However, there must be some changes to these tools and applications in order to include speech and handwriting recognition. The model for developing new applications for InfoPad is to use the tcl/tk [17] scripting and windowing language because it combines the strengths of popularity and ease of use with the ability to be easily extended to include new features such as recognition.

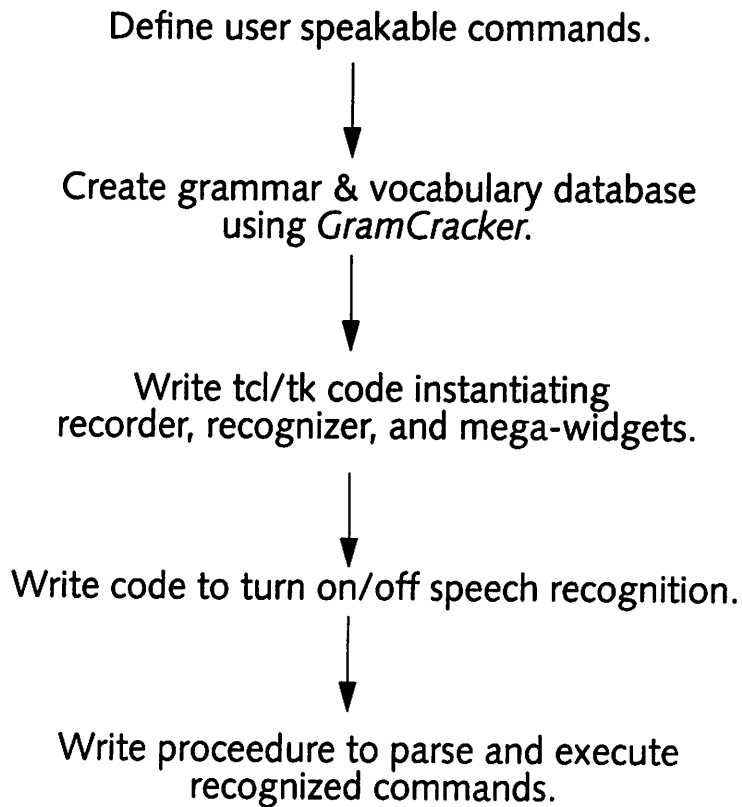


Figure 4-3. Process for adding speech recognition to a tcl/tk application.

#### 4.2.1 Creating and Editing Vocabularies and Grammars

Programmers will create vocabularies and grammars while they are writing applications. They may also have the applications modify the vocabularies and grammars while they are running. Typically, a programmer needs to write a grammar that reflects the overall control flow of the program. For example, the grammar might allow the user to speak any sequence of commands that could be issued by selecting menu items or pushing buttons. The general form of the grammar must be determined while the program is being written because the program must know how to parse any possible recognized sentence. The application, *GramCracker*, described in Section 4.6 on page 62, is a tool for programmers to create vocabularies and grammars.

Run time changes fall into two categories. First, new data such as proper names or file names can be added. For example, the web browser program, described in Section 3.3.1

on page 36, adds new verbal bookmarks during run time. Typically, these changes only involve adding or removing words from existing grammar clusters; the web browser stores bookmark names in a “bookmark” cluster that was created when the program was written. It is possible to make more complex changes to the grammar at run time, but the program’s parsing routines must be able to handle any such changes. These types of changes can be easily controlled using the `spRcg_gramDict` class’ methods combined with phonetic recognition.

The second common run-time change is to adapt a word’s pronunciation to a particular user. Initially, both the MLP’s representation of phonemes and the vocabularies list of pronunciations are speaker independent. If a user finds that the recognizer is having trouble with a particular word, she/he can speak that word one, customizing the dictionary’s pronunciation (but not the MLP). The `spRcg_adapter` mega-widget performs this task.

### **4.3 Low Level Implementation**

This section describes the implementation of the speech recognition system at the tcl level. Usually, this part of the implementation is hidden below the higher level itcl classes and mega-widgets. However, it is useful to understand the tcl level commands in order to modify or extend the high level implementation.

#### **4.3.1 Application Programmer’s Interface**

The `spRcg` speech recognizer is implemented as an extension to the Tcl programming language, allowing simple incorporation in tcl/tk based applications. This extension follows the “object” model for tcl: the programmer first creates a named instance of an object, then invokes the name of that instance to execute commands. The extension contains two objects: a recorder and a recognizer.

##### **4.3.1.1 Recorder**

The recorder is used to open and close connections to audio devices; to record speech; to determine sentence start/stop boundaries by detecting silence; to process recorded speech (using the RASTA algorithm); and to pass on the rasta output to the speech recognizer.

Audio connections are made using the device independent AudioFile (AF) system. Since the InfoPad's networking software (InfoNet) also uses the AF API, the recorder can transparently be used with InfoPad as well as most of common desktop computers.

The programmer can use the recorder to constantly monitor audio input, waiting for a signal of sufficient power and duration to be considered speech. As long as the signal is considered silence, the recorder discards the speech data. When the recorder does detect speech, it saves the speech data until it detects a long enough silence to determine that the spoken sentence has ended. Both the power and duration of the silence threshold can be adjusted while recording. The power threshold must be set high enough so the recorder is not triggered by background noise or breathing sounds, but must be low enough that quiet speech sounds do trigger it. The silence duration must be set long enough that brief pauses between words within a sentence are not misinterpreted as the end of a sentence; it must be set short enough not to cause an annoying delay in recognition while the recorder waits to see if another word will be spoken. Typically, a 0.5 second duration is a reasonable value.

The recorder automatically computes rasta coefficients as it records speech. It can either provide them to the recognizer periodically during recording, so that all recording, processing, and recognition is done in real time. Or it can provide all rasta coefficients all at once when the sentence is finished.

#### **4.3.1.2 Recognizer**

The recognizer object implements a speech recognizer using a particular Multi-Layered Perceptron, vocabulary, grammar, and N-Best depth; all of which are specified during creation of the object's instance. To recognize using a different vocabulary or grammar, the program should create and use another recognizer instance. (Of course, the program could also use another MLP – for example, switching from a speaker independent MLP to a speaker dependent one. However, it is far more common to change the vocabulary or grammar than to change the MLP.)



The main run time commands for the recognizer are to give it rasta coefficients and to output recognized sentences. Rather than providing a single answer, the recognizer provides several answers (hence the name N-Best). The additional answers can be useful for error correction.

One other run time command allows the program to adjust the garbage model, a representation of all words that are not in the vocabulary. The garbage model helps avoid recognition errors when the user speaks a word that does not fit in the vocabulary and grammar. Without the garbage model, the recognizer would return the sentence that fits the utterance the best, even though the fit would not be very good, and even though the answer is wrong. Using the garbage model, the recognizer should report that it could not recognize the sentence. If the garbage threshold is set high, the recognizer is less likely to mistake an out of vocabulary word for an in vocabulary word; unfortunately, it is also more likely to identify an in vocabulary word as out of vocabulary. The garbage threshold can be changed at run time to allow the user to adjust it to his or her needs.

#### **4.3.2 Implementation**

All of the computationally intensive parts of the speech recognizer and recorder are implemented in C++. The code was designed with two goals: to be computationally efficient and to serve as a high level simulation for hardware design.

The computation performance goal was to perform small to medium sized vocabulary (around 100 words) in real time on a sparc 20 workstation. For small to medium sized vocabularies, the most computationally intensive part of the algorithm is the MLP, since the viterbi search scales with the vocabulary size but the MLP computation is independent of the vocabulary and grammar. Fortunately, most of the MLP computation is performed in a single inner loop, which was carefully coded and partially unrolled for maximum performance on the sparc architecture.

#### **4.3.3 Dual Use for Simulating Hardware**

The C++ code assisted hardware design in three ways.

First, the C++ classes modeled and defined the high level architecture of the viterbi decoder. Four C++ classes - Dictionary, State, Grammar, and TagTable - correspond to the four architectural blocks of the hardware design. All of the arrays defined within these classes correspond to memories in the hardware; variables correspond to registers. Thus, these classes help determine the size, number, and access rate of memories in the hardware. Furthermore, by communicating only via public interface functions, the classes define what control and data information must pass between the architectural blocks. Since the memories in each block are implemented as protected class members, it is impossible for a block to have hidden or implicit access to memories in another structure, which would cause problems during hardware design.

Second, the C++ code assisted in designing the bit widths for the data in the hardware. For example, the MLP can have different bit widths for coefficients and for probability data. When acting as part of the speech recognition system, the MLP class represents both probabilities and coefficients as floating point numbers. Fortunately, the operator overloading capabilities of C++ make it easy transform the MLP class into a bit true emulator of the hardware. All that is required is to define a Fixed class with appropriate add, multiply, and type conversion functions, along with a few utility functions such as overflow detection and I/O functions. Then, by redefined the type of the probability and coefficient classes, the same MLP code can be used to evaluate the performance of the algorithm at various bit widths.

Third, the C++ code aided in debugging and verifying the design of the hardware. During the MLP calculation in bit true mode, the code can output any of its intermediate computations, which can then be compared to internal data in the VHDL or switch level simulation of the hardware. Being able to generate correct data values for so many internal buses in the hardware made it easy to verify the design quickly and with a high degree of confidence. When incorrect values did occur in simulations of early phases of the design, the high level simulation made it easy to track them down.

## 4.4 High Level Objects and Procedures: spRcg\_gramDict

Although the recorder and recognizer objects make it easy to record and recognize speech, there is still a considerable amount of work involved in creating and editing vocabularies and grammars. To reduce this burden on the programmer, an additional object called spRcg\_gramDict, implemented in itcl (and also requiring the tclX extension), provides an easy way to store vocabulary and grammar data structures, and to manipulate these data structures at a high level of abstraction. SpRcg\_gramDict also provides all of the appropriate arguments for creating a new spRcg\_recognizer, so the programmer never has to deal with the low-level details of vocabulary and grammar data structures.

### 4.4.1 Data

By internally storing all of the data needed to represent the vocabulary and grammar, SpRcg\_gramDict provides several advantages. First, its functions provide a higher level API to the programmer, so he/she does not have to deal with low-level syntax issues of the data structures. Further, if in the future the data representation should change, or if new recognizers with new data formats are used, the API can remain the same.

SpRcg\_gramDict also stores the file name of the MLP data. There is no need for tcl programs to store the MLP data itself, because it is not useful for applications programs to modify the MLP data. Vocabularies and grammars, on the other hand, should be changed whenever the program moves to a different state.

### 4.4.2 Grammar Methods

Eight class methods allow manipulation and examination of the grammar. The programmer may add or delete words from a grammar cluster, as well as adding or deleting edges between clusters. Access methods can list the contents and connectivity of a cluster without changing the grammar.

These methods should be the most frequently used by applications programs.

```
method addWord2Cluster { cluster word prob }
method destroyCluster { cluster }
method addGramEdge { fromClust toClust prob }
method toggleGramEdge { fromClust toClust prob }
```

```

method delGramEdge { fromClust toClust }
method giveAllClusterNames {}
method giveWordsInCluster {cluster}
method deleteWordFromCluster {cluster word}
method giveToClusters {fromClust}
method giveFromClusters {toClust}

```

#### 4.4.3 Input Output Methods

The following functions store and read vocabulary and grammar from an ascii file.

```

method writeGramDict { fileName }
method readGramDict { fileName }

```

This function returns the arguments needed to create a new instance of a spRcg\_recognizer.

```

method giveRecogConstructorArgs {}

```

This is a constructor method that initializes itself to be equal to another spRcg\_gramDict.

```

method copyOther { other }

```

#### 4.4.4 Dictionary Methods

The main reason for an application program to manipulate the dictionary is to alter the pronunciation of words as a part of a training or speaker adaptation system. However, if the programmer uses gramCracker to create the grammar and uses the adapter mega-widget for speaker adaptation, the application will only need to use the following two methods to add, change or delete pronunciations. The programmer does not need to manipulate the pronunciation data structures, which are provided by the mega-widget.

```

method addCompleteWord2Dict{ word pronunciation {dontOverwrite 0}}
method delDictWord { word }

```

If the programmer does want direct control over the pronunciation (as was the case when writing gramCracker and the adapter mega-widget), she/he can use the following access and edit methods.

```

method giveWordEdges { word }
method giveWordPhones { word }
method toggleEdge { w fromIndex toIndex }
method pastePhones { word phoneIndex addPhones addEdges }
method deletePhones { word indexList }

```

```
method giveWord { word }
method addNewWord2Dict { word }
```

The following methods are utilities to add or remove mandatory or optional silence at the beginning or end of a word. Adding mandatory silence at the beginning or end of a word is equivalent to changing the recognizer to an isolated word recognizer; words can only be recognized if there is a pause between them. Adding optional silence to pronunciations allows recognition whether or not the speaker pauses between words.

```
method wordStartSilence { word case }
method wordEndSilence { word case }
method setAllWordStartSil { option }
method setAllWordEndSil { option }
```

Finally, this utility checks for syntax errors in the pronunciation data structures; it is useful for programmers who directly manipulate the data.

```
method checkPronunciation word
```

#### 4.4.5 Graphics Methods

The following methods create a graphical representation of a pronunciation. They are used by gramCracker and by the adapter mega-widget.

```
method drawWord { canv word x y }
method drawPronunciation { canv word phones edges x y }
```

### 4.5 Mega-Widgets<sup>1</sup>

Each mega-widget can be viewed from two perspectives. For the user, it is a collection of windows such as labels, sliders, and buttons. For the programmer, a mega-widget is also an itcl class, which has member variables and procedures. This section describes the mega-widgets' programming interfaces; their graphical user interface are described in the next chapter.

---

1. The somewhat peculiar term, "mega-widget," is itcl jargon for an itcl class that contains a number of individual tk widgets, but can be created, packed, and configured using syntax as if the whole mega-widget were a widget itself.

#### 4.5.1 spRcg\_controller: Simple Recording and Recognition

After the application creates and initializes a spRcg\_recorder and a spRcg\_recognizer, it can use a spRcg\_controller mega-widget to perform all recording and recognition; this represents the highest degree of abstraction and the simplest programming interface. The spRcg\_controller is associated with a particular spRcg\_recorder and spRcg\_recognizer, at creation, but it can be associated with other ones at run time

The following method allows the program to detect, record, and recognize a spoken sentence by using a single line of code. The method handles silence detection at the start and end of the sentence, records the sentence, and performs recognition on the fly. The N-Best recognized sentences are passed to the answerCallBack function, if one is provided.

```
method recordRecog { answerCallBack }
```

The following methods perform the same functions, but separate the recording and recognition, so the recognition would be performed in batch mode after the recording is finished. The record method executes the callBack function with no arguments when it is finished recording; such a callBack function might inform the user that a sentence has finished recording, for example.

```
method record { callBack }  
method recognize { answerCallBack }
```

Since these functions wait to record an entire sentence, they could be running indefinitely if there is no input audio signal. This is not a problem because they make frequent calls to the tk update command. Since tk programs are event driven, the update commands allow the program to perform normally, even with the recognition process running continually. SpRcg\_controller makes sure that no more than one recording method is running at a time so that they do not steal data from each other. On the other hand, it is easy to send a recorded sentence to multiple recognizers by using the record method, then reconfiguring to use a different spRcg\_recognizer before each call of to recognize.

A call to the method listed below will kill a recordRecog or record method if one is running.

```
method abort { }
```

A call to the method listed below will tell if a `recordRecog` or `record` method is running.

```
method alreadyOn { }
```

The following methods perform functions similar to commands that are built in to the `spRcg_recorder` or `spRcg_recognizer`. The built in commands are simple to begin with, so they are duplicated in the mega-widget for completeness, rather than to provide a higher level of abstraction.

```
method setGarbage { value }
method setGain { value }
method loopback { }
method setPlayGain { value }
method setPowerFloor { value }
```

#### 4.5.2 `spRcg_NBestDisplay`: Simple User Feedback

This mega-widget presents the recognition results to the user. If the first recognition result is incorrect, the user may replace it by clicking on one of the other `NBest` recognized results.

This mega-widget contains only one, simple method, which displays the output of the recognizer:

```
method display { answer }
```

There are four configurable parameters:

```
-NBestDepth (default is 1)
-len length of sliders (default is 8c)
-showProbs bool (default is 0)
-B1Callback
```

The first three control what the user sees. Generally, `showProbs` should be set to 0 since the probabilities of the various answers are of little interest to anyone who is not interested in the speech recognition algorithm itself. The fourth is the callback function, which is passed a sentence if the user clicks on its window.

### 4.5.3 spRcg\_SpeakerAdapter: Simple User Customization

This mega-widget allows the user to change the pronunciation of individual words to match the way she/he pronounces them. The program uses the following config parameters to tell spRcg\_SpeakerAdapter the names of two spRcg\_GramDicts: `generalGramDict` is the name of the currently active grammar/dictionary; its contents are not altered by this mega-widget. `customGramDict` is the name of a grammar/dictionary that contains the words that the user has customized; its contents are altered by this mega-widget.

```
-generalGramDict  
-customGramDict
```

The following parameter, which specifies the name of an active spRcg\_controller is needed because the mega-widget uses it to record and recognize speech.

```
-controller
```

The final three parameters are optional. `Frozenwords` is a list of words that the user should not be allowed to customize. `OkCommand` is a callback command executed when the user pushes the “OK” button. Finally, if `saveStateDirectory` is specified, a file will be stored in that directory which saves the user-specified use/don’t use information for each custom word. (Note that it is the application’s responsibility to save the `customGramDict` if it is to be used again the next time the application is run.)

```
-frozenWords  
-okCommand  
-saveStateDirectory
```

When using `spRcg_speakerAdapter`, the application only needs to call the following method. Its output is identical to a call to the `generalGramDict`’s `giveRecogConstructorArgs` method, except the pronunciations for the customized words replace their pronunciation from `generalGramDict`.

```
method giveRecogConstructorArgs { }
```

### 4.5.4 Phonetic Transcription

Since the `spRcg_speakerAdapter` depends on phonetic transcription, the toolkit includes a special purpose vocabulary and grammar to perform phonetic transcription. This vocab-



ulary and grammar comprises a complete `spRcg_gramDict`. What makes it unusual is its “words” correspond to the individual phonemes; each word’s pronunciation contains only a single phoneme.

The grammar is clustered according to phonetic classes, with the edge probabilities determined by the statistics of the timit database. The clusters correspond to the types of phonemes, e.g. voiced stops, unvoiced fricatives, vowels, etc. Within each cluster, the probability of each phoneme corresponds to its frequency of occurrence relative to the other members of that class. Transition probabilities between clusters are determined by frequency with which the two phonetic classes followed on another in the timit database. For example, it is highly unlikely that a voiced stop is followed by an unvoiced stop, so this transition has only 0.002 times the probability of a transition from a voiced stop to a vowel.

The strengths of this technique of phonetic transcription are its simplicity and its effectiveness. It is simple because it uses the same recognizer that is used for sentence recognition. It is effective because it produces pronunciations that are likely to produce matches, at least for the same user.

However, the chief weakness of this technique is that it fails to accommodate the knowledge that is embedded in any previously used pronunciation. A word in the vocabulary might fail to be recognized because of problems with a single phoneme. Ideally, an adapter should be able to recognize the single problem phoneme and then replace it or put another phoneme in parallel. Thus, the adapter would not be throwing away a pronunciation that is otherwise robust and is based on extensive training or analysis. An algorithm that automatically merges existing pronunciations with phonetic recognition results will be important future work.

#### **4.5.5 `spRcg_PhoneProbDisplay`**

For now, merging existing pronunciations with phonetic recognition must be done by hand using `gramCracker`. The results of phonetic recognition are helpful to the programmer performing this task, but they do not provide enough information. Consider the task of a programmer who wants to alter the pronunciation of a word. If she/he compares the

original pronunciation with the phonetic transcription, several of the phonemes may be different. The problem is that since many phonemes are acoustically similar, they may have similar probabilities of being recognized. In other words, the “correct” phoneme might not be the phoneme that the recognizer matches with the highest probability. However, if it is second or third best and is matched with a reasonably high probability, its presence in the dictionary’s pronunciation should not cause a recognition error (unless there are two words that have very similar pronunciations).

At first glance, it would seem that using an NBest recognizer would solve this problem by providing several choices for phonemes in the parts of the sentence where they are ambiguous. The problem with this is that there are so many phonemes in a sentence, thus so many phonetic transcriptions that have similar probabilities, that N would have to be extremely large before the problem phoneme or phonemes are identified. In other words, using an NBest recognizer asks “what are all the good choices of pronunciations?” which is the wrong question. The right question is “what is wrong with the existing pronunciation?”

The `spRcg_PhoneProbDisplay` mega-widget provides visual feedback that helps the programmer answer that question. Further, it serves as a teaching tool to give programmers more insight into how the recognizer matches sounds with phonemes. Of course, one of the goals of the speech toolkit is to make it unnecessary for the programmer to know the inner workings of the recognizer. However, that does not mean that it is not helpful!

The first type of visual feedback, shown in Figure 4-4, displays a two dimensional array of the probabilities generated by the multi layered perceptron. Each row corresponds to a particular phoneme; each column, to a ten millisecond sample frame. The height of each bar corresponds to the probability of that phoneme occurring in that frame, as determined by the MLP. (Since this is just the output of the MLP, not the viterbi search, the probabilities do not use any information from the vocabulary and grammar). For each frame, the most probable phoneme is highlighted in blue. In the figure showing the author saying the word “reload,” it is interesting to see how several phonemes, namely “r,” “l,” and “w,” have high probabilities during the utterance of the “r” sound in the word. This is not surprising because these three phonemes are acoustically similar. There is no such ambiguity

during the initial and final silence, since this sound is acoustically distinct from other speech!

The second form of visual feedback, called a phonetic backtrace, shows the frame by frame probabilities as determined by a viterbi search through a particular sentence. For each frame, the probability of the most likely phoneme - as determined by the viterbi algorithm, is displayed. By looking at these probabilities, the user can easily see which phonemes in the pronunciation, as specified in the vocabulary - matched well with the utterance and which ones matched poorly.

It is important to distinguish phonetic backtrace from the output of the MLP and from the phonetic transcription described earlier. The output from phonetic backtrace is the list of phonemes and probabilities, given that the sentence **MUST** be the one specified in advance. In other words, it is like a normal recognition, but with two differences. First, the grammar is highly constrained because it contains one and only one sentence. Second, instead of performing a word by word backtrace to determine the sequence of words, the recognizer performs a phoneme by phoneme backtrace to determine the sequence of phonemes (while also preserving probability information, which is usually of no interest in the word by word case.) In contrast, phonetic transcription places very little constraint on the grammar. Thus, it finds a sequence of phonemes that have a high probability, which is good. But, this sequence does not necessarily match any sequence through a real grammar containing words.

`spRcg_PhoneProbDisplay` uses special commands in `spRcg_recognizer` and `spRcg_controller` to access MLP data and to perform phonetic backtraces.

#### **4.6 GramCracker: A Tool to Build Vocabularies and Grammars**

GramCracker is a tool to aid applications programmers in creating, editing, and testing vocabularies and grammars. Its main display is shown in Figure 4-5.

The gramCracker display is divided into two parts: the upper for editing vocabularies and the lower for editing grammars.

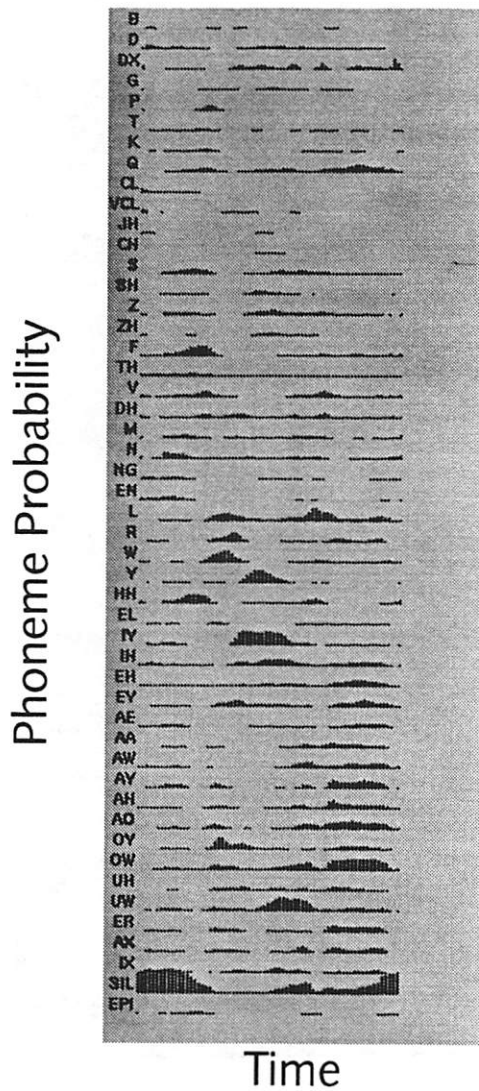


Figure 4-4. Output of MLP for an utterance of "reload."

The vocabulary section shows a graphical representation of each words' pronunciation. Phonemes are represented by circles; transitions, by edges between the circles. The programmer can edit the pronunciation using a mouse and keyboard. Using the mouse, s/he can add or delete edges and can select and un-select phones. Keyboard commands allow cutting, copying, and pasting. The programmer can add phonemes by clicking on the phoneme buttons in the area above the vocabulary display. These buttons also provide a



design. This restriction would not be needed in an all software recognizer, but it leaves enough flexibility to allow most possible pronunciations, anyway. Self loops (edges starting and ending in the same phoneme) are not allowed because they are already present in their individual phoneme models: they model the expected duration of each phoneme.

The grammar display shows the connectivity of the grammar and, optionally, shows the words in each grammar cluster. The grammar connectivity specifies the allowable word order: a sentence is legal if it can be described by a path through the grammar clusters, starting at the "START" cluster and ending at the "END" cluster. There is no left to right restrictions on grammar edges; any cluster may follow any other cluster. Self loops are also legal: they allow repetition of words in a cluster.

As with pronunciations, the grammar connectivity can be edited with the mouse: first select a cluster with a left click, then right click to toggle the connectivity to another cluster. The programmer has complete control over a cluster through the cluster editing dialog, obtained by double clicking a cluster. This dialog, shown in Figure 4-6, controls which words are in the cluster, as well as what edges go in and out of the cluster.

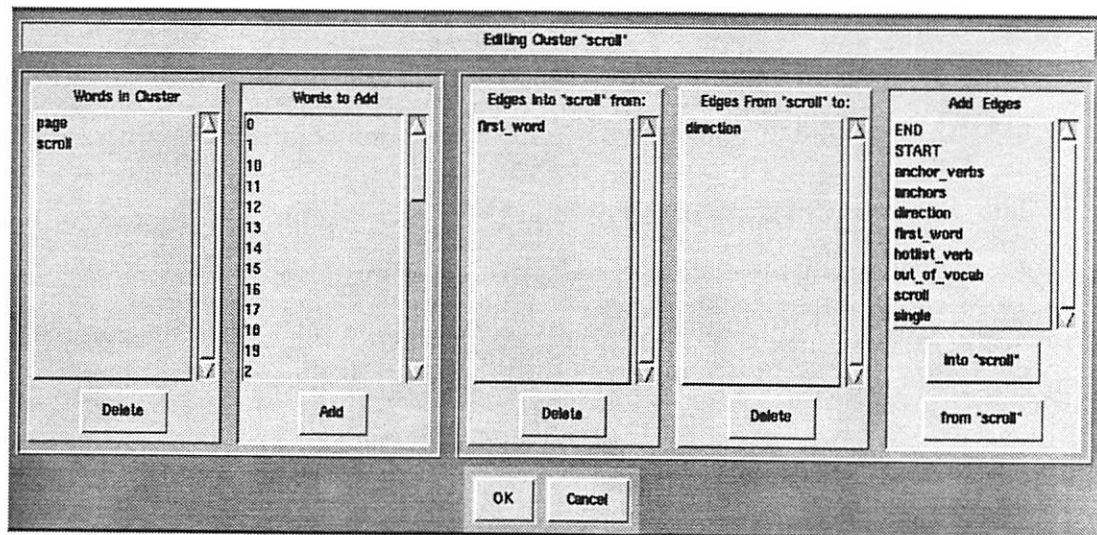


Figure 4-6. Grammar cluster dialog.

GramCracker itself uses speech recognition in two ways. First, the programmer can create new words in the vocabulary by dictating them to gramCracker, which performs phonetic recognition as shown in Figure 4-7. This technique of generating words is the same as is

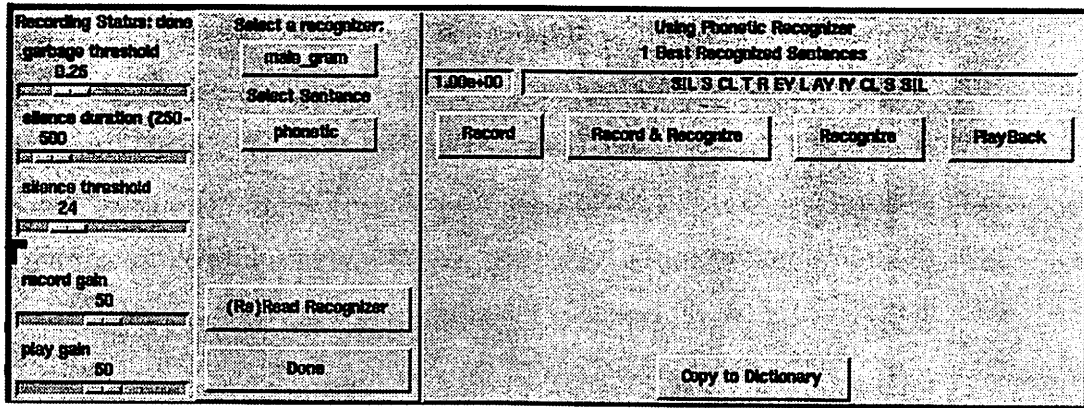


Figure 4-7. Performing phonetic recognition in GramCracker.

used by the `spRcg_adapter` mega-widget to alter the pronunciation of existing words. As such, it is not the best way to generate speaker independent pronunciations; the pronunciations will be that of the programmer. On the other hand, it is a quick and convenient way to generate the first draft of new words, which can then be edited in the vocabulary window. Furthermore, it provides a good way for the programmer to learn how the phonemes correspond to actual speech, so it serves to help educate the programmer.

The second way of using speech is to test the entire grammar. The programmer can run a recognizer using the dictionary and grammar being edited as shown in Figure 4-8. This way, he/she can have a quick edit/use cycle, so he/she can try out the grammar to see if it feels natural.

#### 4.7 Design Example

Let's create a simple application that draws a number of objects as commanded by the user. The user draws from one to four circles, squares, or triangles of any color (as long as the color is red, blue or green). Spoken commands should be something like "draw two green circles."

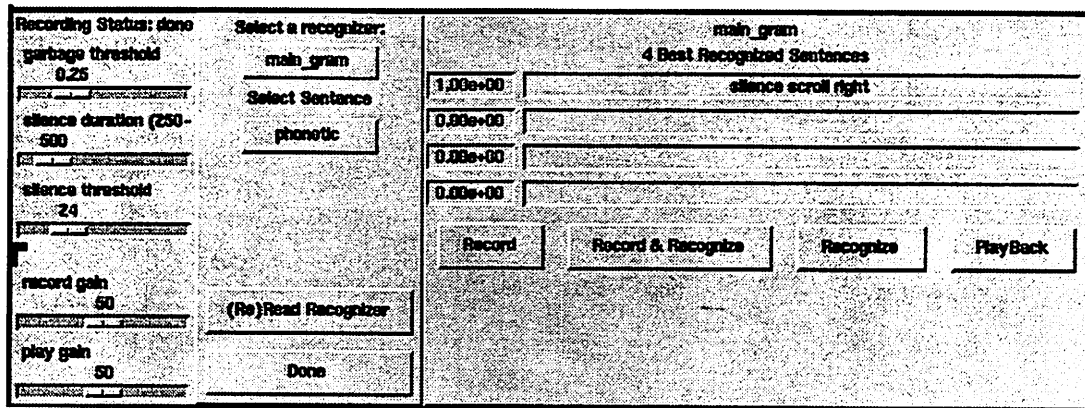


Figure 4-8. GramCracker using the full vocabulary.

#### 4.7.1 Creating the Grammar and Vocabulary

1. Run gramCracker.
2. Choose use recognizer from the recognize menu
3. Click the phonetic button.
4. click *Record & Recognize* then speak the word “one.”
5. Click copy to Dictionary Enter the name of the word (one) and click ok
6. Repeat from step 4 for the words “two,” “three,” “four,” “red,” “green,” “blue,” “circle,” “square,” “triangle,” and “draw.”
7. Select new cluster from the edit menu.
8. Type “action” as the name of the cluster.
9. Select the word *draw*; click the *add* button, click *ok*.
10. Repeat this action to create a cluster called *number* that contains *one*, *two*, *three*, and *four*; one called *color* containing *red*, *green*, and *blue*; and one called *object* containing *circle*, *square*, and *rectangle*.
11. select edit->new word. Call the word not\_recognized Select the first state in not\_recognized’s model, then press the gar button.
12. Create a new cluster called garbage that has one word: not\_recognized.



13. Click and drag the various clusters to arrange them neatly. Add edges between them by left clicking on the source cluster and right clicking on the destination cluster. If you do this again, the edge will go away; you should do this to remove the edge from START to END. The grammar should now look like Figure 4-9.

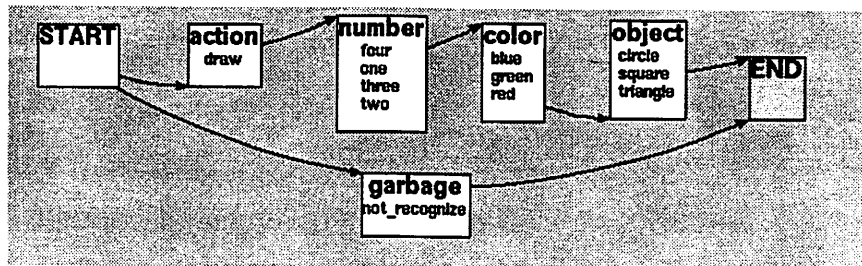


Figure 4-9. Demo grammar.

14. The grammar and dictionary are now complete, and could be used as is. However, we will do a few more enhancements. First, we will add optional silences at the beginning and end of the pronunciation of each word.
15. select edit->initial silence->optional
16. select edit->final silence->optional

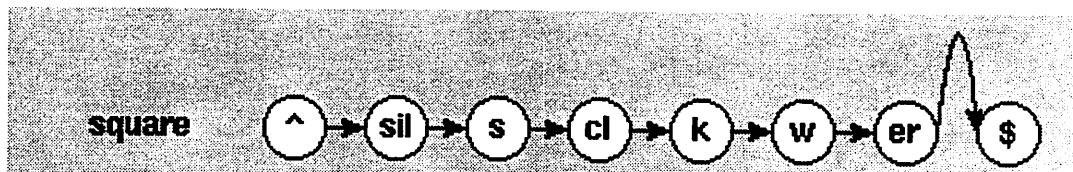


Figure 4-10. Transcribed pronunciation of "square."

17. Next, to demonstrate a more advanced way to modify pronunciations, we will alter the

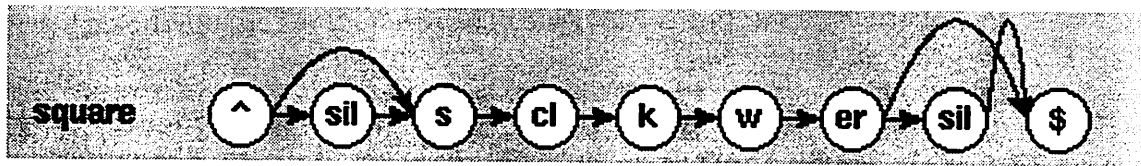


Figure 4-11. "Square" with optional starting and ending silence.

pronunciations of the words "circle," "square," and "triangle" so that they will also match the pronunciations for their plurals: "circles," "squares," and "triangles."

18. First, click on the word's last phoneme before the final sil (silence). It should turn yellow. Now click on the "s" button in the fricatives section of the phoneme buttons at the top of the window. Now right click on the "\$" phoneme to add an edge from the "s" to the end of the word. Now the word has an optional "s" as well as an optional silence at the end of its pronunciation. Repeat the process. For all three words.

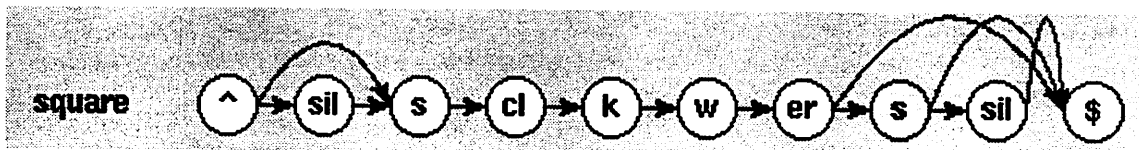


Figure 4-12. "Square" with optional "s."

19. Note that you could also create the plurals and save them as separate words. You can create them one of two ways. First, you could dictate them as you did the other words. Or, you could create them manually by selecting edit->New Word. Then you can copy and paste from the old, singular pronunciations: click to select the first phoneme, then shift-click to add select the rest of the phonemes (do not select "^" and "\$"). Then copy them with Control-C. Select the "^" phoneme in the new word, then paste the phonemes with Control-V. (For completeness, the cut command is Control-X). Now you can add the "s" phoneme as described earlier.

20. As a final step, save the grammar using File.->Save As and enter demo.gram as the file Name. Figure 4-13 shows a listing of the file containing the complete grammar and dictionary. The first line is a keyword. The second line contains the dictionary, for-

```

spRcg
{silence {{p {^ sil sil {$}}} {e {12 1E E}}}} {red {{p {^ sil w r
eh s vcl d ix sil {$}}} {e {12 1 1 1 1 1 1 1 1E E}}}} {green {{p
{^ sil k r iy n sil {$}}} {e {12 1 1 1 1 1E E}}}} {blue {{p {^ sil
l uw sil {$}}} {e {12 1 1 1E E}}}} {draw {{p {^ sil k r ao sil {$}}}
{e {12 1 1 1 1E E}}}} {circle {{p {^ sil s er cl p sil {$}}} {e {12
1 1 1 1 1E E}}}} {square {{p {^ sil s cl k w er sil {$}}} {e {12 1
1 1 1 1 1E E}}}} {triangle {{p {^ sil t w ay sil m el sil {$}}} {e
{12 1 1 1 1 1 1 1 1E E}}}} {one {{p {^ sil w ah n z sil {$}}} {e {12
1 1 1 1 1E E}}}} {two {{p {^ sil t uw sil {$}}} {e {12 1 1 1E E}}}}
{three {{p {^ sil dh ax r iy sil {$}}} {e {12 1 1 1 1 1E E}}}} {four
{{p {^ sil f sil {$}}} {e {12 1 1E E}}}} {not_recognized {{p {^ sil
gar sil {$}}} {e {12 1 1E E}}}} {circles {{p {^ sil s er cl p sil
$}}} {e {1 1 1 1 1 1E E}}}} {squares {{p {^ sil s cl k w er s sil
$}}} {e {1 1 1 1 1 1 1E E1 E}}}} {triangles {{p {^ sil t w ay sil m
el s sil $}}} {e {1 1 1 1 1 1 1 1E E1 E}}}}
{START { }} {END { }} {action {{draw 1.0}} {number {{one 1.0} {two
1.0} {three 1.0} {four 1.0}}} {color {{blue 1.0} {green 1.0} {red
1.0}}} {object {{circle 1.0} {square 1.0} {triangle 1.0}}} {garbage
{{not_recognized 1.0}}}
{from {{START {{action 1.0} {garbage 1.0}}} {action {{number 1.0}}
{number {{color 1.0}}} {color {{object 1.0}}} {object {{END 1.0}}}
{garbage {{END 1.0}}}}} {to {{action {{START 1.0}}} {number {{ac
tion 1.0}}} {color {{number 1.0}}} {object {{color 1.0}}} {END
{{object 1.0} {garbage 1.0}}} {garbage {{START 1.0}}}}}
{START {{x 51} {y 75}}} {END {{x 620} {y 105}}} {action {{x 158}
{y 73}}} {number {{x 299} {y 75}}} {color {{x 413} {y 105}}} {object
{{x 497} {y 84}}} {garbage {{x 339} {y 177}}}

```

Figure 4-13. Listing of demo grammar and dictionary file.

matted as a TclX keyed list, with the words as a key. Within each word are two keys: p for the pronunciation, e for the edges. The third line contains the grammar, also a TclX keyed list, with the clusters as the keys. The final line contains optional position information for gramCracker to draw the clusters.

#### 4.7.2 Creating the Application

The code to create the speech recognizer and its associated mega-widgets is shown in

```
# be sure to set environment variable SPRCG_LIBRARY to the libtcl
directory
source "$env(SPRCG_LIBRARY)/spRcg.tcl"
source "$env(SPRCG_LIBRARY)/spRcg_wigits.tcl"

frame .buttons -relief ridge -borderwidth 2
frame .s -relief ridge -borderwidth 2

# create an object to record and play audio
spRcg_recorder my_recorder

# connect to the AF server
# your machine must already be running AF!
my_recorder open $env(DISPLAY)

# create an itcl object to handle the vocabulary and grammar
spRcg_gramDict my_gramdict

# specify the file that contains speaker independent information
# about phonemes. For now, this is the only such file, so use it.
my_gramdict config -mlpFileName "$env(SPRCG_LIBRARY)/
phone.l6.r6.L128.P3.8khz.M0.Norm.hard.weightsq.mlp"

# read the grammar that you created
my_gramdict readGramDict "./demo.gram"

# create a speech recognizer object;
# let my_gramdict give it the proper arguments in the proper format
eval spRcg_recognizer my_recognizer [my_gramdict giveRecogCon-
structorArgs]

# create an itcl mega-widget that has sliders to control recording
parameters
# it also lets you record, recognize, and play with simple high
level commands
spRcg_controller .s.my_controller -recognizerName my_recognizer -
recorderName my_recorder

# create a display to show the recognizer's results
# show the 2 best matches
# the program doesn't really need this widget, but it's fun to have
spRcg_NBestDisplay .s.my_answers -NBestDepth 2
```

Figure 4-14. Creating the speech objects and widgets.

Figure 4-14. All the programmer needs to do is create a `spRcg_recorder` and open the connection to AF (which is a separate, publicly available program.) Then, read in the `spRcg_gramDict` that was previously created while using `gramCracker`. Next create a recognizer and then a controller widget. Finally, create a display widget to see the n-best answers.

The procedure shown in Figure 4-15 performs the actual speech recognition and turns on

```
# turn on or off recognition depending on the state of $recognitionOn
# when recognition is on, continuously record, recognize, and execute
# verbal commands
proc doRecognition {} {
    global recognitionOn

    if { [ .s.my_controller alreadyOn] } {
        if { ! $recognitionOn } {
            # this will kill the recognizer (invoked in a previous call to
            # doRecognition) in case it is still waiting for a sentence
            .s.my_controller abort
        }
    } else {
        # there are "update" commands in the recordRecog method,
        # so events will be processed even while the loop is running
        while { $recognitionOn } {
            set answer [.s.my_controller recordRecog ""]
            .s.my_answers display $answer
            obey_command [lindex $answer 1]
        }
    }
}
}
```

Figure 4-15. Controlling recognition.

and off recognition as the user turns on and off the *Recognizer On* button. In fact, it takes only one line of code to do the recognition:

```
set answer [.s.my_controller recordRecog ""]
```

Most of the rest of the code responds to the user's button press.

The code in Figure 4-14 and Figure 4-15 is generic, in the sense that almost any speech recognition application could use almost identical code.

The *obey\_command* procedure, shown in Figure 4-16, parses the recognized sentence and draws the objects accordingly. This part of the program is application dependent; each application should have a similar procedure that parses its own specific grammar or grammars according to the application's own program flow. The rest of the code for the demonstration application is shown in Figure 4-17.

```

proc obey_command { sentence } {
  if { [llength $sentence] != 4 } return
  # clear the canvas
  .c delete all
  set quantity [lindex $sentence 1]
  set color    [lindex $sentence 2]
  set shape    [lindex $sentence 3]

  switch $quantity { one { set qn 1 } two { set qn 2 }
    three { set qn 3 } four { set qn 4 }
  }
  # specify the initial positions
  # rectangle needs 3 points; square and circle just need two
  # rename shape to a word recognized by the canvas widget
  switch $shape {
    square {
      set x(1) 10; set y(1) 10; set x(2) 60; set y(2) 60
      set numpoints 2
      set shape rectangle
    }
    circle {
      set x(1) 10; set y(1) 10; set x(2) 60; set y(2) 60
      set numpoints 2
      set shape oval
    }
    triangle {
      set x(1) 35; set y(1) 10; set x(2) 60; set y(2) 60
      set x(3) 10; set y(3) 60
      set numpoints 3
      set shape polygon
    }
    default { return }
  }
  # draw each object
  for {set i 0} {$i < $qn} {incr i} {
    set command [list .c create $shape ]
    # put all of the objects points in the command
    for {set j 1} {$j <= $numpoints} {incr j} {
      lappend command $x($j) $y($j)
    }
    lappend command -fill $color
    eval $command

    # move all the points to the right by 100
    for {set j 1} {$j <= $numpoints} {incr j} { incr x($j) 100 }
  }
}

```

Figure 4-16. Parsing the recognized sentences.

```

proc setup_demo {} {

    canvas .c -height 75
    set recognitionOn 0
    checkbutton .buttons.recogButton -text "Recognizer On" -command
doRecognition \
        -variable recognitionOn
    button .buttons.quitButton -text "Quit" -command exit

    pack .c -side top -fill both -expand 1
    pack .s -side top -fill x -expand 1
    pack .buttons -side top -fill x -expand 1
    pack .buttons.recogButton .buttons.quitButton -side left -fill
both -expand 1
    pack .s.my_controller .s.my_answers -side left -fill both -expand 1
}

```

Figure 4-17. The rest of the application source code.



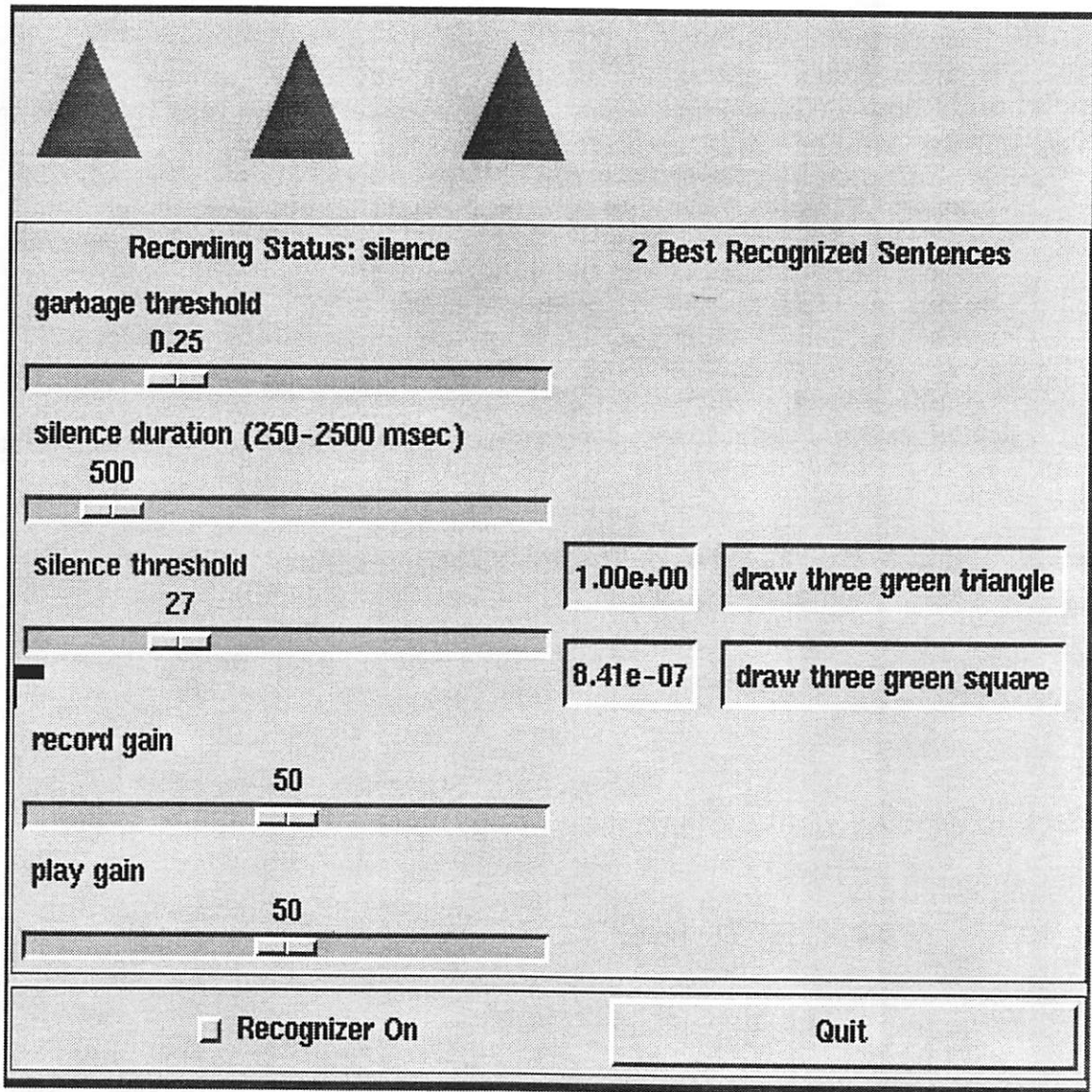


Figure 4-18. The finished demo application.

---

## 5 LOW POWER MEMORY DESIGN

---

### 5.1 Overview

Designing low power speech recognition hardware presented an opportunity to apply existing low power design techniques and also created the necessity of developing new low power techniques. In particular, the design exposed the need for new, low power memory designs. This chapter describes the design and use of new, low power SRAM and ROM designs, implemented as part of the lager[22] CAD system.

Previous research has developed techniques for low power CMOS IC design[3][4][5][10]. In order to understand low power CMOS design, one must consider the equation for power consumption:  $\text{Power} = CV^2f_{0 \rightarrow 1}$ . Where C is the capacitance of the node being switched, V is the supply voltage, and  $f_{0 \rightarrow 1}$  is the frequency with which the node is switching. ( $f_{0 \rightarrow 1}$  should not be confused with the overall system clock; depending on the circuit and the data, a given node can switch at a rate much lower than the clock frequency.)

The most important aspect of the equation for power is its square law dependence on voltage. Since power varies most strongly with voltage, the first priority of low-power design is to have the circuits operating at the lowest possible voltages. In systems that have fixed throughput requirements, the best strategy is to lower the supply voltages to the point where the circuit is barely fast enough to meet the speed requirement.

In order to lower this supply voltage as much as possible, it is often advantageous to design fast circuits since they can meet a speed requirement at a lower voltage than can slower

circuits. At the architecture level, too, some of the most common high-speed techniques—parallelism and pipelining—can reduce the necessary clock speed, thereby allowing operation at lower voltages. In these cases, the higher speed design usually switches more capacitance than does the low speed design. However, the increase in power from the increased capacitance is more than offset by the power reduction from lowering the voltage.

Although the voltage is the most important factor in power consumption, it is also helpful to try to optimize the other two factors. As long as it does not significantly reduce the overall system speed, it is a good idea to reduce capacitances as much as possible. For example, transistors that are not in the critical path should have minimum size geometries in order to reduce gate and parasitic capacitances. Likewise, switching frequencies should be set as low as possible. For example, circuits should not be allowed to switch if their outputs will not be used. Also, circuits should be designed to reduce glitching, since glitches increase the effective switching frequency.

## 5.2 SRAM

The SRAM macrocell is a scalable design incorporated into the Lager and Viewlogic design systems. The chief design goals were low power consumption, high area efficiency, operability over a wide range of word sizes and number of words, operability over a wide range of supply voltages, process independence, and an easy to use electrical interface.

### 5.2.1 Low Power Techniques

Low power consumption was the SRAM's foremost design goal. In keeping with general low power CMOS design techniques, the SRAM is optimized for 1.5 volt operation, and can operate at even lower voltages if used at low speeds. Optimizing for 1.5 volts had two main effects. First, since the typical processes used to implement the SRAM have asymmetric device thresholds—0.7v for NMOS and 0.9v for PMOS, the difference in current drive between the devices is exaggerated. To compensate for this difference, the ratio of device sizes,  $W_p/W_n$  in devices in the critical path was as large as 4/1 (compared with typical values of 3/1 in most higher voltage designs) in order to create equal current drives and hence

symmetric rise and fall times. The second effect was to enable the low-swing bitline circuits described below.

The low power goal influenced the design at both the architectural and circuit level. At the highest architectural level, the SRAM uses a strategy that minimizes the amount of capacitance that switches with each read, thereby minimizing power consumption. The memory is broken up into individual blocks; Each block has an address pre-decoder so that during a memory access, only one block is enabled, so capacitance is switched in only one block. The result of this architecture is to trade a slight increase in delay for a large decrease in power consumption. The sram's architecture causes the time for the address pre-decoding to be added to the overall delay; if the sram performed block level selection at the same time as it performs a read or write, the pre-decode time would not be added to the overall delay, but all of the blocks would be activated every cycle, increasing the power consumption by a factor equal to the number of blocks.

The sram block structure uses two rows; the designer can specify the number of columns. By choosing the number of columns, the designer has control over the aspect ratio, as well as a limited ability to trade area for lower power. With fewer blocks, the sram uses less silicon area because there is less duplication of control, sense, and address decoding logic (although the number of memory cells does not change); however, each block will have more words and hence must consume more power to decode its address and to swing its bitlines.

The architecture within the blocks is also optimized for low power consumption. In particular, the sram uses as little column decoding, 2 to 1, as possible, in order to reduce the amount of capacitance that must be switched to read or write one bit of data. Of course, having no column decoding at all would have been even better, but would have made pitch matching the sense amps nearly impossible. The blocks are most efficient when used with fairly wide words, generally 16 to 32 bits wide, for two reasons. First, the layout is more efficient than for narrow words because wide words reduce the percentage of the area taken up by address decoding and control logic. Second, narrow words would have a larger percentage of the power consumption in the address decoding and control logic.

Fortunately, using wide words fits well with general low power techniques: it is best to access memory data in parallel, allowing slower operation, hence allowing low voltage and low power operation. For example, in the InfoPad frame buffer, the memory is access 32 bits at a time and multiplexed 8 bits at a time by the logic.

Most of the circuit level optimization for low power occurs in the bit lines and sense amps.

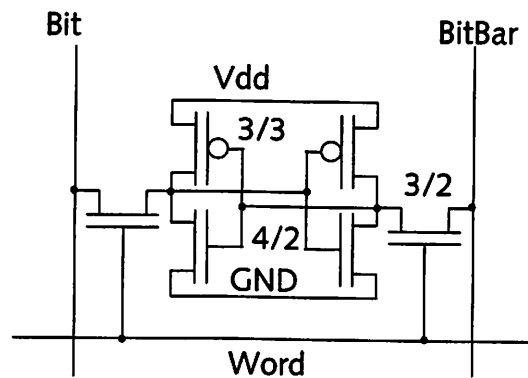


Figure 5-1. SRAM memory cell schematic.

The bitline precharge scheme is designed to reduce their voltage swing, particularly at low supply voltages. The bitlines are precharged through NMOS devices, so their maximum voltage is an NMOS threshold voltage (with body effect) below the supply voltage. With  $V_{dd}=1.5$  volts, the bitlines precharge to approximately 0.7 volts in a typical process. Thus, when the bitline is discharged, the voltage swing is reduced by over 50 percent compared to precharging to  $V_{dd}$ . If slower speed operation is acceptable,  $V_{dd}$  can be reduced, and the percentage savings is even larger.

The sense amps use a cross coupled latch topology because it is fast and consumes no static power (as would a differential pair, which has a constant bias current). In addition, the column select devices also function as cascode amplifiers: the drains of these devices are precharged to the supply voltage, while their sources are attached to the bitlines, which are precharged to a threshold drop below the supply. Thus, before there is any swing on the bitlines, these devices are barely off. When the voltage on a bitline changes a little, the device turns on, discharging the capacitance at its drain. Thus, the column select devices also create voltage gain into the cross coupled latch inputs.

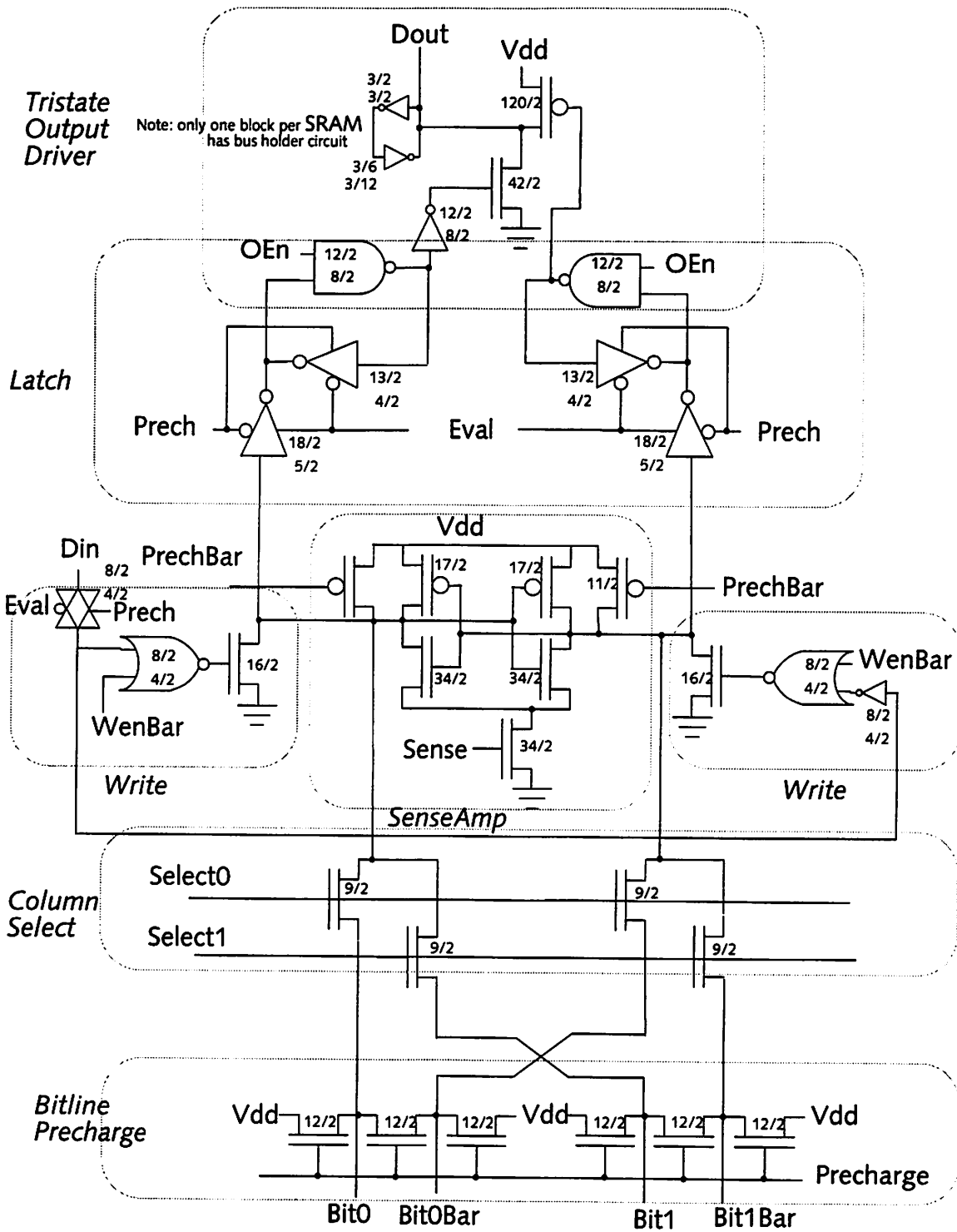


Figure 5-2. SRAM senseamp and write circuit diagram.

The sense amps also use a self-timed, non-glitching output to save power. Initially, the sense amp is precharged so that both the NMOS and PMOS devices are turned off, tristating the output. When the cross coupled latch resolves the data, it turns on one or the other device. Thus, there can be no spurious transitions on the output data bus, which can have very high capacitances.

Since the cross coupled latches of the senseamp provide positive feedback, they must not turn on until a sufficiently large signal - a few hundred millivolts - is present at their inputs. Therefore, the signal that enables the sense amp must be a self-timed signal.

### 5.2.2 Self Timing

In fact, all of the clock phases used by the SRAM are generated using self-timed circuits, for two reasons. First, to ensure correct functionality, as in the case of the sense enable signal, across process variations and different sized memories. Second, to ensure a simple designer interface: the designer only needs to supply one clock edge; the SRAM generates all of its own timing signals.

The self-timing follows the entire critical path through the sram by using “dummy” circuits that have the same delay as the circuits in the actual sram. First, in the address decoder, a circuit tracks the delay to decode a particular address. In essence, this dummy output is just like a word that always is selected, no matter what the input address. The actual address decoder is a tree structured NAND array, driven by the addresses and their complements. The tree structure, with minimum sized devices at the branches and larger devices toward the root, has relatively little total gate capacitance, and hence has reasonably low power consumption. Unfortunately, the nodes between the devices must be precharged to avoid charge sharing; fortunately, they only need to be precharged through NMOS devices to a threshold voltage below the supply

The output of the dummy row decoder drives a dummy word line that emulates the delay through an actual word line. The dummy word line is driven by an extra word line driver. The dummy word line is attached to NMOS gates that are the same size as the gates of the memory cells access devices, so both the rise time from the word line driver and the RC delay through the dummy word line (which, like the actual word lines, is made of polysil-

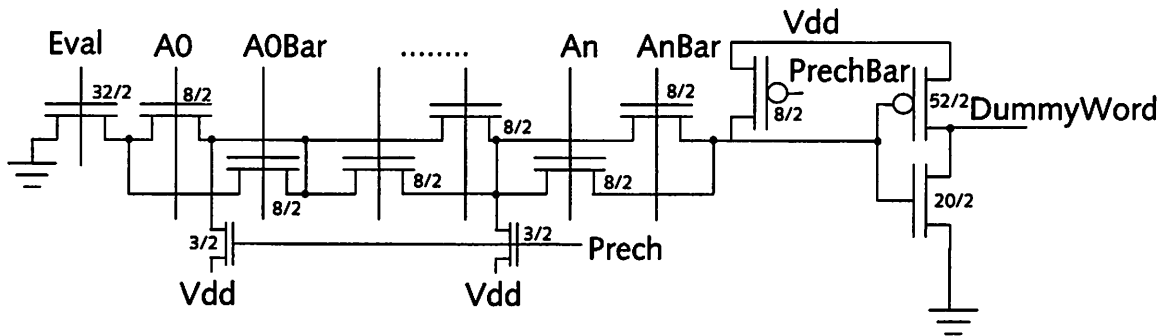


Figure 5-3. SRAM dummy word driver.

icon) should be the same as for the actual word lines, no matter what the supply voltage, number of words, number of bits, and changes in CMOS process

The output of the dummy word line serves two purposes. First, it triggers the senseamp enable signal. The senseamp must not be triggered before a small signal has been placed on the bit lines, which will occur a short time after the selected word line goes high. Since the dummy word line goes high at the same time, the senseamp will be enabled a short time after the word line is selected. The short time is the delay through the control logic and drivers that control the senseamp enable; this short time ensures that a sufficiently large signal is present on the wordlines before sensing. (The senseamp enable signal also waits for one of the two column select signals to become valid. There is no need for a dummy column select signal because it is easy enough to OR the two signals; ORing all of the word lines would entail a much larger delay.)

The second purpose for the dummy word line is to drive a dummy bitline through a dummy memory cell. The dummy memory cell is always precharge to contain the same data, so only one dummy bitline is needed (the other bitline would never change voltage, so it is not needed). The dummy bitline that is used has the same capacitive load as a regular bitline, and hence has the same delay.

Not surprisingly, the dummy bitline drives a dummy senseamp. The output of the dummy bitline is valid at the same time as the data is valid at the output drivers in the actual senseamps. At this time, the cycle for the sram is finished so the output of the dummy



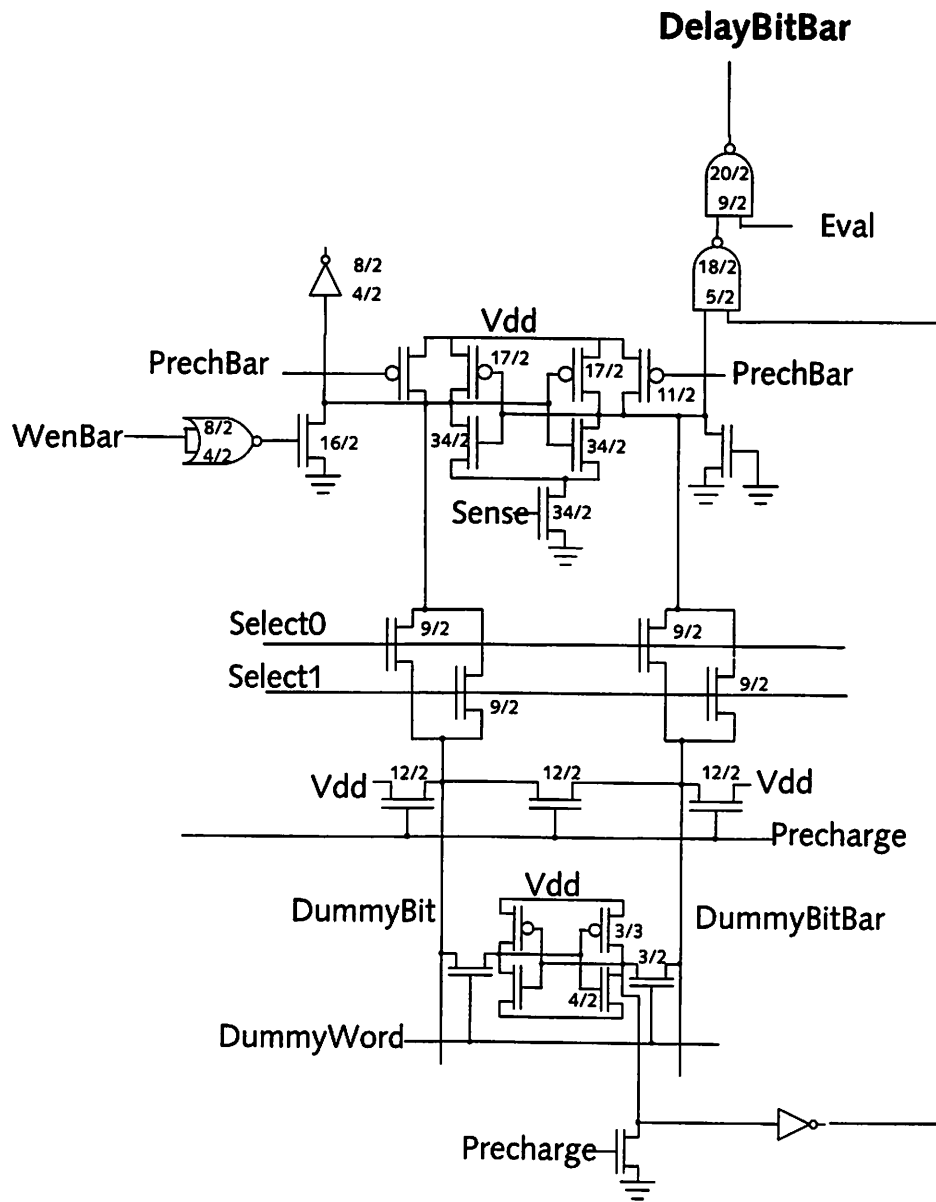


Figure 5-4. SRAM self-timing circuitry.

senseamp triggers the return to the sram's precharge state. (It is not necessary to duplicate the rise or fall time of the signal on the output data bus because the data at the input of the output drivers is latched, so the bus is driven even during the precharge phase.)

In addition to simplifying the interface and ensuring proper timing, self timing helps to maximize the SRAM's speed by ensuring that the various clock phases are properly proportioned.

### 5.2.3 Description of Operation

This section describes the sequence of events that comprise a read or write operation in the SRAM.

#### 5.2.3.1 Set Up and Pre-Decoding

The first step in a memory operation is assertion of valid data to the SRAM: namely, the address, the read/write signal, and (in the case of a write operation) the data. The most significant bits of the address must be valid early enough before the rising clock edge to be decoded so it can enable one of the blocks. The number of MSBs involved depends on the number of blocks; e.g. if there are 8 blocks, 3 MSBs are used. This pre-decoding operation is rather fast because it uses at most 2 stages of static CMOS logic. The output of the pre-decoders must be valid before the rising edge of the clock because they enable (or disable) the blocks' triggering off of the clock edge.

The setup timing of the other input signals is less stringent because they are latched by clock phases generated after the rising clock edge triggers the block. Thus, they are not latched until some time after the rising clock edge.

#### 5.2.3.2 Clock Trigger

A block's internal clock is triggered off of the rising edge of the global clock if that block is enabled by its pre-decoder. The trigger circuit is an SR latch with one edge triggered input and one level trigger input. This latch drives three clock signals: *prechargeBar*, *precharge*, and *eval*. *PrechargeBar* and *eval* have the same value but are separated in time by two inverter delays; *precharge* is their logical compliment. Both *prechargeBar* and *eval* are needed in order to eliminate short circuit currents during clock transitions.

When triggered, the clock signals initiate a read or write cycle. They are reset to the pre-charging state by the output of the dummy senseamp during read operations or by the state of the dummy memory cell during write operations.

### 5.2.3.3 Address Decoding and Wordline Driving

The clock signals cause the address drivers to latch the input address and drive each signal and its complement onto the nodes of the decoding tree. Initially all of these signals are precharged low, so there are no conducting paths in the tree. After each address (or its complement) is driven high, there will be one and only one path through the tree to a single word line driver. (There will also be a separate path for the dummy word line.)

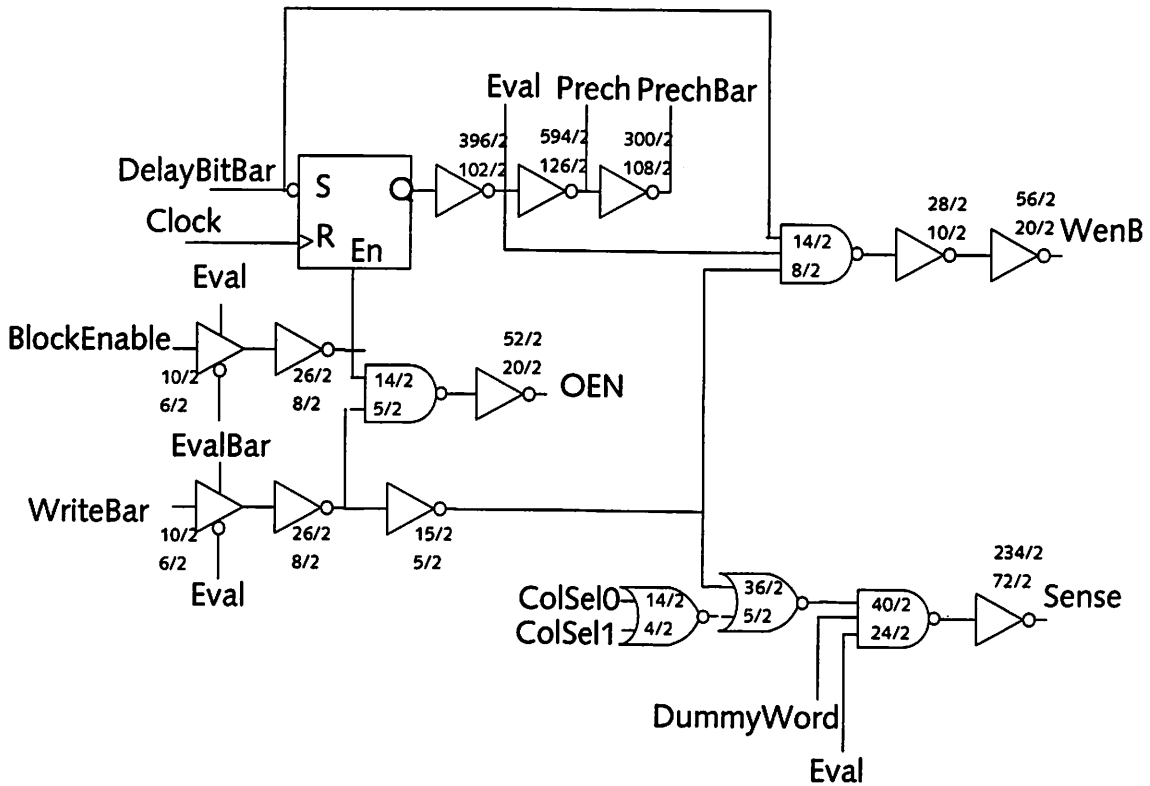


Figure 5-5. SRAM control circuits.

One bit of the address is used to drive the column select devices, which are attached to the senseamp and bit writing circuits. Their signals are also dynamic.

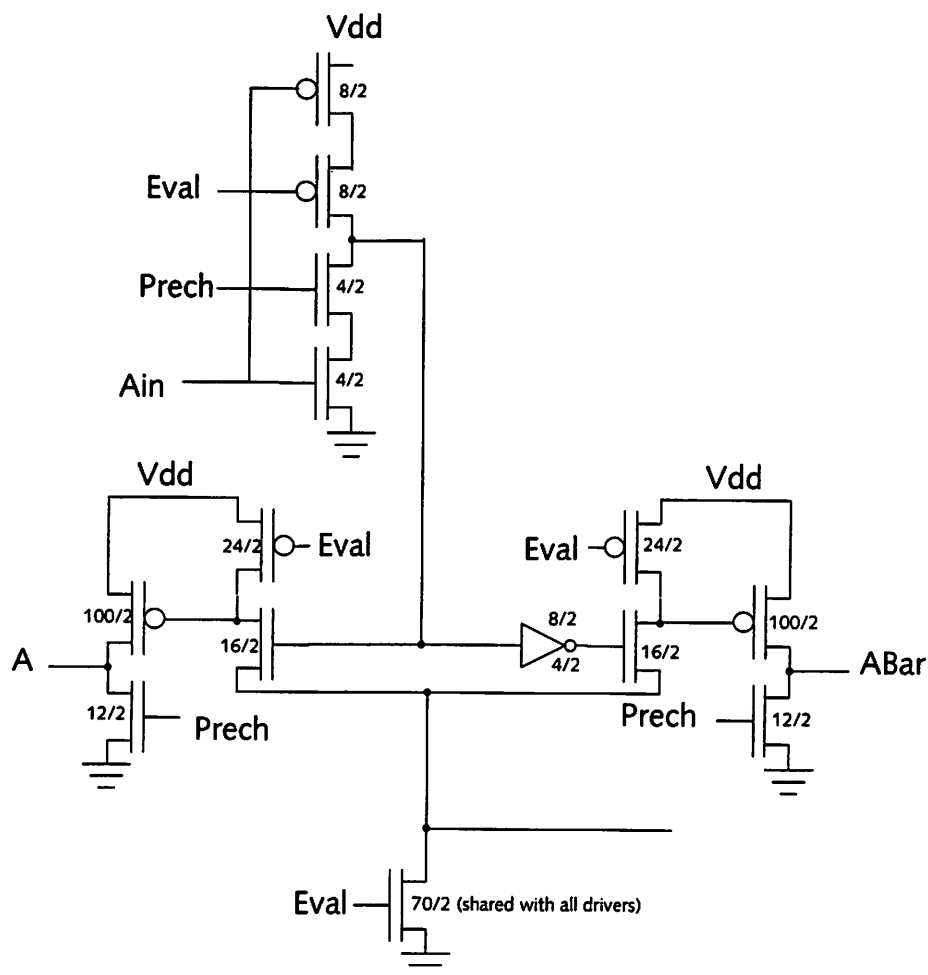


Figure 5-6. Address latch and driver.

Using precharged, as opposed to static, address lines has the benefits of speed and simplicity; its effects on power consumption are unclear. The speed and simplicity derive from the self timed nature of the address decoding: since the proper word line is enable automatically after all of the address lines are valid, there is no need for additional control logic or an additional clock phase to separate the address decoding from the rest of the cycle. The power consumed in the address decoding might be lower if there was a high degree of correlation between addresses from cycle to cycle because the address lines would switch less often. On the other hand, this savings might be offset by an increase in power for the extra control and clock logic needed. Plus, the capacitance on most of the address lines is not large because the tree structure has few transistors on the most significant

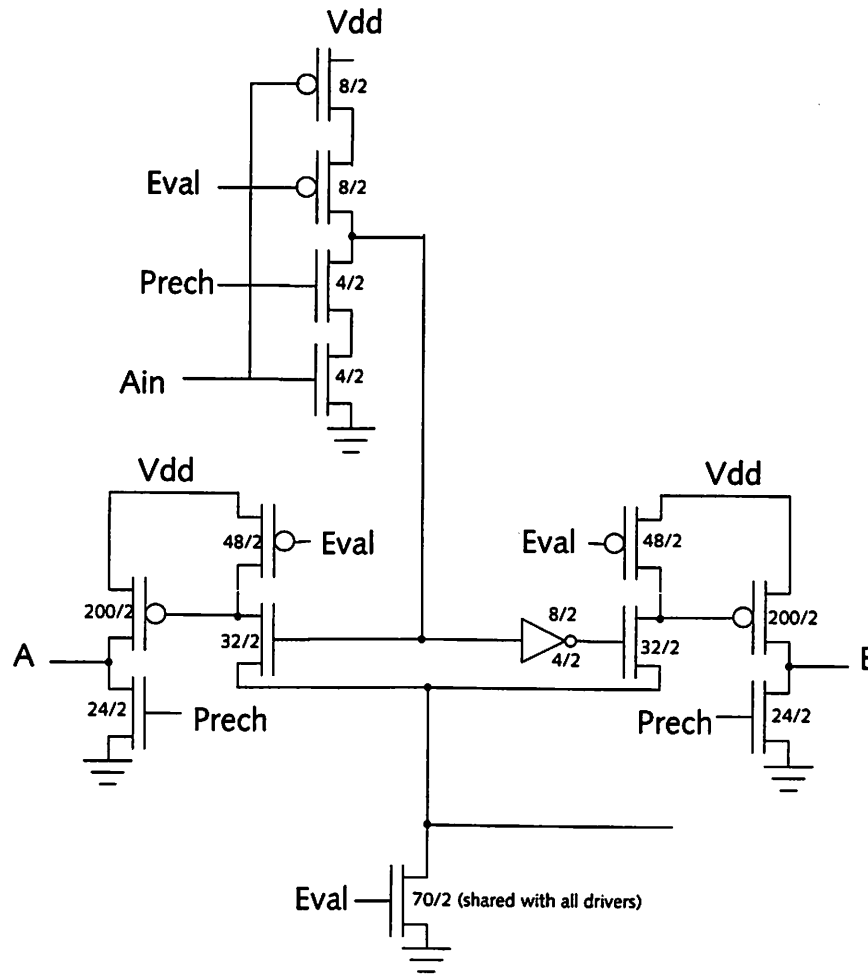


Figure 5-7. Column select latch and driver.

addresses. Finally, the increased speed of the dynamic logic can help with power consumption because it allows the sram to meet the same timing requirements at lower supply voltages.

During precharging and the start of the address decoding, all of the word lines are held low by inverters on the leaves of the decoding tree. When the input to one of these inverters is pulled low by the tree, it drives its wordline high. The wordlines are routed in polysilicon to allow the most compact memory cell in a two metal process, so it is important to make sure that the RC delay through the wordline does not dominate the delay of the SRAM. Two techniques are used to reduce this delay. First, the memory is split into two banks; cutting the length of the wordlines in half, and hence cutting the RC delay by a

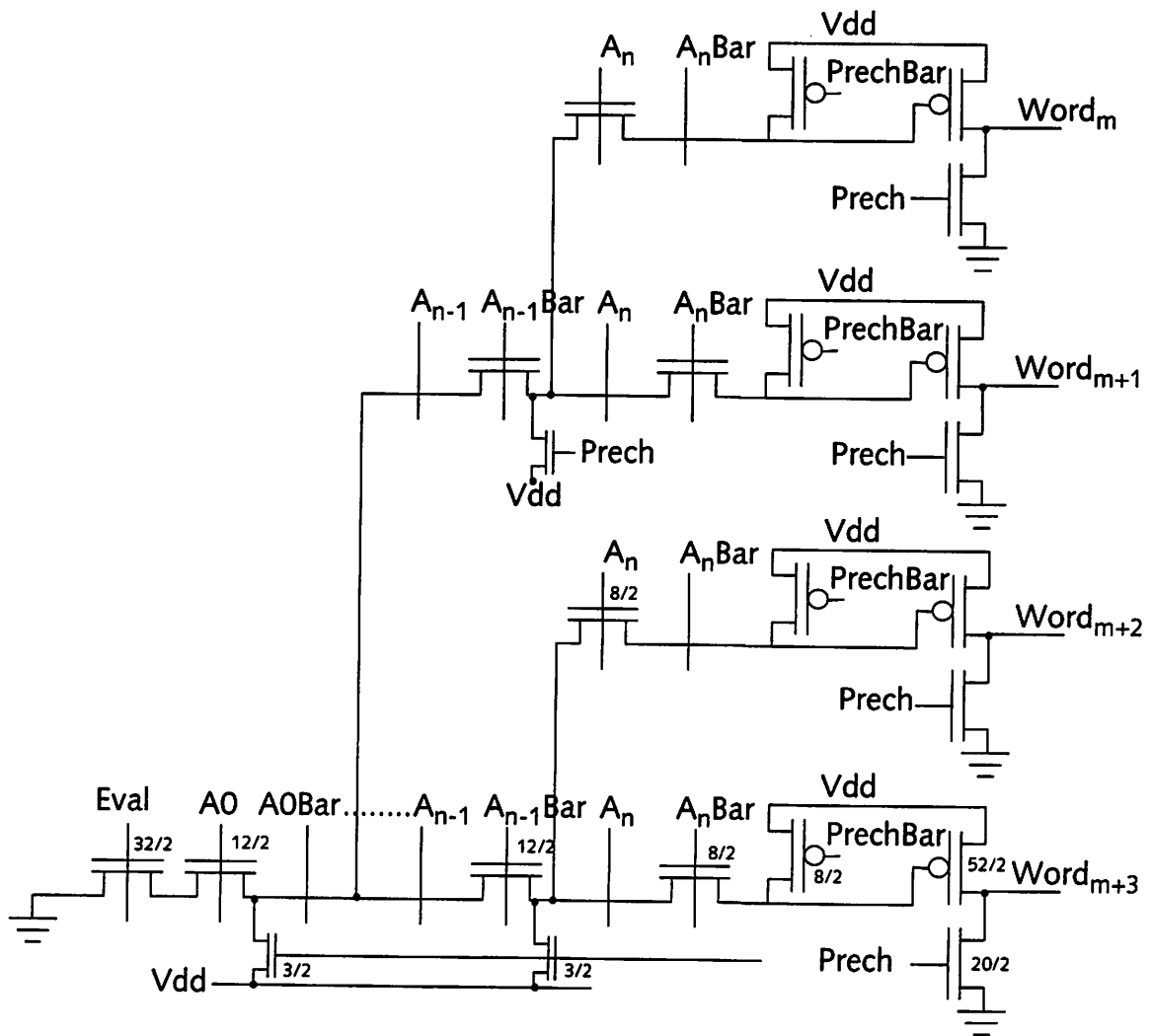


Figure 5-8. Tree-structured SRAM address decoder and row drivers.

factor of four. Second, each word line is actually composed of two polysilicon wires, one connected to each of the two access devices in each bit. Compared to a single word line, the gate capacitance on each line is cut in half, further reducing the RC delay. (Fortunately, using two word wires also produces a very compact memory cell layout, so the area of the memory is reduced!)

#### **5.2.3.4 Bitlines: Writing**

During a write cycle, the write data is latched and asserted on the bitlines simultaneous with the address decoding process, and is valid before the selected word line goes high. During a write, one bitline is pulled down to ground; the other is left at its precharged voltage. The bitlines are written through the column select transistors; in the bitline pairs that are not selected, neither bitline is pulled low. This is not a problem because the selected cells on these bitlines see the same voltages on their bitlines as during a read operation.

The timing between the wordline going high and the bitline being pulled low is not critical. If, for example, the wordline goes high before the column select signal goes high, the memory cell would begin to pull down one of the bitlines (as in a read operation), but the bitline will swing very slowly because the transistors in the memory cell are weak. The write circuit, in contrast, pulls down the bitline with strong devices. Thus, it will pull down the correct bitline faster than the memory cell pulls down the incorrect one, overpowering the memory cell, and thus writing the correct data even though it got a late start.

#### **5.2.3.5 Bitlines and Senseamp: Reading**

During a read cycle, after the word line goes high, the memory cell will pull down one of its two bitlines; the other bitline will not change voltage because the cell is pulling it up through an NMOS device, so it is stable at the precharged voltage.

#### **5.2.4 Layout**

The SRAM layout uses the MOSIS scalable CMOS design rules, so it is suitable for fabrication in a variety of processes. The cell library has also been ported to the HPo.6um process.

The sram cells use polysilicon word lines in order to reduce area. However, each wordline is actually two separate wires, so the RC delay through the wordline is less than it would be for a single poly wire.

The layout of the wiring between the blocks is also part of the SRAM's design. Using the Timplager tiler to do this wiring saves area in two ways as compared to using Lager IV's standard router (which is a part of Flint.) First, the hand-tuned tiling routine achieves the

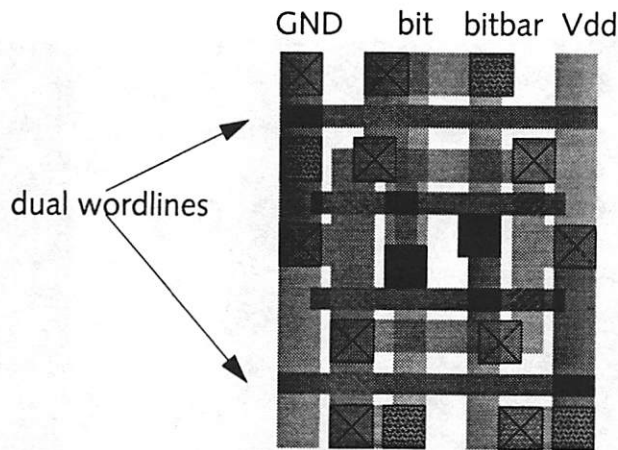


Figure 5-9. SRAM cell layout.

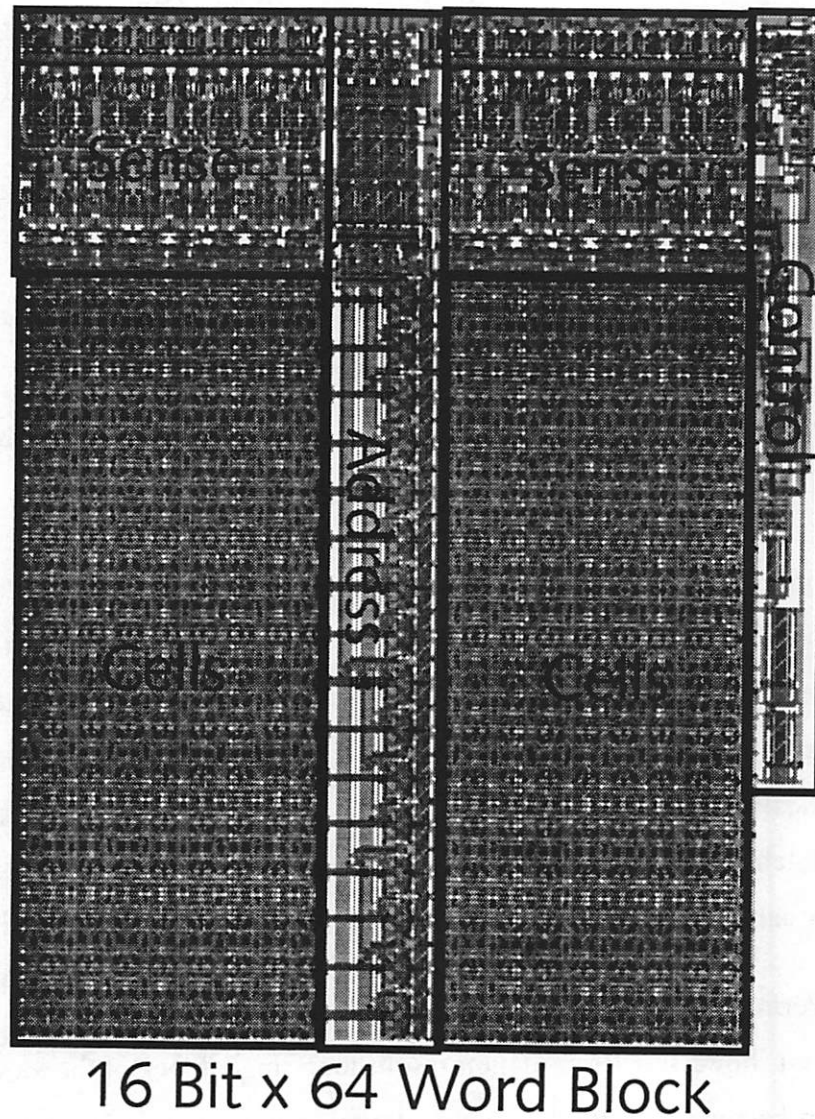
absolute minimum spacing between blocks and between wires in the routing channel; automatic place and route is less efficient. Since this hand optimization only needs to take place once—when the SRAM is designed—it is worthwhile. Second, the SRAM blocks can place some of their active layout in the routing channel, resulting in a further area savings. Specifically, two types of circuits are placed in the routing area: block decoders and weak static latches on the data lines. Both of these circuits reside on metal one and lower layers, so they can inhabit the same area as the metal two wires that are tiled over them.

### 5.2.5 Performance

Table 5-1 shows SPICE simulation results for a single block of the SRAM in a 0.6  $\mu\text{m}$  process. In keeping with the low power design strategy, the designer can trade off speed of operation versus power consumption. Since the global architecture of the sram determines that only one block is activated per operation, the total power consumption is the sum of one block's power plus the power from switching the data buses. Thus, increasing the size of the memory by adding additional blocks only has a secondary increase in power consumption. Namely, the power increase will be due only to the additional capacitance placed on the data bus from the extra blocks and from the wiring to those blocks.

Table 5-2 shows the effects on speed and power of changing the number of bits and the number of words in a block.





16 Bit x 64 Word Block

Figure 5-10. SRAM block layout.

### 5.2.6 Implementations

The SRAM has been fabricated as the core of two memory chips: a 64 kbit chip in 1.2  $\mu\text{m}$ , and the 96 kbit chip in 1.0  $\mu\text{m}$ , shown in Figure 5-11. These chips form the first and second generation InfoPad frame buffers. In addition, two other InfoPad chips use the SRAM as on-chip memories: the transmit chip has four memories with 160 bytes by 16 bits each; the receiver chip has one memory with 256 words by 8 bits. These InfoPad chips were fabri-

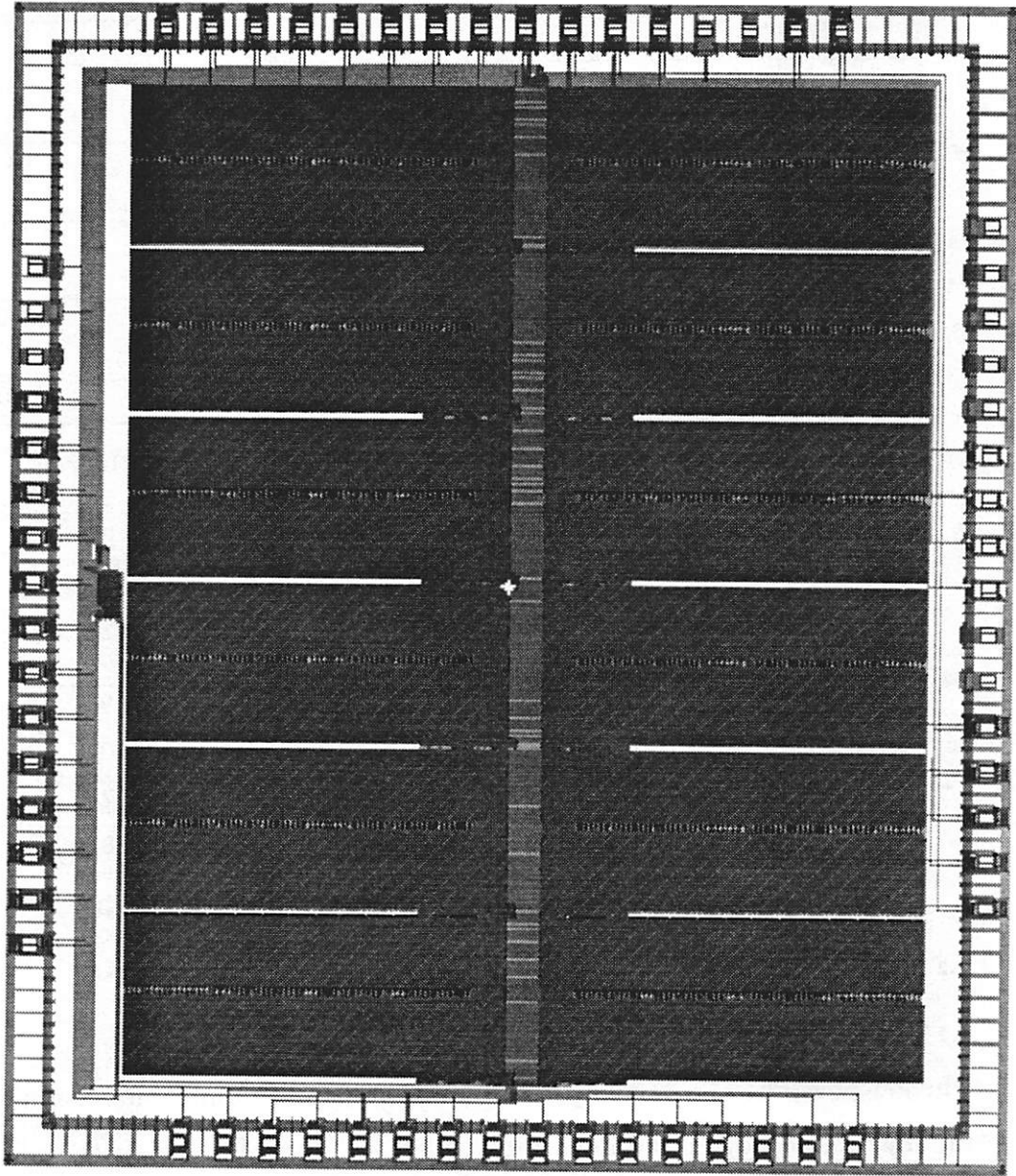


Figure 5-11. Layout of 3 kWord x 32 bit SRAM integrated circuit.

cated in a  $1.2\ \mu\text{m}$  process. The SRAM also was fabricated on the RASTA chip described in Chapter 6 and the MLP chip described in Chapter 7.

Future plans for the SRAM include use as on-chip cache memory and external main memory chips in a low-power, high-throughput microprocessor fabricated in  $0.6\ \mu\text{m}$  CMOS[2].

		Supply Voltage			
		1.25V	1.5	2.0	3.3
Cycle time		34.3ns	19.6ns	10.6ns	5.7ns
Access Time	0.0pF load	25.6ns	14.4ns	7.7ns	4.1ns
	0.5pF load	26.8ns		8.1ns	4.3ns
	1.0pF load	27.7ns		8.4ns	4.5ns
	2.0pF load	29.7ns		8.9ns	4.8ns
Energy/operation		20pJ	32pJ	64pJ	205pJ

Table 5-1. Voltage scaling a 128 word 12 bit 0.6  $\mu\text{m}$  SRAM.

Words	Bits	Access (ns) (2pF load)	Cycle time (ns)	pJ per Operation
128	6	16.1	18.6	25
	8	16.4	18.7	28
	16	17.2	19.8	39
	32	18.3	21.7	50
256	8	17.9	20.1	32
	32	19.7	23.1	78

Table 5-2. SRAM speed and energy vs. size at 1.5V.

### 5.2.7 ROM

### 5.2.8 Architecture

The low power ROM uses many of the same circuit and architectural techniques as does the low power SRAM. The ROM uses a tiled block architecture, with individual ROM blocks connected by a tiled interconnect. These interconnect cells also contain block pre-decoders that enable one and only one block in the entire ROM.

### 5.2.9 Circuits

The ROM uses a NOR array to store the data. When the data is “0,” a transistor connects the bitline to ground; when the data is “1” there is no transistor present.

Figure 5-12 shows the ROM's dynamic address latch. It precharges all of the address line

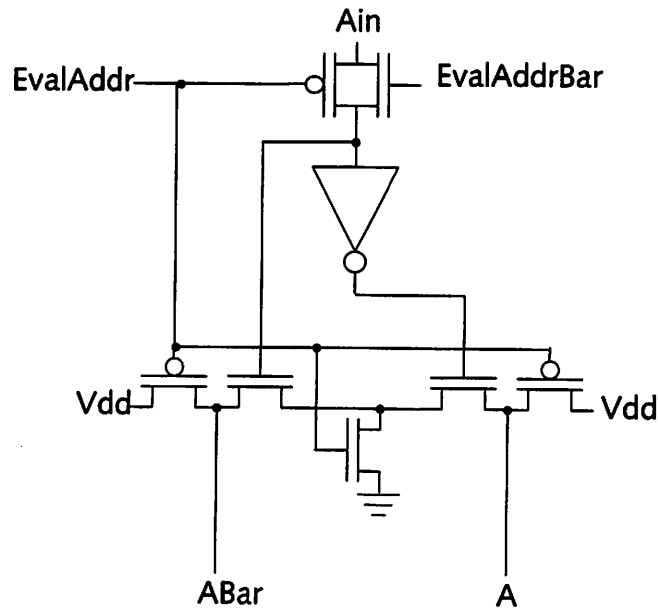


Figure 5-12. ROM address latch.

high, then pulls down either  $A$  or  $A\text{Bar}$  when evaluating. This configuration helps with high speed operation, because only NMOS devices are used during the critical evaluation period. The PMOS devices are active only during precharging, when speed is not as important.

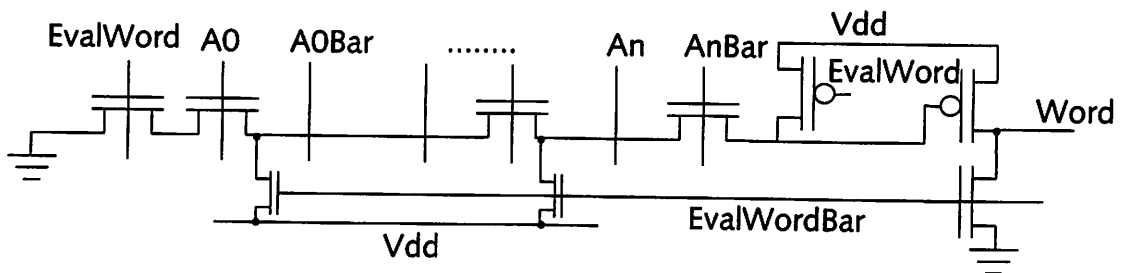


Figure 5-13. ROM address decoder and wordline driver.



the bit cell on that bit line (i.e. the data for that bit is a “1”), then the cascode device’s drain stays at  $V_{dd}$ . Just in case the bitline was not completely precharged, a weak PMOS device always pulls up the drain node to counter any leakage. In the other case, when there is a transistor on the bit line (i.e. the data is “0”), that device pulls down the bitline. This voltage on the bitline is amplified by the cascode device, causing a larger voltage drop at its drain. Although the NMOS device on the bitline is fighting against the weak PMOS pull-up at the cascode’s drain, the NMOS is much stronger because it is a larger device, has a lower threshold voltage, and has greater mobility. The output of the cascode amplifier goes through an inverter and then is sent through two different pass gates.

The rest of the circuit is designed, in combination with some additional self timing circuitry, to latch the data and drive the tristate output without glitching. First, consider the path to the PMOS output device. Initially, all of the clocks and control signals precharge its nodes so the PMOS output device is off. The circuit will stay in this state unless the senseamp output inverter switches (indicating the data is a “1”). In that case, the static latch will change state, turning the output PMOS device on, and sending a high signal to the output. In the other case (data “0”) the inverter never switches, so the rest of this path stays in the same state, keeping the PMOS device off.

Now consider the path to the NMOS device. This path is more complicated because it should change state only when the senseamp output inverter does **not** switch. Thus, the key to this path is to allow the signal to enter it only after sufficient time has passed that it is certain that the senseamp will not switch. This task is performed by dummy signals that generate the *ready* signal that enables a pass gate into this path.

This dummy signal is generated in a fashion similar to that of the SRAM. A dummy address decoder drives a dummy word line (with a worst-case load of a transistor on each bit) that drives a dummy bit (with data of “1”) on a bitline attached to a dummy senseamp. The output of the dummy senseamp’s inverter forms the *ready* signal because it represents the worst-case delay until the output of the senseamps are valid; after *ready* goes high, it is certain that no senseamp will change state.

After the *ready* signal goes high, the latch accepts the senseamp output and sends its signal to the NMOS transistor. When the *evalWord* signal goes low, both latches hold their signal until they are reset by the *OutputEnable* signal going low.

The *OutputEnable* signal only performs an active function when it is low. At that time, it resets the output latches to drive the output tristate. During operation, *OutputEnable* goes low at the start of every clock cycle to tristate the outputs. *OutputEnable* goes high when *EvalWord* is high, so it does not interfere with the senseamp's operation.

The ROM control circuit, shown in Figure 5-15, triggers off the rising edge of the clock. Its first action is to set Output Enable (*oen*) low, which causes the senseamp outputs to become tristated. If the enable signal is low, this is the only action that occurs; if enable is high, the controller generates the rest of the signals in the read sequence. In a multi-block ROM, the block level address pre-decoder generates this enable signal. Thus, all the blocks in the ROM tristate their outputs on a rising clock edge, then the single, enabled block proceeds with a read.

The control's first step in the read sequence is to raise the *EvalAddr* signal, which causes the address latches to pull down the proper address lines. The self-timing circuit that generates *addressReady* makes sure that either each address or its complement has been pulled low before *EvalWord*, which enables address decoding, goes high. In addition, *EvalWord* turns off precharging, turns on *oen*, and enables the senseamp operation.

The self-timing signal, *dataReady*, which comes from the dummy bit, resets *EvalWord* and *EvalAddress* to their precharging state.

### 5.2.10 Results

Table 5-3 shows speed energy and speed numbers for various voltages and memory sizes using a 1.2  $\mu\text{m}$  process.

### 5.2.11 Layout

Figure 5-16 shows the layout of a single ROM block.

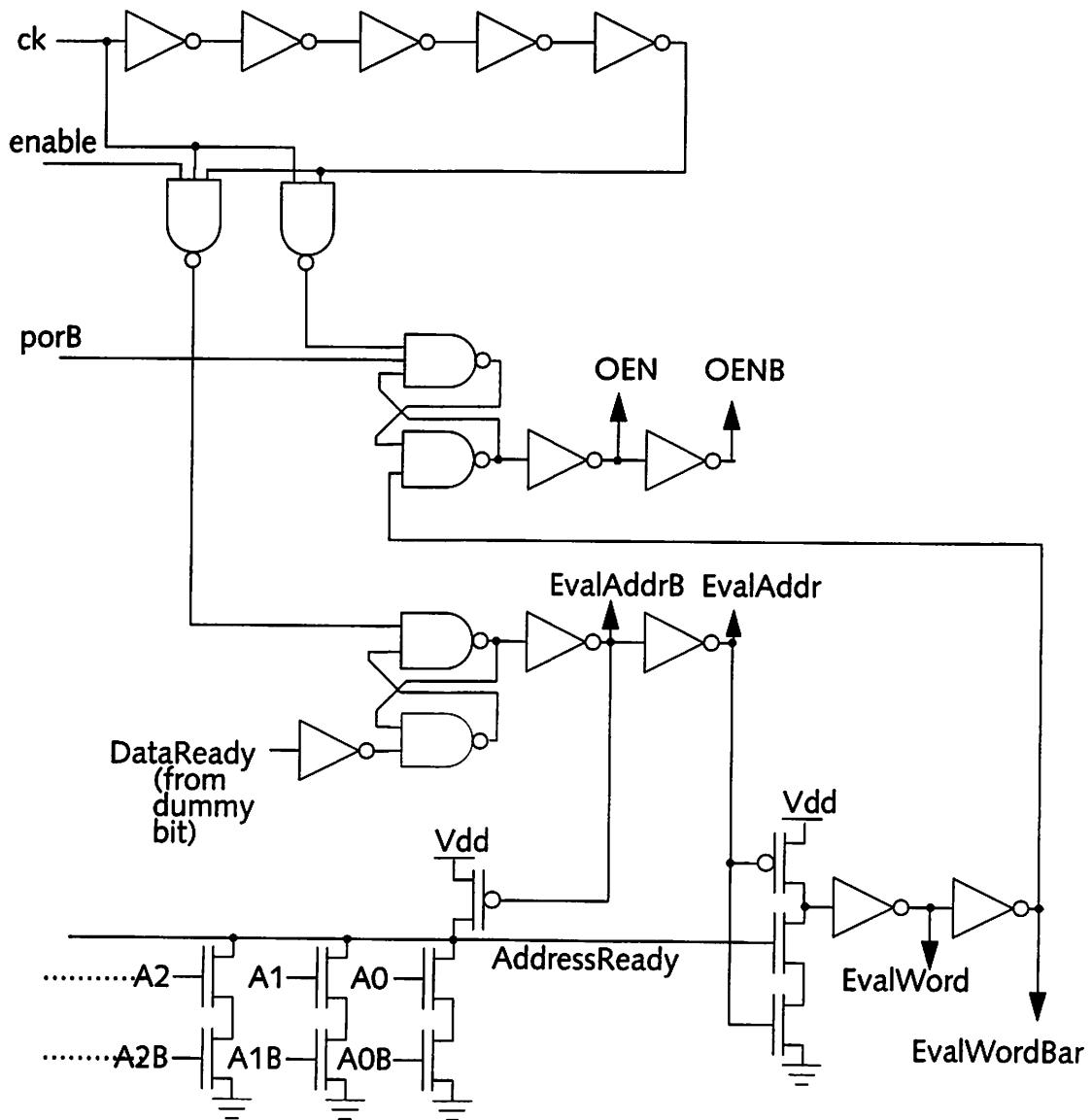


Figure 5-15. ROM control circuits.

## 5.3 How to use the Memories

### 5.3.1 Viewlogic and TimLager

The SRAM and ROM are part of the Lager IV silicon assembler system. To use the SRAM, the user must specify two parameters: WORDS and BITS. There is one optional parameter, BLOCKCOLS, which defaults to "1" if it is not specified. Lager IV calls the TimLager tiling program to create the layout for the memory. At the highest level, the memory is divided



Words	Bits	V <sub>dd</sub> (Volts)	Access (ns)	Cycle (ns)	Energy/ cycle (pJ)
16	4	1.5	52	70	14
16	4	2.1	23	31	29
16	4	3.0	12.3	17	59
16	8	1.5	53	74	20
64	8	1.5	65	85	25
64	8	2.1	29	38	87
64	8	3.0	15.7	21	112
64	16	1.5	68	93	34
256	8	1.5	77	101	31

Table 5-3. ROM speed & energy vs. size and voltage.

into an array of two rows by BLOCKCOLS columns of memory blocks. Thus, the user can use the BLOCKCOLS parameter to control the aspect ratio of the memory.

Lprom has 3 parameters in the sdl file: WORDS, BITS, and dataBits. WORDS gives the total number of addressable words. (The actually number of words in the array may be made slightly larger for tiling purposes, but this will not affect the user.) BITS gives the number of bits contained in each word. (If BITS is odd, lprom will add an extra data line for tiling reasons. The user can leave this line disconnected.) dataBits is a two dimensional array that specifies the actual 1's and 0's to be stored in the ROM. The number of words is specified by WORDS. Each word must be BITS bits wide. The words should be given in order from address 0 to the largest address. The MSB of each word is on the left; the LSB, on the right. (See Figure 5-17 for an example of the parameters).

### 5.3.2 Simulation

In order to use irsim switch level simulator, one must first set the state of two nodes in two flipflops that control the state of the lprom. These nodes are called "setMe" and "resetMe". For an example:, see Figure 5-18.

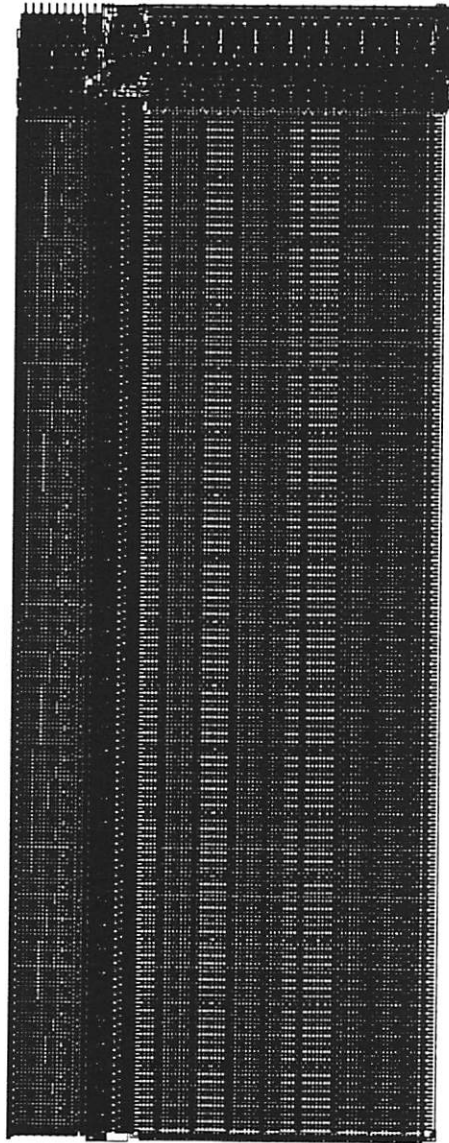


Figure 5-16. Layout of 1024 word x 16 bit ROM block.

In the actual silicon, these nodes will always be properly set when the Power On Reset Bar (PORB) signal goes low; the special procedure in IRSIM is an artifact of that program's inability to recover from undefined states in certain cases.

```

( WORDS 64)
( BITS 8)
(dataBits ("00000000" "11111111" "10101010" "01010101"
"00000000" "00000001" "00000010" "00000011"
"00000100" "00000101" "00000110" "00000111"
"00001000" "00001001" "00001010" "00001011"
"00001100" "00001101" "00001110" "00001111"
"00010000" "00010001" "00010010" "00010011"
"00010100" "00010101" "00010110" "00010111"
"00011000" "00011001" "00011010" "00011011"
"00011100" "00011101" "00011110" "00011111"
"00100000" "00100001" "00100010" "00100011"
"00100100" "00100101" "00100110" "00100111"
"00101000" "00101001" "00101010" "00101011"
"00101100" "00101101" "00101110" "00101111"
"00110000" "00110001" "00110010" "00110011"
"00110100" "00110101" "00110110" "00110111"
"00111000" "00111001" "00111010" "00111011"
))

```

Figure 5-17. Sample parameters to synthesize a ROM.

```

model linear
vector d D[{7:0}]
vector address A[{4:0}]
h enable Vdd porB
l GND ck
set address 00000
l ctrl_0/resetMe
h csEnd_0/setMe
s 50
x ctrl_0/resetMe csEnd_0/setMe

```

Figure 5-18. Example irsim commands to simulate ROM.

---

## 6 LOW-POWER SPEECH RECOGNITION ARCHITECTURE

---

### 6.1 Goals

This chapter describes the design of a low power speech recognition system using custom hardware. Parts of this system have been implemented in hardware; other parts are left as future work. There were two reasons for designing this system: First, to demonstrate the feasibility of performing state of the art speech recognition algorithms on low power, portable hardware. Second, to advance the art of low power CMOS design.

### 6.2 Previous Work

The system design started from the UCB custom speech recognition system [24][25]. This previous system was a successful demonstration of a CAD system designed for rapid prototyping of custom chips and boards for application specific systems. Its design goals were to achieve as large a throughput as possible while maintaining flexibility in the phoneme topologies, word pronunciation, and grammar. Although this system was not designed with any thought to power consumption, its implementation does provide valuable lessons on the bottlenecks inherent in speech recognition algorithms.

This system, pictured in Figure 6-1, ran the algorithm described in Chapter 2. The front end signal processing was performed on one of the 3 TMS320C30's on the grammar board. State processing and phoneme processing occurred on the viterbi board, which contained 6 custom ICs. A separate distribution board stored output distribution probabilities in 80 MBytes of dynamic RAM. Two of the TMS320C30's on the grammar board performed the grammar processing. The Heuricon board, which ran the VxWorks real time operating

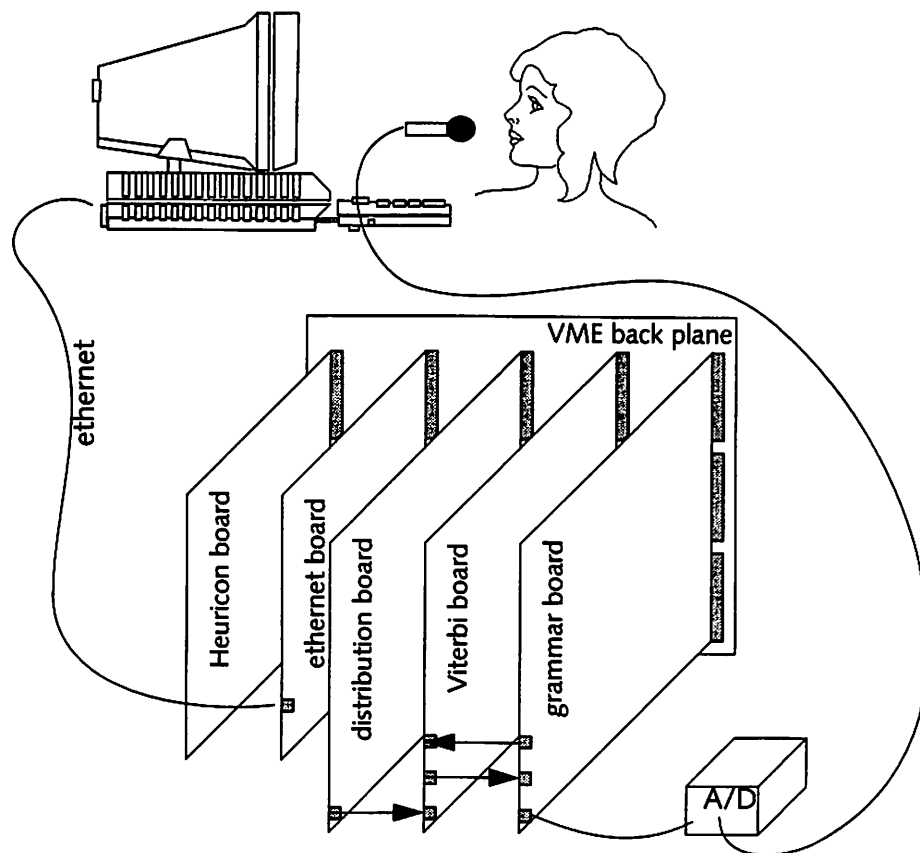


Figure 6-1. Previous UCB speech recognition hardware system.

system on a 68020 microprocessor, performed general command and control functions. Most of the data was transferred over a VME backplane, except for probability distribution data which was transferred over a custom cable between the distribution board and the Viterbi board, and grammar transition information which was transferred between the Viterbi board and the grammar board.

### 6.3 Next Generation: Low Power

The next generation speech recognition system is a scalable, low-power design. It is scalable because the size of the vocabulary can be increased by using more Viterbi decoder chips in parallel. It is low power because all functions will be implemented in low-power CMOS ICs, with all memory implemented on-chip as low-power SRAM and ROM.

The low power design departs from the previous design in the following ways: algorithm changes, reduced programmability, locality of data, memory-centric design, and application of general low power techniques.

The chief goal of the low power system is to maintain locality of data. To see why this is

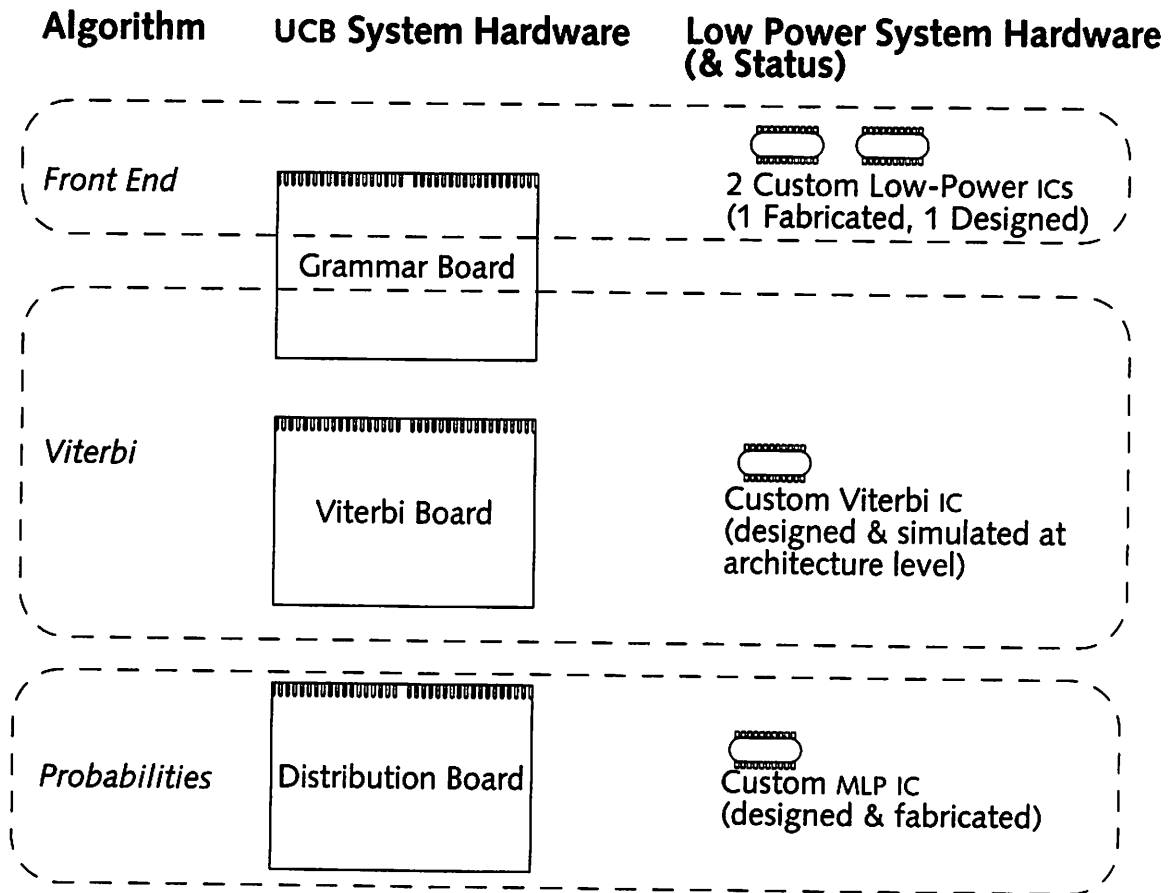


Figure 6-2. Comparison of hardware partitions.

important, first consider the UCB system, which failed to take advantage of much of the structure and locality of data in HMMs. The UCB system did use two levels of hierarchy: phoneme level and what it termed the “grammar” level. (“Grammar” is placed in quotes in this context because the UCB system’s use of the term “grammar” also encompasses what this document terms the dictionary or pronunciation level of the HMM.) Table 6-1 compares the number of bits used to encode state and phoneme data in the UCB system and the low power system. The low power system manages to reduce the amount of data

	UCB System	Low-Power System
Bits/ state	145	25
Bits/ phone	96	11

Table 6-1. Comparison of memory sizes.

stored by keeping data local and by using implicit addressing schemes, so the memories do not need to store a large number of pointers.

The phoneme layer is where the UCB made good use of locality. The UCB had individual HMMs that represented phonemes. These small HMMs were combined at the next higher level of hierarchy to form words and sentences. The phoneme models themselves could have variable topology because they could have any number of states, each of which could connect to any of its three following states. Each phoneme also had a special “source grammar node” and “destination grammar node.” States in the phoneme model could also have edges from the source grammar node and to the destination grammar node.

These grammar nodes are the key to the phonemes locality because transitions into and out of the phoneme could only take place through these grammar nodes. Thus, grammar processing is simpler because a connection to a phoneme only requires a single edge to the phonemes grammar edge, rather than edges to multiple states within the phoneme. Processing within the phoneme is simpler too. Because each state can only be connected to a small number of other states, the number of memory words devoted to describing these connections is small; the size of these words is also small because they only need to describe a small relative offset of one state to the next to describe the connectivity within a phoneme. This phoneme hierarchy also simplifies memory accesses while processing transitions from one state to the next: since transitions only occur between a small number of states, their data can be kept in a cache so the main memory does not need to be accessed several times for each state. In summary, by reflecting the structure of HMMs in the structure of the hardware, and by constraining the topology of phonemes, the UCB system reduced the number of words, the size of the words, and the number of times that words had to be accessed from memory. The low power hardware uses the same strategies,

but takes them even further in the phoneme section and also applies them to other levels of the HMM.

Before examining the low power architecture, it would be useful to look at the “grammar” level of the UCB system to see where its bottlenecks occur. The grammar level encompasses all transitions higher than the phoneme level: namely, transitions between phonemes within words as well as transitions between words. Thus, whenever any phoneme’s destination grammar node is active, it causes data to be sent to the grammar processing board, which then must determine the appropriate transitions and send the data back to the grammar processor on the viterbi board. In practice, this path proved to be a serious bottleneck: the first design of the grammar board could not handle the I/O bandwidth and had to be replaced with a multiprocessor design.

In contrast, the low power system design uses separate systems to process transitions between phonemes in a word and to process transitions between words

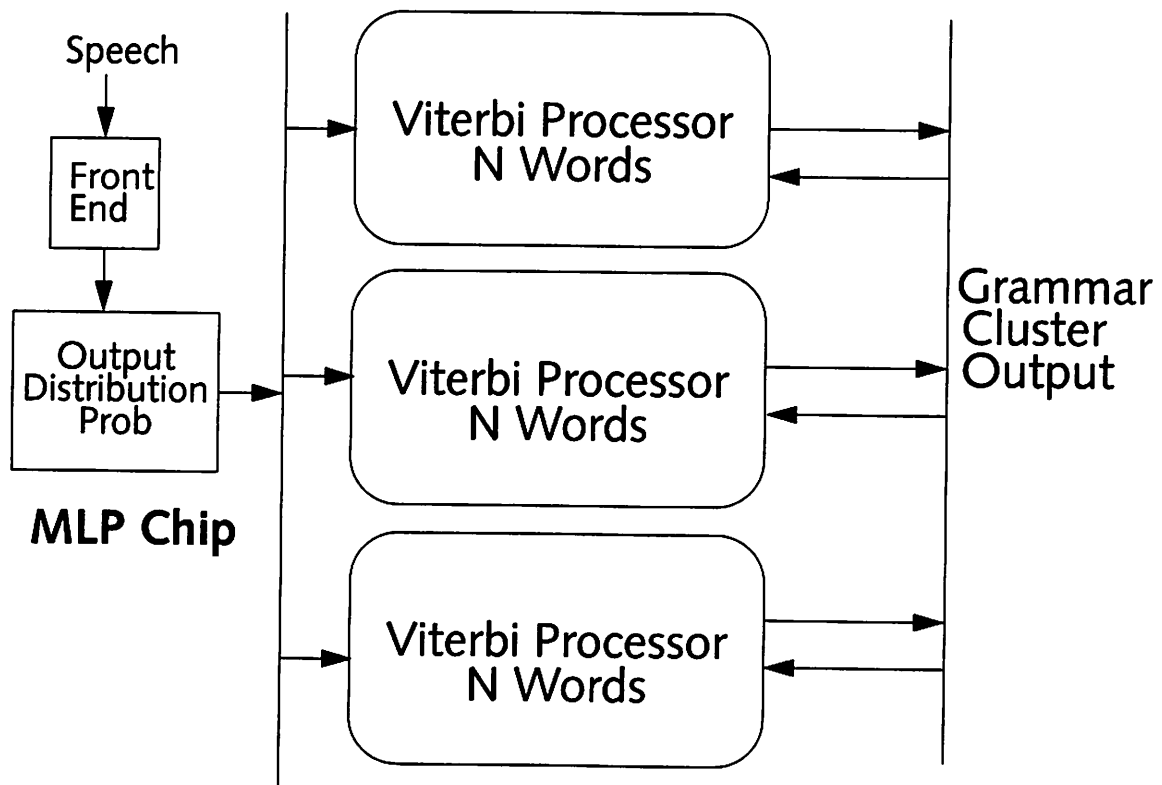
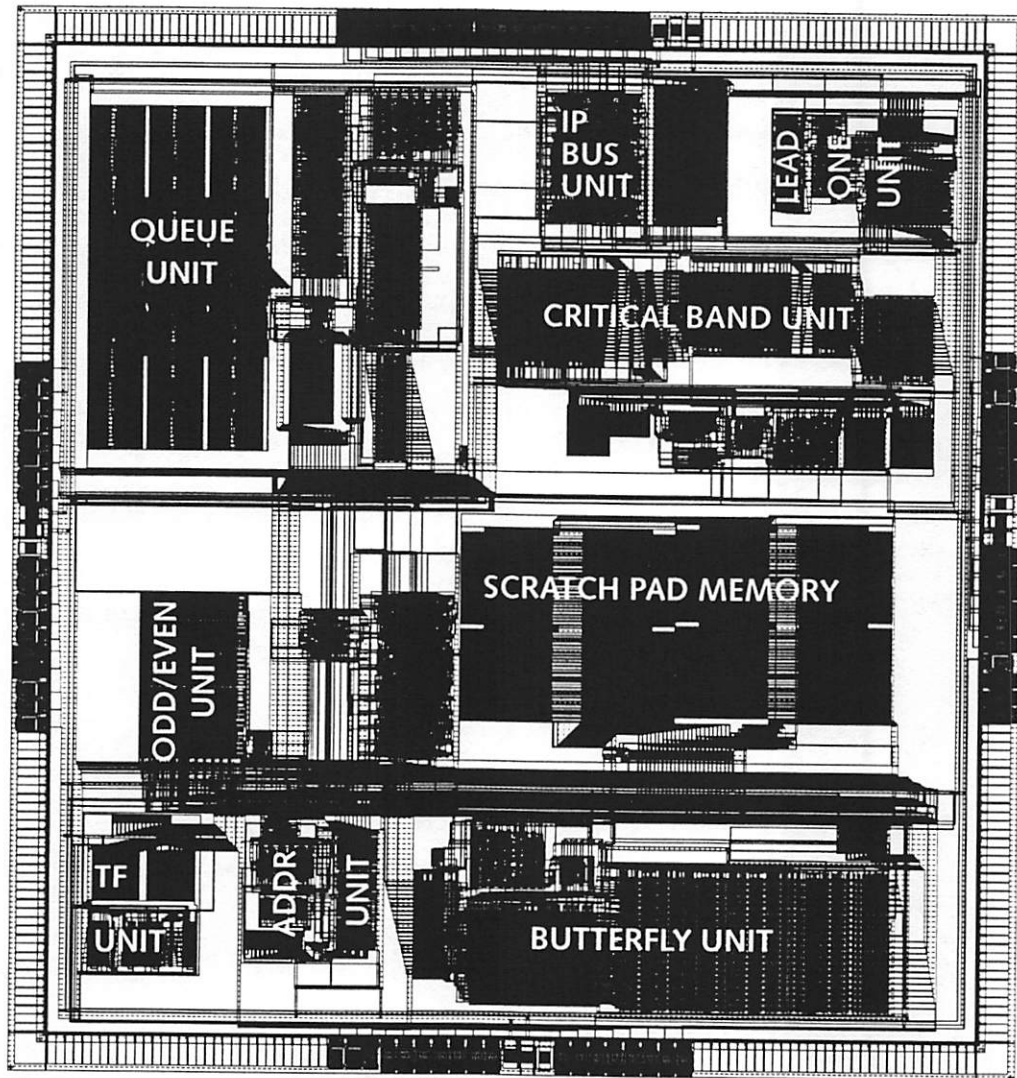


Figure 6-3. Low-power system architecture.



## 6.4 Front End

The design of both front end ICs was done by Steve Stoiber [23]. The SORASTA chip, pic-



Process Technology: 1.2um, SCMOS Design Rules  
Dimensions: 9.8mm x 9.3mm  
Transistors: 222,320

Figure 6-4. SORASTA IC layout.

tured in Figure 6-4, implements the portion of the RASTA-PLP algorithm shown in Figure 6-5.

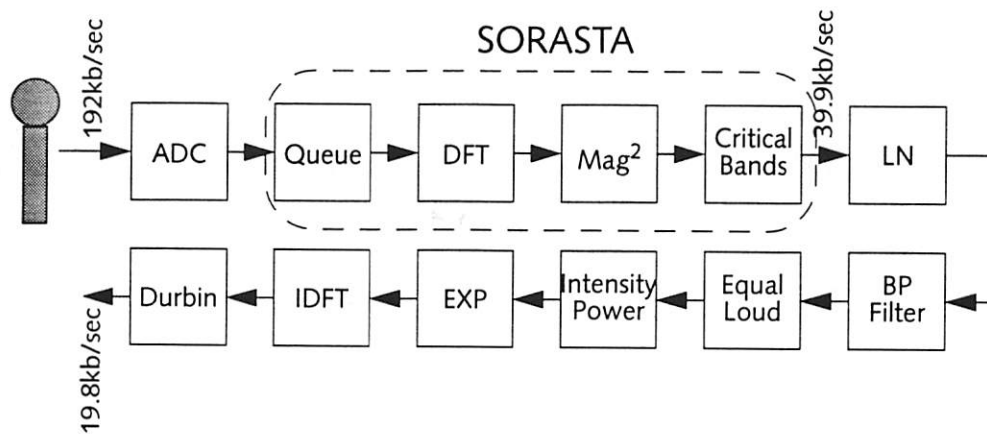


Figure 6-5. SORASTA chip algorithm.

## 6.5 MLP

The MLP is implemented in a single custom IC, described in Chapter 7. Only one MLP is needed in the system, no matter what the size of the vocabulary because all of the words use the same phoneme models. The phoneme probabilities are broadcast from the MLP chip to the Viterbi chips.

Using an MLP, instead of a continuous or discrete probability distribution function, saves a great deal of power and system complexity because the MLP needs much less memory. Since the MLP accounts for the context at its input, it only needs 49 phoneme models, instead of a typical 4,000 triphone models used by the other two methods. Assuming the discrete PDF would use 256 level quantization, it would need 12 million words of memory; the continuous PDF would need 384 thousand words of memory if it used 8 parameters per coefficient. The MLP only uses 22 thousand words of coefficient memory, making it the only one of the three alternatives that allows the memory to be placed on the same chip as the computation units.

Also, using 49 phonemes instead of 4,000 triphones allows the phoneme identification to be encoded in 6 bits, as shown in Figure 6-6 and Figure 6-7, instead of 12 bits. This shrinks the size of the pronunciation memory in Figure 6-7 as well as the phoneme memory in Figure 6-8.

## 6.6 Viterbi

### 6.6.1 Overview

The Viterbi chips will perform the entire Viterbi search through the HMM, including phoneme, word, and grammar processing. Within the chips, each of these levels of the model have their own hardware subsystems in order to take advantage of each level's unique structure and to preserve locality of data. The global data flow between these three sub-

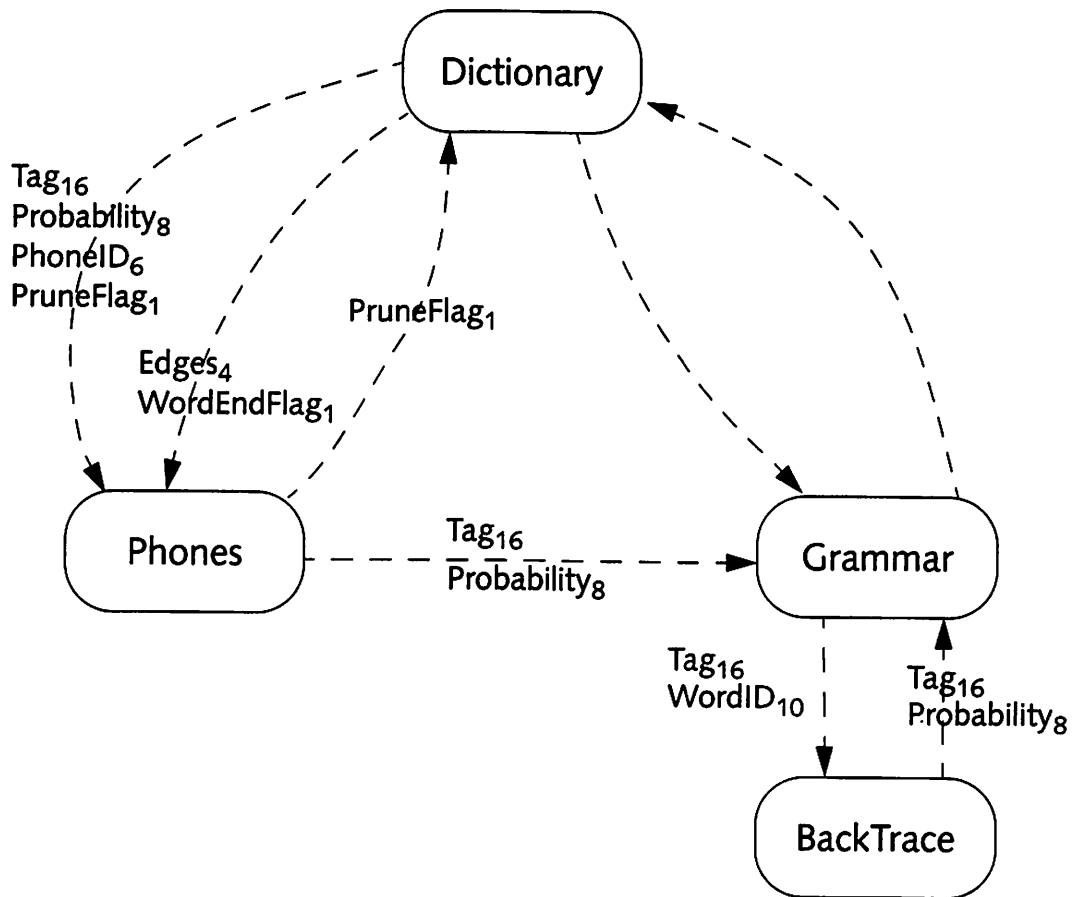


Figure 6-6. Viterbi chip architecture and data flow.

systems, plus the backtrace subsystem, is pictured in Figure 6-6. All of these subsystems operate in parallel.

The Viterbi chip performs a beam search, meaning that it only stores data on states in the HMM that have probabilities above a certain value, called the pruning threshold. States

who's probabilities are below the pruning threshold at the end of a frame are considered pruned: their current data are not stored and no transitions may originate from that state during the next frame. Pruned states may be reactivated by transitions from other states. Pruning reduces the size of the memory in the phones section because only a fraction of all states are active at one time.

Probabilities in the Viterbi chip use a negative log representation, so that multiplication can be replaced by addition. (Fortunately, the Viterbi algorithm does not require additions.) Each frame, the phone subsystem normalizes probabilities to prevent eventual underflow.

### **6.6.2 Dictionary Subsystem**

The memories within the dictionary are shown in Figure 6-7. The pronunciation memory stores the information on how many phonemes are in a word, the identity of these phonemes, and the topology of the edges between the phonemes. Each block ("block" is used instead of "word" in this case to avoid confusion with "words" in the linguistic sense) of 32 bits in the memory contains information on four phonemes. This phoneme information contains the identity of the phoneme and the topology of the edges leaving that phoneme. The four bit encoding scheme for edges allows connections to any of the three subsequent phonemes, as well as a connection out of the word. At the start of a word, the field that is usually used for the phoneme ID actually gives the number of phonemes in the current word; the topology field gives the topology from the start of the word to the first three states.

The pronunciation memory is read in sequence once every frame. It is only written to when the vocabulary changes. Since the vocabulary is written so infrequently, ideally this memory should be implemented in flash RAM in order to save die area. Otherwise, SRAM would suffice.

Since the pruning state data is read and written every frame, it is stored in a separate SRAM. In keeping with low power principles, 32 prune status tags are stored in parallel in each memory block. The controller processes these tags 4 at a time, in keeping with the 4 phonemes per block in the Dictionary memory.

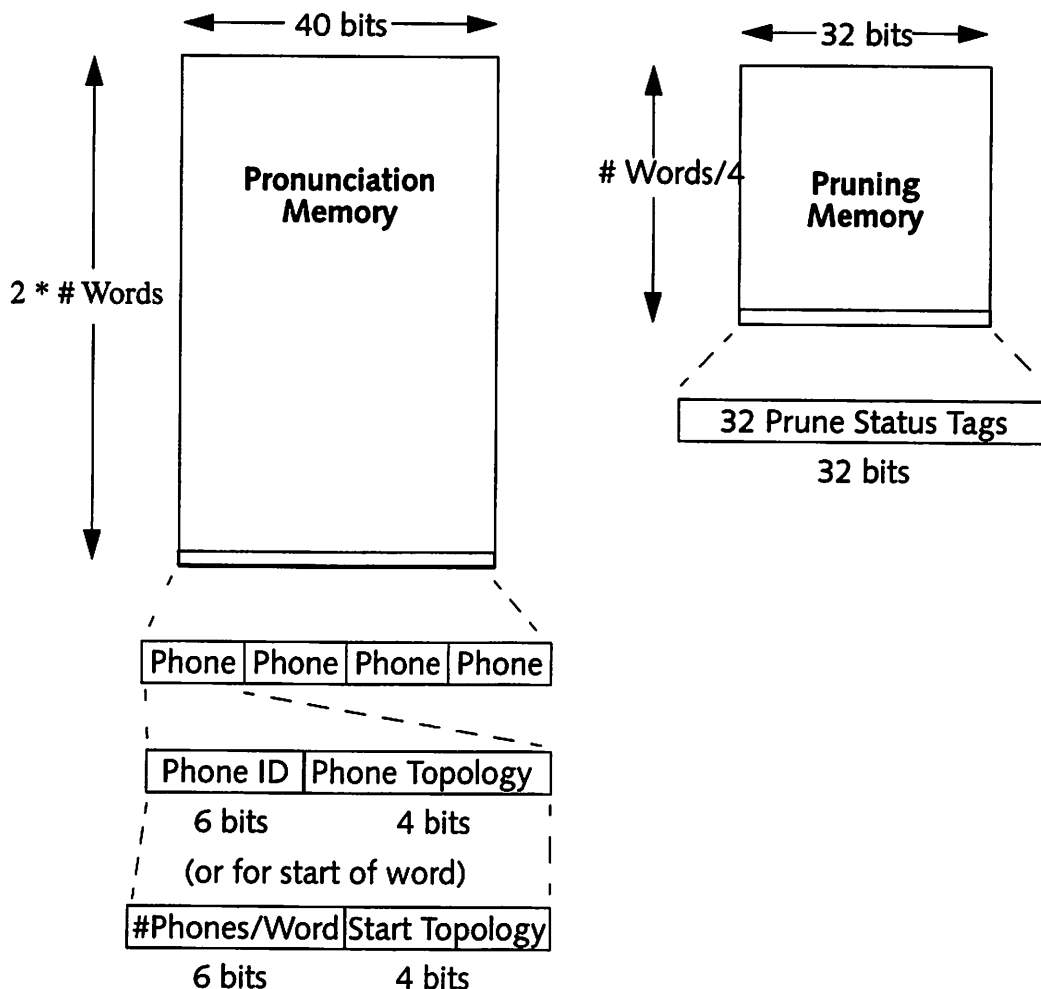


Figure 6-7. Dictionary subsystem memories.

### 6.6.3 Phones Subsystem

The phone subsystem contains two large memories and one small memory, as shown in Figure 6-8. The small memory, which stores data on the phone models, is read once per phoneme processed during recognition. At the beginning of each frame, it stores the output probabilities for each phoneme as determined by the MLP. It also stores the transition probabilities for the phoneme model: namely the edge probability that loops back to the phoneme and the edge probability that leaves the phoneme. These values do not change from frame to frame because they represent the expected duration of each phoneme as determined during training. They are placed in the same memory as the output probabilities as a matter of convenience. Even though these transition probabilities will

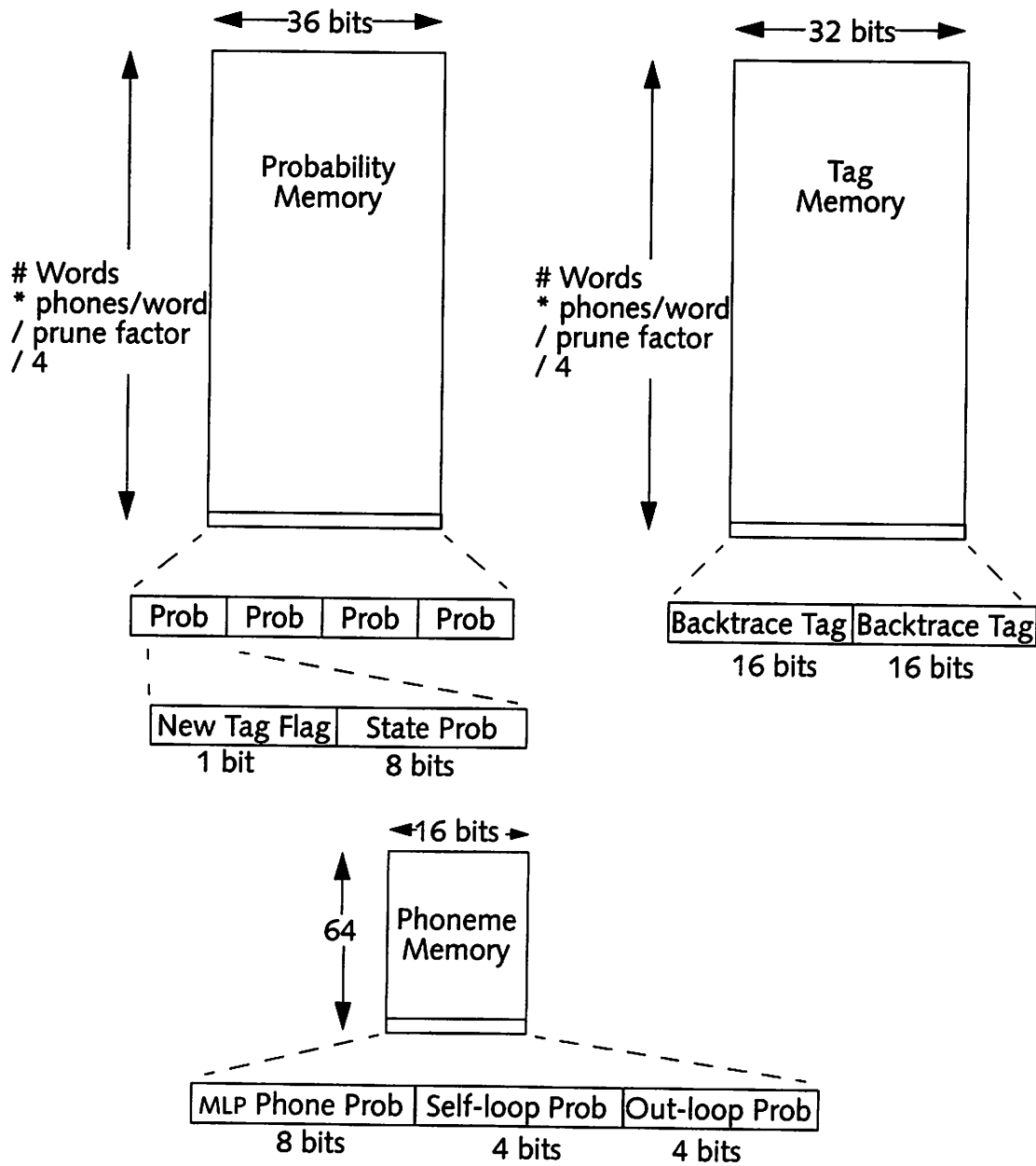


Figure 6-8. Phone subsystem memories.

have to be written back to the memory while writing the phoneme output probabilities, the impact on power consumption is minimal because this writing operation only needs to take place once per frame and there are fewer than 64 phones.

The two large memories store probabilities and backtrace tags for active states. Each memory stores values for both the current frame and newly computed values for the next

frame. Values are stored in order, so the addressing scheme simply consists of two pointers: one for the *current* frame and one for the *next* frame. Data is read from the *current* pointer location if the *current* phoneme was marked as unpruned (as stored in the dictionary pruning status memory); data is written to the *next* pointer location if the probability of being in that state is greater than the pruning threshold. The pointers are incremented after accessing their memory locations. This addressing scheme helps minimize total memory size since after each *current* state probability is read, its memory location is freed for use to store *next* frame data. As long as the *next* pointer does not catch up to the *current* pointer, the single memory is being used correctly. To prevent *current* memory from being overwritten, the phone subsystem marks all states as pruned if the *next* pointer is equal to the *current* pointer, so no *next* data is written until more memory is freed.

The backtrace memory's operation is similar to that of the probability memory, except it uses an additional memory saving technique that takes advantage of the fact that many states in a row have the same backtrace tag. Whenever a state has the same tag as the previous state, that tag is not stored in the backtrace memory; instead, the state sets its *new tag flag* in the probability memory to false.

The probability and backtrace memories at most need to read and write each state's data once per frame because a small context of states store their data in registers in the state subsystem. The algorithm restricts transitions from the current state to go only to the following three states or to the end of the word. Thus, by using successor processing, the phone subsystem can cache the intermediate results of the viterbi search in four registers. After processing each state, it shifts the data in the three successor registers; at the end of the word it passes on the data in the end of word register to the grammar subsystem. This is a prime example of how locality of data in the algorithm can simplify computation and memory access in the hardware.

#### 6.6.4 Grammar Subsystem

The grammar subsystem contains 4 memories, as shown in Figure 6-9. The top two memories store probabilities and backtrace tags for the first and second layer of clusters. These memories are similar to the state memories in the phone subsystem. In this case, the clus-

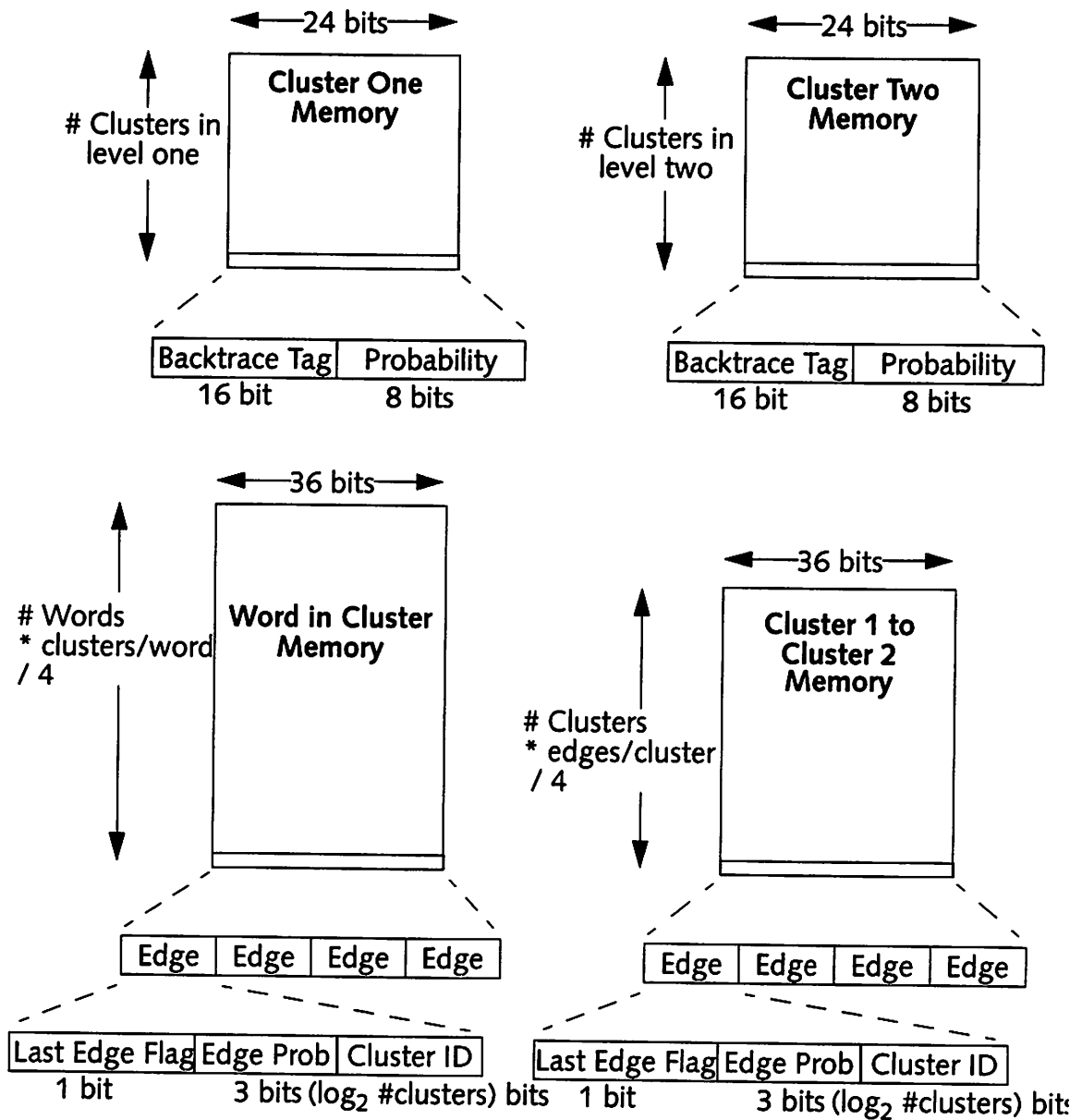


Figure 6-9. Grammar subsystem memories.

ter one memory stores *next* state information while the cluster two memory stores *current* state information. The lower two memories store topology information: the first tells which words are contained in which clusters and with what probabilities; the second tells what are the edges that connect clusters in the first level to clusters in the second level. Since the data in these two memories only change when the grammar changes, they should ideally be implemented in FLASH, or some other compact memory.



The grammar subsystem processes requests from the phone and dictionary subsystems.

Whenever the grammar subsystem receives a probability and a tag from the phone subsystem when a word has ended (and its ending probability is greater than the pruning threshold). It performs successor viterbi processing from the words to the first level of grammar clusters. The first step is to register the transition with the backtrace subsystem (see Section 6.6.5) to see if a new tag needs to be generated. Next, it reads the Word in Cluster Memory to find out to which clusters the word has edges. For each such cluster, the clusters data in the Cluster One Memory is overwritten if and only if the word ending probability, times the word to cluster edge probability, is greater than that cluster's previously stored probability.

Whenever the grammar subsystem receives a request for the starting probability and tag for the next word, it performs a predecessor search on the level 2 clusters. In this case, it reads the Word in Cluster memory to see in to which clusters the word belongs. Then, it reads all of those clusters' probabilities and passes along the largest one, along with its corresponding tag, to the dictionary. (The dictionary will pass along the probability and tag to the phone subsystem).

All of the addressing in the grammar subsystem is simple because all words and clusters are processed in order. Thus, there is no need for storing pointers to find data; all addresses are implicitly known and single bit flags separate data between states and clusters.

Transitions from the level 1 clusters to the level 2 clusters take place in between frames since the clusters are not real states in the sense of the Viterbi algorithm; they are just place holders, so the transition out of a word in one frame should pass through both cluster layers and arrive at the start of another word at the start of the next frame. The computation of the level 1 to level 2 transitions is a simple successor Viterbi process. When it is finished, all of the level one probabilities are overwritten with zero, in preparation for the next frame's computation.

These level 1 to level 2 transitions are crucial to the multi-chip, parallel implementation of the speech recognizer because these transitions are the only ones that travel from one

chip to another. The grammar subsystem passes on out-of-chip cluster transitions to a neighboring chip in a daisy chain fashion. Chips pass these transitions along the daisy chain until they reach the appropriate chip, which is identified by the most significant bits in the level 2 cluster ID.

### 6.6.5 Backtrace Subsystem

The Backtrace subsystem contains two memories: the first contains word ID's and pointers

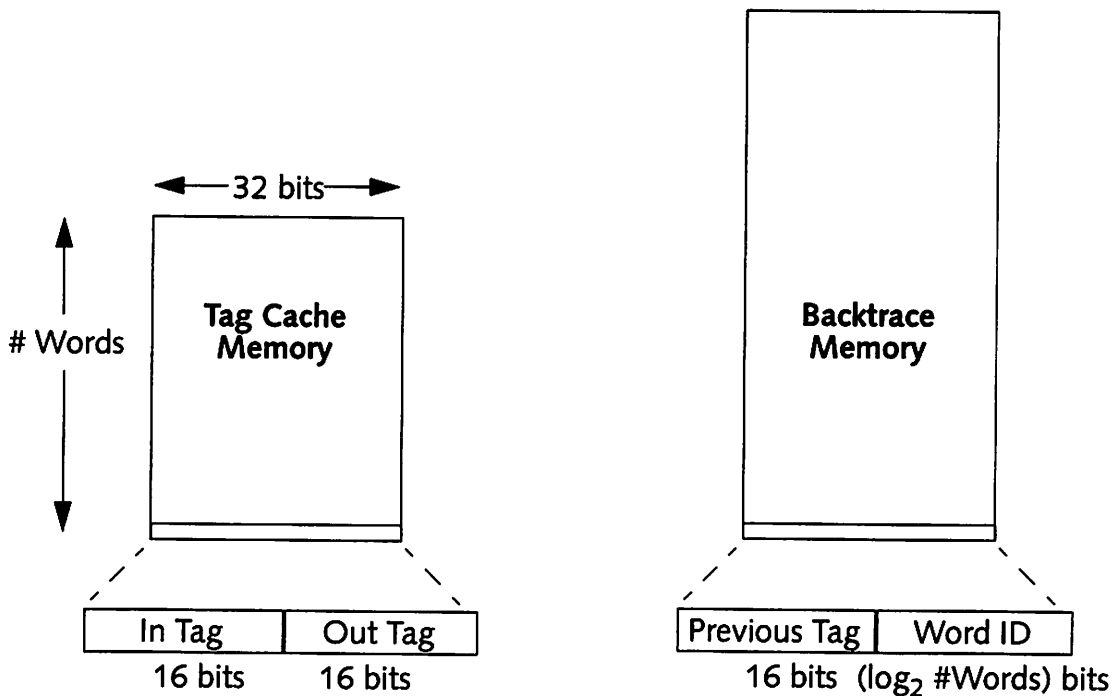


Figure 6-10. Backtrace subsystem memories.

to allow the system to perform a backtrace at the end of a sentence; the second is a cache to reduce the total number of backtrace tags issued in a sentence.

The backtrace memory stores a linked list whose entries contain word IDs. By following the linked list back from the END cluster back to the START cluster, the backtrace subsystem will traverse the most probable sentence.

Backtrace tags are actually pointers into the backtrace memory, with the MSBs identifying the chip in multi-chip implementations.

Since the speech recognizer is only interested in determining the best sequence of words (not states) in the HMM, the backtrace subsystem tries only to create new tags when a word ends with a tag with which it has not ended before. In other words, two paths through the HMM should be treated as identical if they pass through the same sequence of words, even if they did not enter and leave each of those words at the same time. To accomplish this, the tag cache memory stores the most recent tag with which each word has ended. If a word ends with the same tag again, its transition is assigned the same new tag as it was last time. On the other hand, if the tag leaving a word is not the same as it was last time, the transition is assigned a new tag, which is then entered into the cache and into the backtrace memory. Overall, it is important to limit the number of backtrace tags issued in order to reduce the number of words needed in the backtrace memory and hence the word size of the backtrace tags (which are pointers into the backtrace memory.)

#### 6.6.6 Size and Scaling

The factor limiting the number of words on a chip is the amount of memory that can fit on the chip. Table 6-2 shows a breakdown of the memory sizes under a variety of assumptions about the size of the vocabulary and grammar. The first row gives two choices for the number of words in the on-chip vocabulary, either 100 or 500 words. The second row gives choices of 16 or 64 grammar clusters in each level. In the third row is the final parameter, the average number of clusters to which a word belongs and the average number of edges that leave a cluster. (For convenience, these two parameters are assumed to be equal, but there is no requirement that they be so.) Two other parameters are fixed for the purpose of computing the values in this table: the average number of phones per word is 8; pruning leaves  $1/4$  of all states active at a given time. A further assumption, that the number of entries in the backtrace memory should be 4 times the number of words in the vocabulary, was based on heuristics.

Judging from Table 6-2, the dominant factor in the amount of on-chip memory is the size of the vocabulary. Using a 0.6 $\mu$ m process, a 100 word vocabulary should produce a fairly small die area; a 500 word vocabulary would probably be too large. A 256 word per chip design might be a reasonable goal.

Recognizer Parameters	Words	100				500			
	Clusters	16		64		16		64	
	Edges/Cluster & Clusters/Word	2	8	2	8	2	8	2	8
Memory Sizes (FLASH is shaded)	Pronunciation	8k				40k			
	Pruning Status	1k				4k			
	State Probability	2k				9k			
	State Backtrace	2k				8k			
	Phoneme	1k							
	Cluster 1 & 2	1k		3k		1k		3k	
	Words in Clusters	2k	7k	2k	7k	9k	36k	9k	36k
	Cluster 1 to2	0.3k	1k	1k	4k	0.3k	1k	1k	4k
	Tag Cache	3k				16k			
	Backtrace	9k				48k			
	Total FLASH	10k	16k	11k	19k	49k	78k	50k	80k
	Total RAM	19k	19k	21k	21k	87k	87k	89k	89k
	Total Memory	29k	35k	32k	40k	136k	164k	139k	169k

Table 6-2. Memory sizes for various viterbi configurations.

It is interesting to note that clock speed is not the limiting factor in chip design. Allowing 2 clock cycles per phone during the frame and 2 clock cycles per grammar edge between frames, the largest recognizer in Table 6-2 (500 words, 64 clusters, 8 edges/cluster, and 8 clusters/word) would require 800,000 cycles per second for in-frame processing and 100,000 cycles per second for between-frame cluster processing. In other words, the clock rate is less than 1MHz, allowing for very low-voltage, low-power operation. To the first order, this clock rate is proportional to the number of words on a chip.

Now consider scaling the size of the vocabulary by placing more chips in parallel. In this case, the number of cycles to perform the in-frame computation remains constant because each chip is doing its viterbi search through its own words in parallel with all of the other chips. The only additional clock cycles required are those needed to pass along cluster-to-cluster transitions that go from clusters in one chip to clusters in another chip.

As the very worst case, assuming cluster-to-cluster transitions are randomly scattered across all the chips, each transition would have to pass through half of the chips before reaching its destination cluster. Allowing two cycles for a transition to pass from chip to chip, a 10,000 word system using 20 chips would require 1,000,000 cycles per second to perform between frame cluster processing. In practice, this figure should be considerably less if the clusters are intelligently partitioned. Namely, clusters that are connected by many transitions should be placed on the same chips if possible, otherwise, they should be placed on nearby chips.

---

## 7 LOW-POWER MULTI-LAYER PERCEPTRON DESIGN

---

The low power MLP chip serves three purposes. Its first purpose is practical: to be a part of a functional speech recognition chipset. Second, it serves as a test of low power design techniques, both at the architectural and circuit level, with a particular emphasis on low power memory design and usage. Third, its design can be used as a template for future variations. While the chip was fabricated for a particular sized MLP and the data was stored in ROM to reduce die area and hence fabrication costs, the design is meant to be flexible.

### 7.1 Architecture

The MLP chip is divided into four blocks: two layer units and two buffers. Since the “Multi” part of the Multi Layered Perceptron algorithm is “two” in this case, there are two layer units.

#### 7.1.1 Layers

Each layer unit performs all the storage and computation associated with one layer of the MLP. Mostly, this consists of a series of multiply-accumulate operations: each input value is multiplied by its coefficient to each node in that layer; the product is added to the value accumulated at that node. After the multiply accumulate process is finished, the threshold function is applied to the sum at each node. The result is passed on to the next stage (or the output, in the case of the last stage).

Each layer unit contains 5 units: datapath, control, coefficient memory, node memory, and threshold function memory.



### 7.1.1.1 Datapath

The datapath executes the multiply accumulate function on 11 bit-wide data. The datapath contains no multiplier; the multiplication is performed using an adder by employing a modified booth algorithm. Since the data in the datapath is multiplied by 8 bit coefficients, and since the modified booth algorithm multiplies by 2 bits at a time, the entire multiplication takes 4 cycles. The same adder and registers also perform the accumulation function, so the complete multiply-accumulate takes 5 cycles. The 8 bit coefficients do not directly enter the datapath; they go to the controller, which then sends the appropriate select signals to the multiplexor into the adder.

Why not use a hardware multiplier to speed up operations? Because the critical path of the chip is the operation of the memories, not the datapath. The minimum operating voltage, and hence the power consumption of the chip, are determined by the speed of the memories.

### 7.1.1.2 Controller

The control section is composed of standard cells from the Lager low power standard cell library, which employ minimum sized devices and TSPC registers to minimize power consumption. This section was synthesized from a state machine specified in VHDL. The states are shown in Figure 7-3. In addition to having states for the 5 cycles in the multiply accumulate function (states MULT ONE through MULT FIVE), the state machine has states to initialize the layer and fill the pipelines at the start of a frame (START ONE through START FIVE) and to perform the threshold function and pass data to the next layer at the end of a frame. (states DUMP ZERO through DUMP FIVE).

The layer control creates the control signals for the multiplexers, registers, and adder (whose control signal is the carry in signal) in the datapath; creates address, clock, and read/write signals for the coefficient, layer, and threshold memories; and performs handshaking with the buffers before and after the layer.

The layer control accepts four types of input. First, it accepts handshaking signals from adjacent buffers. Each time the layer needs a new input, the controller performs a hand-



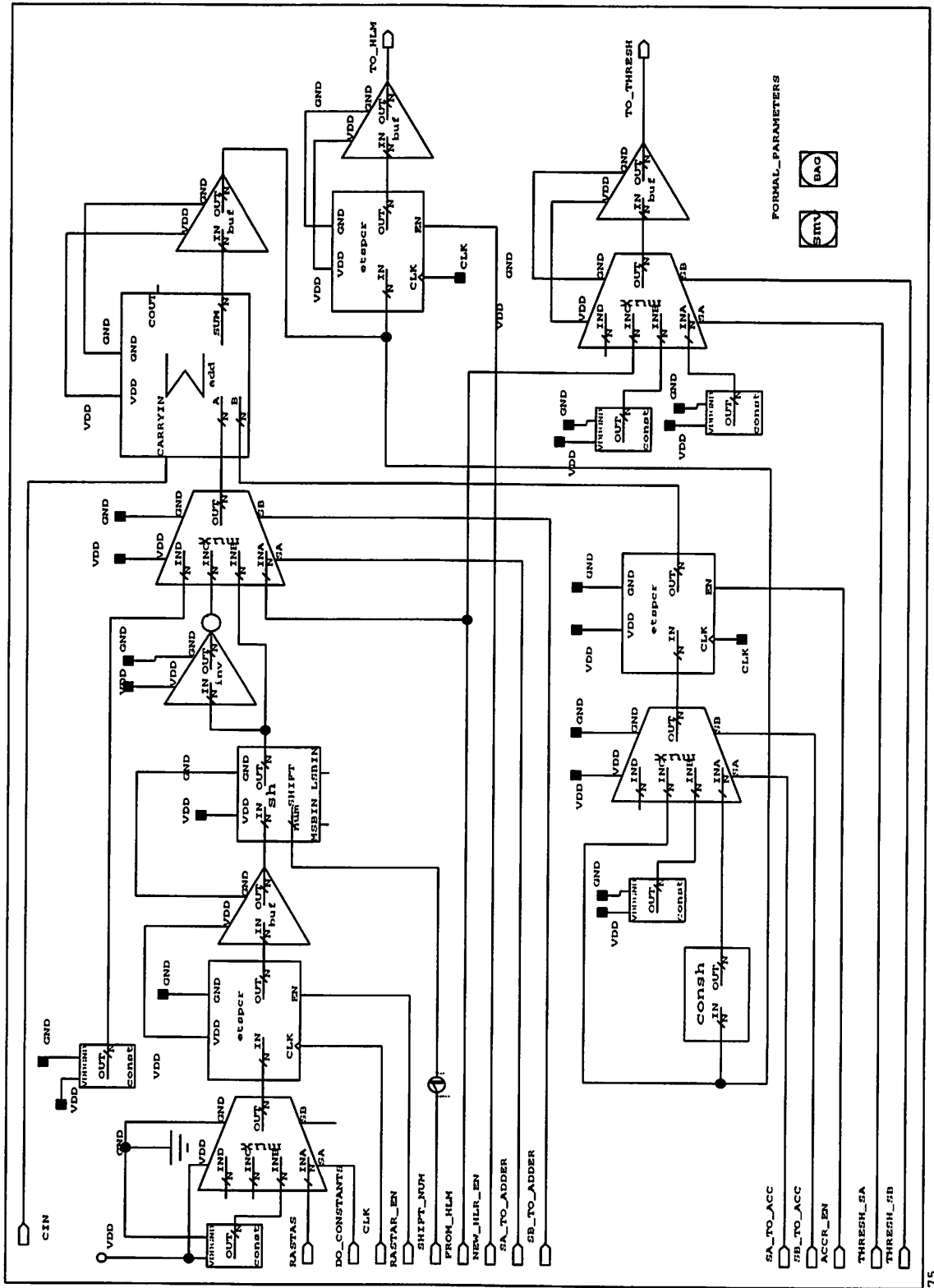


Figure 7-2. Datapath for MLP layers.

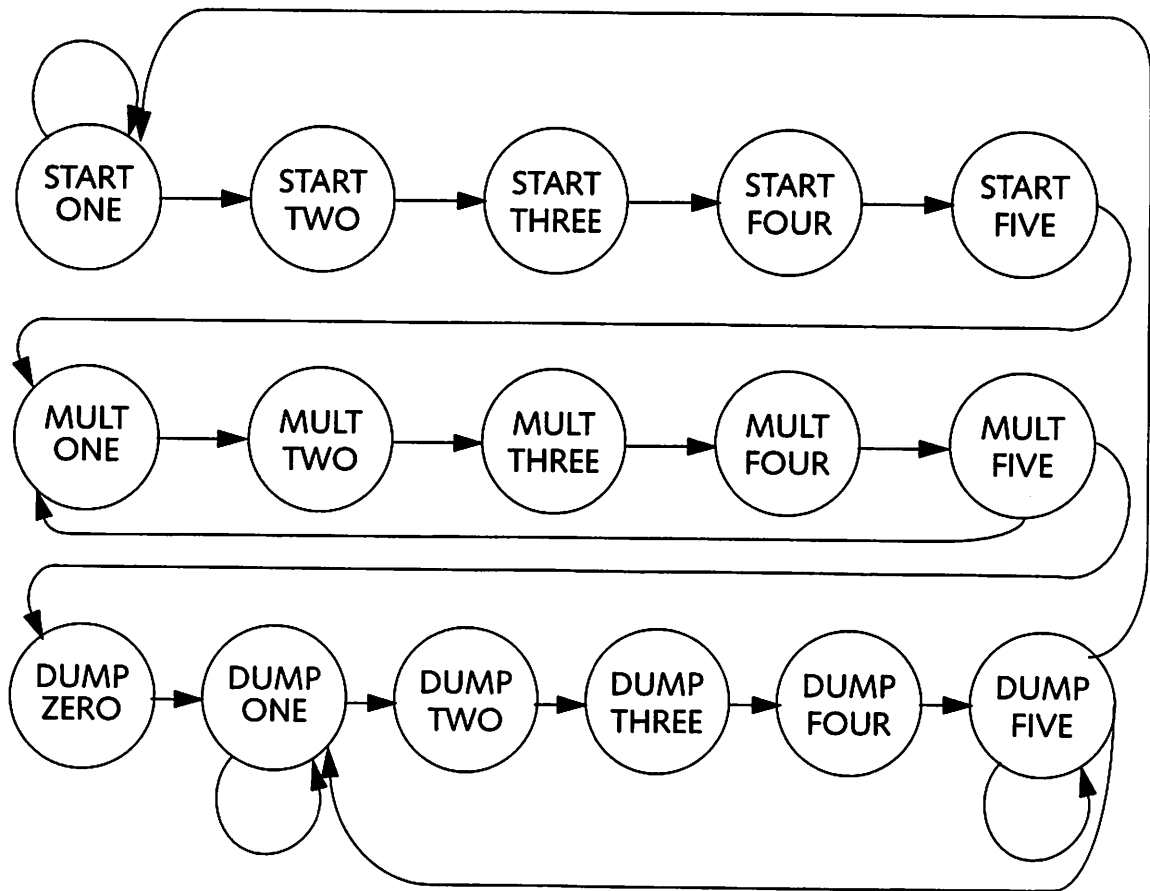


Figure 7-3. State transition diagram of layer control.

shaking operation with the preceding buffer. At the end of each frame, the controller performs a series of handshaking operations with the following buffer to transfer out its completed results. The second type of input is master control, which starts and stops the operation of the layer. Master control comes from off chip. Third, the controller accepts coefficient data from the coefficient memories. Although it is unusual to route data through the controller, instead of the datapath, it makes sense in this case because the coefficients are decoded into mux control signals and adder's carry in signal to perform the modified booth multiplication. In keeping with the slow speed, low power memory strategy, the eight bit coefficients are read four at a time from the coefficient memory and are multiplexed within the controller. Fourth, and finally, the controller accepts parameters that specify the number of nodes in its layer and the number of inputs to its layer.

Parameterizing these values simplifies the design by allowing the same controller to be used for both layers; it simplifies simulation and testing by making it easy to create smaller layers for testing.

#### **7.1.1.3 Coefficient Memory**

The coefficient memories are low power ROMs. Using ROMs was a compromise. On one hand, the coefficient memories dominate the die area, so economically it is best to use the most compact memories possible, which happen to be ROMs. Furthermore, the coefficients themselves need to be changed rarely or never since retraining an MLP is a very slow and computation-intensive process, so the MLP can not be trained on the fly. Furthermore, a well trained, speaker independent vocabulary could suffice for any application. On the other hand, if an application specific or speaker dependent MLP is available, it could provide higher recognition accuracy. Even if the coefficients were changed, they would only need to be changed at most a few times per minute (as applications change) or perhaps a few times per day or a few times ever (as users change). For such slow writing rates, the ideal memory would be a dense, nonvolatile memory such as flash memory. Unfortunately, flash memory is not available through our foundry.

#### **7.1.1.4 Node Memory**

The node memories are low power SRAMs organized in four blocks each. The first layer node memory is 11 bits by 128 words; the second, 11 bits by 64 words.

#### **7.1.1.5 Threshold**

The threshold is a low power ROM that performs a table look up for the nonlinear threshold function of the perceptrons. The ROMs are 512 words by 6 bits. The output of a layer's node memory forms the address to its threshold ROM; the output of the threshold ROM goes to the next layers buffer. Since the node memory's data is 11 bits wide, one might expect to have 2048 words in the threshold ROM. However this is not necessary because the largest and smallest outputs of the node memories saturate the threshold function to zero or one. Rather than storing a large number of zeros and ones in the ROM, the controller intercepts the three MSBs from the node memory and sends control signals to the

datapath to send the first or last address to the threshold ROM whenever the node value would have saturated the threshold function.

### 7.1.2 Input Buffer

The input buffer is responsible for receiving each frame's coefficients as they are input to the chip, storing enough of these coefficients to make up the entire context for the inputs to the first layer, and passing each coefficient in the context to the first layer when it requests them. The input buffer contains a memory, a controller, and a small datapath.

As in the layers, the controller in the input buffer is made from low power standard cells

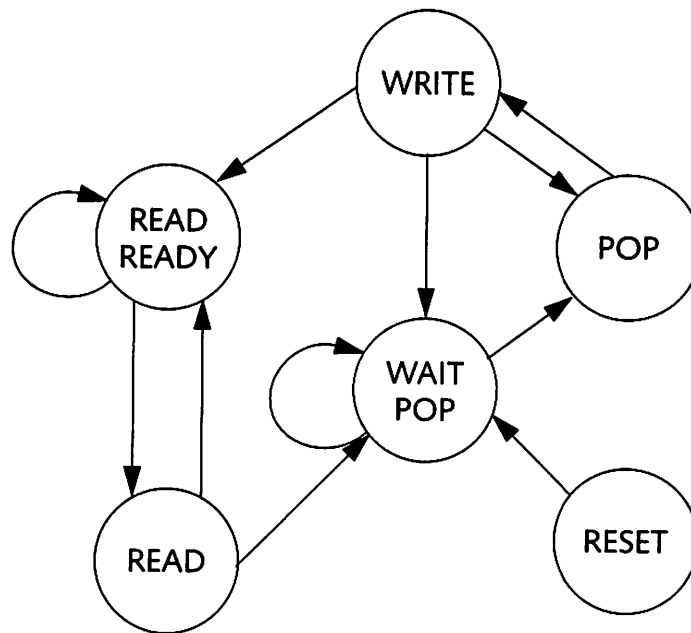


Figure 7-4. Input buffer control state diagram.

synthesized from a VHDL specified state machines. There are two sets of states: WAIT POP, POP, and WRITE accept coefficients from off chip and write them in the memory at the beginning of each frame; READ READY and READ take coefficients from the memory and pass them on to the first layer. Both of these operations are performed using two phase handshaking.

The datapath, shown in Figure 7-5, stores, computes, and compares memory addresses.

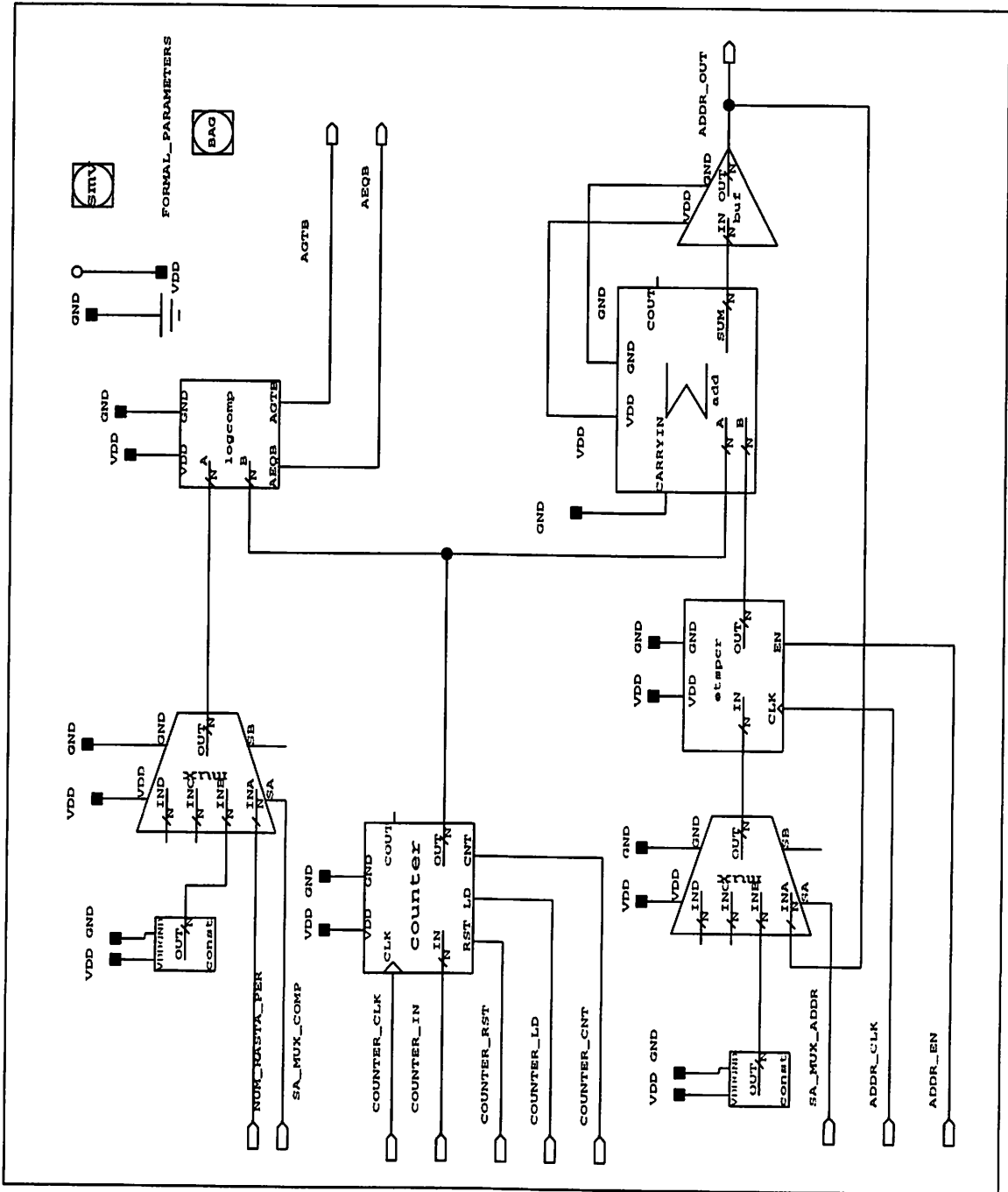


Figure 7-5. Input buffer datapath.

## 7.2 Summary of Low Power Techniques

The limiting factor in power consumption is memory access, so the low power design centered on reducing memory power. The memory sizes are summarized in Table 7-1 and Table 7-2. These memory sizes reflect the size of the MLP, namely 120 inputs (10 frames

of context), 120 hidden nodes, and 49 output nodes. The threshold function ROMs implement the nonlinear threshold function by using a table lookup .

Memory	Words	Word Size	Total Bits
Layer 1 Coefficients	3630	32	113k
Layer 2 Coefficients	2048	32	64k
Threshold Function 1	512	6	3k
Threshold Function 2	512	6	3k
Total			183k

Table 7-1. MLP ROM sizes.

Memory	Words	Word Size	Total Bits
Layer 1	128	11	1.5k
Layer 2	64	11	0.75k
Input Buffer	128	12	1.5k
Middle Buffer	128	6	0.75k
Total			4.5k

Table 7-2. MLP RAM Sizes.

At the architecture level, it is vital to place the memories as close as possible to the sections that use them. In particular, it is best to place the memories on the same chip as the datapath and control, or else power consumption will be dominated by the huge inter-chip capacitance. Placing memories and logic on the same chip also enables the use of wide data buses, which allow the memories to operate slower, and hence at lower voltages and lower power consumption. In the case of this chip, each coefficient memory has its own 32 bit data bus. If the memories were off chip, their data buses would have to be smaller because they would require too many I/O pads, vastly increasing the die area.

Memory scheduling is another important part of low power design. Since the memories are being accessed at less than the clock rate, the memories should be given as many cycles as possible to perform each read or write operation. Reducing the timing constraints on the memories allows the whole chip to meet its timing constraints while operate at lower voltages, thus saving power.

### 7.3 Selecting Bit Sizes

Bit-true simulations determined the proper bit widths and scaling to implement MLP coefficients as well as probabilities and their intermediate values. The C++ MLP performed its usual computations, but coefficients and probabilities were implemented using a Fixed integer class. This class accurately performed fixed width additions and multiplications with appropriate truncations, overflows, and under-flows.

The best fixed point implementations were found experimentally over the following variables: number of coefficient bits, scale factor for coefficient, number of probability bits, and scale factor for the probability. The scale factors are important because if they are too low, precision is lost; if they are too high, overflows will occur. Break points in accuracy gains were found with coefficient bit widths of 8 and datapath bit widths of 11, when scaling was set to eliminate overflows.

### 7.4 Interface

The MLP chip has two reset inputs: *reset*, and *porb*. When *reset* goes high, all of the controllers go to their start-up state. *porb* is an active low signal that resets the control on the ROM. *porb* need only be asserted at power up.

Output Signal or Bus	Pin (On Package)	Pad (On Chip)
Output Probs[0:5]	3:8	1:6
Rasta_Pop	74	65
Second_Output_Ready	9	7
layer_one_done	79	70
layer_two_done	22	20
First_Layer_Output[5:0]	32:37	23:28
Layer_One_State[0:3]	83:86	74:77
Layer_Two_State[3:0]	100:103	91:94
Mid_Buffer_State[2:0]	38:40	29:31
Rasta_State[2:0]	76:78	67:69

Table 7-3. MLP chip outputs.

The single phase clock is sent through the *Clock* input.

Input Signal or Bus	Pin (On Package)	Pad (On Chip)
Clock	99	90
Reset	104	95
PorB	105	96
Rasta_In_Ready	75	66
Next_Ready_Two	11	9
Rasta_In[10:0]	46:56	37:47
Start_layer_one	80	71
Start_layer_two	10	8
Num_Outputs[6:0]	90,91,94:98	81, 82, 85:89
Num_Output_Two[0:6]	12:16,20,21	10:14,19,20
Counter_In[6:0]	57:63	48:54
Num_rasta_per[6:0]	64:70	55:61

Table 7-4. MLP chip inputs.

The input and output data interfaces are simple 4 phase handshaking, in order to simplify interfaces to various other systems. When the *rasta\_pop* signal goes high, it indicates the MLP chip is ready to receive data on the *Rasta* input bus. The chip latches the data after the *rasta\_in\_ready* input signal goes high. After latching the data, the chip sets *rasta\_pop* low, indicating that the data is now free to change. The *rasta\_in\_ready* signal must go after *rasta\_pop* goes low, to acknowledge the transaction.

The same handshaking scheme is used to output the phoneme probabilities on the *OutputProbs* bus. In this case, the handshaking signals are the output, *second\_output\_ready*, and the input, *next\_ready\_two*.

Two other control signals, *start\_layer\_one* and *start\_layer\_two*, should be asserted at the beginning of each frame, after *rasta* data has been loaded on to the chip and probability data has been output from the chip. When the first and second layers have finished their computations for a frame, the assert the *layer\_one\_done* and *layer\_two\_done* signals, respectively.

Four of the inputs supply parameters that tell the chip the size of each layer in the MLP. During normal operations, these parameters are constant. However, they are programma-



Supply	Pin (On Package)	Pad (On Chip)
GND	17, 23, 41, 44, 73, 82, 88, 92, 106	15, 21, 32, 35, 64, 73, 79, 83, 97
Pad Vdd	18, 24, 43, 45, 72, 81, 89, 93, 107	16, 22, 34, 36, 63, 72, 80, 84, 98
Vdd	19, 42, 71, 87	17, 33, 62, 78

Table 7-5. MLP chip power supplies.

ble in order to facilitate simulation and testing, as well as to facilitate reuse of the design for different sized MLPs. Thus, the exact same controller design can be used with different sized memories to create a different sized MLP. *num\_outputs* is set equal to one less than the number of elements in the hidden layer; *num\_outputs\_two*, to one less than the number of elements in the output layer. *Num\_rasta\_per* specifies how many rasta coefficients are used per frame. *Counter\_in* specifies one minus the number of inputs to the first layer of the MLP in two's complement form. (*Counter\_in* is not the same as *num\_rasta\_per* because the input buffers stores 10 frames of context data.) The correct values for these parameters are shown in Table 7-6.

Parameter	Value
Num_Outputs	1110111
Num_Outputs_two	0110000
Num_rasta_per	0001010
Counter_in	0001001

Table 7-6. Parameter input values.

## 7.5 Results

The chip was tested at 10 MHz over a variety of voltages, with the power consumption shown in Table 7-7. The die is 4.6mm x 5.6mm in a 1.0  $\mu\text{m}$  process, as shown in Figure 7-6, and is bonded as shown in Figure 7-7.

Voltage	Power (mW)
3.3	4.6
3.0	2.8
2.5	1.9
2.2	1.5
2.0	1.2
1.8	0.9
1.7	0.5

Table 7-7. MLP chip power consumption.

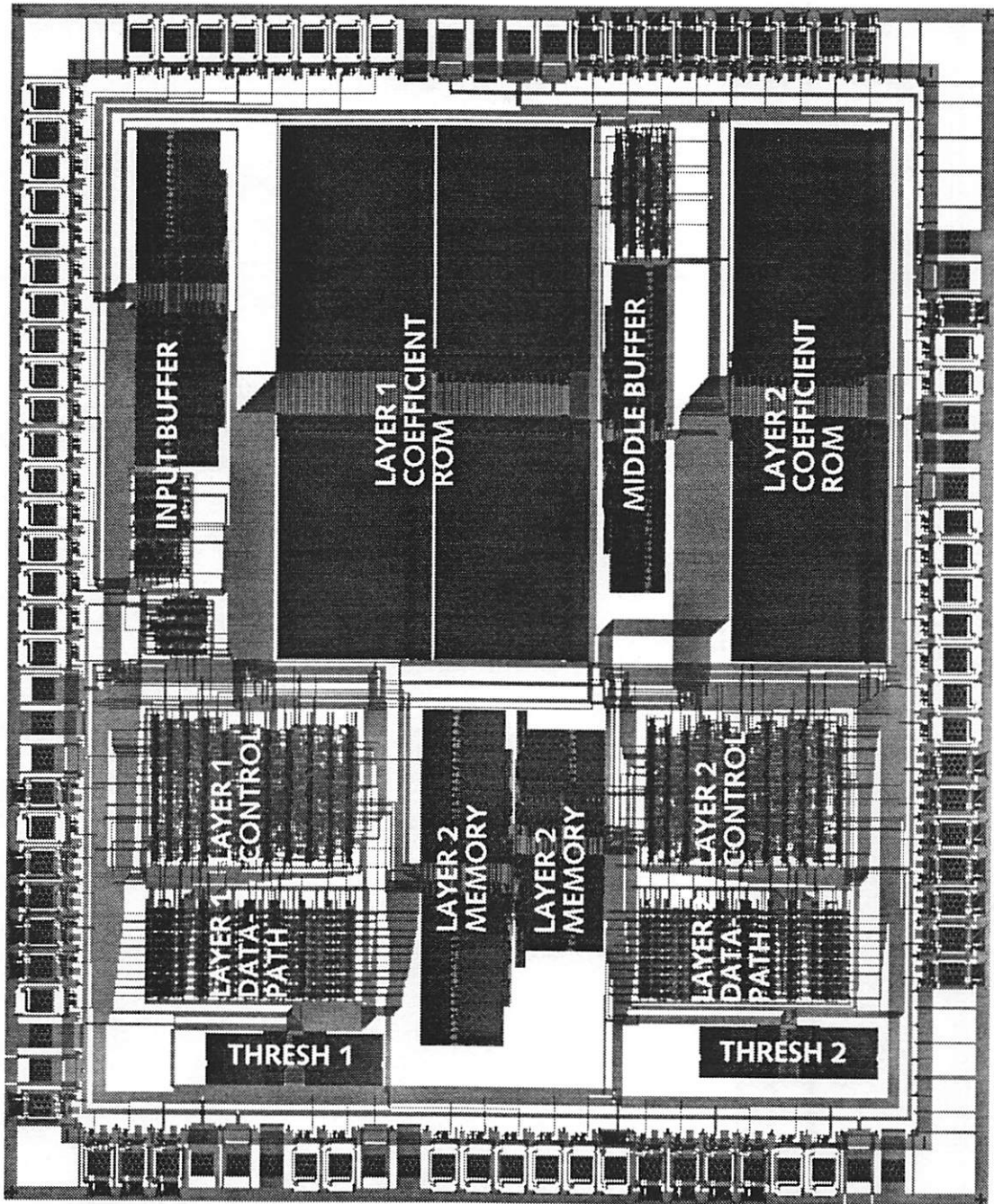
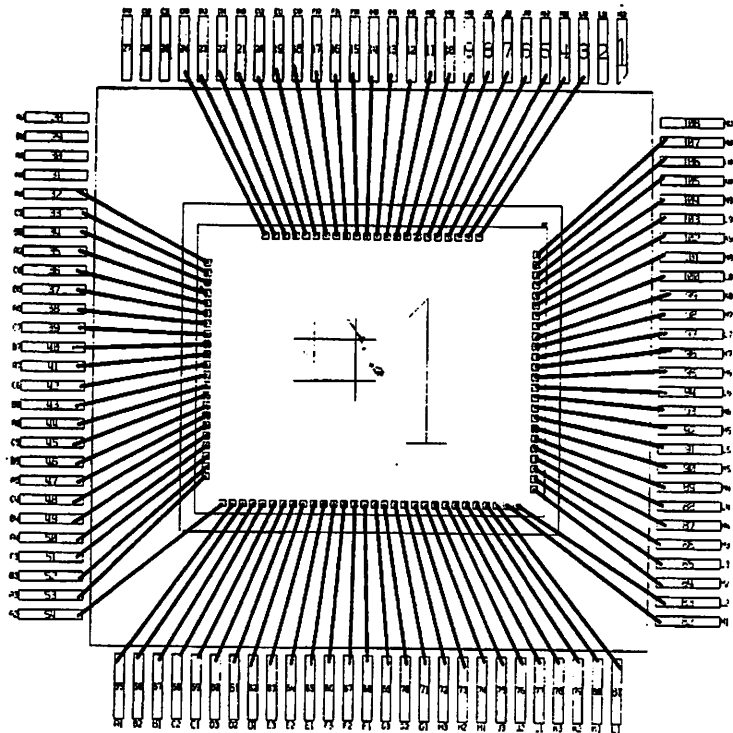


Figure 7-6. MLP chip layout.



N59FAZ 2
TOP VIEW  
 1376-DAR-RES/UOB  
 #1: 46956/BRODERSEN/MLP\_SPEECH  
 FGA108M = 25 PARTS  
 27-SEP-1995

Figure 7-7. MLP chip bonding diagram.

---

## 8 CONCLUSION

---

### 8.1 Contribution

This work involved a broad, vertical slice of the implementation of speech recognition. It ranged from the high level—writing applications—to the very low level—designing low power hardware. The spRcg speech recognition toolkit will provide a fast and easy-to-learn way for programmers to create speech recognition applications. The speech controlled WWW browser demonstrates the capabilities of spRcg, and also provides a good vehicle to demonstrate the InfoPad system. The MLP chip demonstrates a way to implement memory-intensive systems in a low-power fashion. In fact, the low-power memories, which were designed for use in the MLP chip, have been used in several other low-power projects and are planned to be used in several more in the near future.

### 8.2 Discussion & Future Work

The choice of algorithms and architectures for this system reflect a trade-off between the conflicting demands of ease of programming, ease of use, low power implementation, and recognition accuracy. Other work has shown RASTA-PLP to be effective even with few coefficients, making it well suited to low-power implementation. Likewise, a hybrid MLP-HMM system is a well-used system that is suited to low-power implementation when used with a clustered grammar. The clustered grammar itself should have little impact on recognition accuracy because it is topologically equivalent to a bigram grammar. Furthermore, its organization assists with both a low-power, parallel hardware implementation as well as with an easy to program API.

The choice of training the MLP on the TIMIT database, and then using its phoneme models for all applications has its advantages and disadvantages. On the positive side, it makes it much easier to create and rapidly test speech recognition applications: the programmer can create the entire vocabulary and grammar by dictating them to gram-Cracker, and the recognizer will work with enough accuracy to demonstrate the program. Thus, the programmer is freed from having to collect speech data before having a working application, and therefore does not need to collect new data every time the application changes its vocabulary. On the other hand, the recognition accuracy would benefit from more training on the actual vocabulary, as well as from more specific models for keywords—particularly digits. Furthermore, the speaker adaptation widget would benefit from a more sophisticated algorithm than simply replacing the existing pronunciation. As spRcg currently stands, it is well suited for developing new applications, but would benefit from a way to train application specific MLPs once the application has reached a mature stage and the programmer is able to collect data from test users.

Future work should continue to explore new options on where to perform speech recognition. Figure 8-1 shows several possible ways to partition speech recognition between a portable pad and the network. Each method has a different trade-off between bandwidth and hardware complexity. The topmost method, sending speech samples over the radio link to software in the network, is InfoPad's current speech recognition architecture. The second technique, performing RASTA-PLP on the pad and transmitting the coefficients, would entail integrating Steve Stoiber's work (described in Section 6.4) into the terminal. The third method would add the MLP IC, described in Chapter 7, to the RASTA-PLP IC on the terminal. The last method is a complete, custom IC implementation of the speech recognizer, which would also require the implementation of the Viterbi decoder design described in Chapter 6.

Future research should explore partitions between custom and general purpose hardware in the terminal itself. With the inclusion of low-power microprocessors [2] into portable terminals, the partition could be between custom hardware and software running in the terminal. For example, the spRcg system's C++ code could be ported to java, so that it could be downloaded into the terminal on demand. The exact nature of the custom IC/

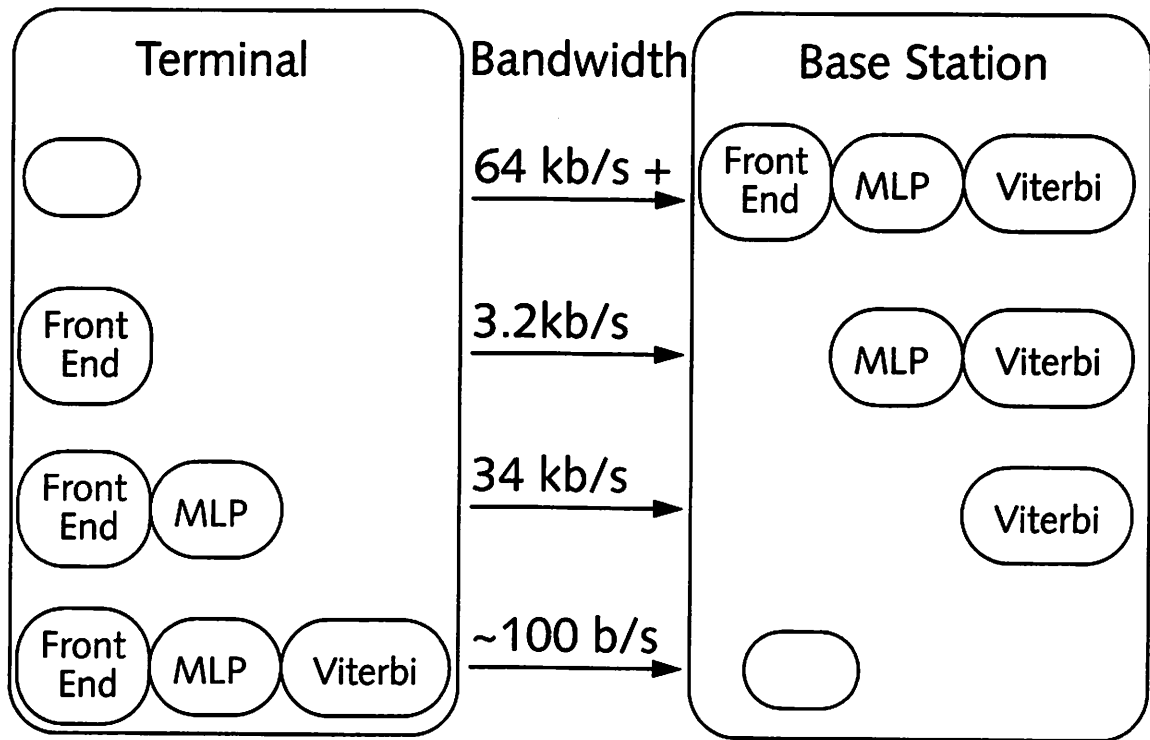


Figure 8-1. Partitioning recognition across the network.

terminal software/network software partition will depend on the cost and computational power of the low-power microprocessor system.

The future of special purpose speech recognition hardware within portable terminals may well depend on the future of general purpose computing in the terminals. The size and complexity of a speech recognizer in the terminal should be closely mated to the size, nature, and complexity of the stand-alone applications that run on the terminal. For example, if the terminal's only local operations are to control its connections to the network, at most it will need a small vocabulary recognizer that has been well trained on that single task. On the other hand, if the terminal runs a number of complex applications, it will need larger, more flexible speech recognition hardware, similar to the system described in this thesis.

---

## BIBLIOGRAPHY

---

- [1] H. Bourlard, N. Morgan, *Connectionist Speech Recognition A Hybrid Approach*, Kluwer Academic Publishers, 1994.
- [2] T. Burd and R. W. Brodersen, "Energy Efficient CMOS Microprocessor Design," Proceedings of the 28th Annual Hawaii International Conference on System Sciences, Volume I, January 1995, pp. 288-297
- [3] A. P. Chandrakasan, A. Burstein, R. W. Brodersen, "A Low-power Chipset for Multimedia Applications", IEEE Journal of Solid State Circuits, Vol. 29, No. 12, pp. 1415-1428, December, 1994.
- [4] A. P. Chandrakasan, "Low-power Digital CMOS Design", UCB/ERL Memorandum No. M94/65, August 30, 1994.
- [5] A. P. Chandrakasan, S. Sheng, R. W. Brodersen, "Low-power Digital CMOS Design", IEEE Journal of Solid State Circuits, pp. 473-484, April 1992.
- [6] R. Gonzalez and M. Horowitz, "Energy Dissipation in General Purpose Processors," Proceedings of the IEEE Symposium on Low Power Electronics, October 1995, pp. 12-13.
- [7] R. Gray, "*Vector Quantization*", IEEE ASSP magazine 1(2), April 1984, pp. 4-29
- [8] H. Hermansky, "Perceptual Linear Predictive (PLP) Analysis of Speech, Journal of the Acoust. Soc. Am., vol 87, no. 4.
- [9] H. Hermansky, A. Bayya, N. Morgan, P. Kohn, "Compensation for the Effect of the Communication Channel in Perceptual Linear Predictive (PLP) analysis of speech", Proc. of Eurospeech 1991, pp. 1367-1370, Genova, Italy.
- [10] M. Horowitz, et al., "Low-Power Digital Design," Proceedings of the IEEE Symposium on Low Power Electronics, October 1994, pp. 8-11.



- [11] K.F. Lee, H.W. Hon, "Speaker-Independent Phone Recognition Using Hidden Markov Models", IEEE Trans. on Acoustics, Speech, and Signal Processing, vol. 37, no. 11, November 1989.
- [12] K.F. Lee, Automatic Speech Recognition - The Development of the Sphinx System, Kluwer Academic, Norwell Mass.
- [13] H. Murveit et al, "SRI's DECIPHER System", Proc. of the Speech and Natural Language Workshop, pp. 238 - 242, Feb. 1989
- [14] K. Ma and N. Morgan, "Scaling Down: Applying Large Vocabulary Hybrid HMM-MLP Methods to Telephone Recognition of Digits and Natural Numbers," IEEE Workshop on Neural Networks for Signal Processing 1995, pp 223-232.
- [15] N. Narayanaswamy, "Pen and Speech Recognition in the User Interface for Mobile Multimedia Terminals", Ph.D. Thesis, UC Berkeley, 1996.
- [16] J. Ousterhout, ACM IEEE 22nd Design Automation Conference, 1984.
- [17] J. Ousterhout, Tcl and the Tk Toolkit, Addison Wesley Publishing Company, 1994.
- [18] D. Patterson, "The Case for NOW," <http://now.cs.berkeley.edu/Case/case.html>.
- [19] L.R. Rabiner, "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition, Proceedings of the IEEE, vol. 77, no. 2, pp. 257-285.
- [20] D.E. Rumelhart, G. E. Hinton, Williams, R.J., *Parallel Distributed Processing. Exploration of the Microstructure of Cognition. vol. 1: Foundations*, MIT Press, 1986.
- [21] S. Sheng, A.P. Chandrakasan, R.W. Brodersen, "A Portable Multimedia Terminal", IEEE Communications Magazine, PP. 64-75, December 1992.
- [22] C. Shung, R. Jain, K. Rimey, E. Wang, M. Shrivastava, B. Richards, E. Lettang, S. Azim, L. Thon, P. Hilfinger, J. Rabaey, R. Brodersen, "An Integrated CAD System for Algorithm-Specific IC Design", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 10, No. 4, Apr. 1991, pp.447-475.
- [23] S. Stoiber, *Low Power Digital Signal Processing for Speech Recognition*, Master Thesis, UC Berkeley, December 14, 1994.
- [24] A. Stölzle et al., "A Flexible VLSI 60,000 Word Real Time Continuous Speech Recognition System", IEEE Press book: VLSI Signal ProcessingIV, ISBN 0-87942-271-8, Chapter 27, pp 274 - 284, Nov. 1990
- [25] A. Stölzle et al, "Integrated Circuits for a Real-Time Large-Vocabulary Continuous Speech Recognition System", IEEE Journal of Solid State Circuits, vol. 26, no.1, pp. 2-11, Jan. 1990

- [26] "*Readings in Speech Recognition*", edited by A. Waibel and K. Lee. Morgan Kaufmann Publishers, Inc., ISBN 1-55860-124-4, 1990.
- [27] TIMIT Acoustic-Phonetic Continuous Speech Corpus, NIST Speech Disc 1-1.1, October 1990, NTIS Order No. PB91-505065.
- [28] TIDIGIT Speaker-Independent Connected-Digit Corpus, NIST Speech Discs 4-1, 4-2, 4-3, February 1991, NTIS PB91-506592.
- [29] Joseph Wang, tkwww, Globewide Network Academy, Macvicar School of Education and Technology, 1993.
- [30] Status of the Library of Common Code, <http://info.cern.ch/hypertext/WWW/Library/Status.html>

---

## APPENDIX: DIRECTORIES

---

This appendix describes the locations of the files and directories that contain the software and hardware designs described in the previous chapters.

The spRcg speech recognition system is located in the User Interface (UI) group's directories: `/tools/ui/speechRecog/spRcg`. Under this directory lie various versions: `vcurrent` is the working directory and `vdev` is the one under development. Within each version are the following directories:

- `src` contains the C++ source files for the spRcg extension to tcl, as well as the `Makefile`.
- `include` contains the file `spRcg.h`, which must be included by source files that include spRcg in a new type of wish.
- `libtcl` contains the tcl/tk and itcl code that make up the rest of spRcg. It also contains the code for gramCracker.
- `example` contains the source code from the example in Chapter 4.
- `man` and `doc` contain documentation.
- `sunos` and `solaris` contain executables ( a wish that includes spRcg) and libraries.

The memories are installed in the low-power Timlager library: `lager/common/LagerIV/cellib/low_power/TimLager`. The ROM is in the `lprom2` directories. There are 3 versions of the SRAM:

- `sram` is the standard scmos design.
- `sramsmall` is the scmos design, but with a single block layout.

- `sram_hp` is designed for the `scmos-hp` design rules.

The files for the mlp chip are in `~burststein/speech/hardware/mlpChip/mlp6`.

The top level master design is `mlp_chip`; the top level instance is `my_mlp_chip`.