# A VECTORIZED PROCESSING ALGORITHM FOR CONTINUOUS SPEECH RECOGNITION AND ASSOCIATED FPGA-BASED ARCHITECTURE

by

Jeffrey William Schuster

B.S., University of Pittsburgh, 2004

Submitted to the Graduate Faculty of

School of Engineering in partial fulfillment

of the requirements for the degree of

Master of Science

University of Pittsburgh

2006

UNIVERSITY OF PITTSBURGH

SCHOOL OF ENGINEERING

This thesis was presented

by

Jeffrey William Schuster

It was defended on

March 29, 2006

and approved by

Raymond Hoare, Ph.D., Assistant Professor, Department of Electrical and Computer Engineering

Steven Levitan, Ph.D., John A. Jurenko Professor, Department of Electrical and Computer Engineering

Alex K. Jones, Ph.D., Assistant Professor, Department of Electrical and Computer Engineering

Thesis Advisor: Raymond R. Hoare, Ph.D., Assistant Professor, Department of Electrical and Computer Engineering

# A VECTORIZED PROCESSING ALGORITHM FOR CONTINUOUS SPEECH RECOGNITION AND ASSOCIATED FPGA-BASED ARCHITECTURE

Jeffrey William Schuster, M.S.

University of Pittsburgh, 2006

This work analyzes Continuous Automatic Speech Recognition (CSR) and in contrast to prior work, it shows that the CSR algorithms can be specified in a highly parallel form. Through use of the MATLAB software package, the parallelism is exploited to create a compact, vectorized algorithm that is able to execute the CSR task. After an in-depth analysis of the SPHINX 3 Large Vocabulary Continuous Speech Recognition (LVCSR) engine the major functional units were redesigned in the MATLAB environment, taking special effort to flatten the algorithms and restructure the data to allow for matrix-based computations. Performing this conversion resulted in reducing the original 14,000 lines of C++ code into less then 200 lines of highly-vectorized operations, substantially increasing the potential Instruction Line Parallelism of the system.

Using this vector model as a baseline, a custom hardware system was then created that is capable of performing the speech recognition task in real-time on a Xilinx Virtex-4 FPGA device. Through the creation independent hardware engines for each stage of the speech recognition process, the throughput of each is maximized by customizing the logic to the specific task. Further, a unique architecture was designed that allows for the creation of a static data path throughout the hardware, effectively removing the need for complex bus arbitration in the system. By making using of shared memory resources and applying a token passing scheme to the system, both the data movement within the design as well as the amount of active data are continually minimized during run-time. These results provide a novel method for perform speech recognition in both hardware and software, helping to further the development of systems capable of recognizing human speech.

## TABLE OF CONTENTS

1.0 INTRODUCTION....1
2.0 PREVIOUS WORK....7
2.1 MAJOR RESEARCH TOPICS....7
2.1.1 Signal Processing....7
2.1.2 Discrete Vs. Continuous Models....8
2.1.3 Gaussian Selection....10
2.1.4 Feature Selection....11
2.1.5 Token Passing....12
2.2 COMMERCIAL SOFTWARE SYSTEMS....13
2.2.1 IBM’s Via Voice....14
2.2.2 BBN’s Byblos....14
2.2.3 SRI’s DECIPHER....15
2.2.4 CMU’s SPHINX....16
2.3 CUSTOM HARDWARE ARCHITECTURES....18
2.3.1 Sensory Inc....19
2.3.2 University of Birmingham....19
2.3.3 University of California at Berkley....20
2.4 VECTOR PROCESSING ARCHITECTURES....22
2.4.1 VLIW Processors....22
2.4.2 SIMD Processors....24
2.4.3 Sony, Toshiba, IBM CELL Processor....25
2.4.4 Stretch Inc. S5000 Chip Family....27
3.0 SYSTEM ARCHITECTURE....29

3.1 SYSTEM OVERVIEW....29
3.2 PRELIMINARY ANALYSIS....33
3.3 SYSTEM LEVEL HARDWARE....35
3.3.1 System controller....35
3.3.2 Central Data Structure....37
3.4 TEST / DEVELOPMENT ENVIRONMENT....38
4.0 ACOUSTIC MODELING....39
4.1 GAUSSIAN PROBABILITY EVALUATIONS....39
4.1.1 One-Dimensional Gaussian Probability Density Functions....40
4.1.2 D-Dimensional Gaussian Probability Density Functions....44
4.2 DESCRIPTION OF CALCULATIONS....47
4.3 MATRIX MATLAB REPRESENTATION....50
4.4 HARDWARE ARCHITECTURE....55
4.4.1 Gaussian Distance Pipelined Processor....56
4.4.2 Log-Add LUT....58
4.4.3 Find MAX / Normalizer....61
4.4.4 Composite Senone Evaluator....63
4.5 HARDWARE PERFORMANCE RESULTS....66
5.0 PHONEME EVALUATION....71
5.1 HIDDEN MARKOV MODELING....71
5.1.1 Mathematics....72
5.1.2 HMM Topologies....73
5.1.3 Viterbi Searching....76
5.2 DECSRIPTION OF CALCULATIONS....78
5.3 MATRIX MATLAB REPRESENTATION....82
5.4 HARDWARE ARCHITECTURE....85
5.4.1 HMM Control Logic....86
5.4.2 HMM Pipeline....89
5.4.3 HMM Pruner....91
5.5 HARDWARE PERFORMANCE RESULTS....92
6.0 WORD MODELING....97

6.1 TREE SEARCHING.....97
6.2 MATRIX MATLAB REPRESENTATION.....100
6.3 HARDWARE ARCHITECTURE.....103
6.3.1 Token Deactivator.....103
6.3.2 Token Activator.....104
6.4 HARDWARE PERFORMANCE RESULTS.....108
7.0 PERFORMANCE PROFILES.....113
7.1 LITERATURE PERFORMANCE REVIEW.....113
7.2 MATRIX MATLAB PERFORMACE PROFILE.....115
7.3 HARWARE PROFILE SUMMARY.....121
8.0 CONCLUSIONS, CONTRIBUTIONS, & FUTURE DIRECTIONS.....125
8.1 CONCLUSIONS.....125
8.2 CONTRIBUTIONS.....125
8.3 FUTURE DIRECTIONS.....127
APPENDIX A: VECTORIZED MATLAB CODE.....131
A.1 RM1_top_level.m.....131
A.2 data_loader.m.....133
A.3 acoustic_modeler_comp_senones.m.....135
A.4 logs3_add_look_up_float.m.....136
A.5 HMM_eval_no_comps.m.....136
A.6 word_model.m.....138
APPENDIX B: VHDL CODE.....141
B.1 AM_calc.vhd.....141
B.2 gaus_calc_top_v2.vhd.....152
B.3 composite_senone_calc.vhd.....166
B.4 find_max.vhd.....176
B.5 gaus_dist_full_pipe_V2_FINAL.vhd.....180
B.6 log_add_calc_V1_FINAL.vhd.....192
B.7 normalizer.vhd.....203
B.8 hmm_top_struct.vhd.....209
B.9 hmm_pipe_w_mdef.vhd.....229

B.10 hmm_calc_vNEW.vhd....241
B.11 mdef_rom_struct.vhd....251
B.12 ph_ptr_ram_NEW_struct.vhd....260
B.13 new_pruner.vhd....266
BIBLIOGRAPHY....271

## LIST OF TABLES

Table 1. Number of Compute Cycles for 3 Different Speech Corpuses...................................... 35
Table 2. Number of Calculations Necessary for Various Dictionaries........................................ 48
Table 3. Parallelism in Acoustic Modeling...................................................................................... 53
Table 4. Summary of Synthesis and Place-and-Route Results for Virtex-4 SX35....................... 71
Table 5. Summary of Synthesis Results for HMM Pipeline using both Precision Synthesis and Synopsys Synplify ........................................................................................................ 97
Table 6. Summary of Place-and-Route Results for PE.................................................................. 98
Table 7. Summary of Word Model Synthesis Results................................................................ 112
Table 8. Summary of Software Performance Results................................................................. 115
Table 9. Summary of Hardware Performance Results................................................................ 124

## LIST OF FIGURES

Figure 1. Continuous vs. Discrete Gaussian PDFs ........ 9

Figure 2. A Simple Tree Structure ........ 13

Figure 3. Sample HMM Topology ........ 17

Figure 4. Diagram of Decoder Architecture ........ 22

Figure 5. SIMD Processing With and Without Filter Vector ........ 25

Figure 6. Block Diagram of Cell Architecture ........ 27

Figure 7. Stretch Inc. S5000 Processor Engine ........ 29

Figure 8. Block Diagram of Software-Based ASR System ........ 31

Figure 9. Block Diagram of the *Speech Silicon* Hardware-Base ASR System ........ 33

Figure 10. FSM for System Controller ........ 37

Figure 11. Organization of Fields in Database ........ 38

Figure 12. Sample Set of Gaussian Distributions for Multiple $\sigma^2$ Values ........ 42

Figure 13. Distribution of 2,100 Cepstral Inputs for 4 Different Coefficients ........ 44

Figure 14. Percentage Error Vs. Percentage of Database Removed for Varying n Values ........ 45

Figure 15. Percentage Contribution of Each Dimension of a Multivariate Gaussian PDF ........ 47

Figure 16. Initialization Code for AM ........ 54

Figure 17. Calculation of Component Scores ........ 54

Figure 18. Block Diagram of Vectorized Gaussian PDF Calculation ........ 55

Figure 19. Code for Vectorized Calculation of Log-Summation ........ 56

Figure 20. Code Necessary to Calculate Senone Scores ........ 57

Figure 21. Code for Composite Senone Calculation ........ 58

Figure 22. Block Diagram for Acoustic Modeling Co-Processor ........ 59

Figure 23. Data Flow Graph for Gaussian Distance Pipe ........ 60

Figure 24. Data-Flow Graph for Log-Add LUT ........ 63

Figure 25. Data-Flow Graph for Find Max Unit ........ 65

Figure 26. Data Flow Graph for Composite Senone Calculation ........ 68

Figure 27. Analysis of $f_{max}$ vs. Pipeline Stages for Virtex-4 SX and Spartan-3 FPGAs ............ 70
Figure 28. Speed Improvement vs. Percentage of Design Constrained for Gaussian Distance Pipe ........................................................................................ 72
Figure 29. Post Place-and-Route Layout for AM Pipeline on a Virtex-4 SX35 FPGA ............ 73
Figure 30. Linear HMM Topology ........................................................................................ 77
Figure 31. Comparison of Bakis and Left-to-Right Topology HMMs ..................................... 77
Figure 32. Sample Alternative Paths HMM Topology ............................................................ 78
Figure 33. Sample Ergodic HMM Topology ........................................................................... 79
Figure 34. Sample HMM Trellis ............................................................................................ 80
Figure 35. HMM Trellis with Viterbi Algorithm ...................................................................... 81
Figure 36. Data-Flow Diagram for Memory Access for HMM Calculation ............................. 83
Figure 37. HMM Topology with Labeled Values ..................................................................... 84
Figure 38. Code for PE initialization ....................................................................................... 86
Figure 39. Code for Data Allocation in PE .............................................................................. 87
Figure 40. Code for HMM Calculation and Data Storage ........................................................ 87
Figure 41. Code for HMM Pruning ......................................................................................... 88
Figure 42. Control for Phoneme Evaluator ............................................................................. 90
Figure 43. Finite State Machine for PE Control ....................................................................... 91
Figure 44. Data-Flow Graph for HMM Pipeline ...................................................................... 93
Figure 45. Data-Flow Graph for HMM Pruner ......................................................................... 95
Figure 46. Precision Synthesis Vs. Synopsys Synplify ............................................................ 96
Figure 47. Final Layout for PE ................................................................................................. 99
Figure 48. Sample Tree Topologies ........................................................................................ 102
Figure 49. Code for Token Deactivation and Frame Initialization ........................................... 104
Figure 50. Code for Obtaining Link-List Information ............................................................. 105
Figure 51. Code for Obtaining Leaf-Node Propagation Information ....................................... 105
Figure 52. Code for Main WM Loop ....................................................................................... 106
Figure 53. Sample Word Tree ................................................................................................. 108
Figure 54. Sample Search Space Node Mapping ...................................................................... 109
Figure 55. Data Flow Graph for Token Activator .................................................................... 111
Figure 56. Performance analysis of MATLAB Model for Acoustic Modeling ......................... 117

Figure 57. Results of Code Profile for PE .......... 119
Figure 58. Code Profile for WM .......... 120
Figure 59. Code Profile for Entire MATLAB Code .......... 121

## PREFACE

I would like to pay special thanks to all of the people who helped me along this journey. To all my lab mates past and present, especially Dara, Shenchih, & Eliott, for providing both the intellectual and moral support to continue pursuing my education. To the members of my research team, Kshitij and Johnny, for correcting me when I was wrong and for all the long nights spent in debate. To all of my professors for the knowledge they passed on, the questions they answered, and most importantly the questions they posed that forced me to reach further and learn more. A special thanks goes to Dr. Hoare, Dr. Jones, and Dr. Levitan for appearing on my committee and helping me fine tune the end result of many years of hard work. Dr. Hoare, who gave me the opportunity that not even I thought I deserved, my deepest thanks to you for the chance to expand my horizons and the chance to begin developing into the engineer I hope to be. I would also like to thank my friends and family for their support and encouragement through all of the highs and lows of the past few years. Finally, a world of thanks to my mother and father for their unwavering belief in my ability to achieve anything and for giving me the attitude and poise to attempt it.

# 1.0 INTRODUCTION

As speech recognition technology was being developed in the 80's and 90's researchers examined numerous techniques for reducing the computational complexity of the process, attempting to reach middle-ground between speed and accuracy.  One the one side of the argument were isolated word systems that could only operate on a single word at a time but accurate on medium to large scale dictionaries.  In the opposing camp were systems focused on the recognition of conversational speech, allowing the user to interact with the device as they would a person but severely limiting the size of the dictionary as well as the potential accuracy. Many of the first systems to attempt speech recognition used discrete models in which isolated words were able to be deciphered as opposed to conversational speech.  By removing any cross-word articulation effects, and limiting the number of recognizable words to those necessary for a specific task, the accuracy of a system could be increased substantially without dramatically affecting the computational load.  Using discrete models does however require the user to speak to the machine in isolated words which sounds very different from conversational speech.  This difference can take some time for a person to adjust to and it most situations is highly undesirable.

In contrast to discrete speech systems, other groups worked with single-user, systems, able to handle more conversational speech, but only from one user.  These systems require that the user train the system to understand them, leading to the need for numerous systems in a single building, each dedicated to one person's voice.  Using speaker-dependent systems also requires the storage of large amounts of data, since each speaker needs their own specific model, which can become a very costly process in systems where multiple speakers are using large-scale dictionaries.  Additionally, these types of systems have no certainty that they can recognize words outside of the set they were trained on.  A simple example of a speaker-dependent system can be seen in current mobile phone technology where by recording a voice command, that command can be repeated at a later time to create an action. Unfortunately, these systems are semi-accurate, and can be cumbersome if multiple recognitions are necessary to complete a task.

With the rise of high performance computing, new techniques have been developed that take a more broad approach to the problem, attempting to exploit computational ability for the sake of multiple user, continuous speech recognition. Most of these systems rely on complex software- based algorithms to analyze incoming speech requiring state-of-the-art processors and large amounts of memory. In these speaker-independent systems, large numbers of probability evaluations are performed to determine the most likely sequence of sounds heard, which in turn can be correlated to the most likely sequence of words heard by the system. In order to achieve acceptable recognition rates extensive off-line training must occur to ensure that the models being used by the system are both general enough to cover a large base of different speakers and yet unique enough to accurately represent the different sounds in a given speech corpus. Even with modern computers these methods still suffer serious computational overhead, taking between 0.6X real-time on a 1.7GHz Athalon Processor to upwards of 10X real-time when running on a 450MHz Pentium III style device, to analyze a full English speech corpus. This results in systems that perform transcription oriented tasks quite well but fall short of user expectations when continuous real-time recognition is required [1, 7, 8, 9].

The current generation of ASR technologies can be broken down into three basic functional units: a Feature Extraction unit, an Acoustic Modeler, and a Language Modeler [2, 6, 18]. Acoustic modeling is consistently the most computationally intensive of the three phases, taking up anywhere from 30% to 95% of the computation time [1, 3, 4]. Acoustic modeling can be broken down into two major components, the actual Acoustic Modeler and the Phoneme Evaluator. The Acoustic Modeler is responsible for comparing the incoming data to a pre-defined set of Gaussian Probabilities via the evaluation of multi-dimensional Gaussian PDFs. These evaluations are computationally intensive and can require upwards of 4.9 million floating point operations per second to completely evaluate the database [5]. To further the problem many of the state-of-the-art recognition systems like BBN's *Byblos,* CMU's *Sphinx,* Cambridge University's *HTK Toolkit,* IBM's *Via Voice,* and SRI's *DECIPHER,* perform statistical pattern matching using Hidden Markov Models (HMMs) during phoneme evaluation, requiring the calculation of hundreds of thousands of state probabilities based on the results obtained by the

Acoustic Modeler [6, 7, 8, 9]. In the Language Modeling phase a high-level tree structure is created to link the phonemes together into words. This tree can become cumbersome and can cause severe pipe-line stalls, resulting in an overall IPC that ranges between 0.37 and 1.2 [5]. Additional details on previous work and processing technologies can be found in Chapter 2 of this document.

The first major contribution of this research examines the calculations being performed in software and exposes the potential for parallelism. It has been shown that systems such as the SPHINX recognition engine have very poor IPC rates, between 0.5 and 0.6 for a 1.7GHz AMD processor, but the equations begin performed imply much larger potentials [5]. This potential parallelism is illustrated through the reorganization of the data to allow for large matrix-based operations in MATLAB. These operations while executing sequentially on a Pentium 4 processor, show the potential for parallelism upwards of 600,000 elements for certain operations. Additionally a token passing scheme is employed, resulting in reductions in the computational workload of the system. The derivation of MATLAB code based on point-wise matrix operations serves to illuminate the potential for parallelism in the speech recognition process and to highlight the portion of the algorithm that stand to benefit most from large parallel operations. Combining the potential increase in parallelism with a method for reducing unnecessary data accesses, answers are provided for both the performance and memory problems associated with ASR leading to the possibility for ASR systems able to perform speaker-independent recognition in real-time.

The second major contribution of this work uses the potential parallelism for the design of pipelined hardware on a 90nm FPGA capable of operating at over 100MHz. Being able to operate with a clock frequency over 100MHz ensures real-time operation based on the amount of calculations necessary for a 1,000 word dictionary given that the job takes less than 1 million cycles. The specific details correlating the 100MHz clock to the ability to recognize speech in real-time are discussed in depth in Chapter 3. Using the matrix-based MATLAB code as a template for the creation of a hardware system the portions of the code that benefited most from parallelism could be specially designed to exploit it. Developing custom hardware blocks for each portion of the speech recognition task allowed for the creation of an architecture that is able to efficiently manage the data in between the blocks as well as efficiently processing it within a given block. Each block in the design was created to allow for the use of a token passing scheme to manage the active data in the system. Through the use of shared RAM arrays to store the active data and FIFOs to manage the active workload of the system, the designed system is able to perform a minimal amount of work at all times while simultaneously maximizing the bandwidth in each cell. Additionally, the Acoustic Modeling hardware block is flexible and its computations can be configured through the data stored in RAM; thus, a variety of Gaussian calculations can be computing using this block.

The following chapter of this work presents an array of previous works in both hardware and software to help characterize the current state-of-the-art. Additionally, the major research areas in speech recognition are discussed to illuminate the corners of the field that are currently of special interest. The chapter will conclude with a brief analysis of some existing processor architectures that while not specifically designed for speech recognition, highlight some unique architectures that serve to illustrate the desired architecture in an ideal speech recognition system.

After a summary of the current research field, Chapter 3 will present the high-level system details for the designed system. First, a summary of the major operations for the speech recognition process is given to lay the foundation for the remainder of the document. A preliminary analysis of the task to be performed is then given to help quantify the complexity of the project and the potential resources required for a functional system. The next section of this chapter describes the system-level hardware needed to interconnect the individual hardware cells. This hardware represents a joint effort between myself and the members of my research group to establish an efficient means for moving data throughout the system and while not pertinent to the contributions of this thesis, is necessary for sake of complete understanding of the final design. Chapter 3 then concludes with a description of the testing and development environments used throughout the project.

Chapter 4 begins the bulk of the contributions of this thesis by presenting the work done on the Acoustic Modeling portion of the speech recognition process. Acoustic Modeling involves the evaluation of thousands of multi-variant Gaussian distributions for each new input to the system and is a mathematically complex operation. After presenting the preliminary mathematics for the process, the development of the MATLAB algorithm is discussed. This discussion will cover the complete code used by MATLAB to perform the Acoustic Modeling algorithm and show how the use of large-scale matrix operations can help to simplify the operations needed for the process. Next, the hardware development will be examined, paying special attention to derivation of the pipeline used for the core of the Acoustic Modeling calculation. In this section, the specific operation of the pipeline will be discussed, and its ability to change functionality based on the input configuration will be examined. Having detailed the hardware necessary for completion of the Acoustic Modeling task, this chapter will conclude with an analysis of both the synthesis and the post place-and-route results for the derived hardware. Additionally, this chapter briefly discusses the ability to use state-of-the-art design tools to help increase the performance of the design without presenting any additional designer effort.

After summarizing the Acoustic Modeling process this document will go on to describe the next major portion of the design, the Phoneme Evaluation block. Phoneme Evaluation is the process of utilizing the data generated by the Acoustic Modeler to evaluate the active set of Hidden Markov Models in the system. Hidden Markov Models, HMMs, are used to represent the individual phonetic units of a given language, and their evaluation presents its own unique set of computational problems. Chapter 5 follows a similar format to Chapter 4 beginning first with an introduction to Hidden Markov Models, followed by a description of the major mathematics needed for the process. From here, the software development will first be described concentrating on the ability to use compact pointer vectors to perform sparse-matrix operations, leading to the ability to remove all loops from the phoneme evaluation process. As in the previous chapter the description of the software development will next lead to the derivation of the custom hardware created to perform phoneme evaluation. During the description of the hardware development special attention will be paid to the large data allocation problem presented by phoneme evaluation and how these conflicts were resolved. Chapter 5 concludes with a summary of the synthesis and place-and-route results for the hardware design and examines the effects of different synthesis tools on the end performance of the designed logic.

In the same style as the previous chapters, Chapter 6 presents the work done for the Word Modeling process. This process involved the evaluation of a large tree structure and the propagation of information throughout it. While the MATLAB code for this block represents a unique contribution to this work and helps to quantify MATLABs ability to perform search based operations, the hardware development is less critical and presented only for the sake of completeness in the document. While the hardware cell for this portion of the design is based on the derived MATLAB code, it is not unique work nor is it a performance critical portion of the design worth focusing significant attention on. As in Chapters 4 and 5 a summary of the place-and-route results will be given although with not as much detail as in the preceding chapters.

Having examined the development of the hardware and software cells for the speech recognition system, Chapter 7 will spend some time quantifying the performance of the MATLAB code written for this thesis. While profiling of the MATLAB code does not lead directly to a performance profile of the associated C-code, nor does it exemplify the performance boundaries of the hardware device, it does provide a unique insight into the capabilities of the MATLAB computing engine and helps to show how MATLAB can greatly accelerate some operations while impeding others. This chapter is presented solely to help characterize the MATLAB code and the associated programming environment in an effort to show the completeness of the work performed on the software algorithms.

This work concludes with a summary of the major contributions of the research and presents the potential future directions for research in this area. Chapter 8 will discuss the final results of the project and help to summarize the specific contributions that make this work unique and beneficial to the speech recognition community. After discussing the benefits of this specific work, Chapter 8 will conclude with a contemplation of the potential to develop this work into real-world products capable of improving the quality of life of generations to come.

# 2.0 PREVIOUS WORK

Since the first researchers began to look at the speech recognition problem in the early 70's the amount of data required has always been a limiting factor. The amount of information that humans use to process and understand speech is much greater than what a modern computer can process in real-time, resulting in the need to trade speed for accuracy or visa versa. Exploration of these trade-offs has resulted in some major advancements in both the signal processing used to transform the incoming audio into some useful information, as well as the architectures of the software systems used to process this information in an efficient and meaningful way.

## 2.1 MAJOR RESEARCH TOPICS

During the past 40 years of ASR research a number of developments have helped to bring the technology to the level that we are currently familiar with. These advances have occurred in a number of different research areas from signal processing to computer system architecture and each new development has helped push the industry forward. The following section will take a look at some of the key developments in the ASR field over the past few decades and illustrate how each of these events played into the current state-of-the-art technology.

### 2.1.1 Signal Processing

Before any actual speech processing can begin, it is first necessary to convert the incoming audio into some useful information that uniquely represents the sounds heard in a given sample. Modern ASR systems employ a number of standard frequency-domain transforms such as the Fast Fourier Transform (FFT), Discrete Cosine Transform (DCT), and Cepstral Transform to extract information about the features, frequency content, of a sample [10, 12, 14]. As Signal Processing evolved along side high-performance computing in the 80's and early 90's, it was found that using greater numbers of features can noticeably increase word recognition accuracy [11, 12]. Early systems generally relied on low-order Liner Prediction Coefficients (LPCs), usually only the first 10-13, to extract the information regarding the relative frequency content of a speech sample. Most of these systems also relied on a frequency-warping transform, usually a bilinear, to warp the frequency axis and give the frequencies around the human speech band more weight. Eventually it was found that converting LPCs to Cepstral Coefficients and also making use of the first and second derivatives enabled a significant improvement in recognition accuracy [13, 14]. Also aiding in ability to obtain such high recognition rates was the application of a new frequency transform, the Mel-scale transform which is described in detail by [19]. This transform breaks the frequency band into as many as 40 separate regions and quantizes the information in each of the regions separately to allow for maximum frequency resolution in each band. Based on these high word accuracy rates, the use of a 39-dimensional Mel-Frequency Cepstral Coefficient (MFCC) vector to quantize the incoming audio has become the methodology of choice for feature extraction, resulting in the "standard" acoustic front-end as described by [15]. Despite the increase in computation this standard acoustic front-end is used in most state-of-the-art systems including BYBLOS, DECIPHER, and SPHINX 3 due to the substantial increase in recognition that can be obtained [9, 16, 17].

### 2.1.2 Discrete Vs. Continuous Models

The majority of state-of-the-art speech recognizers rely on the use of Hidden Markov Modeling (HMM) techniques to correlate the data provided by the Feature Extractor to a known database of phonetic units, phonemes [6, 7, 8, 9, 16]. HMMs can be viewed as either continuous or discrete based on the types of data they attempt to model, with the continuous models requiring significantly more calculations than their discrete counterparts. Figure 1 shows the difference between the Gaussian Probability data used to score the HMMs in a discrete versus a continuous system.

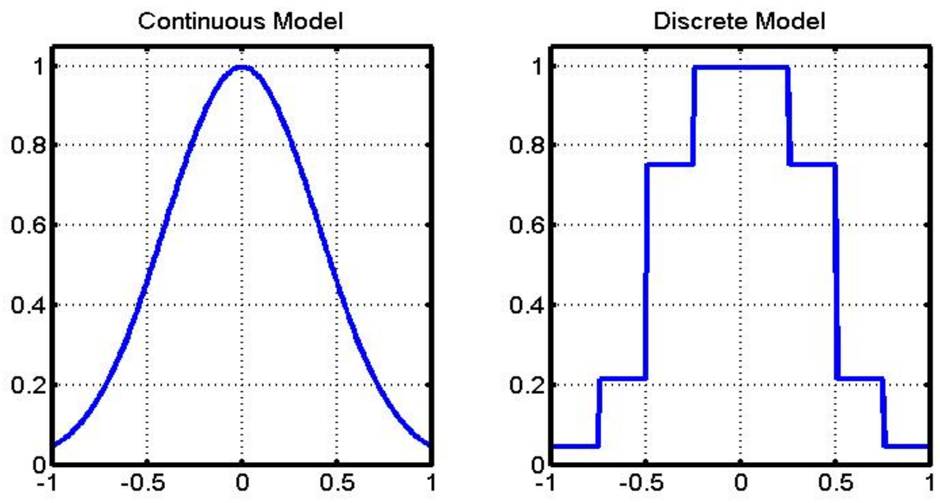


**Figure 1.** Continuous vs. Discrete Gaussian PDFs

In a system using discrete HMMs, each Gaussian is pre-calculated, quantized, and stored in memory such that when the system is in use, no actual calculation is required and the values necessary may simply be looked up. While this does provide a very efficient manner acquiring the result of the Gaussian evaluation it introduces a significant amount of error into the system through the quantization of the Gaussian PDF, and in systems where multiple similar Gaussians are necessary there results quickly become highly confusable. One way to remedy this problem is through reducing the size of the quantization step but this leads to large increases in the amount of memory required, which in most current systems may be an even larger problem than the one it attempts to solve. In semi-continuous models, a manageable set of parent-Gaussians is chosen to represent the entire desired set, and when a result is needed from the full set, the parent Gaussian is calculated and a weighting factor is applied to make it unique. Because this method actually calculates some Gaussians precisely, it is an inherently more accurate method than using a discrete model, but due to the fact that not all the Gaussians are directly calculated there is still a noticeable loss in precision versus using a continuous model. A Continuous HMM system actually stores the means and variances for each Gaussian in the knowledgebase and fully calculates each probability as it is needed. It has been shown that Continuous Density HMMs can increase the recognition accuracy upwards 6% when compared to Discrete or Semi-Continuous HMMs and in turn the number of likelihood calculations has increased from a few hundred to multiple thousands in most systems today [3, 12, 18, 22]. Coupled with this, it was found that greater the number of mixtures, where each additional mixture contains a unique cluster of Gaussians, that were used to model each state, the better the recognition accuracy [12]. Hence, it is not just the increase in dimensionality of the problem but also the increase in the number of quantities that need evaluation that has caused the computational requirements to drastically increase over the years. Since these calculations take a majority of the computational effort, between 58% and 70% of the total run-time, in-depth research has been focused on minimizing the number of these calculations without sacrificing accuracy [5].

Early ASR systems were limited by the performance of the available microprocessors and consequently relied on discrete HMMs in an effort to reduce the volume of Gaussian probability calculations necessary to evaluate the knowledgebase. Certain combinations of these limited Gaussians make up the states in an HMM and thus by computing the likelihoods over this limited set, recognition can be performed with significantly fewer calculations [6, 18]. An example of this method can be found in the SPHINX 2 system that used only 256 distributions as compared to SPHINX 3's 50k [1, 6, 16, 28]. Although discrete HMMs do provide a substantial amount of computational savings they also introduce large error rates when compared to continuous systems and restrict the system to smaller dictionary sizes, making them an undesirable solution in situations requiring high word accuracy rates on dictionaries over 5,000 words.

### 2.1.3 Gaussian Selection

Noting the need for continuous HMMs to achieve acceptable recognition rates, a significant amount of research has been done to expose other possible optimizations for the phoneme evaluation problem. One method, Gaussian Selection, has become one of the more widely accepted techniques in ASR. Originally proposed by Boccheiri [20], this method uses a process of Vector Quantization that utilizes, a set of coarse Gaussians to map multiple Gaussians in the full model to one of the vector quantized Gaussians [4, 20, 21]. For every input speech frame, the best matching coarse Gaussians are found and used as pointers to clusters of high-probability Gaussians that need to be evaluated fully. The remainder of the set can then either be approximated or completely ignored [16, 21, 23]. This method has been able to reduce the number of Gaussians that need to be calculated by a factor of up to 16 but can increase the word error rate by as much as 28% [23, 24]. Though this method does significantly reduce the computations necessary, it can also create issues involving branch misprediction and memory access bottlenecks, that can lead to pipeline stalls up to 52% on a Pentium III platform [3, 5, 6]. In the case of the SPHINX 3 system, the sub-vector quantization algorithm is used to reduce the number of Gaussians being analyzed during any single frame by a factor of 12 during the first pass of the search [20, 25]. In this initial pass a short list of Gaussians is returned, determining the candidates from the full model to evaluate on the second pass of the search as potential matches for the current input frame. While this method does provide improvement over a direct analysis of the full model, the system still takes ~1.8X real time on a 1.7GHz AMD Athlon processor, requiring 800MB/sec main memory bandwidth [5].

### 2.1.4 Feature Selection

Another school of thought involves Feature Selection, in which specific features are used based on their impact on the likelihood calculations [26]. This method can reduce the amount of computation between 33% and 66% depending on how the features are defined [11, 27]. A basic method of Feature Selection is to only evaluate the low order Cepstral coefficients (c1~c9) and their 1st and 2nd derivatives while simply ignoring any data generated in the higher-order coefficients (c10~c13). This reduced or 'First 24' evaluation can be very effective for small dictionary tasks but as the amount of variability increases in the system, the effects of the high-order coefficients become more evident.

To account for this problem, work done at the University of Washington and AT&T's Bell Labs used data driven approaches to try and determine which information is most relevant and prune the search accordingly [11, 27]. In the University of Washington's approach, the D-dimensions were broken into 3 groups according to their importance in the calculation. First, a summation of the primary group is performed and compared to a preset threshold. If the summation has not crossed the threshold for a particular component, then that component is still considered a valid candidate and must be further analyzed using the secondary and tertiary sets, checking after each pass to ensure the on-going validity of the component. This system was able to speed up the process by 40% while only increasing the word error rate by 0.2% for the 1400 word TIMIT database [27]. In contrast, Bell Labs performed a full statistical analysis to find the 24 most important dimensions, and then removed the others from any calculation ever. This method has shown a 0.3% increase in word error rate, but provides less overall speed up than the University of Washington's system [11]. Both of these systems highlight a very interesting model for reduction of computation with minimal reduction in accuracy but suffer limitations due to the derivation of a static sub-set for use in the likelihood evaluation.

### 2.1.5 Token Passing

The token passing algorithm as described by Young [66], provides a methodology for controlling the flow of data within an ASR system in an optimal fashion. Rather than trying to optimize the calculations being performed by merging Gaussians or intelligently selecting features to operate on, this algorithm minimizes the total work done by the system by monitoring the active data in the system and only performing the operations necessary to update the active data. By considering each HMM in the system as a unique token and then stringing tokens together to create words, a tree-style architecture is created. This tree is then used to determine which tokens to calculate in the next turn based on the locations of the active tokens in the present turn. In the simple case shown in Figure 2 the un-shaded node (i.e. token) is active indicating that the two nodes connected to the active node need to be calculated next while the others do not.

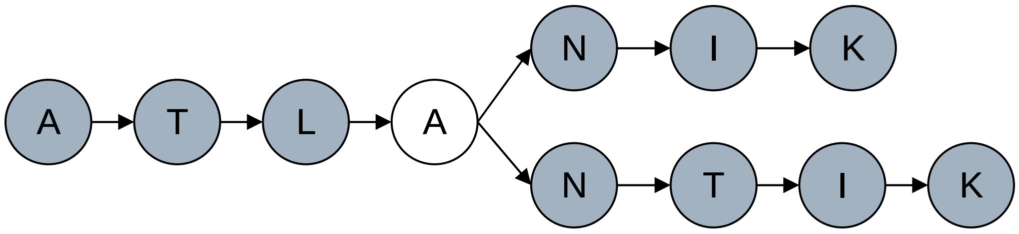


**Figure 2.** A Simple Tree Structure

Using the token-passing scheme can result in substantial savings noting that in the simple example above only two of the 11 possible nodes are evaluated yielding an 82% reduction in the work done by the system. In systems with large dictionaries where the tree-structure can exceed 1,000 nodes, the amount of savings can be quite large and allows for a more precise evaluation of the active tokens since fewer total tokens need to be evaluated at a given time. This model has been widely accepted and is currently used with great success in systems from Carnegie Mellon University, the University of California at Berkley, the Helsinki University of Technology, and Tsinghua University in Beijing [6, 17, 29, 30, 31].

## 2.2 COMERCIAL SOFTWARE SYSTEMS

As the personal computer began to evolve in the late 80's and early 90's many research organizations, both academic and corporate, started pursuing the idea of performing speech recognition on these platforms in attempts to make voice the primary means of data entry. These research efforts focused primarily on software-based algorithms that would be able to be run in conjunction with the computers operating system without causing severe memory access problems or other pipe-line stalls. The following section highlights a few of these research endeavors and provides a summary of their functionality and performance characteristics.

### 2.2.1 IBM's Via Voice

One of the more successful commercial speech recognition products on the market today is the Via Voice system from IBM, originally designed for the 1996 DARPA HUB-4 evaluations [8]. This system is based on a set of 5.7K HMM states comprised of 170K Gaussians, and was trained using 35 hours of data from the broadcast news (BN) corpus distributed by the LDC for the DARPA evaluations. The feature extraction unit for Via Voice uses a 60-dimensional input vector as opposed to the standard 39-dimensional vector presented in [15] to achieve higher recognition rates in languages other than English, specifically Mandarin Chinese [32]. Via Voice also uses Speaker Adaptive Training (SAT) and Cluster Adaptive Training (CAT) models to increase the speaker variability of the recognition engine. The SAT and CAT models define different classes of speakers in terms of their gender, age, or dialect and when a new speaker begins to use the system the are assigned to one of the pre-defined groups and the phoneme models are adjusted accordingly. By adapting the knowledge base dynamically even the earliest incantations of the Via Voice system were able to achieve recognition rates over 83% for both English and Mandarin Chinese speakers.

### 2.2.2 BBN’s BYBLOS

Another popular commercial system BBN’s BYBLOS, was also originally designed for the 1996 DARPA HUB-4 evaluations. This system only makes use of 4K three-state HMMs, but uses 64 Gaussians per state for a total of 768K Gaussians [34]. The earliest versions of BYBLOS had word error rates as high as 30% but as of the 1999 DARPA evaluations the error had been reduced to 15% [9, 33]. A sub-vector quantization algorithm is used during the first pass of the search to help minimize the amount of work done by the system and SAT models are also used to help adapt the system to a particular speaker. The original BYBLOS system ran on the standard 39-dimensional Cepstral vector, but later versions have been updated to accept multiple different sizes of input vectors allowing the end user to further customize the performance of the system. Recent versions of the system have also gotten away from the discrete densities used in the early manifestations and have become more speaker-independent with each new generation. In recent years, the BYBLOS recognition engine has been incorporated into a conference transcription and archive software suite called Rough’n’Ready with a dictionary of over 45K words [35]. This product can not only transcribe speech but can also archive the data it records by speaker, topic, or ‘named entity’. The named entity archive method looks for specific words in a conversation, generally the name of a product or business account, and will file the transcription with a header attached to it such that it may be queried from the archive at a later date. The ability of this system to perform topic spotting as well as speaker identification sets this system apart from most other ASR systems on the market, but these abilities come at the cost of a system that runs at 40 times real time.

### 2.2.3 SRI’s DECIPHER

The research team from SRI international has also been involved in developing speaker independent recognition software since the field first became a popular topic. The first cut of their DECIPHER system came out in 1989 as part of one of the earlier DARPA evaluations. This system uses the standard front-end (39 Mel-scaled Cepstra) and the widely accepted 3-state HMM models [7], but also incorporates a Gaussian Merging-Splitting Algorithm as described in [36]. Use of this algorithm allows for the models to be trained in a very simplistic manner, providing both a shorter training cycle and a method by which the models can be adapted while the system is in operation. This allows the system to constantly learn new patterns as opposed to being limited to a finite set of speaker-groups as in the SAT and CAT training methodologies. During the 1989 DARPA evaluation the DECIPHER system was able to achieve recognition rates over 75% on a database of 1300 words and by the time the 1997 evaluations took place, the system was able to achieve recognition rates over 80% on a dictionary of 48,000 words. This system also benefits from the use of information from 4 different knowledge sources to derive the final probability of a given word. By combining the results of the evaluation with and without cross-word articulation models, as well as the 5-gram language model and the total number of hypothesis for a given recognition, DECIPHER is able to consistently recognize easily confusable words, and words with strong cross articulation effects [7].

### 2.2.4 CMU's SPHINX

The SPHINX Large Vocabulary Continuous Speech Recognition (LVCSR) engine, designed by CMU, has for the past 15 years been one of the most successful research projects in the speech recognition industry. During the DARPA evaluations in 1989 the SPHINX-I system was able to recognize continuous speech from a 997 word vocabulary with between 70% - 95% accuracy using discrete HMMs [17]. By the 1992 DARPA, evaluations the SPHINX-II system had been developed using semi-continuous HMMs and applying an A* search algorithm, as described by [39], to the language model. A* is leading one-time computation algorithm used in path-finding research and has been shown to effectively minimize the portion of the search space that needs evaluation at a given time. By only focusing on area in the search space that are actually encountered as opposed to evaluating the entire space and then choosing the ideal path, the amount of calculation is reduced. While the performance of this algorithm does degrade as the path length increases is benefits are apparent noting that the SPHINX II system scored higher than any other in the evaluation with recognition rates consistently over 95% on a 5,000 word dictionary, setting the standard for future generations of software base speech recognition systems [28]. The SPHINX-II system was also one of the first systems to switch to the use of Mel-Frequency Cepstral Coefficients (MFCC) for the input feature stream as opposed to the previously accepted bi-linearly transformed LPC Cepstral coefficients.

For the 1996 evaluations, the SPHINX-III system was introduced and by using continuous HMMs as well as multi-pass search strategy, was able to achieve greater than 85% for a dictionary of over 51,000 words [37, 38]. Another major improvement in the SPHINX-III system was the ability to choose different HMM topologies to formulate the phoneme models. This ability allows for a system that can be modified according the specific recognition needs of the application without having to redesign the entire knowledgebase. For example, in situations where the users are talking very quickly, co-articulation effects can be very severe and using a three-state topology may not be ideal. In a situation like this where the individual phonetic units may not have a clear beginning, middle, and end, the topology of the HMM can be altered to allow for transitions straight from the beginning to the end states as shown in Figure 3, where the solid lines represent the standard transitions and the dotted lines show the alternative transitions.

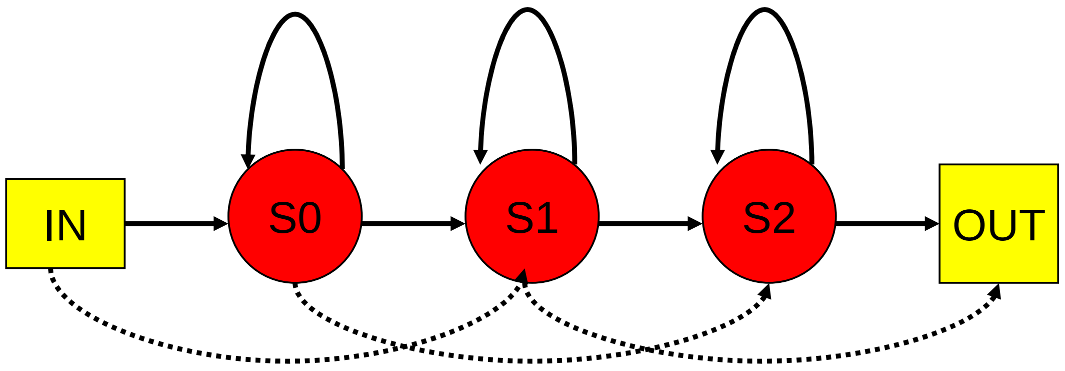


**Figure 3.** Sample HMM Topology

In addition to being one of the most consistently successful projects presented in the DARPA evaluations over the past 15 years, the SPHINX recognition engine has also been one of the more widely researched speech recognition engines with projects from research facilities such as the University of Utah and the University of Texas at Austin dedicated solely to the characterization and performance analysis of the SPHINX recognition engine as the golden model for the design of future ASR systems [5, 6]. Projects such as these as well as the continued research efforts of CMU have resulted multiple revisions and optimizations of the SPHINX-III platform all helping to create an industry benchmark system, able to perform large scale recognition tasks at near real-time speeds with very high levels of recognition accuracy.

The most recent revisions to the SPHINX project have been aimed at moving the code from a C++ design over to a JAVA based system. The SPHINX-4 project is a joint venture between Carnegie Mellon University, SUN Microsystems, and the Mitsubishi Electric Research Laboratory aimed at developing a speech recognition toolkit from which the end user could build their own custom speech recognition system based on the SPHINX recognition engine [40]. SPHINX-4 also benefits from a redesigned decoder architecture, the inclusion of a stand-alone graph construction module, and the application of the Bushderby classification algorithm to the language model. The graph construction module is responsible for creating and managing the trellis created as the tree-structure is evaluated over time. This module controls the transitions out of one tree and into another, and also the removal of branches as their probabilities become undesirable. This structure has been static in previous versions of the SPHINX system so by creation of the graph construction module, a new method for dynamic creation is introduced to the recognition engine, furthering its abilities to adapt to new scenarios and different applications. The Bushderby classification algorithm, described in detail in [41], is a direct extension of the Viterbi algorithm and by incorporating it into the langue model, the system gains the ability to classify mismatched data and adapt the system accordingly. The SPHINX-4 system is not currently finished and resultantly no quantification of the word error rates were available at the time of this paper but based on the success of the previous generations of the SPHINX recognition engine, SPHINX-4 promises to provide a user-friendly, highly customizable speech recognition platform capable of large-vocabulary recognition with impressive accuracy.

## 2.3 CUSTOM HARDWARE ARCHITECTURES

In recent years Application Specific Integrated Circuit (ASIC) architectures and embedded system design have become increasingly popular. As these systems become larger and more widely accepted the possibility to implement speech recognition on such devices has become an appealing alternative to the software based solutions currently on the market. Both the governmental and private sectors have spent significant amounts of money over the past decade attempting to determine the feasibility of a single-chip speech recognition engine, all the while fighting a battle between the amount of memory required to perform accurate recognition and the amount of logic required to fabricate such intelligent systems. The consistently improving computational ability of Field Programmable Gate Arrays (FPGAs) has allowed for a number of research institutions to experiment with the potential of these devices to be configured for speech recognition applications.

### 2.3.1 Sensory Inc.

Sensory Inc. offers a single chip speech recognition microcontrollers based on a simple 8-bit microcontroller, the RSC-4128 [42, 43]. These systems have both onboard memory for storage of speaker-dependent models and off-chip storage for speaker independent models. In speaker independent mode the RSC-4128 is capable of recognizing a set of up to 20 words, while in speaker dependent mode the set size can be increased to 100. The number of possible sets of words in either mode is limited only by the size of the off-chip memory, but if no external memory is available then it is only possible to recognize a set of 10 speaker dependent words. The RSC device is also capable of using either HMM-based model or Neural Network Models depending on the desired user configuration. These devices also have word spotting and continuous listening capabilities, wherein the system will listen for one of a set of key-words and either enables a device based on that word or begin the recognition process from that word forward. While these devices do provide a good solution for small vocabulary speech recognition in real-time, they are limited in terms of their applications since the majority of speech recognition tasks require significantly more than 20 words. Unfortunately, word error rates are not widely published for these devices, but it is not expected that an 8-bit microcontroller is capable of performing with similar quality to a desktop system. This intuition is confirmed by the observation that Sensory's primary audience for their chips are the manufacturers of children's toys whose devices would not be hindered by mediocre recognition rates.

### 2.3.2 University of Birmingham

The research group from the University of Birmingham in the United Kingdom has also produced promising results in the field of hardware-based speech recognition engines with a system they implemented on a Xilinx Virtex XCV1000 FPGA. While this system is only capable of recognition rates around 56% for the 500-word TIMIT knowledgebase it only occupies 45% of the entire device and is capable of performing at over 75X faster than real-time [46, 47]. A major reason for the reduced accuracy of this system lies in the use of mono-phones as the primary phonetic unit as opposed to bi- or tri-phones as are used in other commercially used systems [7, 37, 44, 46]. This choice was made in an attempt to prove the performance of the architecture while minimizing the amount of external memory required for basic recognition and serves well to highlight the impact that co-articulation effects have on continuous speech recognition. One of the primary goals of this project was to show the effectiveness of using an off-the-shelf FPGA as a dedicated speech co-processor, capable of perform the recognition at speeds much greater than real-time allowing for multiple input streams to be analyzed at once. The second generation of this design took advantage of the less than real time abilities of the preliminary system and was effectively able to process three speech file simultaneously. This new version of the architecture was released in both a mono-phone and a bi / tri-phone version, capable of speeds of 250X less than real-time and 13X less than real-time respectively [48]. Although this work provides a compact and efficient architecture for processing data in hardware based ASR systems, there are still obvious issues with respect the completeness of the knowledgebase and the overall scalability of the design that keep this research from providing a complete solution to the hardware speech recognition problem.

### 2.3.3 University of California at Berkley

In 2003 the University of California at Berkley completed work on a custom ASIC design for a speaker independent recognition device capable of recognition rates over 80% for dictionaries up to 50 words [44, 45]. This project was aimed at creating a system capable of small vocabulary (< 100 words) focusing on low power considerations for handheld devices. Because of their small dictionary objective, the design team chose to implement a system based on a large array of identical processing elements connected to one another via aggregator units. The recognition system is based off of the traditional 3-state HMM topology for the phoneme models, and also uses a vector quantization algorithm to reduce the complexity of the Gaussian probability evaluations. As described in section 2.1 of this paper, it is the large amount of data that needs to be processed not the amount of work that needs done on the data that becomes the limiting factor. So by creating a system containing multiple identical blocks in parallel, the throughput can be greatly increased and the cycle count lowered. Using this ideology, the research group determined the number of processing elements, memories, and aggregators that could fit on one-chip and worked backwards to determine the total number of nodes possible in the tree and subsequently the number of words allowable in the dictionary. By intelligently analyzing the language model for the given dictionary it was determined that the majority of the HMMs are only used in certain branches of the tree, enabling the HMMs in the knowledgebase to be clustered together into groups of highly associated nodes. By allowing each processing element to act only on HMMs within a given cluster, the routing of the data from one element to the next is greatly simplified and can be directly extracted from the tree-structure derived for a given dictionary. While this does reduce the complexity of the data path, the scheduling of the data on the busses, the intercommunication between aggregator nodes and the global control logic remain fairly complex alluding to potential problems as the design is scaled upwards. To help with this scheduling / synchronization issue, the computations within in a cycle are divided into two phases; first the aggregators and all nodes with no new inputs complete their operation, and secondly the nodes receiving data from other nodes or the aggregator are allowed to execute. To help further stream-line the operation of the system the token passing algorithm was employed to manage the communication and transfer of information between the aggregator nodes. Figure 4 shows the architecture of the designed decoder.

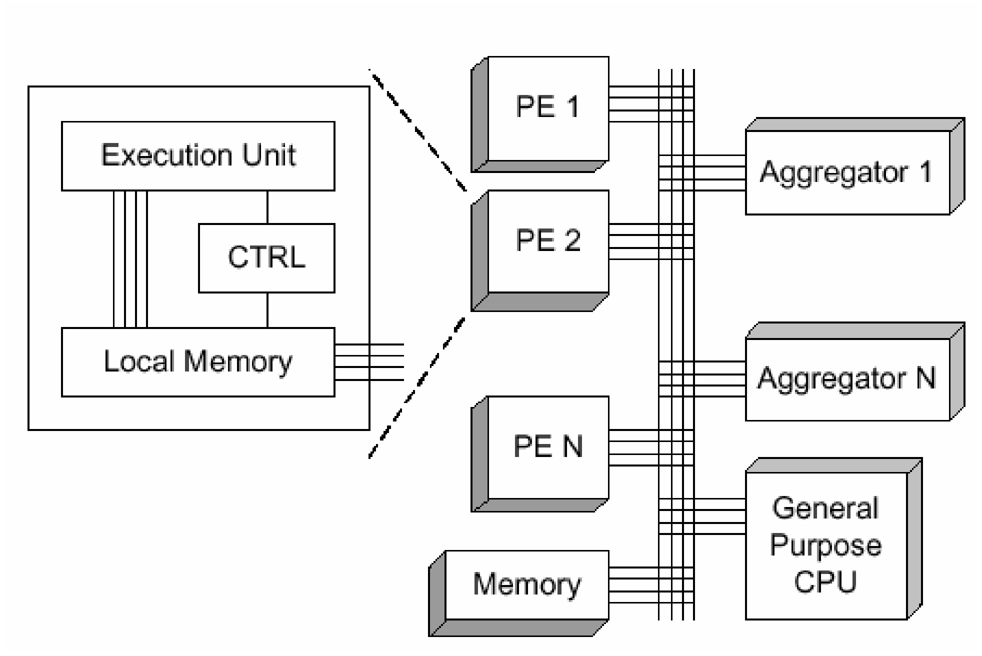


**Figure 4.** Diagram of Decoder Architecture

For achieving a low power system the design team utilized gated clocks as well as a single-cycle operation flow. This was possible since the system requires relatively slow clock speeds (< 5MHz) to run at real-time. Additionally, voltage scaling was used to further reduce the amount of power used by the system but this can only be applied to systems with moderate supply voltages, since the delay added by this measure become prohibitive as the supply voltage decreases. The Berkley system was able to achieve word accuracy rates as high as 80% for dictionaries under 50 words but was only able to achieve sentence accuracy around 25% [31]. This system provides a highly effective architecture for low power speaker independent recognition, but does not seem to offer the scalability required to adapt the architecture for large dictionary tasks.

## 2.4 VECTOR PROCESSING ARCHITECTURES

Vector Processors have been a topic of major interest in recent years, with a number of major processor manufacturers beginning research on new vector processing architectures. From the new generation of VLIW and SIMD processors to fully-custom architectures such as the Cell processor from Sony, Toshiba, and IBM, the potential gains from parallel processing have become evident and the number of applications benefiting from this technology is constantly rising. Among the benefactor technologies, speech processing appears to have large potential gains from the use of parallel architectures, noting that the crux of the problem lies in the inability to exploit the inherent parallelism due to the limitations of the current generation of processors.

### 2.4.1 VLIW Processors

Very Long Instruction Word (VLIW) processors provide a unique architecture for the development of speech recognition systems due to their ability to perform multiple instructions in a single cycle. Observing the latent parallelism in the speech recognition process, the ability to perform loop unrolling becomes obvious, allowing for a direct implementation in a VLIW architecture. As described in [49], the higher the number of instructions able to be implemented in parallel the greater the advantage of VLIW systems over other parallel architectures such as superscalar systems. Further, VLIW systems benefit from the fact that the scheduling of the instructions is performed by the compiler, as opposed to necessitating special hardware to perform this scheduling during run-time as in the superscalar architectures, leading to the need for less overall hardware in the design [50]. In a speech processing system the number of elements that can be operated on in parallel is quite large, hence the advantages of a VLIW become very evident, however this paradigm does saturate at some point due to the fact that the size of the device necessary to perform extremely large sets of data in parallel becomes prohibitively large. Another limiting factor for VLIW implementation lies in the need to perform both floating point and integer operations in large parallel groups. Due to the fact that VLIW systems require separate function units for each issue in the instruction word most traditional implementations reserve certain issues for integer operations and others for floating point operations. For a speech processing system it is desired to have each issue be capable of executing both floating point and integer operations at different times, but this would require the inclusion of both integer and floating point ALUs, as well as a multiplexor to select between them. This leads to an architecture with double the number of ALUs of a traditional VLIW processor. For example if we derived an 8-wide VLIW with 4 memory read issues, 2 floating point issues, and 2 integer issues, during run-time half of the issues (2 memory and either both floating point or both integer) would be idle resulting in a significant amount of unused resources. This reduction in ability to fully utilize the device would lead to a system that functions more like a 4-wide VLIW than an 8-wide, which is an extremely undesirable result when the name of the game is massive parallelism. Even in heterogeneous systems where all function units are capable of all instructions, the necessity for a shared register file for the processor creates its own limitations. The shared register file does not scale well, resulting in the limitations that while VLIW processor perform well for widths less than 16, as the size of the processor increases the potential benefits start to decrease. In the research presented in [51] by R. Hoare, *et al.* at the University of Pittsburgh, a modified VLIW system is presented that incorporates dedicated hardware functions that can be executed along side the VLIW instructions to assist in the speed-up of algorithms that are not able to be efficiently implemented in sequential code. This research was able to speed up the GSM speech coding algorithm by 7X and shows a very interesting method for accelerating the operations of traditional VLIW processors. While the notion of dedicated hardware functions working in parallel with VLIW processors solves some of the problems found when trying to implement speech recognition on a VLIW platform. it does not provide a complete solution to the problem.

### 2.4.2 SIMD Processors

Another popular parallel architecture, Single Instruction Multiple Data (SIMD), takes a different approach to increasing Instruction Level Parallelism (ILP) from the VLIW architecture and gives a another perspective on how to implement speech recognition systems. In SIMD architectures, a single instruction is applied to all elements of an input vector, creating a system that can perform the same action on large amounts of data in a single cycle [52]. These systems can perform intra- as well as inter- element operations and also support saturation arithmetic, which is commonly used in video and signal processing algorithms. Some SIMD processors, such as the AltiVec processor are also capable of using a filter vector to rearrange the elements of the input vectors. Figure 5 illustrates the functionality of a SIMD processor with and with out the inclusion of the filter vector.

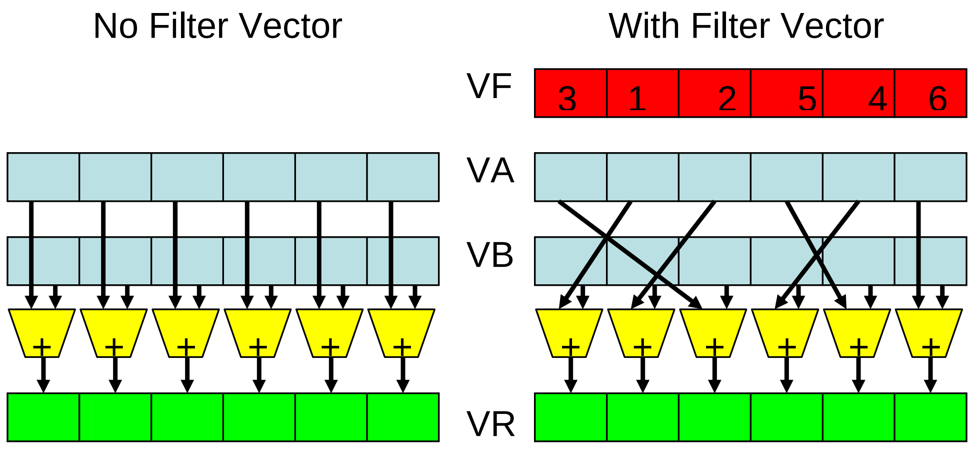


**Figure 5.** SIMD Processing With and Without Filter Vector

SIMD architectures fit the speech processing paradigm quite well since the majority of the parallelism observed in a speech recognition system occurs in the form of identical processing of large numbers of elements. Since this architecture only requires the construction of a single integer and a floating point ALU capable of vector ops, the necessary hardware need not share registers as in the case of the VLIW. While SIMD processors seem to provide some additional benefits over VLIW systems, they are still restricted in the number of parallel elements that can be processed before the size of the hardware become a limiting factor.

A purely SIMD processor would also encounter some difficulties handling the word modeling process in an ASR system. This is largely due to the complexity of the data fetch that would need to occur in order to fill the input vectors with the proper amount of data to take full advantage of the SIMD processors capabilities. During the word modeling phase a tree structure, described briefly in Section 2.1.5, need to be evaluated resulting in the traversal of a large number of link-list style elements. Assuming that all of these lists are contained in unified memory bank, each active token in the system must be read and its corresponding link-list starting address decoded. Then, the addresses must be applied to the memory bank sequentially and the elements in the link-lists fed out of the memory and into the SIMD processors input vector buffer. The amount of overhead required to execute this process would render the full capabilities of the SIMD processor inactive during the time that the memory fetch was occurring, resulting in significant processor stalls. From this observation, it would seem that while the SIMD processor does provide a reasonable platform for the development of speech recognition systems, much like the VLIW processor discussed in Section 2.4.1 it only provides a partial solution to the entire problem, leading to the pursuit of other more ideal architectures.

### 2.4.3 Sony, Toshiba, IBM Cell Processor

During the first quarter of 2005, a joint research venture between Sony, Toshiba, & IBM filed the patent documents on a revolutionary processor architecture code-named Cell. The Cell processor takes a multi-core computing approach to System-on-Chip design, resulting in a network of eight Attached Processing Units (APUs) connected to a master Processing Unit (PU) via the Element Interface Bus (EIB) [53]. Figure 6 illustrates the overall architecture of the Cell processor.

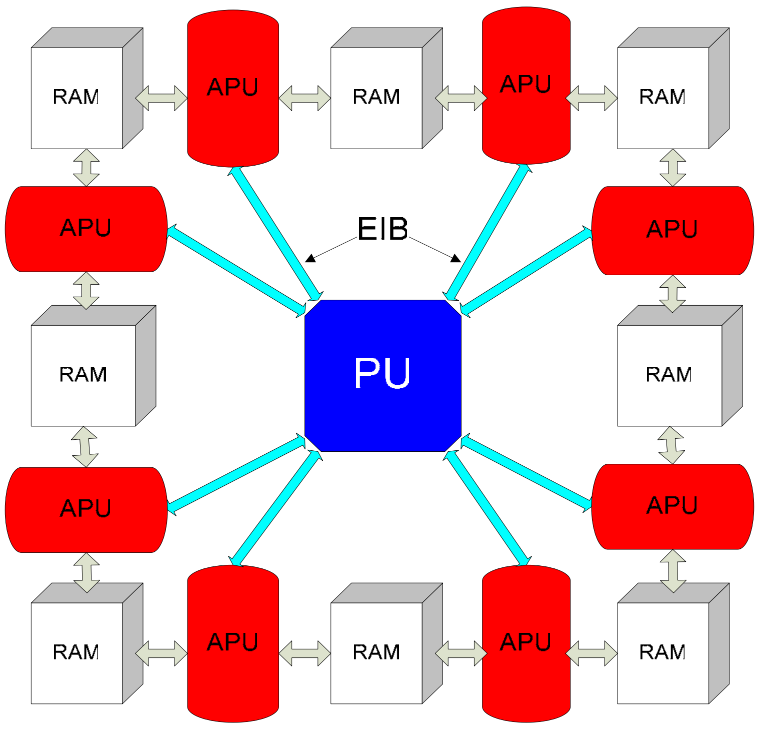


**Figure 6.** Block Diagram of Cell Architecture

Each Cell processor is capable of 250 GFLOPS and has a 6.4 Gigabit/sec I/O bus to allow for the creation of ad-hoc networks of Cell processors to perform massively distributed processing operations. The master PU is based on the 4.6GHz 64-bit Power PC architecture with each of the APUs operating as a 4x32 (128 bit) SIMD processor with four integer and four floating point units, each capable of 32 billion operations per second. The eight APUs are set-up in a ring with shared memory banks between each APU as well as a common external RAM location attached to the EIB for use by all APUs. One of the truly unique and powerful capabilities of the Cell is the ability to run in stream mode. In stream mode, one of the APUs performs an action on some data and then puts it to a specified location in its shared RAM bank. The data is then immediately read out of the RAM bank by the other APU connected to the shared memory, processed further, and then written to the next shared memory. This process creates a constant flow of data between each APU allowing for highly pipelined processing to be executed, with a level of efficiency order of magnitude larger than any commercially available processor today. Additionally, the Cell is capable of accessing the external memory at well over 10 Gigabytes/sec, which it highly advantageous in applications that require both extremely large amounts of memory and constant access to the memory, such as speech and other signal processing operations. Use of Cell processors for speech processing applications appears to solve both of the major problems encountered when designing ASR systems; the need for highly parallel processing and the need for extremely high throughput. Further this architecture has the benefits of both SIMD processors and VLIW processors; in that each of the APUs is in fact a SIMD unit, while having eight of them on one chip allows for the multi-functionality of a VLIW processor. Unfortunately it may be years before the research community gets access to the Cell chip on a level that will allow for the development of custom embedded systems based on this revolutionary processor architecture.

### 2.4.4 Stretch Inc. S5000 Chip Family

The S5000 family of processors from Stretch Inc. also provide an interesting potential solution for speech processing with in a hardware device. Released in 2004, the S5000 family is the first family of processors to embed reprogrammable logic inside the processor core, creating a multi-purpose RISC style processor capable of being tailored to each user's specific needs. At the core of the device is a 300 MHz 32-bit Xtensa processor from Tensilica which is capable of performing any of the tasks assigned to a traditional microprocessor. The innovation in this technology comes in the form of Stretch Inc.'s Instruction Set Extension Fabric (ISEF), which functions as a reconfigurable logic device capable of being programmed to tackle computationally difficult tasks in a custom hardware environment [54]. Through use of Stretch's C++ code profiler the "hot-spots" in a C/C++ code are found and given to the compiler such that the ISEF can be programmed to perform the most computationally intensive software operations in a single hardware instruction. The potential performance gains from this architecture are quite large and this point was made even more obvious during an independent certification by EEMBC where the S5000 received a Telemark score of 877, substantially higher than any other device available on the market today [55]. A block diagram of the S5000 processor engine can be seen in Figure 7.

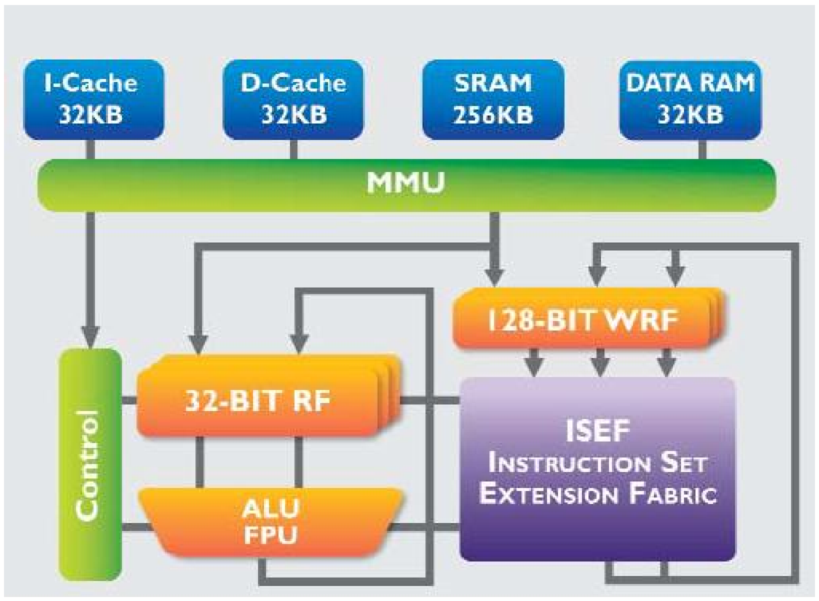


**Figure 7.** Stretch Inc. S5000 Processor Engine

As discussed in Section 2.3 there are a few calculations that dominate the workload in a speech processing system. Existing hardware implementations have shown the benefit of using dedicated hardware to solve this problem. These systems however, all require the inclusion of another chip in addition to a microprocessor when attempting to creating speech recognition devices. This can be very undesirable in situations where size are power consumption are limiting factors. Use of the S5000 processors solves this problem by having the ISEF and the processor on a single chip and allows for the development of more complex software algorithms for processing speech. Complex search algorithms and Language Model constraints are well suited for software implementation, while the phonetic modeling and feature extraction operations have been shown to be well suited for hardware implementation. Through use of the S5000 processor both of these criteria can be met with a single chip. As this technology becomes more established there promises to be significant amounts experimentation with the S5000 to expose the true abilities of the processor to tackle speech recognition and other signal processing tasks.

# 3.0 SYSTEM ARCHITECUTRE

Based on the SPHINX 3 system described in Section 2.2.4, this research first maps the major algorithms of SPHINX into highly vectorized MATLAB code which is then used as the template for a custom hardware architecture for performing ASR. Each of three main components of the SPHINX system, the Acoustic Modeler (AM), the Phoneme Evaluator (PE), and the Word Modeler (WM) were designed separately so as to be able to tailor each design to meet the specific challenges of each portion of the system. Their development in both MATLAB and VHDL is presented in the following chapters along with a performance analysis of the resultant hardware implementations. This chapter begins with an overview of the system level for our project as well as some preliminary analysis and then moves on to describe the system-level hardware that was created to control the architecture. This chapter then concludes with a brief description of the development and test environments used for this project.

## 3.1 SYSTEM OVERVIEW

This project, known as Speech Silicon [62], has been working for the past two years to characterize and model state-of-the-art speech recognition technology to help forward the development of hardware-based solutions to unique challenges facing the speech recognition industry. Preliminary work on this topic has been discussed in [59, 60, 61] and these papers will be referenced throughout this document. Figure 8 shows a block diagram for the interaction between the major components of a traditional software system, with inputs from a DSP being shown on the left of the diagram as the Feature Extractor block.

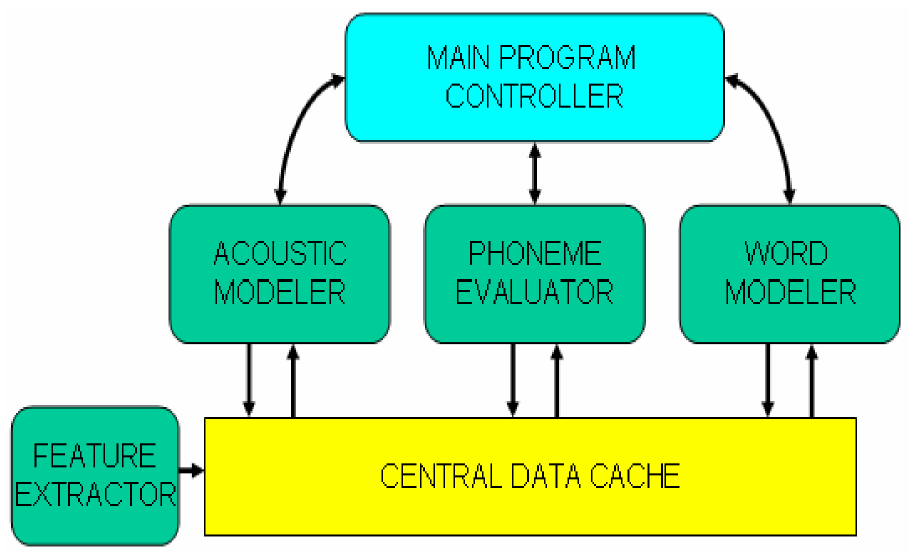


**Figure 8.** Block Diagram of Software-Based ASR System

Feature Extraction is the process of transforming the incoming speech in to its frequency content via the Fast Fourier Transform, and the subsequent generation of Mel-Scaled Cepstral Coefficients through Mel-Frequency Warping and the Discrete Cosine Transform. These operations can be performed on most currently available DSP devices with high precision and in real-time. Therefore, FE will not be considered within the scope of this paper.

Acoustic Modeling is responsible for evaluating the inputs received from the DSP unit with respect to a database of known Gaussian probabilities and for producing a normalized set of scores (i.e. senones) that represent the individual sound units in the database. These sound units represent sub-phonetic components of speech and are traditionally used to model the beginning, middle, and end, of a particular phonetic unit. Each of the senones in a database is comprised of a mixture of multi-variant Gaussian Probability Density Functions each requiring a large number of complex operations. It has been shown that this phase of the speech recognition process is the most computationally intensive taking up to 95% of the execution time [3, 26], and therefore requires a pipeline with very high bandwidth to accommodate the calculations.

The Phoneme Evaluator (PE) associates groups of senones into to HMMs representing the phonetic units, phonemes, allowable in the systems dictionary. The basic calculations necessary to process a single HMM are not extremely complex and can be broken down into a simple Add-Compare-Add pipeline, and are described in detail in Chapter 5. The difficulty in this phase is in managing the data effectively, so as to minimize unnecessary calculations. When the system is operational, not all of the phonemes in the dictionary are active all the time. It is the PE that is responsible for the management of the active/inactive lists for each frame. By creating a pipeline dedicated to calculating HMMs and combining it with a second piece of logic that acts as a pruner for the active list, a two-step approach was conceived for implementing PE allowing for the efficiency of the block to be maximized.

The Word Modeler (WM) uses a tree-based structure to string phonemes together into words based on the sequences defined in the system dictionary. This block serves as the linker between the phonemes in a word as well as the words in a phrase. When the transition from one word to another is detected, a variable penalty is applied to the exiting words score depending on what word it attempts to enter next. In this way, basic syntax rules can be implemented in addition to pruning based on predefined threshold for all words. WM is also responsible for resetting nodes in the tree when they become inactive. While some nodes in the tree will propagate during on a given cycle, other will achieve scores beyond an acceptable threshold and need to be removed to avoid unnecessary calculation. The pruning stage of PE will pass two lists to the WM, one for active tokens and the other for newly inactive tokens. Much like PE, WM takes a two stage approach, first resetting the inactive tokens and then processing the active tokens. By doing the operations in this order we ensure that while processing the active tokens, all possible successor tokens are available if and when they are needed.

When considering such systems for implementation on embedded platforms the specific constraints imposed by each of these components must be considered. Additionally, the data-dependencies between all components must be considered to ensure that each component has the data it requires as soon as it needs it. To make matters worse, the overall size of the design and its power consumption must also be factored into the design if the resultant technology is to be applicable to small hand-held devices. The most effective manner for accommodating these constraints was determined to be the derivation of three separate cells, one for each of the major components considered, with shared access RAMs creating the boundaries between cells. To minimize the control logic and communication between cells, a token-passing scheme was implemented using FIFOs to buffer the active tokens across cell boundaries. A block diagram of the component interaction within the system is shown in Figure 9.

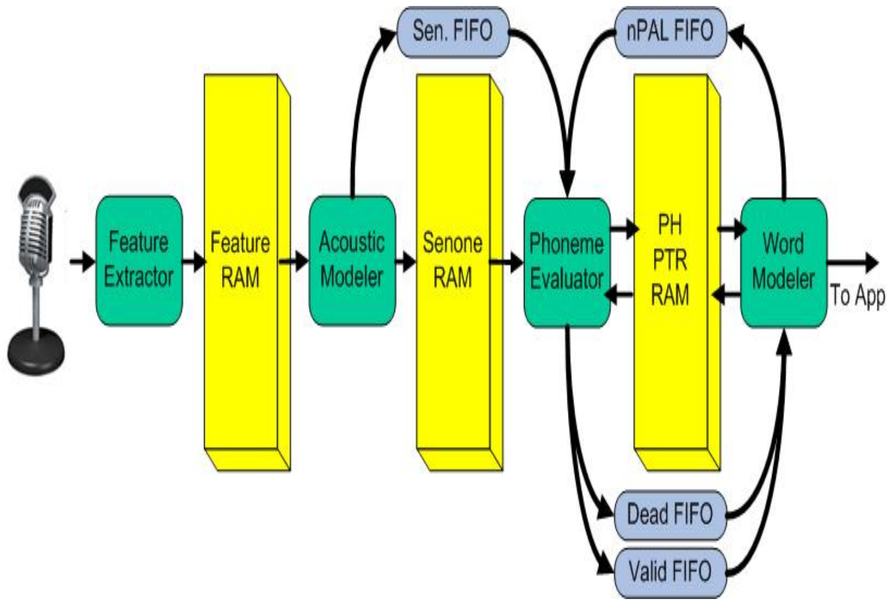


**Figure 9.** Block Diagram of the Speech Silicon Hardware-Base ASR System

By constructing the system in this fashion and keeping the databases necessary for the recognition separate from the core components, this system is not bound to a single dictionary with a specific set of senones and phonemes. These databases can, in fact, be reprogrammed with multiple dictionaries in multiple languages, and then given to the system for use with no changes to the architecture. This flexibility also allows for the use of more or less complex models in any of the components allowing for a wide range of input models to be used, and further aiding in the customizability of the system.

The Active Senone RAM is simply a large dual-ported RAM with the Acoustic Modeler tied to the write port of the device and the Phoneme Evaluator tied to the read port. By creating the RAM in this manner there is no need to any multiplexing of the input address to the RAM and it can be allowed to operate without supervision from the global controller. The Active

Phone RAM is noticeably more difficult and contains a total of 4 access ports. Also known as the Central Data structure, both the Phoneme Evaluator and the Word Modeler have dedicated read and write ports to the device allowing for either device to simultaneously read and write from the RAM. This RAM contains a large number of fields used for storing both information about the current state of a specific HMM and information about the location of the HMM in the word-tree. The specific details of the Active Phone RAM are given in Section 3.3.2.

## 3.2 PRELIMINARY ANALYSIS

During the conceptual phase of the project, one major requirement was set: the system is able to process all data in real-time. It was observed that Speech Recognition for a 64k word task was 1.8 times slower than real-time on a 1.7 GHz AMD Athlon processor [5, 63]. Additionally, the models for such a task are 3 times larger than the models used for the 1,000 word Command & Control task our project was focused on. Therefore, extending this linearly in terms of the number of compute cycles required, it can be said that a 1,000 word task would take 0.6 times real-time to process at 1.7 GHz. While this proves the need for custom hardware it also asks the question of how many cycles would a full custom architecture require?

In modern speech processing incoming speech is sampled every 10ms leading directly to the notion that any system able to perform in real-time must be able to execute its entire work-load within a 10ms window. For a system running at 100MHz this translates to a 1 million cycle budget for completing all operations. To find our actual budget a series of experiments were conducted on open-source Sphinx models [64, 65] to observe the cycle counts for different recognition tasks. Table 1 summarizes the results of these tests for three different sized tasks: digit recognition [TI Digits], command & control [RM1], and continuous speech [HUB-4].

Table 1 shows the number of cycles required for the computation of all Gaussians for different tasks, assuming a fully pipelined design. It can be seen that assuming one-cycle latency for memory accesses, the RM1 task would require 620k compute cycles just for the Acoustic Modeling calculation while HUB4 would require 2M cycles. Knowing that we need to process all of the data within a 10ms window we observe that the minimum operating speeds for systems performing these tasks would be approximately 62 MHz and 200 MHz respectively.

**Table 1.** Number of Compute Cycles for 3 Different Speech Corpuses

| **Speech Corpus** | **# of Words** | **# of Gaussians** | **# of Evaluations per Frame** |
|---|---|---|---|
| **TI Digits** | 12 | 4,816 | 192,600 |
| **RM1** | 1,000 | 15,480 | 619,200 |
| **HUB-4** | 64,000 | 49,152 | 1,966,080 |

Since the computation of Gaussian probabilities in AM constitutes the majority of the processing time, keeping some cushion for computations in the PHN and WRD blocks, it was determined that 1 million cycles would be sufficient to process data for every frame for RM1 task. Therefore an operating speed of 100 MHz was set for our design. While a completely pipelined design is possible in the case of AM and PHN, computations in the WRD Block don't share such luxury. This is a direct result of the variable branching characteristic of the word tree structure. Further, the number of cycles required by the PHN and WRD Blocks is completely dependent on the number of phones/words active at any given instant. Therefore, an analysis of the software was performed to obtain the maximum number of phones active at any given time instant. It was observed from Sphinx 3.3 for a RM1 dictionary, a maximum of 4000 phones were simultaneously active. Based on this analysis a worst case estimate of the number of compute cycles necessary for the complete calculation, it was determined that a 1 million cycle budget was more than sufficient to complete the workload. Having determined the timing budget for the project the next task was to begin design the custom hardware cells to execute each phase of ASR as well as the high-level control logic that connects all of the components. The remainder of this research focuses on design of these components.

## 3.3 SYSTEM LEVEL HARDWARE

Aside from the three major components described in this thesis there are two other central components needed to complete the development of a complete speech recognition system. Both the central data structure, the PH RAM, mentioned in chapters 4 and 5, as well as the control logic needed to combine the three separate components into a cohesive system are critical in understanding how the systems functions on its highest levels. These structures were designed over the course of our research and implemented in various forms before coming to a standard system interface that would define the top-level structure of our designs. The design for the top-level control logic presented in Section 3.3.1 and the data structure presented in Section 3.3.2 both represent VHDL written for the ECE 2121 course during the spring term of 2005. As with the hardware design for the WM stage, these designs are presented for completeness rather than for their unique contributions to this thesis.

### 3.3.1 System Controller

The system controller designed to manage the hardware system is largely responsible for managing potential error responses from each block and ensuring the system timing through a series of handshaking signal associated with each of the primary hardware cells. During regular operations the controller will wait in the idle state until some external stimulus excites the system. Once this has happen the controller will pass a start signal to the AM block which will start processing the data bring received from the external DSP source. Once all of the senones have been written to the RAM AM will signal to the control that it has finished and a start signal will correspondingly be sent to the PE block. Likewise, when the PE block is done a series of hand-shakes occur to turn on the WM block. This block will run either until all data has been processed or until the next new frame of data is ready from the DSP processor. By forcing the WM to exit if too much time is spent searching the tree we ensure both static system timing as well as creating a known break point at which the system can be forced back to a known state. Aside from managing the handshaking between the different stages of the design, the controller is also responsible for monitoring the status of each of the cues in the system to ensure that no overflow situations are created where a token could possible be dropped from system entirely. In the event that a cue does become full and cannot be emptied before a new token is placed on it, the cue will send an error flag to the controller that will put the system into an error state requiring the user to restart the operation there were in the process of executing. A diagram of the system controller FSM is shown in Figure 10.

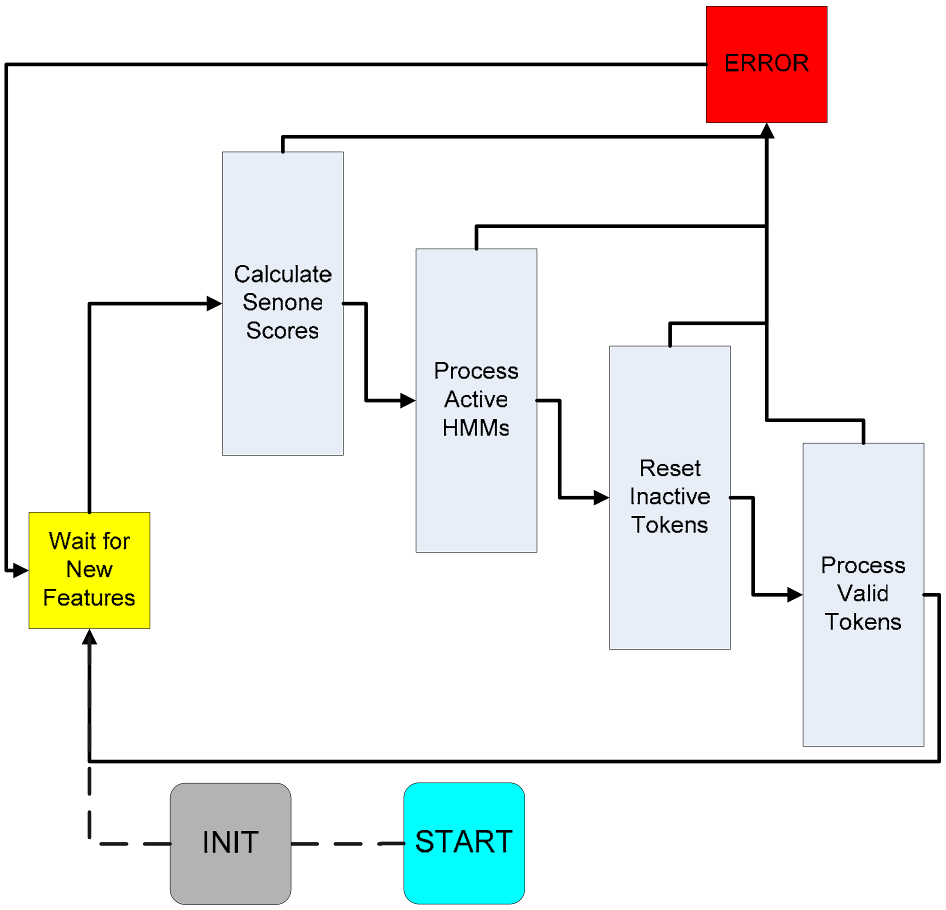


**Figure 10.** FSM for System Controller

## 3.3.2 Central Data Structure

The central data structure for architecture is responsible for storing all of the data about the active nodes in the word-tree and for managing data accesses from both the PE and the WM blocks. This data structure is shown as the *Active Phone RAM* in Figure 8 and also commonly called the PH_PTR RAM or simply the PH RAM. In software this translates to a large database where each entry contains multiple fields. For the hardware implementation this means the creation of a dual-ported RAM array with very long word length. Specifically, the PH RAM contains one entry for each node in the WM search space and each of these entries contains eleven different fields that must be referenced by either the PE, the WM or both. Of the eleven fields in each entry only seven of them are variable fields while the other 4 provide static information about the particular nodes location in the search space. Figure 11 shows the eleven fields used for each entry in the database with the number of bits used to represent each field shown below its name.

| RAM | | | | | | | ROM | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Active | In_Scr | H0_Scr | H1_Scr | H2_Scr | Out_Scr | Max_Scr | LM_Scr /Diff | SSId | WID | CS/WEnd |
| 1 | 32 | 32 | 32 | 32 | 32 | 32 | 32 | 13 | 11 | 1 |

**Figure 11.** Organization of Fields in Database

The first field in the database (Active) is a single bit that is set every time the WM launches a given token and is reset every time that token ends up the token deactivation cue. The next six fields (In_Scr, H0_Scr, H1_Scr, H2_Scr, Out_Scr, Max_Scr) are all 32 bits wide and are used to store the information about the current HMM state probabilities calculated by the PE block. These fields and their values are discussed in more detail in Chapter 5. The remaining four fields

(LM_Scr/DIFF, SSId, WID, CS/Wend) represent the static information about the tokens location in the search space and are described in more detail in Chapter 6.

## 3.4 TEST/DEVELOPMENT ENVIRONMENT

During the term of the project many different stages of design were performed, requiring the use of multiple development and test environments to fully characterize the work performed. Our original model was the SPHINX 3 open-source speech recognition engine from CMU, running on a 3.2GHz Pentium 4 processor with 1.5GB of RAM installed. The original code was modified with numerous '*printf()*' statements to allow for more visibility of the algorithm in action. From this base model vectorized MATLAB code was designed to emulate the process and create a stable, compact design for illustrating the latent parallelism of the SPHINX algorithms. This design was performed in MATLAB 7 release 14 making use of SIMULINK as well as many of the toolkits included in the MATLAB suite. Initial hardware development was performed using FPGAdvantage 6.1 along with Modelsim 6.1 for simulation and Precision Synthesis for logic synthesis. The work presented in this thesis for the system level hardware, Sections 3.3.1 and 3.3.2, and the WM hardware development, Section 6.4, was completed in this environment and then put through place-and-route using ISE 7.1. The more computationally intensive portions of the design, the AM and the PE, were developed in a separate environment to ensure maximum performance and take advantage of additional tool suites that had become available. Specifically, these components were designed in Xilinx ISE 7.1 through the derivation of pure VHDL and were then synthesized using the Synplicity Synplify 8.2 synthesis tool. The final place-and-route for these designs was complete in ISE using hand-generate PACE constraint files to guide the PAR tool.

For verification of our implementations the SPHINX 3 baseline model was modified to output text files at the barriers between the individual stages of the algorithm. Then as the corresponding MATLAB blocks were completed they could be fed the same inputs as the SPHINX system and have their outputs checked against the baseline text files. Once the MATLAB code had been verified against SPHINX it was then used to generate the test vectors to drive the final hardware design. While this method of verification provided bit-accurate comparisons between the Acoustic Modeling and Phoneme Evaluation blocks, such was not the case for the verification of the Word Modeler. In the full SPHINX system there is an additional level of feedback present in the word model that helps to enforce grammar constraints on sequences of words. This level of feedback, while not implemented in any of our proposed methods, is integrated very tightly into the rest of the SPHINX system and therefore cannot simply be ‘turned-off’ to allow for a direct comparison of the Word Modeler operations. Given this constraint, verification of the Word Modeler operations was limited to supplying the SPHINX code, the MATLAB code, and the hardware the same phoneme information and then only observing to see if the same word strings were observed, not if the associated probabilities were identical. The hardware and MATLAB results were able to be verified against each other to ensure that both version of our method produced identical results but it is important to note the indirect comparison of these results to the SPHINX 3 baseline.

# 4.0 ACOUSTIC MODELING

In modern speech recognition systems the first stage of processing after Feature Extraction is called the Acoustic Modeling phase. It is during this phase that the results of the signal processing operations performed at the front-end of the system are referenced against a database of know Gaussian distributions and relatively scored to find the most likely matches. These comparisons require the computation of thousands of multi-dimensional Gaussian PDFs and consume a majority of the run-time of systems operating on even the most modern of desktop computers. This chapter takes a closer look at the operations performed during Acoustic Modeling and expands on the work presented in [59, 60, 61]. Through this work, a solution is found to reducing the amount of execution time through the use of highly vectorized code and the subsequent development of a high throughput hardware device.

## 4.1 GAUSSIAN PROBABILITY EVALUATIONS

In the world of statistical modeling there is a large number of distribution functions used to represent the different types of populations observed in the world around us. Among the most popular of these is the Gaussian Probability Density Function (PDF), both because of its ability to model a wide variety of populations and because of its ability to be used in single dimensional as well as multi-dimensional problems. Gaussian PDFs can also be used to model other types of distributions, such as the Gamma or Poisson, under certain circumstances and so they become a very appealing option for all but the most special cases of statistical population analysis problems. The following section will describe in detail the mathematics necessary for both single and multi dimensional Gaussian analysis and also prove the correlation of Gaussian distributions to the observed input populations seen when performing ASR.

### 4.1.1 One-Dimensional Gaussian Probability Density Functions

The Gaussian or Normal Distribution is perhaps one of the most widely used distributions in statistical modeling today. The equation for a basic Gaussian Probability Density Function (PDF) is given in Eq. 1, where *x* represents the input, *μ* represents the mean of the Gaussian, and $\sigma^2$ represents the variance.

$$P(x) = \frac{1}{\sqrt{(2\pi)}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \qquad \text{[Eq. 1]}$$

A set of sample Gaussian distributions is shown in Figure 12 for the case where $\mu = 0$, and $\sigma^2 = \{0.5, 1, 2\}$, evaluated over the range -5:5.

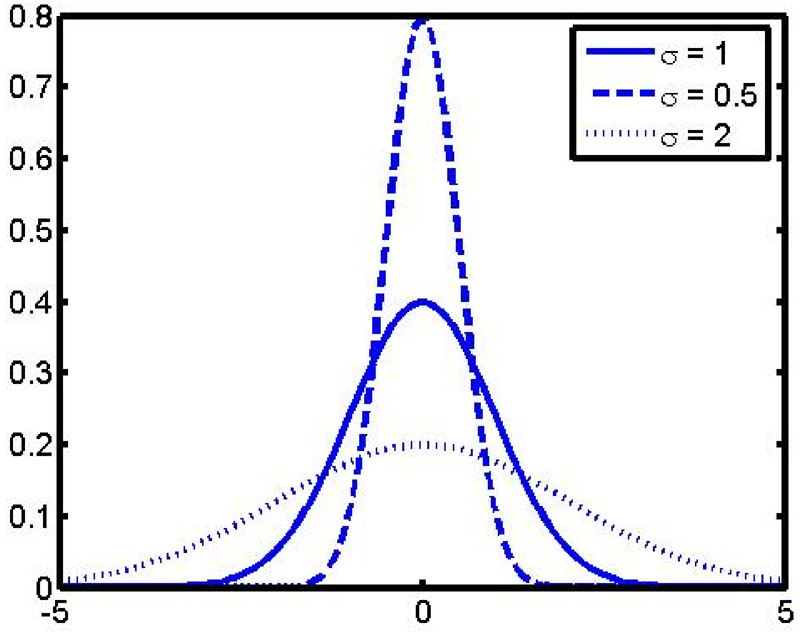


**Figure 12.** Sample Set of Gaussian Distributions for Multiple $\sigma^2$ Values

There are a wide range of applications for these distributions as they serve well to model a vast array of populations, including speech signals. In a modern ASR system, the input being fed to the system from the front-end signal processing element is a vector of Cepstral coefficients usually 39 elements long. Each element represents one of the 13 Mel-Frequency Cepstral Coefficients (MFCCs) as well as their first and second derivatives [15], and can be modeled by Gaussian distributions for input sets with large populations. This is achieved through the recording of large amounts of data and then analyzing the results to find the best set of Gaussians to represent the entire input set. During run-time the actual input to the system is evaluated with respect to this set of Gaussians to see which ones most accurately represent the observed data. The more complete the set and the higher the ability to adapt the set, the better recognition that can be achieved. To verify that the MFCC values obtains during feature extraction are in fact normally distributed, a series of experiments were conducted to analyzed the results of feature extraction on 2,100 different frames of input speech from the RM1 speech corpus. The results of these experiments are shown in Figure 13 for 4 of the 39 elements of the input vector. Figure 13 clearly illustrates that even though some outliers exist for every coefficient, Gaussian distributions will accurately approximate the populations found when examining large input populations.

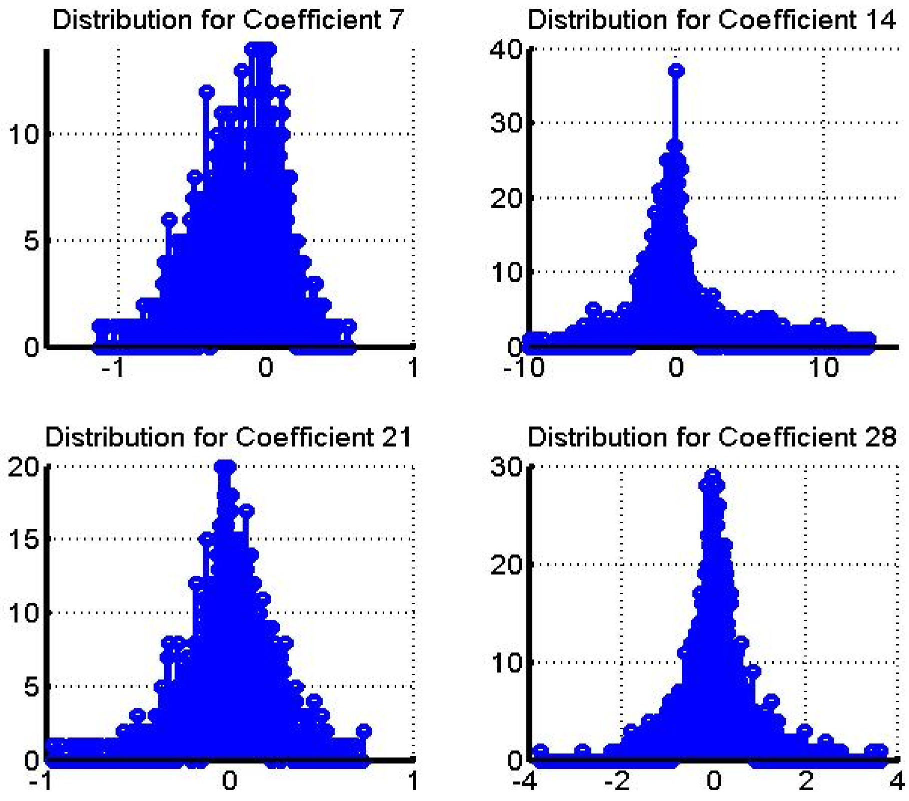


**Figure 13.** Distribution of 2,100 Cepstral Inputs for 4 Different Coefficients

Another helpful property of the Gaussian is what is called the *empirical rule.* This rule states, 68% of the data represented by a Gaussian distribution lies within ±1σ of the mean, 95% within ±2σ of the mean, and 99% within ±3σ of the mean [56]. Given that σ is a known value then the ±Nσ values for N = {1,2,3} can be calculated off-line and used to calculate the quality of the input without having to evaluate Eq. 1. This can be very useful in system where inputs beyond a certain threshold do not need to be considered due to their poor quality. Speech recognition systems have been shown to behave in this manner observing that during the later stages of processing a beam pruning algorithm is used to remove poor results from the search space. If these results can be pruned out by a simple comparison before calculation, then the amount of poor solutions being evaluated by the system can be minimized allowing for more correct solutions to be included. To verify this paradigm experiments were conducted using input data from the RM1 speech corpus to find the percentage error introduced into the system for varying levels of database approximation. To control the amount of the database that was approximated the *N* value in the Eq. 2 was varied across the range 0:6, and the results for the experiment can be seen in Figure 14.

$$\mu - N\sigma \leq x \leq \mu + N\sigma \qquad \text{[Eq. 2]}$$

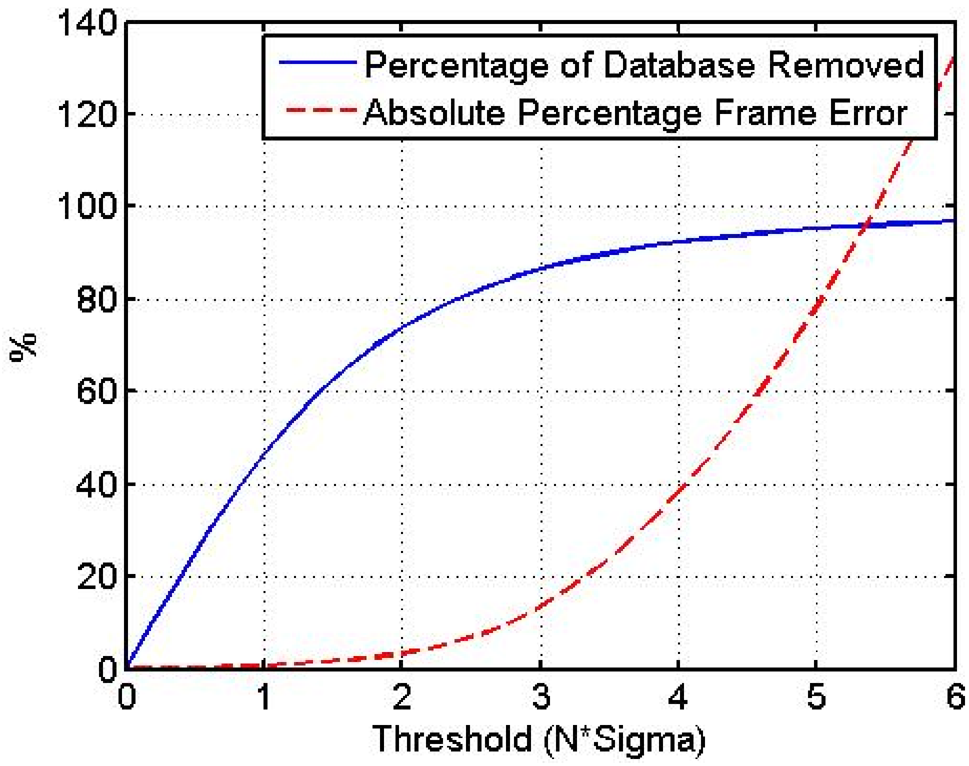


**Figure 14.** Percentage Error Vs. Percentage of Database Removed for Varying n Values

The results shown in Figure 14 appear to be very promising and looking at the 2σ mark it is observed that for a 75% reduction in the database, less than 10% potential error is introduced. Unfortunately, the implementation of an algorithm capable of achieving this potential savings is not straight forward, but the observation does create for an interesting mathematical relationship between accuracy and savings within a Gaussian probability evaluation. This relationship is shown in Eq. 3-5 and results in the acquisition of a constant *N* that is capable of controlling the trade-off between savings and error in the system. In the following equations *T* denotes value obtained at the threshold distance, and *N* denotes the specific sigma distance from the Mean ($\mu$) at which *T* lies.

$$T = \frac{[x-\mu]^2}{2\sigma^2} \qquad \text{[Eq. 3]}$$

$$let :\rightarrow x = \mu + N\sigma$$

$$T = \frac{[\mu + N\sigma - \mu]^2}{2\sigma^2} = \frac{[N\sigma]^2}{2\sigma^2} \qquad \text{[Eq. 4]}$$

$$T = \frac{N^2}{2} \rightarrow \therefore \ N = \sqrt{2T} \qquad \text{[Eq. 5]}$$

It can be seen from the Eq. 5 that for a given threshold *T*, the value of *N* is independent of the mean and variance of a Gaussian thereby enabling it to be set as a constant for the entire system.

### 4.1.2 D-Dimensional Gaussian Probability Density Functions

Like other statistical distributions, Gaussians can be used to model *n*-dimensional distributions allowing for the evaluation of the entire input vector as a single multivariate Gaussian distribution. The equation for a multivariate Gaussian distribution is,

$$P(\overline{X}) = \frac{1}{\sqrt{(2\pi)^D \left|\overline{V}^*\right|}} e^{-\sum_{d=1}^{D} \frac{(X_d - \mu_d)^2}{2*\sigma_d^2}} \qquad \text{[Eq. 6]}$$

where $D$ represents the number of dimensions in the distribution, $\overline{X}$ represents the Cepstral coefficient input vector, $\overline{V}^*$ defines the covariance matrix, and $\overline{\mu}$ and $\overline{\sigma^2}$ represent the mean and variance vectors respectively. As *D* increases so does the complexity of the model, allowing for increasingly more diverse data to be represented by a single equation and creating a very powerful tool for comparing two sets of data. From analysis of these types of equations not only can a set of elements be compared to one another on many different levels but the amount of similarity in any one level can be found leading to the ability to weight the results obtained in the different dimensions to produce an optimized model for representing a given population. When evaluating speech signals, each one of the 39 coefficients in the MFCC input stream is assigned to a separate dimension of a 39 dimensional multivariate Gaussian PDF. Experiments were conducted based on the RM1 speech corpus to determine the weighting necessary for each of the 39 dimensions, and the results can be seen in Figure 15. During the experiment 4,000 MFCC input vectors were analyzed to see what percentage of the total summation in Eq. 6 was supplied by each coefficient.

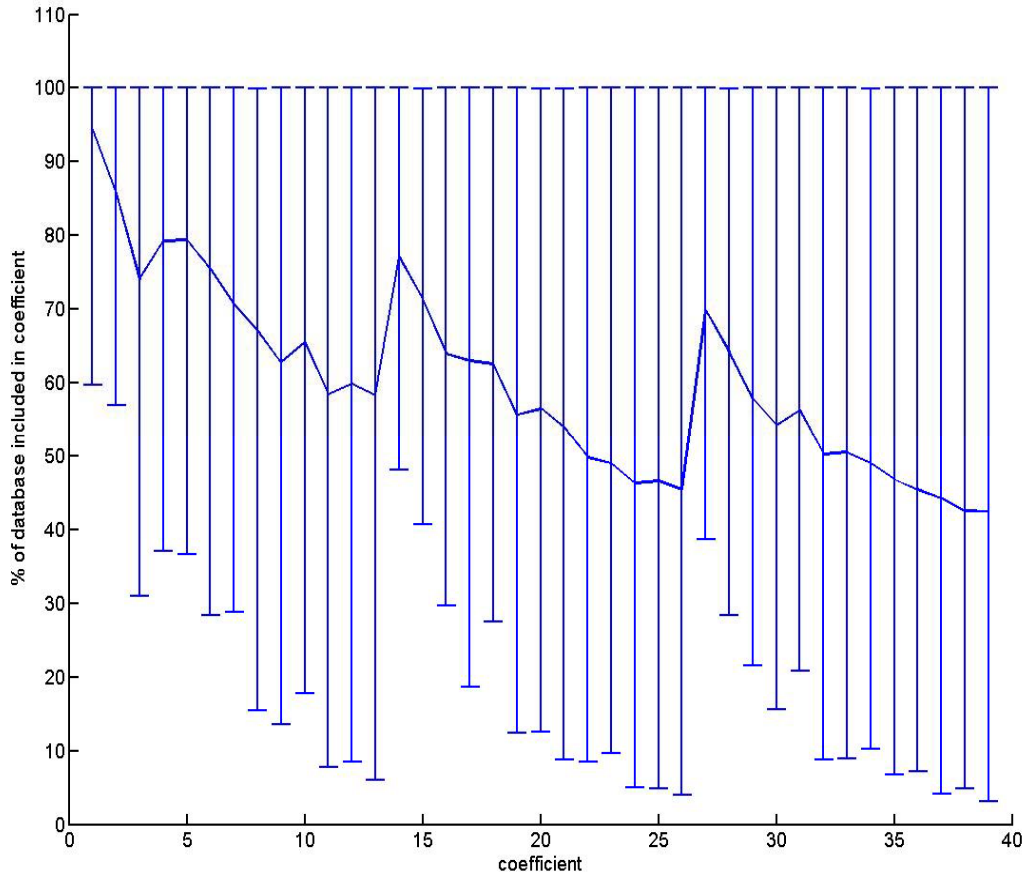


**Figure 15.** Percentage Contribution of Each Dimension of a Multivariate Gaussian PDF

Figure 15 shows the average percentage contribution of each coefficient as well as its maximum and minimum contributions and leads to the observation that while some coefficients do contribute less to the total summation on average, all components are individually responsible for the entire summation for at least one of the inputs. In mapping each of the coefficients to its own dimension of a multivariate Gaussian, the need for extremely large amounts of calculation is created, as illustrated in Table 2, and presents a pressing need for optimization of the equations if real-time operation on a modern computer system is to be considered.

**Table 2.** Number of Calculations Necessary for Various Dictionaries

| **Speech Corpus** | **Dictionary Size (words)** | **Number of Senones** | **Number of Components** | **Number of Gaussian Evaluations** | **Number of Ops. per Frame** | **Number of Ops. per Sec.** |
|---|---|---|---|---|---|---|
| TI Digits | 12 | 602 | 4.8K | 188K | ~1.4M | ~140M |
| RM 1 | 1,000 | 1,935 | 15.5K | 604K | ~5M | ~500M |
| HUB-4 | 64,000 | 6,144 | 49.5K | 1.2 M | ~14M | ~1.4G |

Given the nature of the equation for a normal distribution, the application of a log-domain transform leads to significantly easier calculations in modern computer systems. This reduction in complexity arises from the transformation of the exponentiation in Eq. 6 to an addition, a substantially easier computational task, as shown in Equations 7-8.

$$\ln\left(P\left(\overline{X}\right)\right) = -0.5\ln\left[\left(2\pi\right)^{D}\left|\overline{V}^{*}\right|\right] - \sum_{d=1}^{D}\frac{\left(X_d - \mu_d\right)^2}{2 * \sigma_d^2} \qquad \text{[Eq. 7]}$$

$$\textit{let: } K = -0.5\ln\left[(2\pi)^D \left|\overline{V}^*\right|\right]$$

$$\textit{let: } Dist(\overline{X}) = \sum_{d=1}^{D} \frac{(X_d - \mu_d)^2}{2 * \sigma_d^2} = \sum_{d=1}^{D} (X_d - \mu_d)^2 * \Omega_d \ , \ \Omega_d = \left(\frac{0.5}{\sigma_d^2}\right)$$

$$\ln\left(P\left(\overline{X}\right)\right) = K - Dist\left(\overline{X}\right) \qquad \text{[Eq. 8]}$$

The *K* value in Eq. 8 is now independent of the input and can be calculated off-line and simply looked up during computation, while the rest of the mathematics required to find *Dist(X)*, commonly called the Mahalanobis distance, may be implemented in a simple three state pipeline, wherein the subtraction is done first, followed by the squaring, and then the multiplication by Ω. These results are then buffered into a *D*-deep buffer and summed when the buffer is full. These simplifications of the mathematics can decrease the run-time of a Gaussian calculation algorithm by a noticeable amount, and are widely used in applications involving statistical signal processing. In ASR systems using vector or sub-vector quantization algorithms, as described in section 2.1.3, it is only the Mahalanobis distance of the parent Gaussians that are calculated to find the best candidate clusters for full evaluation [20, 25]. While this may not appear to have significant savings over calculating the entirety of Eq. 8 for each parent Gaussian, once these equations are put into the equations necessary to solve for the component and senone scores during Acoustic Modeling, described in detail in section 3.2, the results of this simplification become non-trivial and provide significant savings during the first pass of the quantization algorithm.

## 4.2 DESCRIPTION OF CALCULATIONS

Having a basic understanding of Gaussian PDFs it next becomes necessary to understand how these equations are used to find the scores for the basic phonetic units, senones, used in the Acoustic Modeling process. In modern ASR systems it is not a single multivariate Gaussian PDF but groups of them that are used to represent the individual senones, leading to even more complex equations than the ones presented in Section 4.1.2. Different systems have used anywhere from 1 to 64 different Gaussians, traditionally called components, to represent senones and as would be assumed the greater number of components used, the more successful the recognition [8]. This does however create the need for extremely large databases of Gaussians and drastically increase the run-time of the system, so most commercially available products have chosen to work with 8 components per senone, as the return on investment for component counts greater than eight does not warrant the extra computational effort. Adhering to this convention allows the work presented in this paper to be compare directly to systems such as SPHINX 3, and for that reason the rest of the discussion involving senones will be understood to be referring to an eight component mixture of multivariate Gaussian PDFs. Noting that an ASR system has *i* total senones, each with *c, d*-dimensional components, Eq. 6 can be modified to include these new dimensions, resulting in Eq. 9.

$$P_{i,c}(\overline{X}) = \frac{1}{\sqrt{(2\pi)^D \left|\overline{V}^*\right|}} e^{-\sum_{d=1}^{D} \frac{(X_d - \mu_{i,c,d})^2}{2^*\sigma_{i,c,d}^2}} \qquad \text{[Eq. 9]}$$

Each of the *c* components in a given senone has a unique mixture weight $W_{i,c}$ that helps scale each Gaussians contribution to the final senone score. This mixture weight is a pre-defined constant that simply needs to be looked up in a ROM during run-time and when this constant is incorporated into Eq. 9, Eq. 10 is obtained which represents the basic equation of a senone, $S_i(X)$, in an ASR system.

$$S_i(\overline{X}) = \sum_{c=1}^{C}\left[W_{i,c} * P_{i,c}\left(\overline{X}\right)\right] \qquad \text{[Eq. 10]}$$

As mentioned in section 3.1.2, it is important to convert the mathematics from the natural domain to the log domain in order to simplify the operations being performed by the CPU. In order to facilitate this conversion a scalar conversion factor, *f*, can be derived as per Eq. 11-14, that allows for a direct mapping between the natural and log domain values for a given quantity. To allow for a direct comparison to the SPHINX 3 system in later sections of this paper the *Ψ* value used to find *f* in Eq. 11-14 is based off of data extracted directly from the SPHINX 3 source code.

$$f = \frac{1}{\ln(\psi)} \qquad \text{[Eq. 11]}$$

$$f * \ln[S_i(\overline{X})] = \log_{\psi}[S_i(\overline{X})] \qquad \text{[Eq. 12]}$$

$$\log_{\psi}[S_i(\overline{X})] = LOG\sum_{c=1}^{C}\left[\log_{\psi}\left(W_{i,c}\right) + \log_{\psi}\left(P_{i,c}(\overline{X})\right)\right] \qquad \text{[Eq. 13]}$$

*let:* $W^{'}_{i,c} = \log_{\psi}\left(W_{i,c}\right)$

$$\log_{\psi}\left[S_i\left(\overline{X}\right)\right] = LOG\sum_{c=1}^{C}\left[W^{'}_{i,c} + \log_{\psi}\left(P_{i,c}\left(\overline{X}\right)\right)\right] \qquad \text{[Eq. 14]}$$

The log-summation ($LOG\sum$) term seen in the right hand side of Eq. 14 represents an extension of the log-add function called the log-add, as defined in [10]. Eq. 15 decomposes the $\log_{\psi} P_{i,c}(\overline{X})$ term from Eq. 14 to highlight additional simplifications to the overall senone calculation.

$$\log_{\psi}\left(P_{i,c}(\overline{X})\right) = f * \ln\left(P_{i,c}\left(\overline{X}\right)\right) = f\left[-0.5\ln\left[(2\pi)^{D}\left|\overline{V}^{*}\right|\right] - \sum_{d=1}^{D}\frac{\left(X_d - \mu_{i,c,d}\right)^2}{2 * \sigma^2_{i,c,d}}\right] \qquad \text{[Eq. 15]}$$

The right side of Eq. 15 consists a summation term and a constant term defined in section 3.1.2 as *Dist(X)* and *K* respectively, and when the necessary substitutions are made in Eq. 15, Eq. 14 can then be re-written as Eq. 16.

$$Sen_i(\overline{X}) = \log_{\psi}[S_i(\overline{X})] = LOG\sum_{c=1}^{C}\left[W_{i,c}^{'} + f*[K_{i,c} - Dist_{i,c}(\overline{X})]\right] \qquad \text{[Eq. 16]}$$

Eq. 16 represents the actual calculation performed the Acoustic Modeling block in order to derive the senone score set for use by Phoneme Evaluation block in the next stage of processing. This score represents the combination of each of the individual Gaussians in the mixture, represented by Equation 8, scaled by the necessary log-domain conversion factor. Even with the log-domain transformation this equation still represents the majority of the work load within an ASR system and can account for 30% to 95% of the total computation time [1, 3, 4].

While Equation 16 represents a large, complicated portion of the speech processing algorithm, it also shows the potential for substantial optimization of the process. Specifically, each of the *i* senones can be calculated independently of each other allowing for a minimum parallelism of *i* for Acoustic Modeling process. Inside the senones, each of their *c* components may be calculated independently, with each components *d* dimensions also being independent. This means that during the calculation of $Dist_{i,c}(X)$ a potential parallelism of *i*c*d* may be achieved equating to over 624K parallel operations for a 1,000 word dictionary. The potential parallelism for each portion of the Acoustic Modeling calculations is summarized in Table 3 where the second column indicates the total potential parallelism implied by the equation and the third column indicates the parallelism achieved in the RM1 speech corpus.

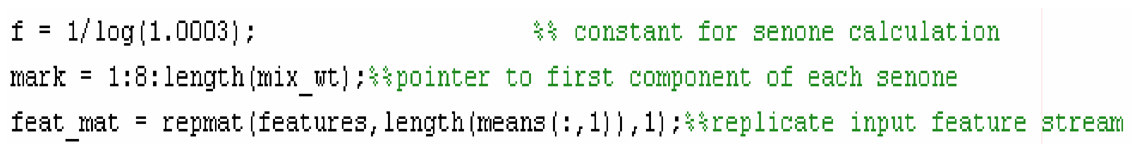
**Table 3.** Parallelism in Acoustic Modeling

| Calculation | Potential Parallelism | RM1 parallelism |
|---|---|---|
| $Dist(\overline{X}) = \sum_{d=1}^{D}(X_d - \mu_d)^2 * \Omega_d$ | ***i*c*d*** | **624,000** |
| $W'_{i,c} + f * [K_{i,c} - Dist_{i,c}(\overline{X})]$ | ***i*c*** | **16,000** |
| $LOG\sum_{c=1}^{C}\left[W'_{i,c} + f * [K_{i,c} - Dist_{i,c}(\overline{X})]\right]$ | ***i*** | **2,000** |

## 4.3 MATRIX MATLAB REPRESENTATION

In current software implementations, the calculations described in Sections 4.1 and 4.2 are performed sequentially using nested for-loops and require large amounts of C code to execute. While this is necessary given the architecture of modern desktop computers it is by no means the most efficient way to execute this process. When the calculations are examined in detail it becomes quite clear that there is only a small number of operations that actually need to be performed, the problem lies in the extremely large number of times that each of these operations needs to be executed. This paradigm provides a perfect fit for a vector processing solution in that, if all of the actionable data can be lined-up into a large input vector, then only one operation would need to be performed on the entire vector, emulating a large-scale SIMD processor. While this is not completely possible on either a SIMD or a VLIW machine since the size of the input vectors (>1000 elements) is far too large, it is possible to simulate this type of operation using the MATLAB software suite. MATLAB is capable of executing large scale vector operations in ways not possible using other coding languages and enables the creation of simulation models of systems that would potentially be capable of executing such operations. All of the code written for this project can be found in Appendix A at the end of the document with relevant sections shown as figures within the body of the document. The code for Acoustic

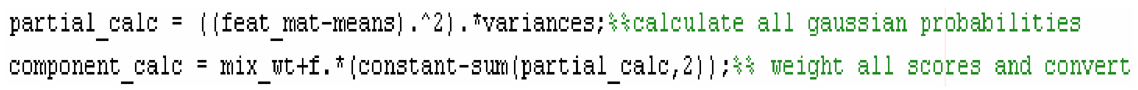
Modeling highlights the abilities of MATLAB code to perform large vector operations, noting that the entirety of the MATLAB code is only 19 lines long. The first 3 lines of code, shown in Figure 16 are responsible for declaring the constants necessary to execute the AM routine.



```
f = 1/log(1.0003);                    %% constant for senone calculation
mark = 1:8:length(mix_wt);%%pointer to first component of each senone
feat_mat = repmat(features,length(means(:,1)),1);%%replicate input feature stream
```

**Figure 16.** Initialization Code for AM

Line one creates the *f* constant described in section 4.2, line two creates a vector to hold the address of the first component of each senone score, and line three replicates the input feature stream to create a large input matrix for use in the AM calculation. The creation of the *mark* vector allows for the starting point each senone summation to be found so that once all of the components are calculated the result vector can be broken into smaller sub-vectors wherein all the elements of the sum-vector are summed together to obtain the senone scores. The next two lines of code, shown in Figure 17, execute Eq. 8 and Eq. 16 respectively.



```
partial_calc = ((feat_mat-means).^2).*variances;%%calculate all gaussian probabilities
component_calc = mix_wt+f.*(constant-sum(partial_calc,2));%% weight all scores and convert
```

**Figure 17.** Calculation of Component Scores

In these two lines of code every Gaussian PDF in the entire system is calculated and their resultant component scores are found. This is done through the subtraction and multiplication of two large *(i*c)* x *D* matrices on an element per element basis and then summing the result matrix along the *D* dimension. After the summation the resulting *(i*c)* x 1 vector is subtracted from the

*K* constant and then scaled by the log-conversion factor *f*. For clarification purposes this process is shown in a block diagram form in Figure 18. The ability to represent these operations comes from use of the '.*' operator in MATLAB as opposed to a traditional '*' for the multiplication of the matrices. This special '*dot-star*' operator represents an element-wise multiplication of the two arrays instead of a true matrix multiplication. By using this form the potential parallelism of the algorithm being performed is translated directly into the size of the matrix being operated on. As noted in section 4.2 the potential parallelism for the RM1 speech corpus is approximately 620K for the *partial_calc* matrix and 16K for the *component_calc* vector and this is confirmed by looking at the size of the input arrays presented in Figure 18 where $i$ = 2,000, $c$ = 8, and $D$ = 39.

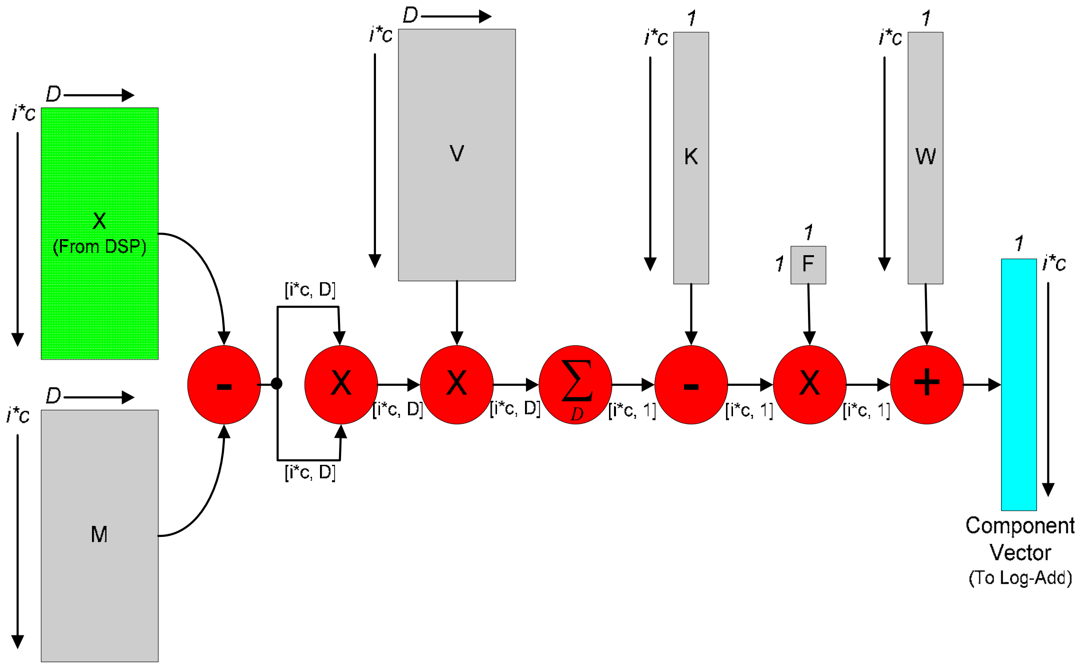


**Figure 18.** Block Diagram of Vectorized Gaussian PDF Calculation

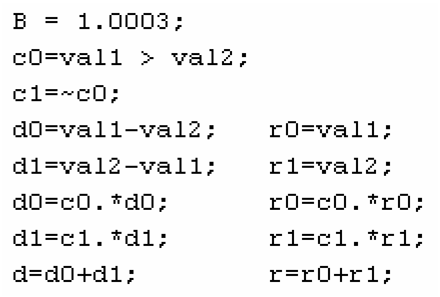
Once all of the components have been calculated it is necessary to execute the log-summation of Eq. 16 to obtain the set of senone scores to be sent to the Phoneme evaluator. This log-summation operation must be executed in a sequential manner for each senone and cannot therefore be completely vectorized, but it is possible to reshape the *(i*c)* x 1 component vector into a $i$ x $c$ matrix and then perform $i$ log-summations in a vectorized form. The calculation necessary to calculate *RES*, the partial result of the log-summation of $input_A$ and $input_B$ , is shown in Eq. 17.

$$let:\ VAL_1 = MAX[input_A, input_B]\ \&\ VAL_2 = MIN[input_A, input_B]$$

$$let:\ B = 1.0003\ \&\ d = VAL_1 - VAL_2$$

$$RES = VAL_1 + 0.5 + f * \log(1 + B^{-d}) \qquad \text{[Eq. 17]}$$

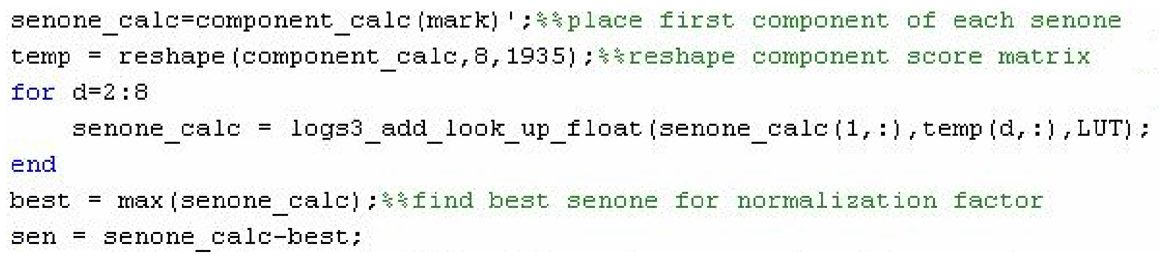
To help minimize the execution time for the log-summation operation, a look-up table of partial results was created such that at each step in the summation the results can simply be looked-up, as opposed to having to execute Eq. 17. Although substituting a look-up table for Eq. 17 does reduce the amount of computation done to obtain a partial result of the log-summation there is still some additional processing that needs to be done in order to allow for this operation to be done in a vectorized fashion. Figure 19 shows the additional steps needed to calculate *d, $VAL_1$, and $VAL_2$,* for all $i$ senones at once noting that the variable $r$ is used to represent the vector of $VAL_1$ entries.



```
B = 1.0003;
c0=val1 > val2;
c1=~c0;
d0=val1-val2;    r0=val1;
d1=val2-val1;    r1=val2;
d0=c0.*d0;       r0=c0.*r0;
d1=c1.*d1;       r1=c1.*r1;
d=d0+d1;         r=r0+r1;
```

**Figure 19.** Code for Vectorized Calculation of Log-Summation

Once all of the senone scores have been calculated for a given frame the entire set must be normalized with respect to the best senone score found during that frame. The code necessary to reshape the component data, calculate the senones, and normalize the results is shown in Figure 20.

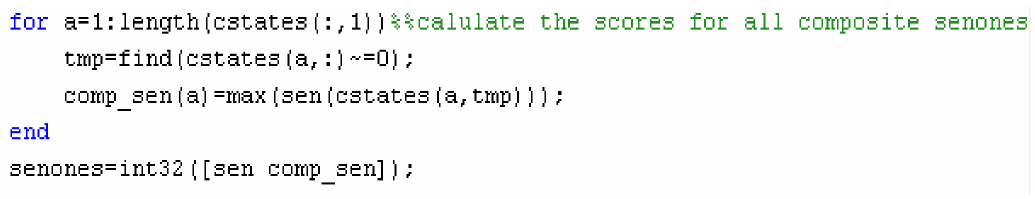


```
senone_calc=component_calc(mark)';%%place first component of each senone
temp = reshape(component_calc,8,1935);%%reshape component score matrix
for d=2:8
    senone_calc = logs3_add_look_up_float(senone_calc(1,:),temp(d,:),LUT);
end
best = max(senone_calc);%%find best senone for normalization factor
sen = senone_calc-best;
```

**Figure 20.** Code Necessary to Calculate Senone Scores

This process of normalization finishes the bulk of the computation for the acoustic modeler leaving only the calculation of the special case or composite senones to be performed. In the RM1 speech corpus there are two distinctly different types of senones. The first are what can be considered ‘normal’ or ‘base’ senones and are calculated via the processes described in equations 6-17. The second type of senone is a sub-set of the normal senones called composite senones. Composite senones are used to represent more difficult or easily confusable sounds, as well as non-verbal anomalies such as silence or coughing. Each composite senone is pointer to a group of normal senones, and for a given frame the composite senone takes the value of the best scoring normal senone in its group.

In terms of computation this equates to the evaluation of a series of short link-lists, where the elements of the list must be compared to find the greatest value. Once this greatest value is found it is written to a unique location in the senone RAM at some address above the address of the last normal senone. By writing this entry into is own location in the senone RAM instead of creating a pointer to its original location, the Phoneme Evaluation block is able to treat all senones equally, thus simplifying the control for that portion of the design. When implementing the composite senone evaluation in MATLAB we can create a simple loop that executes once for each composite senone and is able to find the composite value in a single line of code. The code for this loop is shown in Figure 21.



```
for a=1:length(cstates(:,1))%%calulate the scores for all composite senones
    tmp=find(cstates(a,:)~=0);
    comp_sen(a)=max(sen(cstates(a,tmp)));
end
senones=int32([sen comp_sen]);
```

**Figure 21.** Code for Composite Senone Calculation

The first line of the loop finds the locations of the normal senone addresses in the *cstates* matrix and the second line uses this information to obtain the normal senone scores and find the max of all acquired values. Once all composite senones have been found they are appended to the end of the normal senone vector as seen in the last line of code in Figure 20. After concatenating the composite senone vector and the normal senone vector the process of AM is complete and phoneme evaluation may begin.

## 4.4 HARDWARE ARCHITECTURE

Once the MATLAB model was completed for AM it was next used as a guide for developing a custom hardware co-processor. Since the parallelism implied by the MATLAB model cannot be fully exploited, it was chosen to develop a fully pipelined hardware block small enough to be tiled on an FPGA to increase the overall throughput. Figure 22 shows the block diagram for the proposed hardware system with the primary interconnect busses shown for clarity.

Each of the stages in the pipeline sends a ‘go’ signal to the following stage along with any data needing processed, allowing for the system to be stalled anywhere in the pipe without breaking. The first three stages also receive data from a status bus regarding the particular nature of the calculation being perform (ie. is this the first, middle, or last element of a summation), which removes the need for any local finite state machine to control the pipeline. In addition to removing the need for internal finite state machines the use of a input control bus also helps to minimize the number of pipeline stalls in the system and creates a system capable of adapting to multiple different configurations without having to perform any hardware re-design.

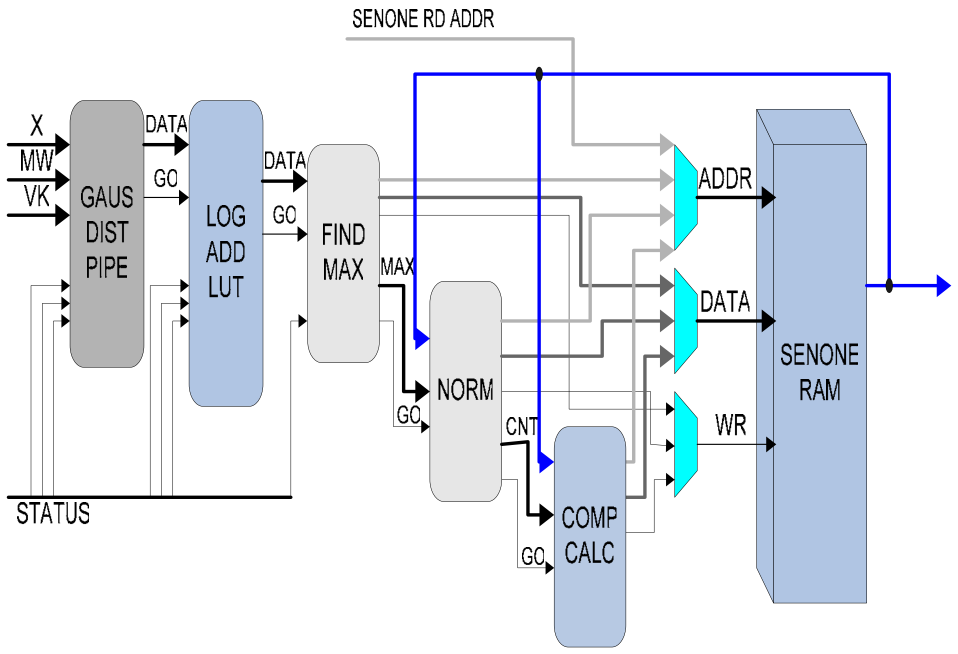


**Figure 22.** Block Diagram for Acoustic Modeling Co-Processor

### 4.4.1 Gaussian Distance Pipelined Processor

The Gaussian Distance pipe is the heart of AM block and is responsible for calculating Eq 6-8 for each senone in the database. This pipe must execute Eq. 6 over 620,000 times for each new frame of data and therefore must have the highest throughput of any component in the system. To accommodate this requirement while still trying to minimize the resources consumed by pipeline, the inputs to crucial arithmetic operations are muxed, allowing the inputs to the operation to be selected based on the bits of the Status Bus. Figure 23 shows a data flow graph for the order of operations inside the Gaussian Distance Pipe.

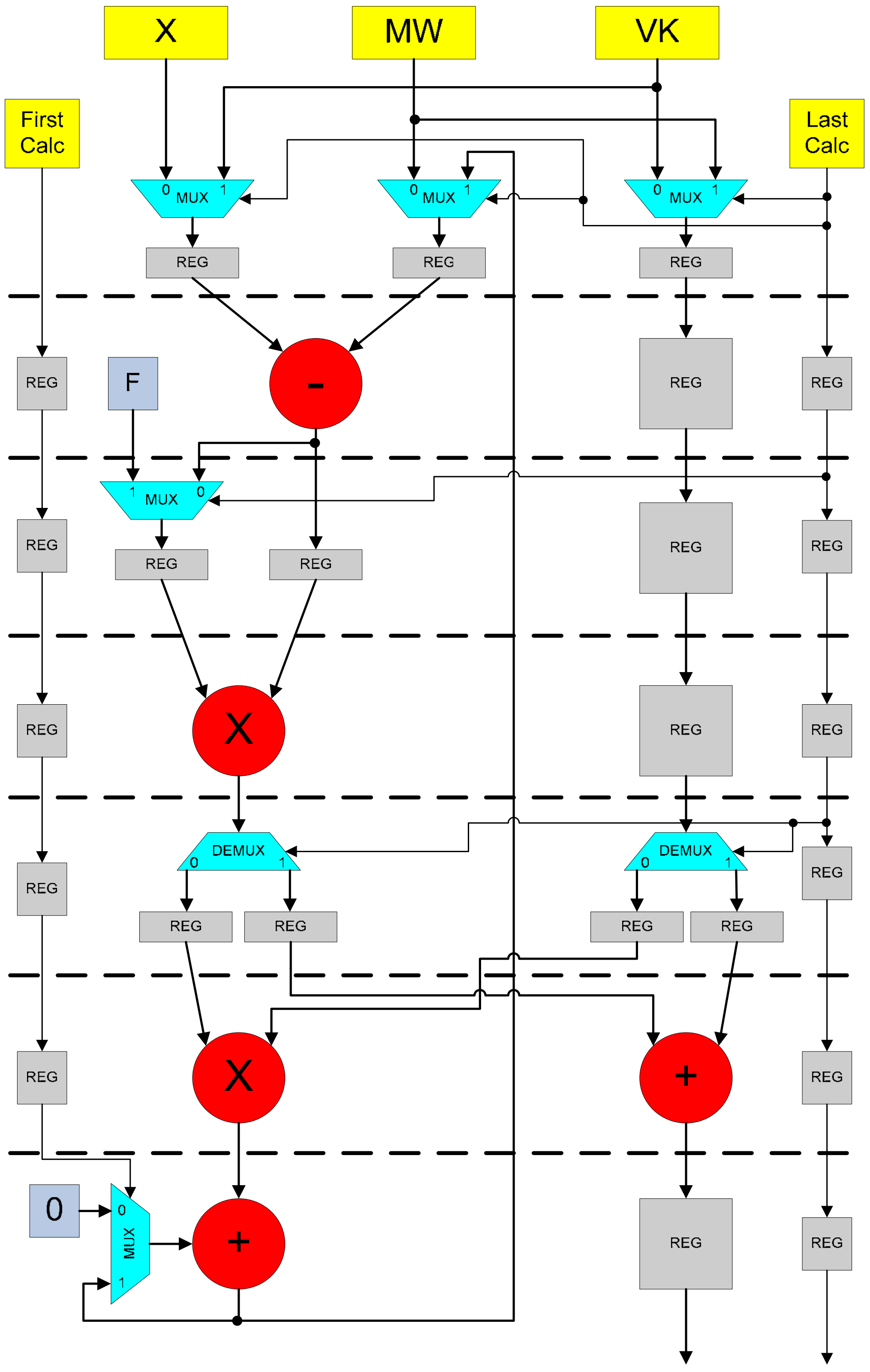


**Figure 23.** Data Flow Graph for Gaussian Distance Pipe

Figure 23 indicates a 7 cycle for the pipe, however, the next stage of the design the Log-Add LUT, described in Section 4.4.2, takes 10 cycles to traverse and therefore we have added 3 extra cycles to the Gaussian distance pipe to keep both stages in sync with one another. This synchronization is necessary since it takes the log-adding of multiple components to calculate a single senone, and buffering schemes would have to be implemented to account for the cycle mismatch. Having the next component ready on the same cycle that the current one finishes is the most efficient way to execute the calculations therefore the decision to extend the depth of the Gaussian distance pipe becomes most appealing. In order to ensure that the addition cycles would not be detrimental to the performance of the system a series of experiments were conducted examining the effects of additional pipeline stages on the achieved $f_{max}$ of the system. The results of these experiments as well as the synthesis and post place-and-route results for this block are summarized in section 3.5.

In order to help with low power applications, the Gaussian Distance pipe has a pipeline stall feature included which is not shown in the data flow graph. If the last calc bit is seen at the end of the pipe before a new first calc bit has been seen the pipe will completely shut down and wait for the presence of a new first calc bit. Internal to the pipe each stage passes a valid bit to the successive stage that serves as a local stall, which will freeze the pipe until the values of the predecessor stage have become valid again.

### 4.4.2 Log-Add Look-Up

After completing the scoring for one component, that component is sent to the Log-Add look-up for evaluation of equations 10-16. This block is responsible for accumulating the partial senone scores and outputting them when the summation is complete. The primary function of this block can be summarized by Equation 17. Due to the complexity of Equation 17, it has been replaced by a look-up table where *D* serves as the address into the table. By using this look-up Equation 17 can be simplified to the result seen in Equation 18.

$$RES = R + LUT(D) \qquad \text{[Eq. 18]}$$

While use of a look-up table to perform the bulk of the computation is a more efficient means of obtaining the desired result, it creates the need for a table with greater than 20K entries. In an effort to maximize the speed of the look-up the table was divided into smaller blocks and the process was pipe-lined over 2 clock cycles wherein the address is de-muxed on the first cycle and the data is fetched and muxed onto the output bus during the second. The operations necessary to find the address to this look-up and the operations can be summarized by stating that the absolute difference of the inputs is used to address a look-up table whose data is added to the larger of the two inputs. In order to complete these operations with minimal delay we chose to implement them as a three stage pipeline. The first stage of operation performs a subtraction of the two raw inputs and strips the sign bit off of the output. In the second cycle the sign bit is used as a select signal to a series of muxes that assign the larger of the two inputs to the first input of the subtraction, and the smaller to the second input of the summation. The third cycle of the pipe registers the larger value for use after the LUT and simultaneously subtracts the two values to obtain the address for the LUT. Figure 24 shows a detailed data-flow graph of the operations being performed inside the Log-Add LUT.

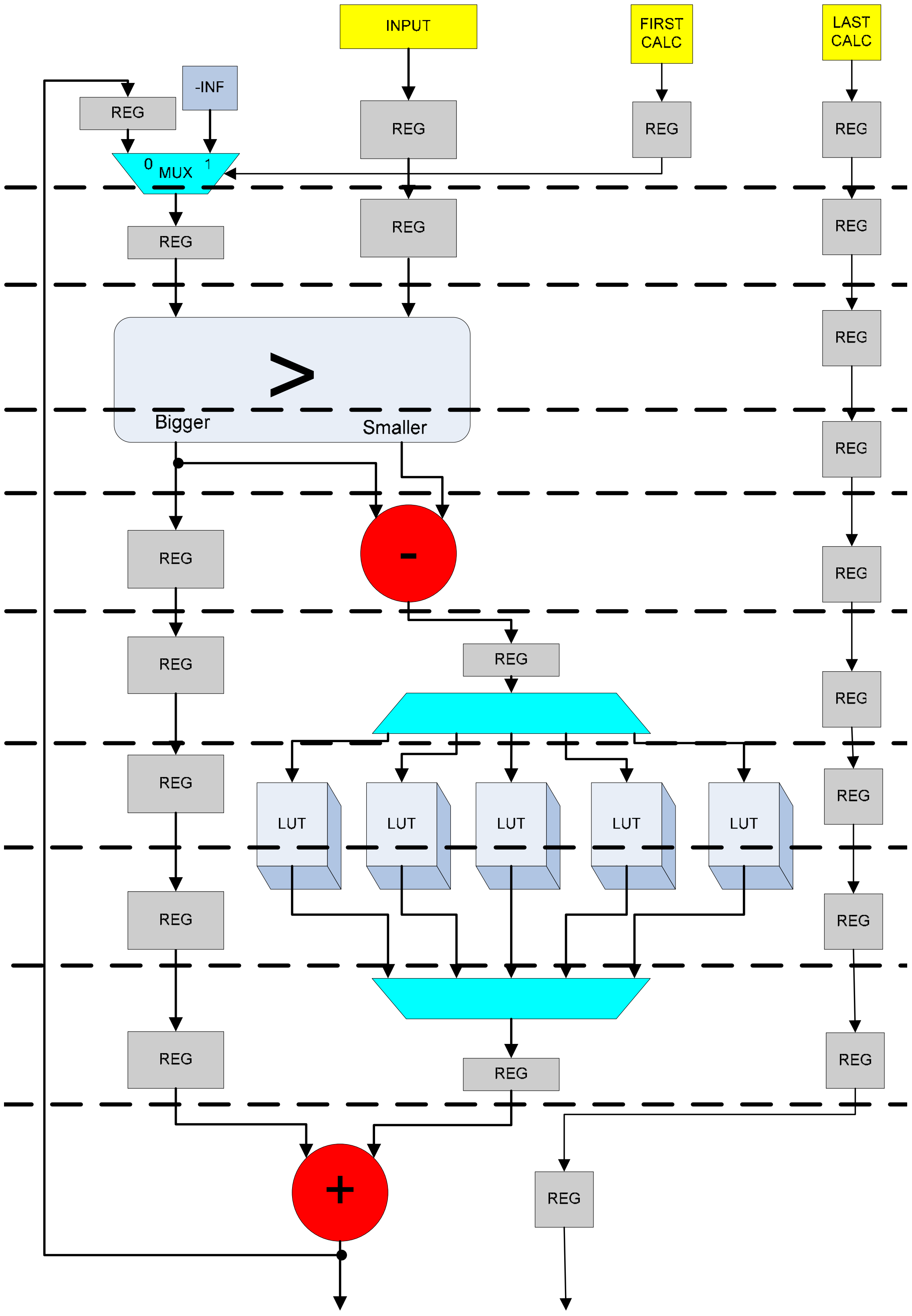


**Figure 24.** Data-Flow Graph for Log-Add LUT

Similar to the Gaussian Distance pipe the Log-Add LUT also has a pipe-stall function built in. This function performs exactly as the Gaussian Distance pipe freeze with respect to the first and last calc bits, and will also perform a local stall if the partial log-add has been updated before a new component value is available. As mentioned at the end of Section 4.4.1, the entire log-add calculation takes a minimum of 10 clock cycles to process a single input and return the partial summation for use by the next input. When this block is combined with the Gaussian Distance Pipe to form the main pipeline structure for the AM block the result is a 20-stage pipeline capable of operating at over 140MHz, and requiring no local finite state machine for managing the traffic through the pipe.

### 4.4.3 Find MAX / Normalizer

Once a senone has been calculated it must first pass through the Find Max block before being written to the Active Senone RAM. This block is a 2-cycle pipeline that compares the incoming data to the current best score and overwrites the current best when the incoming data is larger. Once the larger of the two values has been determined the raw senone is output to the senone RAM along with a write signal supplied by a registered version of the valid signal supplied to the block by the log-add LUT. A data-flow graph for the Find Max block is shown in Figure 25.

As was pointed out in Section 4.4.2, the Find Max unit only needs to operate once every 10 cycles, or whenever a new senone is available, therefore the values being fed to the compare are only updated when the senone valid bit is high. Aside from this local stall, the Find Max unit has a pipe-stall function similar to the one described in the previous sections to minimize the amount of power consumed by the device during run-time.

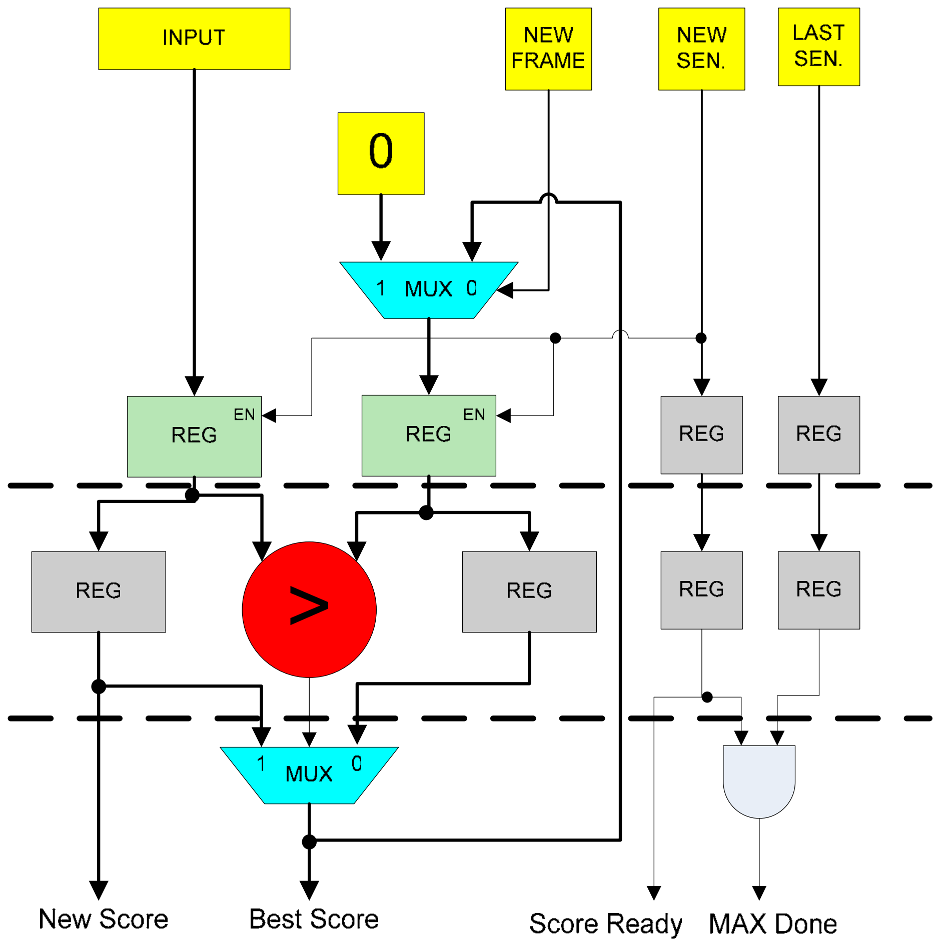


**Figure 25.** Data-Flow Graph for Find Max Unit

When the last raw senone is put into the senone RAM the *MAX done* signal in Figure 25 is set high, signaling to the Normalizer block that it can begin. During the process of normalization the raw senones are read sequentially out of the senone RAM and subtracted from the value seen at the *Best Score* output of the Find Max block. The Normalizer block consists of a simple 4-stage pipeline that first reads from the RAM, then registers the input, then performs the normalization, and finally writes the value back to the RAM. As per the blocks discussed in previous sections, the Normalizer block also has pipe-stall and local stall capabilities.

### 4.4.4 Composite Senone Calculation

In the RM1 speech corpus there are two types of senones. The first are what can be considered *normal* or *base* senones and are calculated via the processes described in Sections 4.4.1-4.4.3. The second type of senone is a sub-set of the normal senones called *composite senones*. Composite senones are used to represent more difficult or easily confusable sounds, as well as non-verbal anomalies such as silence or coughing. Each composite senone is pointer to a group of normal senones, and for a given frame the composite senone takes the value of the best scoring normal senone in its group. In terms of computation, this equates to the evaluation of a series of short link-lists, where the elements of the list must be compared to find the greatest value. Once this greatest value is found it is written to a unique location in the senone RAM at some address above the address of the last normal senone. By writing this entry into is own location in the senone RAM instead of creating a pointer to its original location, the Phoneme Evaluation block is able to treat all senones equally, thus simplifying that portion of the design.

The composite calculation works through the use of two separate internal ROMs to store the information needed for processing the link-lists. The first ROM (*COUNT ROM*) contains the same number of entries as the number of composite senones in the system, and holds information about the number of elements in each composites link-list and each lists start address. When a count is obtained from this ROM it is added to the start address and used to address a second ROM (*ADDR ROM*) that contains the specific address in the senone RAM where the normal senone resides. Once the normal senone has been obtained from the senone RAM it is passed through a short pipeline similar to the Find MAX block except that only the best score is output to be written back to the senone RAM. The count is then decremented and the process repeated until the count equals zero. At this point the next element of the count ROM is read and the process is repeated for the next composite senone. Once all elements of the count ROM have been read and processed, the block will assert a done signal indicating that all the senone scores for a given frame have been calculated. A data flow graph for the Composite Senone calculation is shown in Figure 26.

Like the other blocks of the AM calculation, the Composite Senone calculation has the built-in ability for locally stalling during execution and freezing completely when no new data is present at the input.  These features become more significant when considering this block however, because the Composite Senone calculation can only be performed once the all of the normal senones have been completely processed.  This results in a significant portion of the run-time where this block can be completely shut down leading to notable savings in terms of power consumption for the system.  Specifically, it takes approximately 650,000 clock cycles to calculate all of the normal senones, during which the Composite Senone calculation block may be totally shut down, then once awoken the block need only run for 2,200 cycles to calculate the composite senones and may then be shut off again.

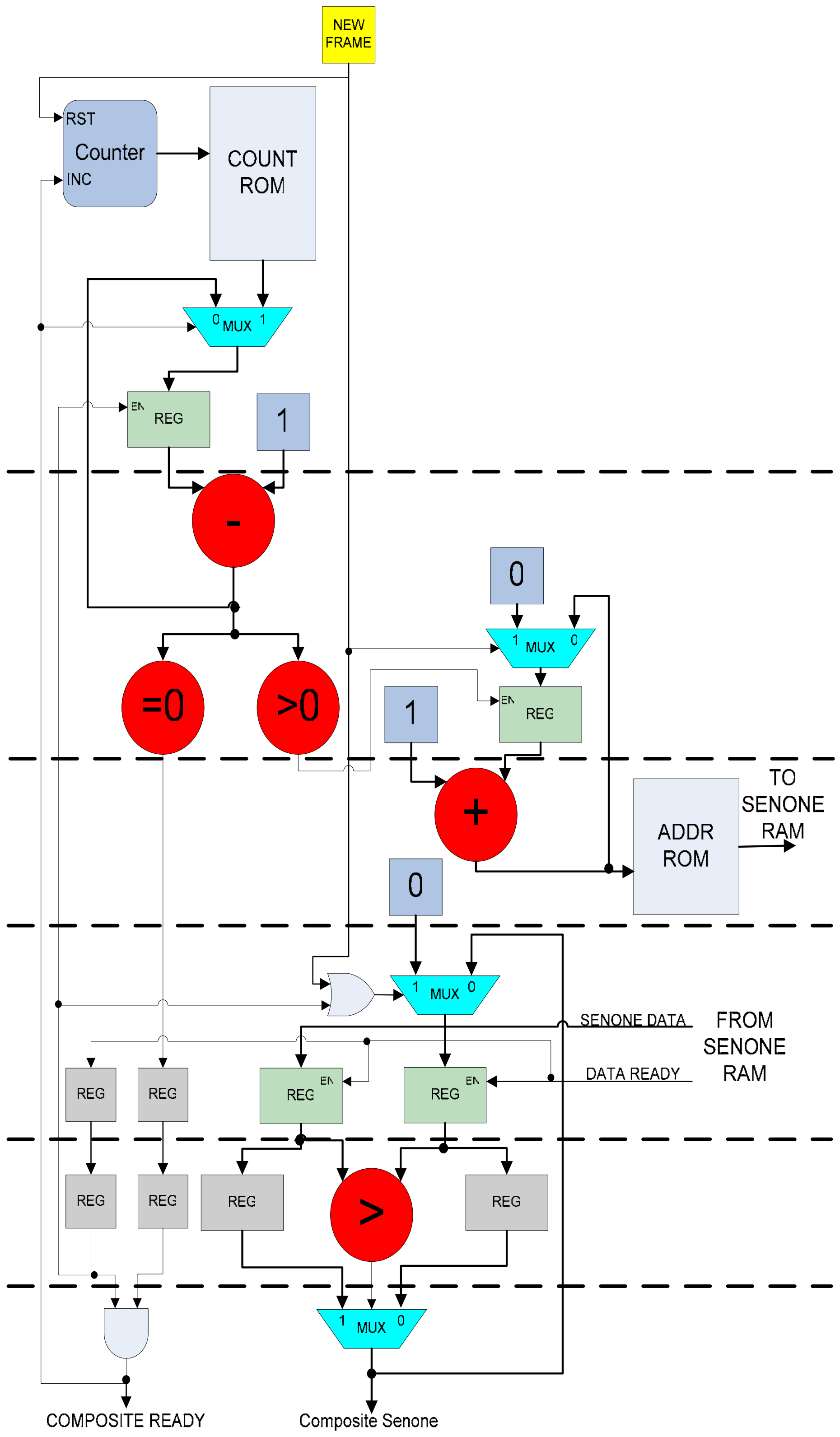


**Figure 26.** Data Flow Graph for Composite Senone Calculation

## 4.5 HARDWARE PERFORMANCE RESULTS

Having completed both software and hardware systems for Acoustic Modeling it was then necessary to analyze the performance of the derived hardware. In Section 4.2 the potential for parallelism is shown through the equations and then in Section 4.3 this parallelism is illustrated through the use of element-wise matrix operations. For the hardware this parallelism was exploited to created the pipelined system described in 4.4 and while the derive pipe does not take advantage of all the parallelism available it does utilize enough to ensure its ability to operate in real-time.

As discussed in Section 4.4.1 the Gaussian distance pipe was extended by three cycles to keep it in synch with the log-add LUT operation. To ensure that this did not adversely affect the system an experiment was conducted to observe the effects of pipelining and retiming on the $f_{max}$ of the system. To do this, extra registers were put into the design and the pipelining and retiming options were enabled in the synthesis tool. When the synthesis was executed the tool was able to move these registers to what it determined were the optimal locations in the design, minimizing the amount of analysis done by the designer. Our primary target in these experiments was the Xilinx® Virtex-4 SX35 FPGA due to its high performance and large number of embedded DSP (DSP48) & RAM (BRAM) cells. For sake of comparison we also targeted a smaller device, the Xilinx® Spartan-3, a 90-nm FPGA with embedded 18x18 multipliers. The graph in Figure 27 shows the results of these experiments for a pipeline between 7 and 19 stages deep, with the dotted lines representing the projected $f_{max}$ of the system from the synthesis engine and the solid lines representing the post place-and-route $f_{max}$. While the $f_{max}$ obtained for the Vertix-4 device is noticeably higher than the $f_{max}$ of the Spartan-3 devices, it is also observed that increasing the number of pipeline stages improves the speed of the Spartan-3 device more significantly than the speed of the Virtex-4 device. It is also observed that for both devices there are an optimum number of stages beyond which the performance of the device actually degrades due to the increased amount of area consumed by the design.

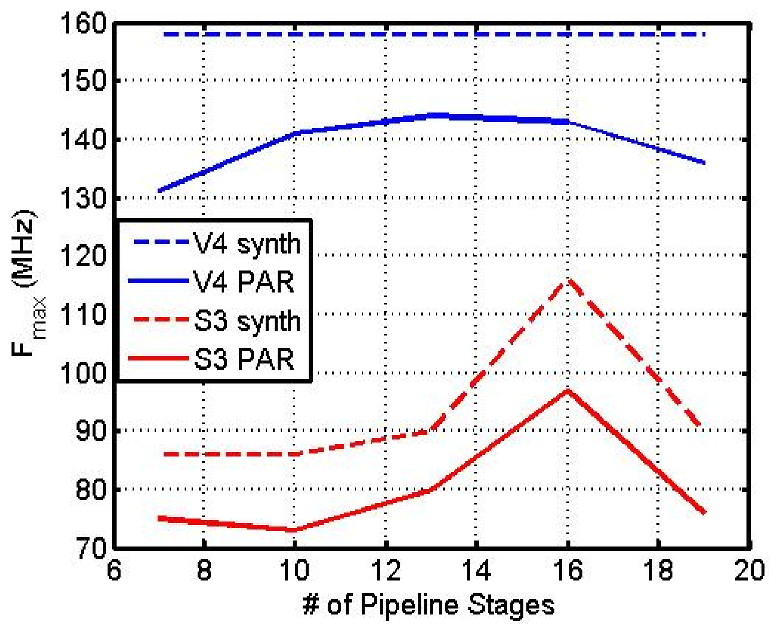


**Figure 27. Analysis of $f_{max}$ vs. Pipeline Stages for Virtex-4 SX and Spartan-3 FPGAs**

Another interesting result of these experiments was that regardless of the number of pipeline stages the projected synthesis speed for the Virtex-4 did not change. This implies that even when the pipeline is configured with the minimum number of allowable stages, the results of the pipelining and retiming processes are the same. The post place-and-route timing results for the Virtex-4 however; do change with the number of pipeline stages implemented. Since we know we are utilizing the embedded DSP slices on the chip and we can trace the critical path of the circuit we can conclude that the physical distance between two individual DSP cells is great enough that adding extra registers along the path will in fact increase the speed of the design. Figure 10 further shows that when targeting the Virtex-4, a 10-stage pipe will provide an acceptable operating frequency for the system with only minor improvements being gained with each additional pipe stage. This is a promising result because we know the depth of the Log-Add LUT is 10 cycles as well, allowing us to match the depths of the two pipelines with out having to sacrifice a considerable amount of speed.

By setting the final depth of the Gaussian Distance/Log-Add logic at 20 cycles the total amount of parallelism utilized was also set. Given that only a single memory bank is used to drive the pipe and that the pipe is 20 stages deep, the total parallelism utilized in the hardware is fixed to being able to operate on 20 things at once. Multiple memory banks and replicated logic could easily increase the amount of parallelism, but given that a single memory configuration provides enough computational effort to complete the process within our real-time operation constraint no additional time was spent investigating this option. In addition to the experiments described above all individual components of the Acoustic Modeling block were synthesized and routed on the chip to fully characterize their performance. Table 4 summarizes the results of these tests and makes note of any special ASIC cells used by each stage of the design.

**Table 4.** Summary of Synthesis and Place-and-Route Results for Virtex-4 SX35

| Component | Synthesis (MHz) | Place-and-Route (MHz) | AREA |
|---|---|---|---|
| **Gaussian Dist. Pipe** | 157 | 145 | 6 DSP Tiles, 411 Slices |
| **Log-Add LUT** | 164 | 150 | 13 BRAMs, 307 Slices |
| **Find Max** | 181 | 160 | 90 Slices |
| **Normalizer** | 197 | 172 | 144 Slices |
| **Composite Senone Calc.** | 197 | 140 | 2 BRAMs, 147 Slices |
| **AM Block (TOTAL)** | **164** | **125** | **6 DSP Tiles, 30 BRAMs, 1328 Slices** |

Looking at the table it can be seen that all portion of the logic operate above the 100MHz restriction placed on the design, allowing the system to operate in real-time. As mentioned in Chapter 3, 100MHz with a 10ms input window leads to a 1 million cycle budget for completion of all operations. The AM unit designed is capable of executing all of its operations in approximately 650,000 cycles, leaving the remaining 350,000 cycles for the processing of the Phoneme Evaluator and the Word Modeler blocks. As mentioned in Chapter 3 the hardware was verified against both our SPHINX baseline and the MATLAB code by checking the outputs of the software against the waveform output of Modelsim. These comparisions were made on random sample frames out of a set of 300 frames on continuous speech. By comparing the results of the hardware to the results of the software at various points in the progression through incoming utterances the flow of data in the system as well as the correctness of the outputs could be verified.

In order to further increase the performance of the SoC we hand coded our own placement constraint files for each block of the design. By putting restrictions on the placement of each block we were able to ensure minimal delays between blocks as well as with in the block itself, and were also able to confine portions of the design that utilized special ASIC cells to the slices of logic adjacent to those cells. The graph in Figure 28 examines the post PAR speeds obtained for Gaussian Distance pipe when constrained to various percentages of the FPGA. The effects of placement constraints are quite evident from Figure 28, noting a 14 MHz in performance between an unconstrained and a fully constrained design. Figure 29 shows the final floor-plan for the entire Acoustic Modeling Pipeline.

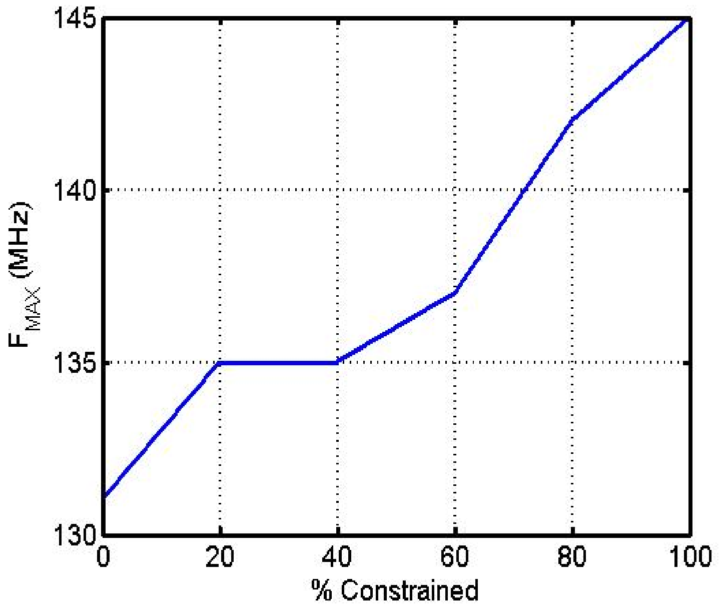


**Figure 28.** Speed Improvement vs. Percentage of Design Constrained for Gaussian Distance Pipe

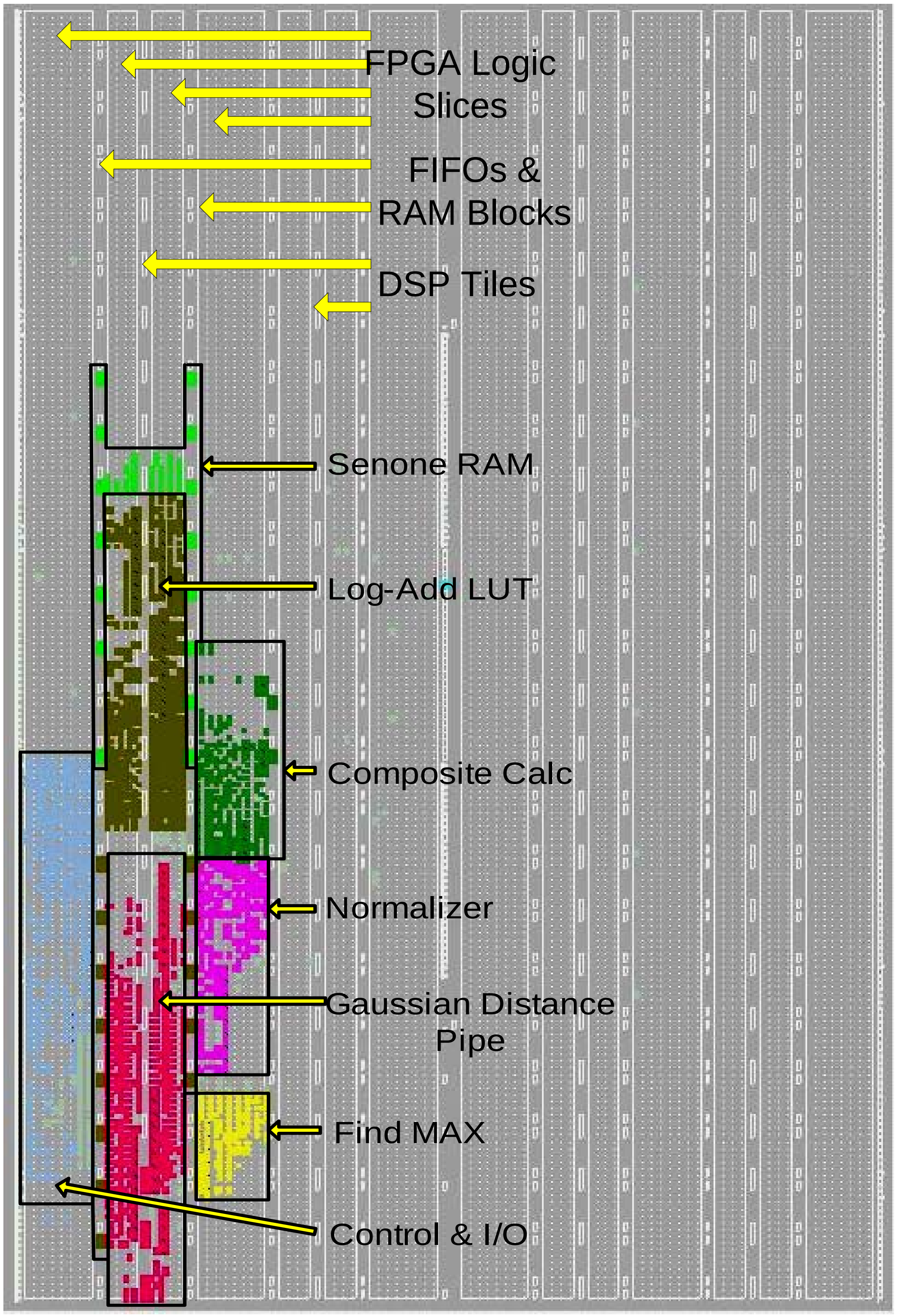


**Figure 29. Post Place-and-Route Layout for AM Pipeline on a Virtex-4 SX35 FPGA**

# 5.0 PHONEME EVALUATION

The process of phoneme evaluation (PE) in modern ASR systems refers to the evaluation of a predefined set of Hidden Markov Models, with respect to the newly acquired set of senones provided by the Acoustic Modeler. Unlike AM, the amount of work done by the phoneme evaluator at any one point in time is dependent on the number of active inputs to the block, and the architecture for the block must reflect this dynamic nature.
The HMMs being evaluated during PE represent time-varying statistical models for set of phonetic units allowable in the system dictionary. For any one frame, all of the active HMMs must be evaluated and pruned based on a basic beam pruning algorithm to ensure that only promising data is forwarded in the system for future processing. This chapter will take a closer look at the operations necessary to calculate a single HMM and then take a step backward to examine how this single calculation fits into the overall architecture of the block.

## 5.1 HIDDEN MARKOV MODELS

Hidden Markov Models (HMMs) are just one of an array of non-time-dependent (NTD) statistical models used by engineers to model many of the phenomena observed in the world around us. Unlike time-dependent models that rely on the regularity of the data as the basis of the model, HMMs and other NTD models are adapted to observe and emulate processes that evolve over time and posses no core regularity. The human voice is one such phenomenon and has been shown by numerous research projects to be quite accurately modeled by HMMs [36, 38, 46, 48]. One of the key concepts that make HMMs so useful is their ability to maintain a potentially unlimited number of previous transitions even though the core process only has a single cycle memory. To better understand this concept it is first necessary to understand the underlying features of a Markov chain and then to present a formal definition of a HMM and the variables associated with it.

### 5.1.1 Mathematics

A Markov chain is a special set of random variables where each variable has some knowledge of the variable before it [2]. Given that $X_1, X_2, \ldots, X_n$ is a sequence of random variables all with values in the same finite alphabet, then Bayes formula can be used to represent the probability associated with the set as shown in Equation 19.

$$P(X_1, X_2, \ldots, X_n) = \prod_{i=1}^{n} P(X_i \mid X_1, X_2, \ldots, X_{i-1}) \qquad \text{[Eq. 19]}$$

If it is found that the input sequence fits the equation given in Equation 20 then the sequence can be considered a Markov Chain and Equation 19 can be rewritten as Equation 21.

$$P(X_i \mid X_1, X_2, \ldots, X_{i-1}) = P(X_i \mid X_{i-1}) \qquad \text{[Eq. 20]}$$

$$P(X_1, X_2, \ldots, X_n) = \prod_{i=1}^{n} P(X_i \mid X_{i-1}) \qquad \text{[Eq. 21]}$$

In order to introduce more freedom into the system the actual state sequence can be hidden from the observer and only the states themselves allowed to generate observable data. To apply this freedom to Equation 21, the following lemma must be presented to define the new variable necessary for the process. The terms defined in the lemma are taken from the definition of an HMM as defined by Jelinek [2].

*Let:* ***y = {0,1,…,b-1}*** *➔ the output alphabet*

*Let:* ***L = {1,2,…,c}*** *➔ the state space*

*Let:* ***p(s'|s)*** *be the probability distribution of transitions between states*

*Let:* ***q(y|s,s')*** *be the output probability distribution associated with transitions from state s to state s'*

With these four terms it is now possible to define the probability of observing the output string $y_1, y_2, \ldots, y_k$ for a given HMM. This equation is given in Equation 22 as a modified version of Equation 21 where we are now observing the output sequence $y_1, y_2, \ldots, y_k$ instead of the actual state sequence $X_1, X_2, \ldots, X_n$.

$$P(y_1, y_2, \ldots y_k) = \sum_{s_1, \ldots s_k} \prod_{i=1}^{k} p(s_i \mid s_{i-1}) q(y_i \mid s_{i-1}, s_i) \qquad \text{[Eq. 22]}$$

From Equation 22 it can be seen that the output sequence is determined by the summation of the scores for each state given their predecessor state only. This means that while only the values from the previous time need to be maintained for processing in the current frame, these values contain information about the entire preceding chain of inputs, enhancing the memory capabilities of the model.

### 5.1.2 HMM Topologies

Understanding the basic mathematics at play, the properties of a HMM can begin to be visualized and possible state orientations can be surmised. While by definition any combination of states can be used to form a HMM as long as the specified criterion are met, for purposes of statistical modeling there are a few common classes of HMMs that are widely used through out the world [57]. The simplest configuration is what's known as a linear topology where each state contains only a self-transition and a next state transition, and is shown in Figure 30.

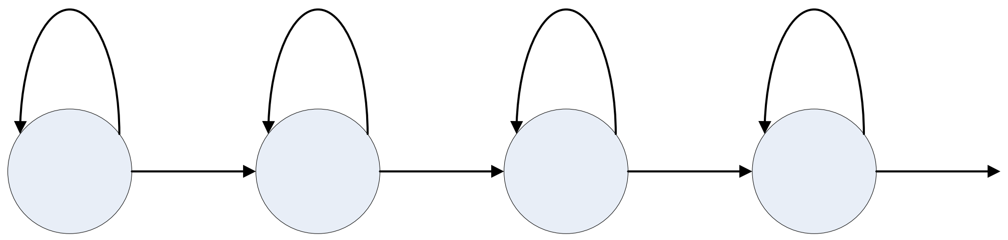


**Figure 30.** Linear HMM Topology

Two other forms of a linear topology are known as the left-to-right and Bakis topologies, differing only in the number of sequential states that can be skipped while traversing the HMM. In a Bakis topology each state may either transition to itself, the state immediately next to it, or the state located one hop away. In a left-to-right topology however each state can transition to itself or any of the following states in the model [12]. Figure 31 helps to visualize the distinction between the two models showing a Bakis topology HMM with the addition left-to-right topology transition shown as a dotted line.

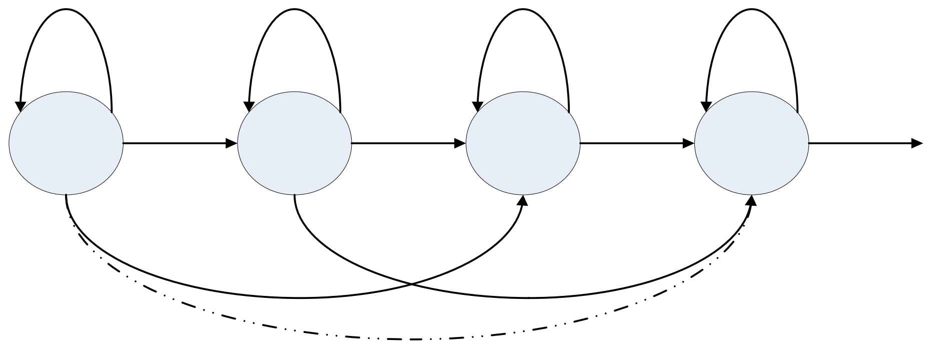


**Figure 31.** Comparison of Bakis and Left-to-Right Topology HMMs

Another basic HMM topology is called the Alternative paths topology which utilizes multiple distinct sets of states to transition from the beginning of the model to the end. Within an alternative path topology both Bakis and left-to-right transitions can be applied to all but the start and end nodes helping to further extend the complexity of the model. Figure 32 illustrates the basic alternative path topology with the additional transitions shown as dotted lines.

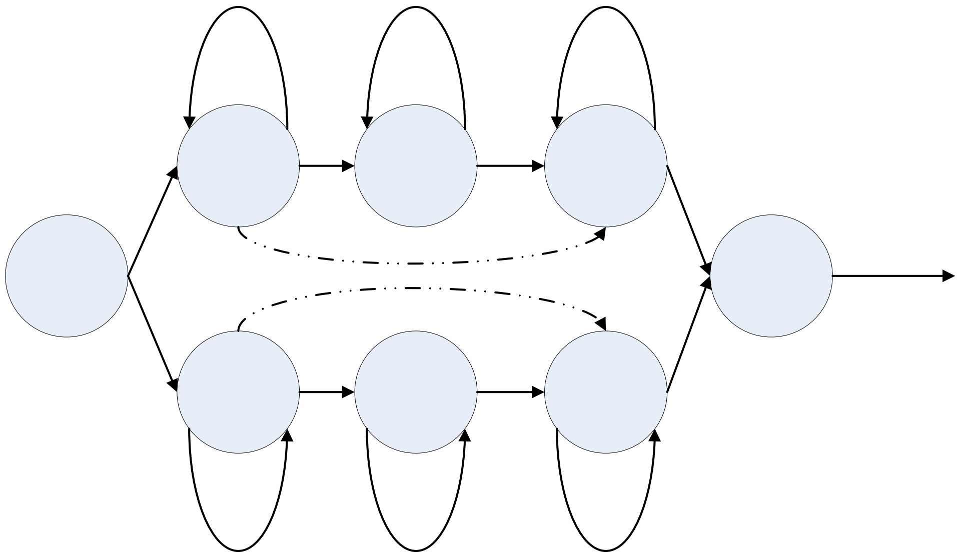


**Figure 32.** Sample Alternative Paths HMM Topology

While all of the topologies shown so far contain only forward transitions this is not a requirement of HMM topologies although it may be a requirement of the systems using them. In statistical processes such as speech recognition it is known that speech cannot "go backward" therefore having reverse transitions make no real sense. Other systems do have such constraints and in these cases it is highly desirable to use a class of topologies known as Ergodics to represent the data. Ergodic HMM topologies have no restrictions on the allowable transitions as shown in the example topology of Figure 33.

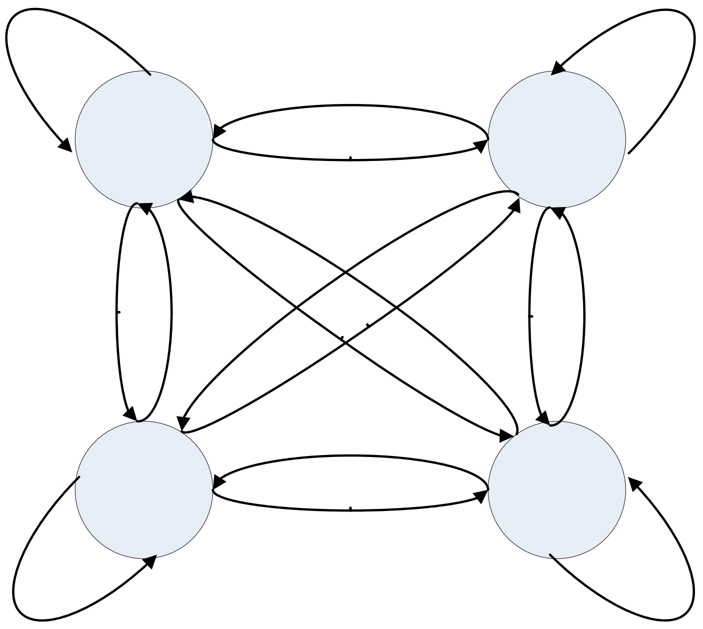


**Figure 33.** Sample Ergodic HMM Topology

### 5.1.3 Viterbi Searching

In all of these topologies when the HMM is left unconstrained, all information about all possible paths through the HMM must me retained. This can lead to a very large amount of data needing stored the longer one stays in a single HMM. This can be quite problematic in systems where memory is a constraint or where many HMMs all need to be run in a real-time environment. To help mitigate this problem a number of optimization algorithms have been proposed, one of the most popular of which is the Viterbi search algorithm [12]. The Viterbi search algorithm simply states that at any time while evaluating an HMM, if two paths converge only the best path need be maintained. By only keeping the best possible path through the HMM at any time, the amount of data needing to be updated is significantly reduced while the overall impact on the accuracy of the model is impacted very little. Figure 34 shows a sample HMM oriented on the y-axis of a trellis to help visualize the number of possible paths through an HMM.

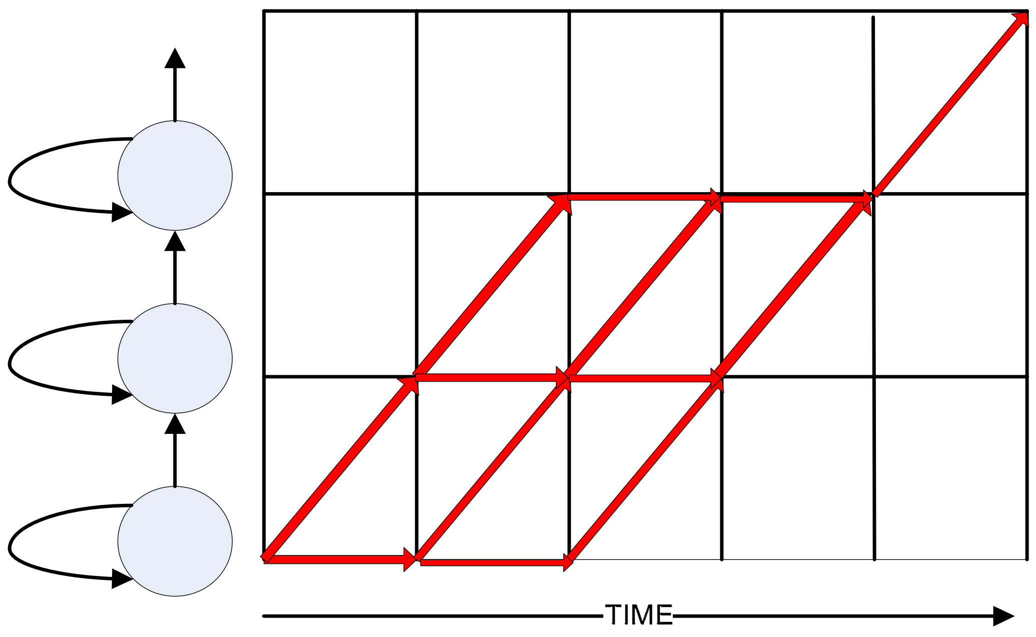


**Figure 34.** Sample HMM Trellis

Looking at the sample trellis it is observed that even for only five time ticks there are already 6 possible paths through the HMM. If the Viterbi algorithm were not used all of these paths would need maintained throughout the entire evaluation of the HMM and only upon exiting the HMM would the best path be determined. With the Viterbi algorithm however only the information about the best current path need to be retained. This concept is visualized in Figure 35, with the Viterbi path shown as a sold line and all other transitions shown as dotted lines.

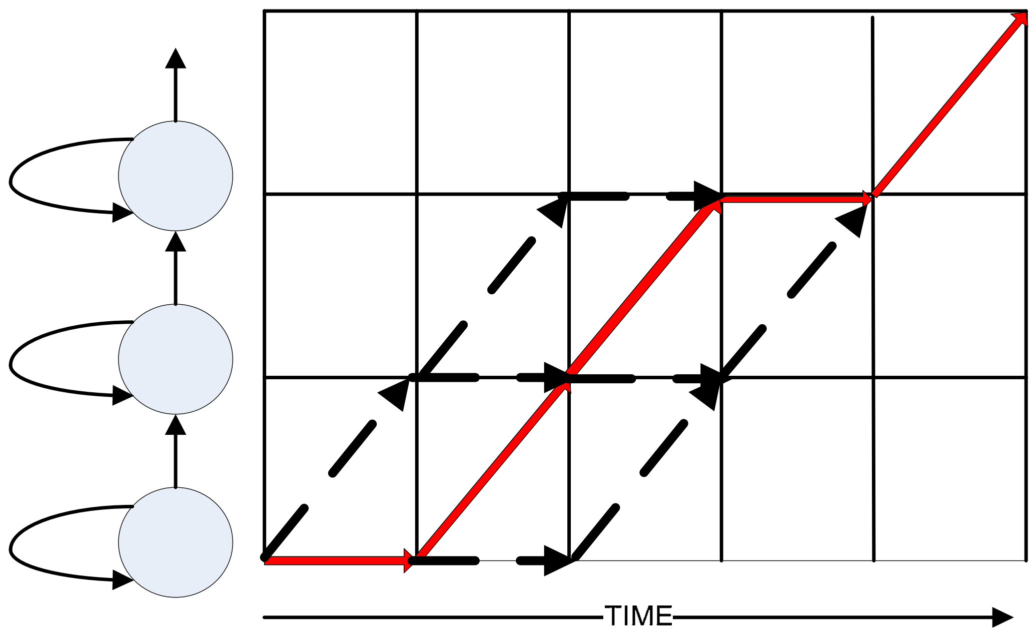


**Figure 35. HMM Trellis with Viterbi Algorithm**

With a basic understanding of HMMs and their ability to model random processes the specific nature of the calculations being performed in the derived architecture can be investigated and understood. The remaining sections of this chapter will focus on the calculations necessary to calculate the HMMs in the systems as well as their implementation as both fully-vectorized software and a hardware co-processor.

## 5.2 DESCRIPTION OF CALCULATIONS

During each new frame in speech recognition each of the active phonemes must be evaluated and pruned based on the best score for the entire active set. Each HMM requires the retrieval of a large amount of static information as well as information relating to the previous state of the HMM and the present state of the inputs to the HMM. This results in a process that is dominated largely by memory access with only a few compact calculations being performed to update the database. While this advantageous in that only simple arithmetic units need to be used to perform the operations, it has the distinct disadvantage that each simple calculation requires complex memory access, leading to sub-optimal conditions for highly pipelined designs.

Each HMM calculation begins by obtaining the token for the HMM from the active cue. The tokens relate to a specific HMM location in the word tree, described in Chapter 5, and are important because while it is possible to have multiple copies of an HMM in the word tree, each one must be treated uniquely due to different contexts. Using tokens allows each node in the tree to have a unique ID even though many nodes may contain the same HMM and this provides the necessary uniqueness to the problem. In the proposed implementation this token is used to address a large shared RAM as well as ROM containing additional pointers needed for the HMM. The RAM provides information about the previous state of the HMM and will be updated by the Phoneme Evaluator once the calculation is complete. The ROM contains pointers to the senones needed from Acoustic Modeling and the transition scores needed to evaluate the HMM. The senone pointers are then passed to the senone RAM and the necessary scores are retrieved. At the same time the senone scores are being obtained the transition scores are obtained from a set of small local ROMs. Figure 36 helps to visualize the full scope of the data access required for a calculation of a single HMM.

Once the previous state information, the current senone scores, and the transition scores have been acquired the calculation of the HMM may occur. To keep the calculations in line with the SPHINX models being used as the baseline a 3-state Bakis topology HMM was implemented using the Viterbi search algorithm. The equations necessary to calculate an HMM are shown as equations 23-27 with a sample HMM shown in Figure 37 with all values labeled for clarity.

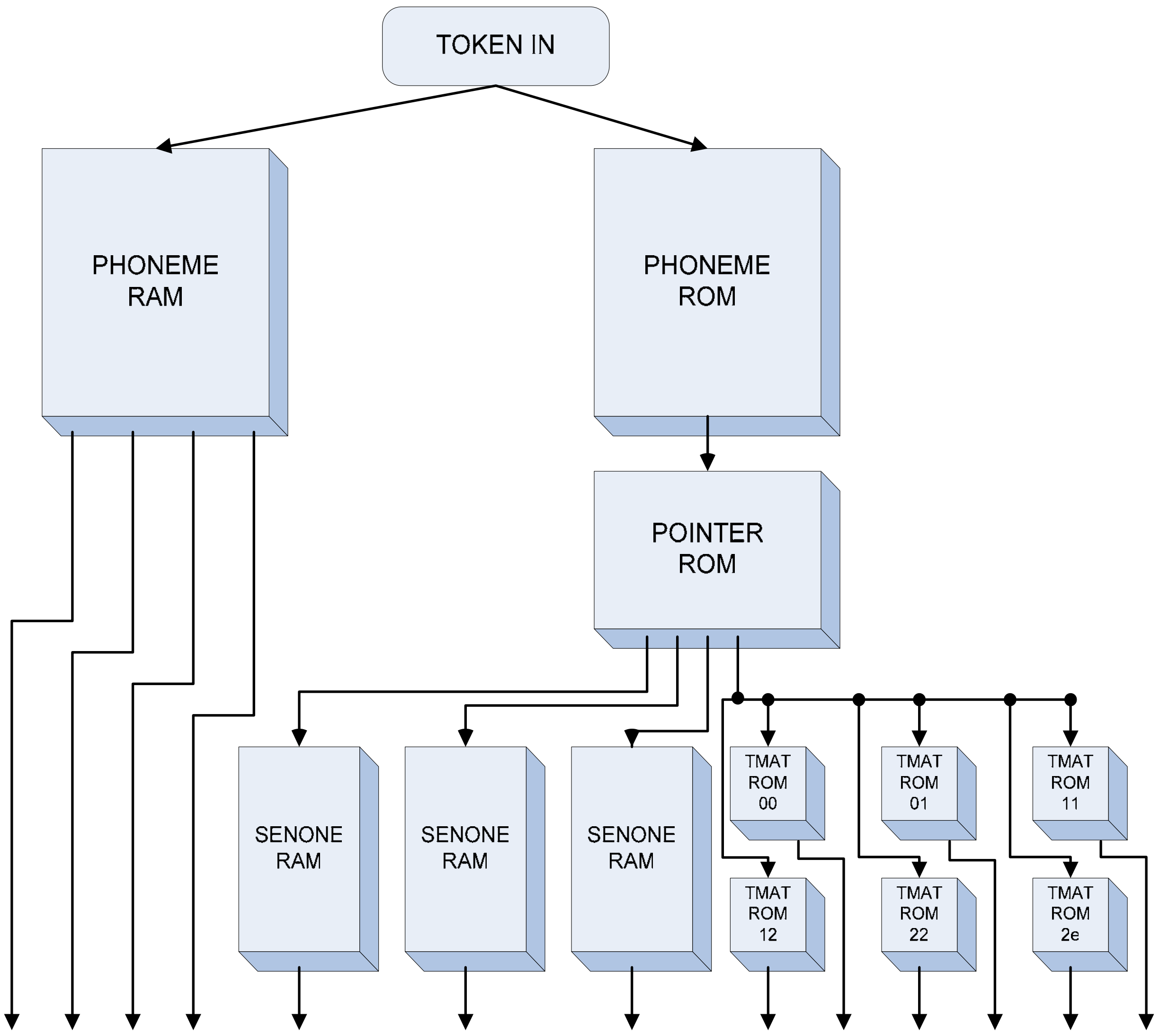


**Figure 36.** Data-Flow Diagram for Memory Access for HMM Calculation

$$H_1(t) = MAX\{H_0, H_1(t-1) + T_{00}\} + S_0(t) \quad \text{[Eq. 23]}$$

$$H_2(t) = MAX\{H_1(t-1) + T_{01}, H_2(t-1) + T_{11}\} + S_1(t) \quad \text{[Eq. 24]}$$

$$H_3(t) = MAX\{H_2(t-1) + T_{12}, H_3(t-1) + T_{22}\} + S_2(t) \quad \text{[Eq. 25]}$$

$$H_{BEST}(t) = MAX\{H_1(t), H_2(t), H_3(t)\} \quad \text{[Eq. 26]}$$

$$H_{EXIT}(t) = H_3(t) + T_{2E} \quad \text{[Eq. 27]}$$

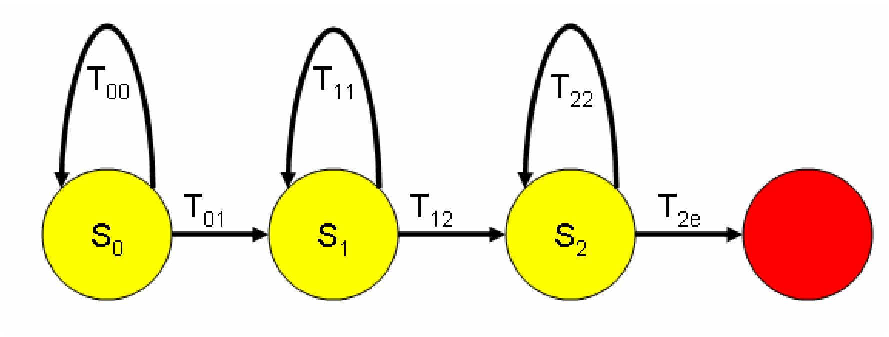


**Figure 37.** HMM Topology with Labeled Values

Unlike Equations 24 and 25, Equation 23 shows that one of the input terms, $H_0$, is not time dependent. This term is defined as the input probability to the HMM and only has a valid value during the single time tick when the HMM is actually entered. This value is a function of the score of the previous node the word tree and will take on a value of negative infinity for all time except the time of the transition from one node to another. For this reason the term is not considered strictly time-dependent and is treated differently from the other previous state scores, $H_1$, $H_2$, & $H_3$.

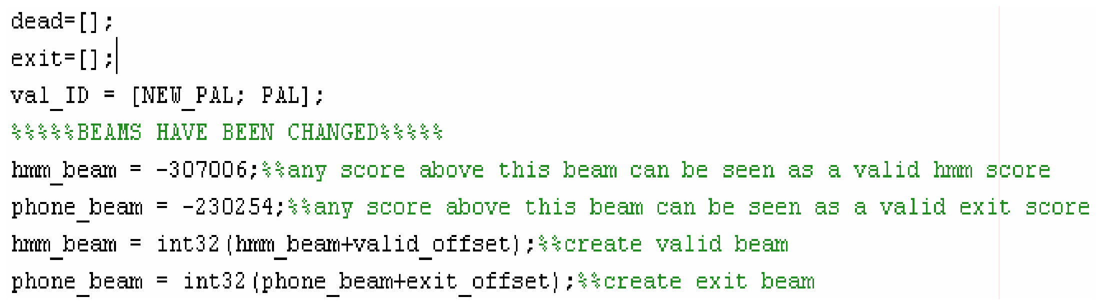
The $H_{BEST}$ & $H_{EXIT}$ scores are calculated along with the other values but are not used until during the pruning phase of the phoneme evaluation. Once all HMM have been calculated the best of the $H_{BEST}$ values is found and used to define the beams used during pruning. This global best score is used to calculate both the 'valid beam' and the 'exit beam' which get compared to the $H_{BEST}$ and $H_{EXIT}$ values respectively for each active HMM. If a given HMMs best score is above the valid beam then it will remain in the active cue for the next frame, and if its exit score is above the exit beam then it will also be placed in the exit cue for processing by the word model. In the case that the best score for the HMM is not above the valid beam then the HMM will be removed from the active cue and placed into the dead cue for later processing by the word model. After all HMMs have been pruned and their tokens placed in the appropriate cues the process of phoneme evaluation has been completed and the word modeler may begin its operation. Having described the steps necessary to complete the phoneme evaluation it is next necessary to describe the vector code written to execute the desired operations.

## 5.3 MATRIX MATLAB REPRESENTATION

Unlike the calculations necessary for Acoustic Modeling, Phoneme Evaluation (PE) requires no multi-element summations, allowing for the creation of fully vectorized MATLAB code requiring no looping to calculate the HMM dataset. By organizing the data for the calculations into large matrices, all off the HMMs can be calculated in a single set of operations. Further, the data management problem described in section 5.2, can be reduced to a simple set of dynamic pointers allowing for rapid random accesses on very large arrays.

Similar to the Acoustic Modeling block the majority of the parallelism found in this portion of the algorithm is in the ability to calculate all HMMs independently of one another. While certain operations within a given HMM may also be calculated simulatneaously it is the ability to calculate all HMMs at once that provides the most benefit. Unlike Acoustic Modeling however, the workload for Phoneme Evaluation varies from frame to frame meaning that the potential parallelism in the task is not constant. The token passing scheme helps to manage this variability by maintaining a series of token vectors indicating the present workload of the system. While this provides an effective means of obtaining numerous data elements in a single line of MATLAB code, it also limits any potential hardware to having to operate sequential to obtain the data given that only a single memory bank exists. The impact of this limitation are described in more detail in Section 5.4.

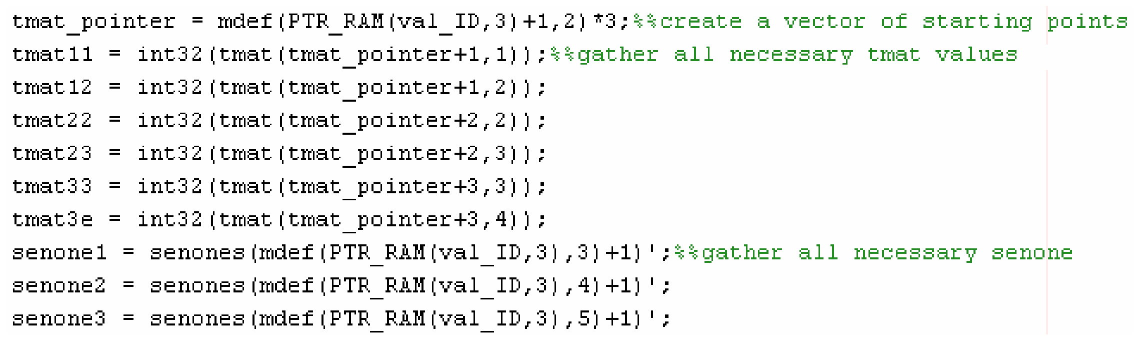
The first step in performing PE is to reset the output token vectors, merge the input token vectors into a single large list, and pre-calculate the beam offsets to be used during pruning. For each beam used in the pruning process there is both a static beam and a variable offset. The static portion of the beam relates to the value needed to achieve maximum recognition accuracy. This will apply the strictest thresholds to the active scores creating a system capable of only seeing words that are observed with very high probability. While this can be very beneficial for situations where accurate recognition is absolutely critical, it can also cause the performance of the system to degrade very quickly if any noise is present in the environment. To counter this effect the user may want to add some variable offset to the static beam that can be tailored to the specific application. These initialization steps can be performed quite simply and the necessary code is shown as Figure 38.



```
dead=[];
exit=[];
val_ID = [NEW_PAL; PAL];
%%%%%BEAMS HAVE BEEN CHANGED%%%%%
hmm_beam = -307006;%%any score above this beam can be seen as a valid hmm score
phone_beam = -230254;%%any score above this beam can be seen as a valid exit score
hmm_beam = int32(hmm_beam+valid_offset);%%create valid beam
phone_beam = int32(phone_beam+exit_offset);%%create exit beam
```

**Figure 38.** Code for PE initialization

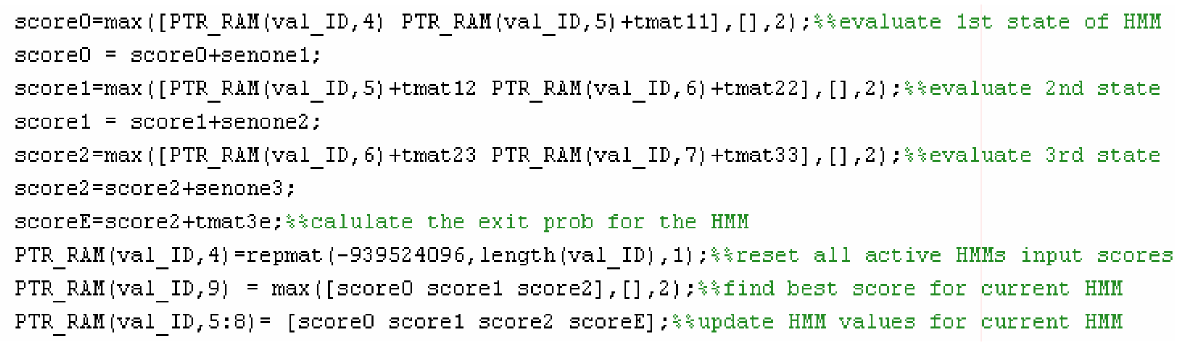
After initialization, the transition matrix scores must be collected into compact vectors for use in the calculation. This process involves using the input token vector to obtain the necessary transition matrix pointer IDs and then using those IDs to obtain the actual transition scores for each arc of each HMM. A single vector is created for each transition in the HMM containing as many values as there are active HMMs in the system. At the same time the transition matrix scores are being obtained, the senones scores corresponding to each HMM can be obtained from the Acoustic Modeler. As with the transition matrix scores, the input token vector is used to obtain a list of senone IDs that is then used to obtain a list of senones scores. The code for the data allocation process is shown in Figure 39.



```
tmat_pointer = mdef(PTR_RAM(val_ID,3)+1,2)*3;%%create a vector of starting points
tmat11 = int32(tmat(tmat_pointer+1,1));%%gather all necessary tmat values
tmat12 = int32(tmat(tmat_pointer+1,2));
tmat22 = int32(tmat(tmat_pointer+2,2));
tmat23 = int32(tmat(tmat_pointer+2,3));
tmat33 = int32(tmat(tmat_pointer+3,3));
tmat3e = int32(tmat(tmat_pointer+3,4));
senone1 = senones(mdef(PTR_RAM(val_ID,3),3)+1)';%%gather all necessary senone
senone2 = senones(mdef(PTR_RAM(val_ID,3),4)+1)';
senone3 = senones(mdef(PTR_RAM(val_ID,3),5)+1)';
```

**Figure 39.** Code for Data Allocation in PE

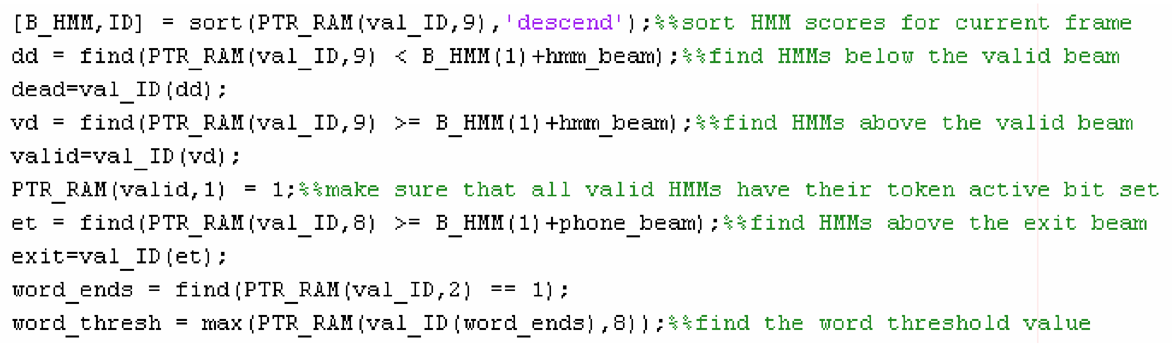
Once all of the data has been collected for the HMM evaluations Equations 23-27 can be executed for all of the active HMMs. During calculation the scores are stored into temporary vectors allowing for the previous times scores to be read from the database and used before being overwritten by the current times results. For each state of the HMM the best score for the transition is found first and then the senone score for the state is added to the result. After calculating all the current states for each HMM the $H_{BEST}$ and $H_{EXIT}$ values are found and all the data is then written back to the central database. Additionally the input score for each HMM must be reset to prevent it from being seen as new again in the next frame. Figure 40 shows the code used for the calculation and storage of all HMM values.



```
score0=max([PTR_RAM(val_ID,4) PTR_RAM(val_ID,5)+tmat11],[],2);%%evaluate 1st state of HMM
score0 = score0+senone1;
score1=max([PTR_RAM(val_ID,5)+tmat12 PTR_RAM(val_ID,6)+tmat22],[],2);%%evaluate 2nd state
score1 = score1+senone2;
score2=max([PTR_RAM(val_ID,6)+tmat23 PTR_RAM(val_ID,7)+tmat33],[],2);%%evaluate 3rd state
score2=score2+senone3;
scoreE=score2+tmat3e;%%calulate the exit prob for the HMM
PTR_RAM(val_ID,4)=repmat(-939524096,length(val_ID),1);%%reset all active HMMs input scores
PTR_RAM(val_ID,9) = max([score0 score1 score2],[],2);%%find best score for current HMM
PTR_RAM(val_ID,5:8)= [score0 score1 score2 scoreE];%%update HMM values for current HMM
```

**Figure 40.** Code for HMM Calculation and Data Storage

Having stored the new HMM scores in the database the pruning operation are able to begin. These operations can be performed very efficiently with the use of the *'sort()'* and *'find()'* functions in MATLAB. Both of these functions are specifically designed to operate on large arrays of numbers and can be given a number of different arguments to allow them to function in different capacities. The first step in pruning is to find the global best state score that will be used as the origin of the beam. Once this is found the beam offsets are applied and the database is searched to find the list of HMMs passing each beam. These lists are then assigned to their appropriate token vectors based on whether they are to remain active for the next time, be reset to inactive for the next time, or be kept active and passed onto the word model for additional processing. While pruning is occurring an additional beam value must be calculated for use by the Word Modeler. Inside the HMM database each token is flagged with a special bit to indicate whether it represents the end of a word, or some point inside a word. All scores corresponding to end-of-word HMMs must be compared and the best of the set passed on to the Word Modeler. This value will be used to determine if the observed word was seen with a high enough probability to be output from the system and is described in more detail in Chapter 6. Figure 41 shows the MATLAB code used to prune the HMM database.



```
[B_HMM,ID] = sort(PTR_RAM(val_ID,9),'descend');%%sort HMM scores for current frame
dd = find(PTR_RAM(val_ID,9) < B_HMM(1)+hmm_beam);%%find HMMs below the valid beam
dead=val_ID(dd);
vd = find(PTR_RAM(val_ID,9) >= B_HMM(1)+hmm_beam);%%find HMMs above the valid beam
valid=val_ID(vd);
PTR_RAM(valid,1) = 1;%%make sure that all valid HMMs have their token active bit set
et = find(PTR_RAM(val_ID,8) >= B_HMM(1)+phone_beam);%%find HMMs above the exit beam
exit=val_ID(et);
word_ends = find(PTR_RAM(val_ID,2) == 1);
word_thresh = max(PTR_RAM(val_ID(word_ends),8));%%find the word threshold value
```

**Figure 41.** Code for HMM Pruning

The calculation of the word threshold and the creation of the *dead, valid,* and *exit* token vectors marks the end of the PE. From here the *dead* and *exit* vectors are passed to the Word Modeler along with the word threshold and the next stage of the recognition process may begin.

## 5.4 HARDWARE ARCHITECTURE

When implementing PE in hardware the majority of the logic necessary resides in the large amount of constant data that must be stored and retrieved in order to process an HMM. As was shown in Figure 35 in section 5.2, the process of retrieving the constant data requires the use of numerous cascaded RAMs / ROMs. The TOKEN IN (HMM ID) input spawns the addresses for 4 separate look-up Tables, one for the transition score ROMs, and one for each of the senones needed for the HMM. Within the POINTER ROM structure of Figure 35 resides the TMAT ID ROM which serves as a decoder to map one of a large set of HMM IDs to one of the relatively small set of TMAT IDs. This single TMAT ID is then used to address six TMAT score ROMs in parallel in order to decrease the latency of the data access. The other three LUTs receive the same HMM ID, but are used to decode the appropriate senone IDs needed for the HMM. In an effort to decrease latency, these senone IDs are found in parallel and output from the block back to the shared senone RAM described in Chapter 4. Once the TMAT scores have been retrieved along with the current senone scores and previous state scores, the actual processing of the HMM can begin. As mentioned previously the remainder of the calculation can be implemented as a high throughput pipeline and is described in detail in the following Sections.

### 5.4.1 HMM Control Logic

While developing the architecture it was necessary to consider the fact that unlike AM, the amount of work that needs done at any one time is variable and therefore some control must be included to monitor the amount of active data in the system. The data needing processed by the PE is most efficiently managed by a series of FIFOs containing lists of active HMM IDs. Specifically, data entering PE is provided via either the new phoneme active list (nPAL) FIFO, and data exiting PE is written to either the exit (VALID) FIFO, the inactive (DEAD) FIFO, or the phoneme active list (PAL) FIFO. Figure 42 illustrates the relationships between these FIFOs.

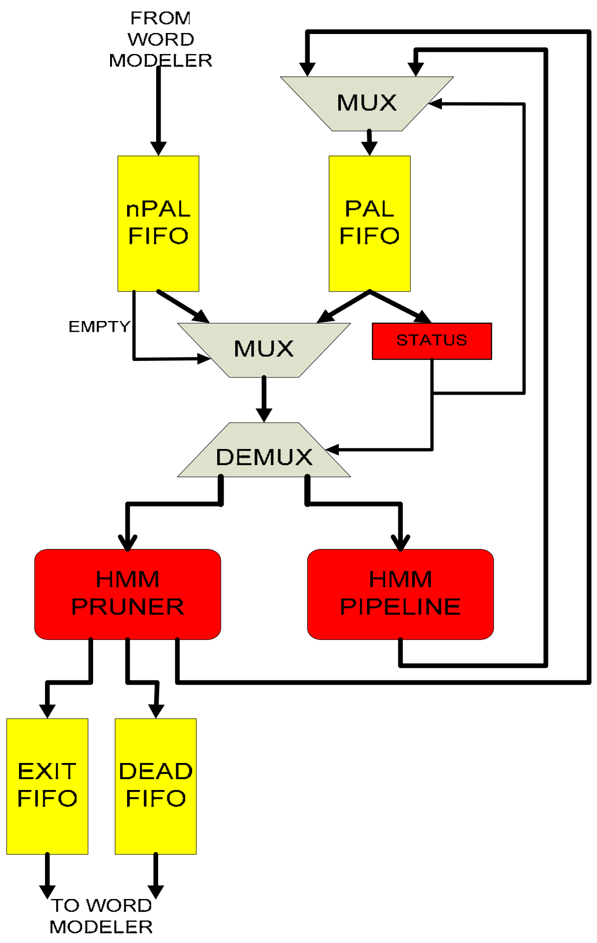
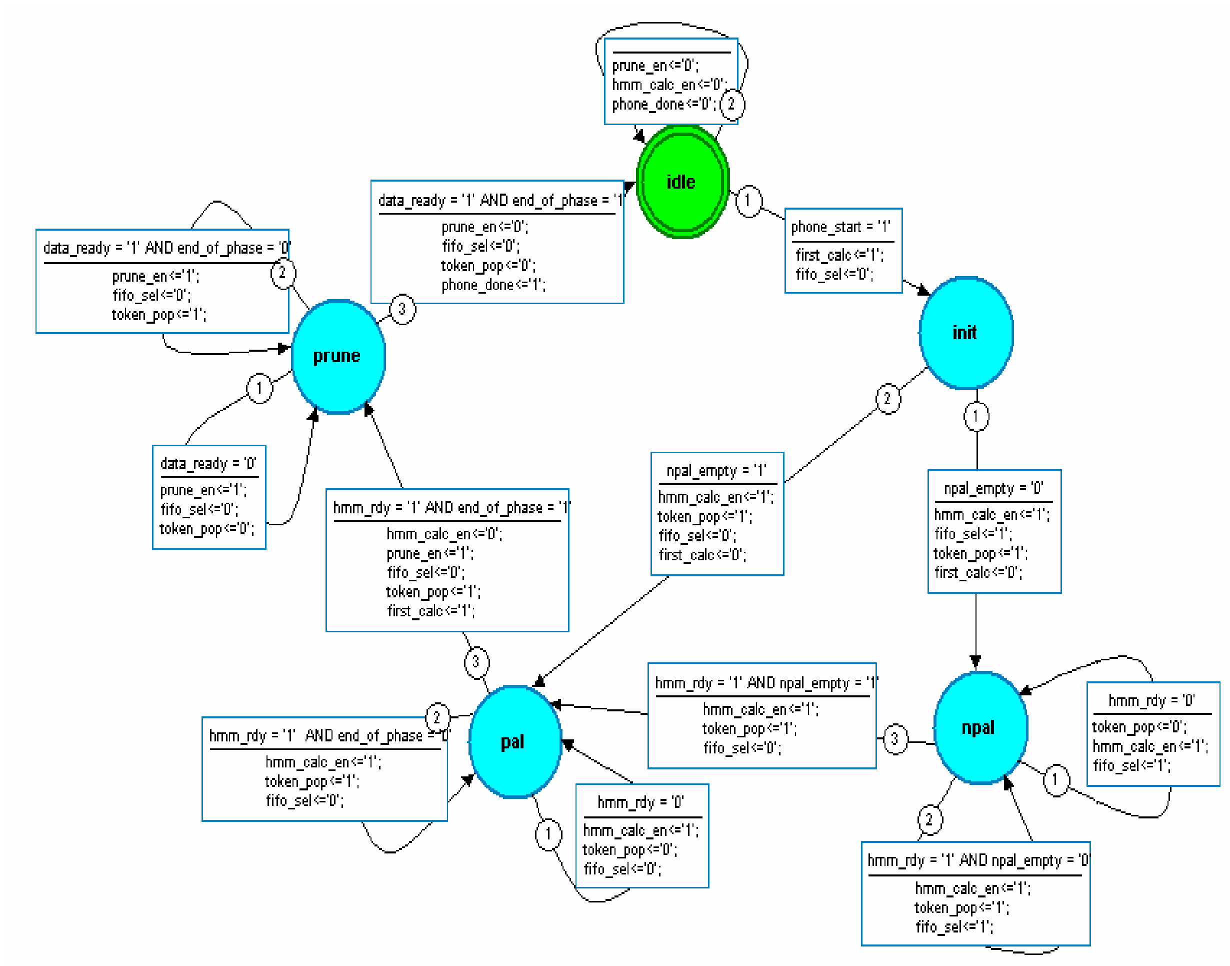


**Figure 42.** Control for Phoneme Evaluator

The first observation made when looking at Figure 42 is that the PAL FIFO is actually completely internal to the PE block and can be loaded by either the HMM pipeline or the pruner. In order for this to work properly a special end of phase (EOP) token was created to serve as a place marker in the FIFO. To create the EOP token the PAL FIFO was designed to be 1-bit wider than the other FIFOs so that the extra bit could be used to hold information about the EOP token. All standard tokens in the queue will have logic zero in this extra bit, but the EOP token will have logic one, making it very easy to detect the presence of the EOP token.

At the beginning of each new frame PE will start pulling tokens from the nPAL FIFO until the FIFO is completely empty. When the FIFO is emptied, or in the case there was nothing there to start, the PAL FIFO is then read from until and EOP token is detected by the STATUS block. When the EOP token is seen it is then known that all HMMs needing calculated have been done and pruning may commence. At the beginning of pruning the EOP token is written back to the PAL FIFO, and pruning is executed until the EOP token is popped back out of the FIFO again. Once this second observation of the EOP token is made it is known that all data has been processed for that frame the word modeler may begin processing the tokens in the DEAD and EXIT FIFOs. Since this process is not fully pipelined as was the case for the AM design, removing all of the FSMs from the architecture was not entirely possible. Through inclusion of objects like the EOP token as well as basic handshaking signals between PE and the other portions of the design however, the control for PE was able to be reduced to a single 5-state FSM. Figure 43 shows the FSM used to control the PE block.



idle
init
npal
pal
prune
prune_en<='0'; hmm_calc_en<='0'; phone_done<='0';
phone_start = '1'
first_calc<='1'; fifo_sel<='0';
npal_empty = '0'
hmm_calc_en<='1'; fifo_sel<='1'; token_pop<='1'; first_calc<='0';
npal_empty = '1'
hmm_calc_en<='1'; token_pop<='1'; fifo_sel<='0'; first_calc<='0';
hmm_rdy = '0'
token_pop<='0'; hmm_calc_en<='1'; fifo_sel<='1';
hmm_rdy = '1' AND npal_empty = '0'
hmm_calc_en<='1'; token_pop<='1'; fifo_sel<='1';
hmm_rdy = '1' AND npal_empty = '1'
hmm_calc_en<='1'; token_pop<='1'; fifo_sel<='0';
hmm_rdy = '0'
hmm_calc_en<='1'; token_pop<='0'; fifo_sel<='0';
hmm_rdy = '1' AND end_of_phase = '0'
hmm_calc_en<='1'; token_pop<='1'; fifo_sel<='0';
hmm_rdy = '1' AND end_of_phase = '1'
hmm_calc_en<='0'; prune_en<='1'; fifo_sel<='0'; token_pop<='1'; first_calc<='1';
data_ready = '0'
prune_en<='1'; fifo_sel<='0'; token_pop<='0';
data_ready = '1' AND end_of_phase = '0'
prune_en<='1'; fifo_sel<='0'; token_pop<='1';
data_ready = '1' AND end_of_phase = '1'
prune_en<='0'; fifo_sel<='0'; token_pop<='0'; phone_done<='1';
1
2
3

**Figure 43.** Finite State Machine for PE Control

When the FSM receives the start signal it will first enter the init state to allow for the EOP token to be placed onto the PAL FIFO. At the end of a given frame the last operation to occur is the reading of the EOP token from the FIFO signaling the end of the pruning phase. While this read does trigger the end of the PE evaluation it does not trigger an event to place the EOP token back onto the FIFO again. Therefore this operation must happen at the beginning of each new frame, to ensure the proper operation of the control logic. After placing the EOP token on the PAL FIFO, the status of the nPAL FIFO will be checked and the FSM will transition accordingly. If the nPAL FIFO has new data in it then the FSM will transition to the *nPAL* state and begin to evaluate the active HMMs. If this FIFO is empty however, the FSM will transition directly into the *PAL* state and begin processing the HMMs that are still active from the previous time. The *nPAL, PAL,* and *PRUNE* states all function identically in that they will self-loop through one arc while the evaluation is being performed and then self-loop through the other arc only on the cycle that the HMM processing is finished so that the score may be written to the RAM. This process will continue until the nPAL FIFO is emptied or the EOP token is seen, depending on which FIFO is being evaluated, at which point the FSM will transition to the next state in the process. After observing the EOP token while in the *PRUNE* state, the FSM will raise a *done* signal and return to its idle state until the next receiving the next *start* signal.

### 5.4.2 HMM Pipeline

The execution of Equations 23-27 constitute the majority of the calculations necessary to perform PE and therefore a significant amount of time was put into examining the optimal way to perform these operations. After establishing the control for this pipeline the calculations were examined and it was found that to find all *H* values for a given HMM a simple ADD-COMPARE-ADD-COMPARE pipeline can be constructed as shown in the data flow graph in Figure 43. The data-flow graph in Figure 44 highlights the regularity of the structure for the pipeline and leads directly to a high-throughput low-latency design for calculation of the HMM scores in the system. Further, the complexity of the pipeline actually becomes very low and requires a noticeably smaller amount of logic than even the ROMs required to drive it. As each of the active HMMs are evaluated, the five output values are written to the HMM database and the HMM ID token is written into the pruner queue.

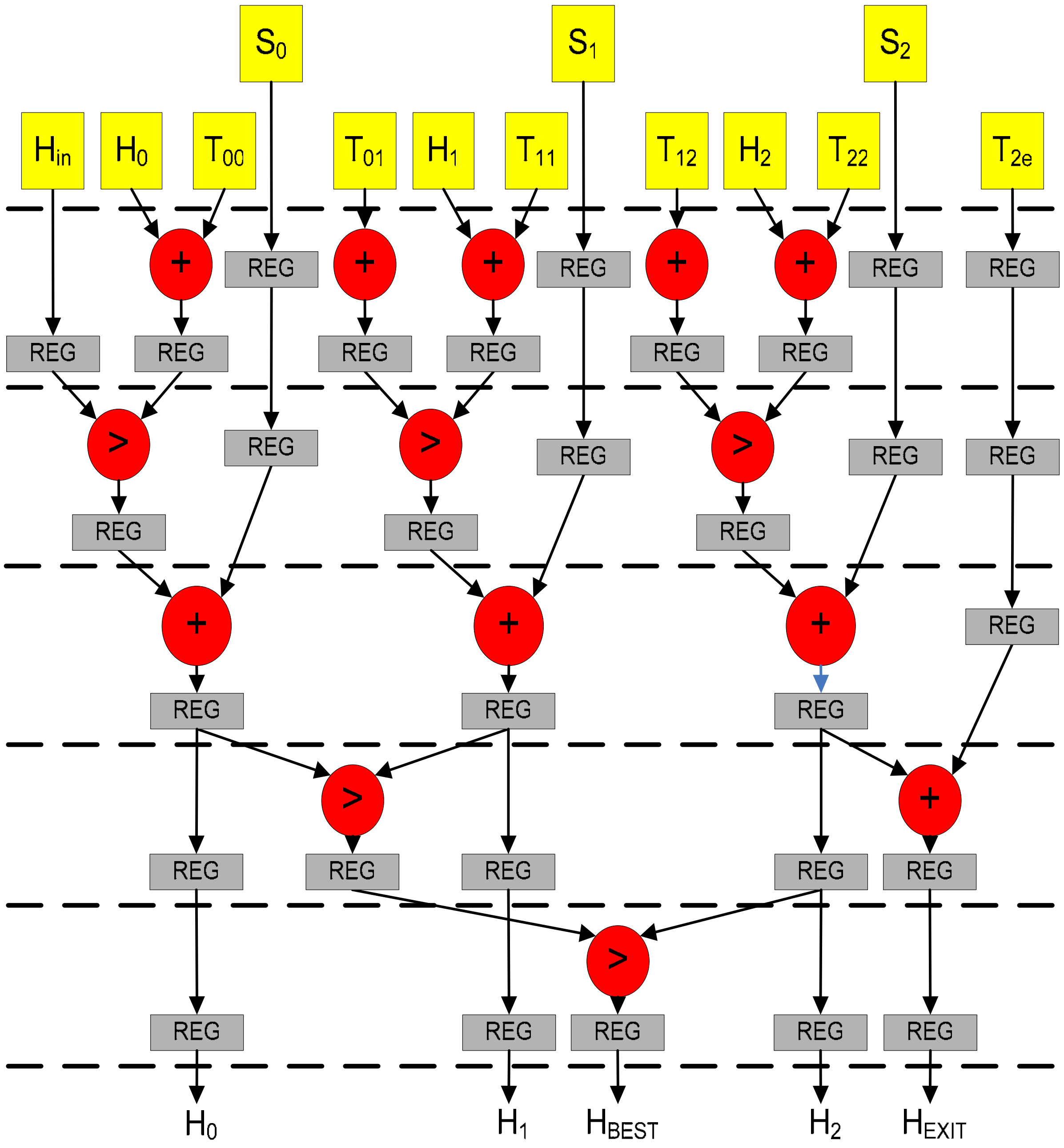


**Figure 44.** Data-Flow Graph for HMM Pipeline

### 5.4.3 HMM Pruner

After having calculated all HMM scores for a given frame the scores are then read back out of the RAM and compared to the beams. As mentioned in section 4.3 two different beams are used to prune the HMMs based on both their exiting score and their best score. If an HMM has a valid exit score it will be passed to the word modeler as well as remain in the active queue. If the HMMs score is not above the exit beam however, it will be checked against a second beam to see if the HMM should remain in the active queue. This two step approach helps to minimize the number of HMMs mistakenly pruned from the system and significantly increases the recognition accuracy of the system. It also helps to maintain a time-varying system, in that a HMM can exit and remain active so that in successive frames the HMM could exit again, but with a higher probability.

To implement the beam pruning algorithm in hardware a simple pipelined approach was taken. Since all of the active tokens must be pruned before moving on and because each HMM is pruned in the same manner tokens can be popped from the FIFO each cycle until the EOP token is seen. Additionally, while the data regarding the best and exit score for each HMM is needed from the central database, nothing needs written back to the database helping to simplify the pipeline. Figure 45 shows the data-flow graph for the HMM pruner pipeline.

A third beam is also calculated by the pruner and is passed forward to the word modeler for later use. This beam is calculated based off of the exit score for any active HMM in the cue that represents the end of a word in a dictionary. Just as transitioning from one HMM to another incurs a penalty so does transitioning from one word to another, and the word beam helps to prune out unlikely sequences of words. While the HMM pruner does not actual process the data in the PH RAM based on the result of pruning, it does establish the work order for the word modeler and helps to greatly simplify that stage of processing.

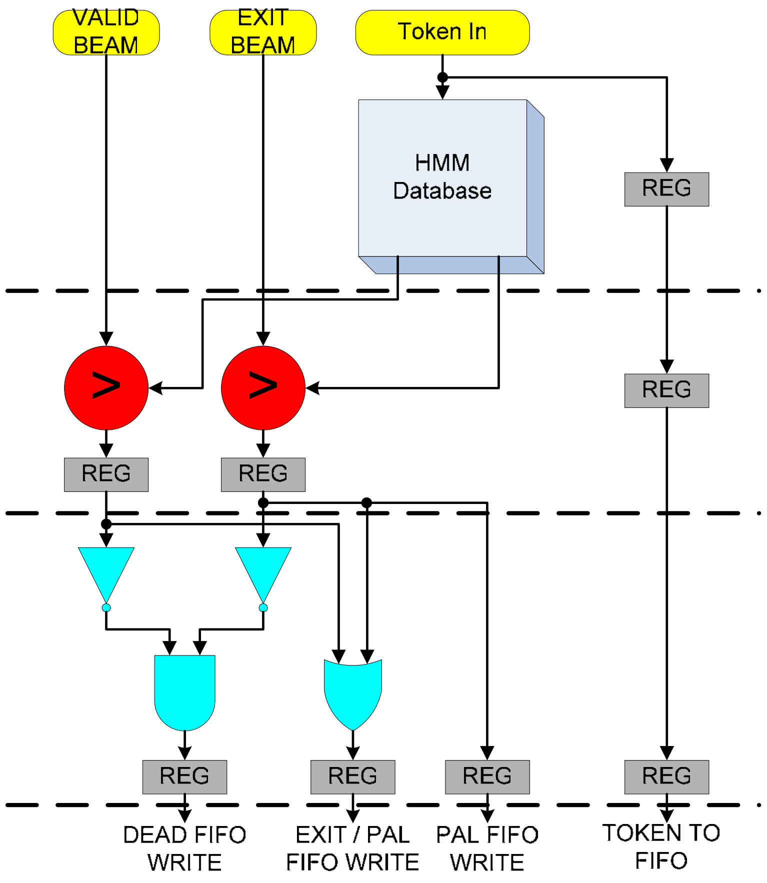


**Figure 45.** Data-Flow Graph for HMM Pruner

## 5.5 HARDWARE PERFORMANCE RESULTS

After finishing the design of both the hardware and software components for PE the focus was then turned to the analysis of the hardware designed to execute the process. During this analysis an experiment was derived to examine the effects of the synthesis engine on final chip utilization. To conduct the experiment the HMM pipeline was synthesized using both the Synopsis Synplify tool and the Precision Synthesis tool from Mentor Graphics. The results of the experiment are shown in Figure 46 with a summary of the results given as Table 5.

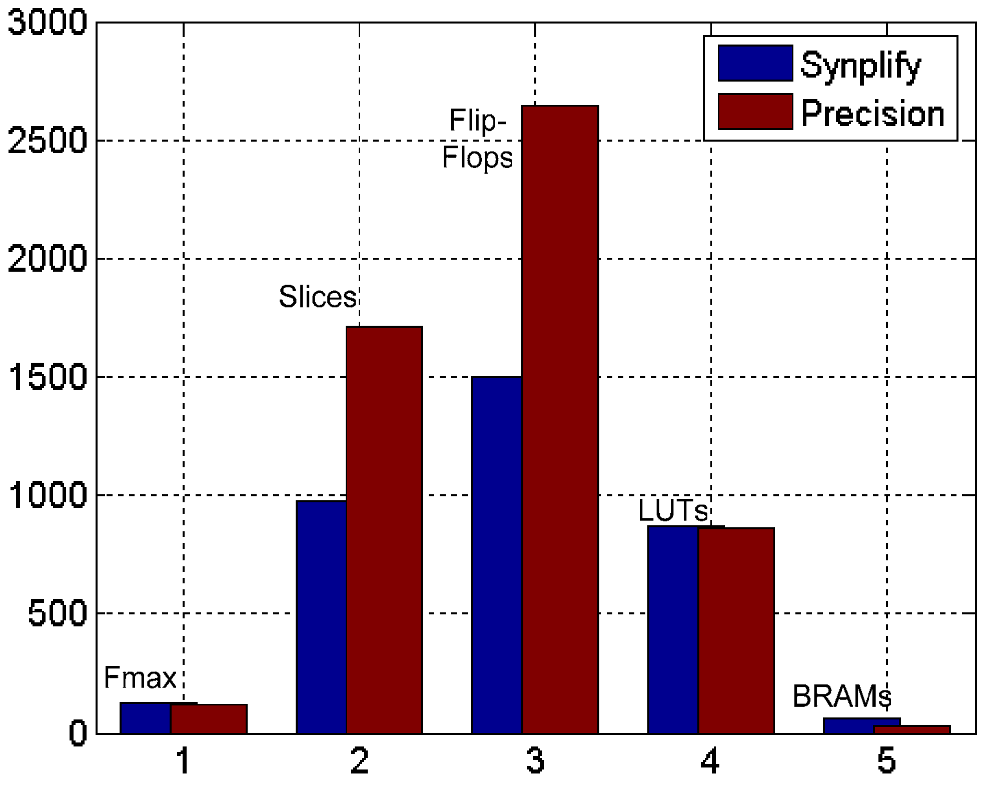


**Figure 46.** Precision Synthesis Vs. Synopsys Synplify

**Table 5.** Summary of Synthesis Results for HMM Pipeline using both Precision Synthesis and Synopsys Synplify

| **Synthesis Tool** | **$f_{MAX}$ (MHz)** | **Slices** | **Flip- Flops** | **LUTs** | **18K RAMs** |
|---|---|---|---|---|---|
| **Precision** | 117 | 1713 | 2645 | **861** | **25 RAMs** |
| **Synplify** | **127** | **978** | **1497** | 874 | 60 RAMs |

From this experiment it is observed that for the exact same functionality the Synplify synthesis tool was able to create a design with a slightly higher $f_{MAX}$ and a significantly lower gate count. It is also observed however that in the Synplify case 60 BRAMs were used as opposed to the 25 used in the Precision version. This is related to the notion that during our custom design some small local ROMs were made using purely combinational logic, whereas when Synplify created these ROMs for us it chose to use embedded BRAM cells. Given these observations the rest of the components for PE were synthesized using Synplify and put through PAR by Xilinx ISE using hand-generated constraint files as described in Section 4.5. The results of both synthesis and place-and-route for all components of PE are summarized in Table 6 with the final layout for PE shown in Figure 47. It is important to note that the results obtained for the number of RAMs required by each block is strictly a function of the small database chosen as our test set. In a fully system the number of RAM blocks required would be far greater leading to the necessity to extend the system into external RAM usage for very large scale operations.

As with the Acoustic Modeling block, the functionality of the Phoneme Evaluator was verified against both the SPHINX 3 baseline model and the MATLAB hardware. The three implementations of the algorithm were first directly compared to each other on random sample frames over the course of three spoken utterances. Having ensured the correctness of the MATLAB code to the SPHINX baseline, random test vectors were fed through both the MATLAB and the hardware to compare the output results. Due to the fact that all calculations during Phoneme Evaluation operate only on 32-bit integer values, a bit-accurate comparison was able to be made between the hardware and the MATLAB models.

**Table 6.** Summary of Place-and-Route Results for PE

| Component | Synth (MHz) | PAR (MHz) | Slices | 18K RAMs |
|---|---|---|---|---|
| **HMM Pipeline** | 261 | 140 | 775 | -- |
| **MDEF ROM** | 454 | 151 | 141 | 24 RAMs |
| **Pruner** | 277 | 177 | 112 | -- |
| **PH PTR RAM** | 572 | 118 | 16 | 8 RAMs |
| **Pipe w/ MDEF & Sen. RAMs** | **164** | **113** | **1605** | **69 RAMs** |
| **PE TOTAL** | **115** | **111** | **1866** | **84 RAMs** |

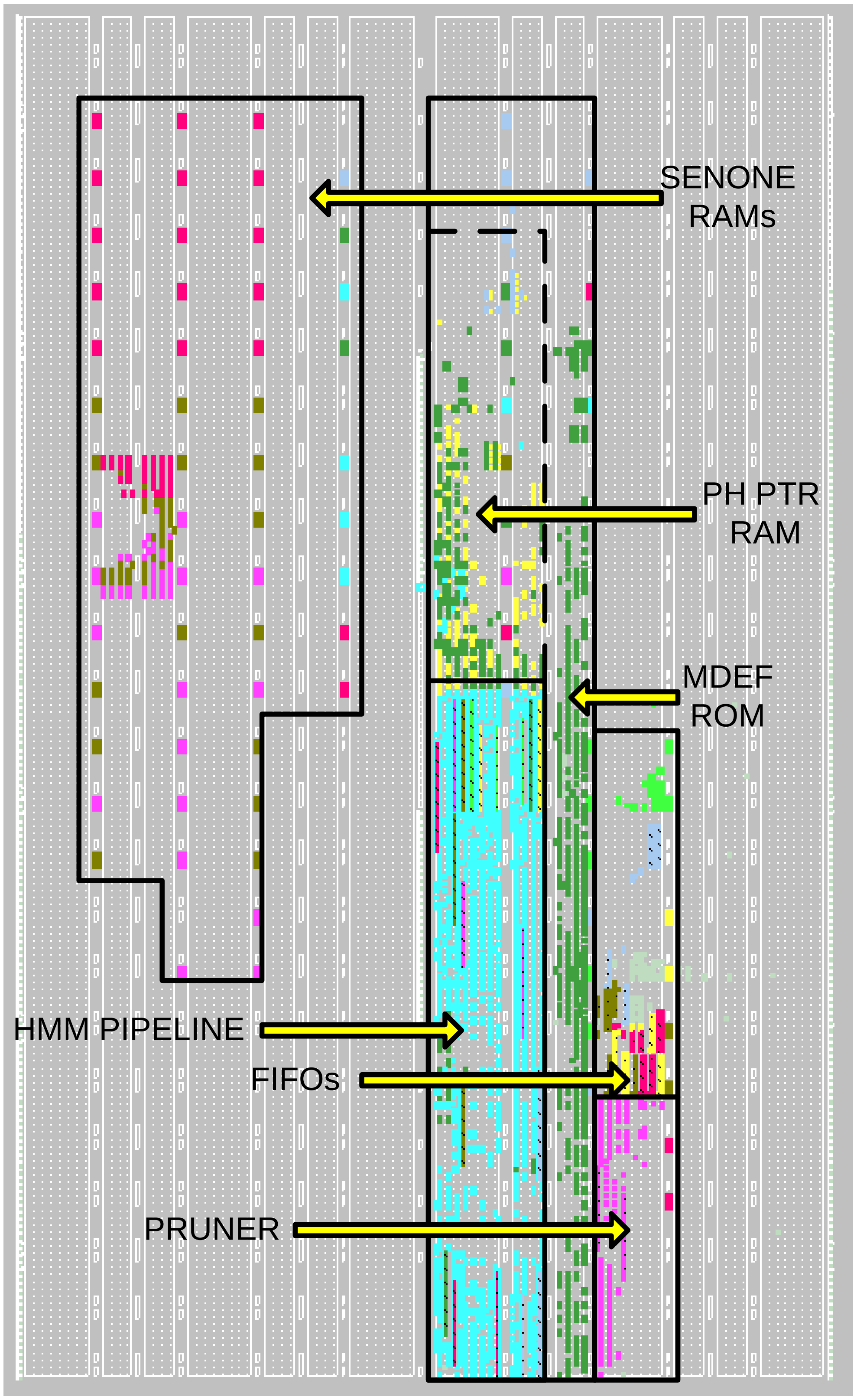


**Figure 47.** Final Layout for PE

# 6.0 WORD MODELING

Unlike the definitions for AM and PE, Word Modeling (WM) can refer to multiple levels of abstraction, depending on the specific system under consideration. For some systems WM may actually include PE, and effectively divide the system into an Acoustic Modeler and a Word Modeler. Other systems maintain PE as its own unique event but consider WM to include both the linking of the phonemes within a word as well as the linking of multiple words. These systems can employ constraints based on the language being recognized to help increase the accuracy of the system. The constraints work by preventing or creating heavy penalties for groups of words that are impossible or at least highly unlikely for the given language and provide the system with some knowledge of what is being said, while it is being processed. For our purposes however, WM will be defined solely as the linking of phones into words, including the linking of the end of one word to the beginning of the next. This definition allows for the separation of the WM process from what is commonly called the Language Modeling (LM) process, and creates a concise portion of the design for discussion and development. In the designed system WM is responsible for managing a tree-style data structure and updating nodes in the tree based on information provided by the PE portion of the design. The rest of this chapter will focus on describing the functional nature of the data structure, and quantifying the software and hardware developed to execute the algorithms.

## 6.1 TREE SEARCHING

The basic search tree is one of the fundamental data structures in computer systems today. A tree in its most general sense refers to any set of connected data points, nodes, without any specification as to their priority. Data structures that fit this loosest definition are often called *free trees* since they allow for numerous entry and exit points to the data set. Tree structures can be used to represent various types of systems, with speech systems being one particularly good example. Most languages including English contain large numbers of words with similar beginning sounds but distinctly different endings. These groups of similar words can be grouped together into what are called *rooted trees* helping to minimize the search space through lumping similar paths. A rooted tree as defined by [58] represents a special kind of free tree wherein one of the vertices is distinguished as the root of the search space. In this type of tree there exists only one entry point and subsequently the direction of propagation becomes static unlike in the case of the free tree. Figure 48 illustrates the difference between a free tree and a rooted tree data structure.

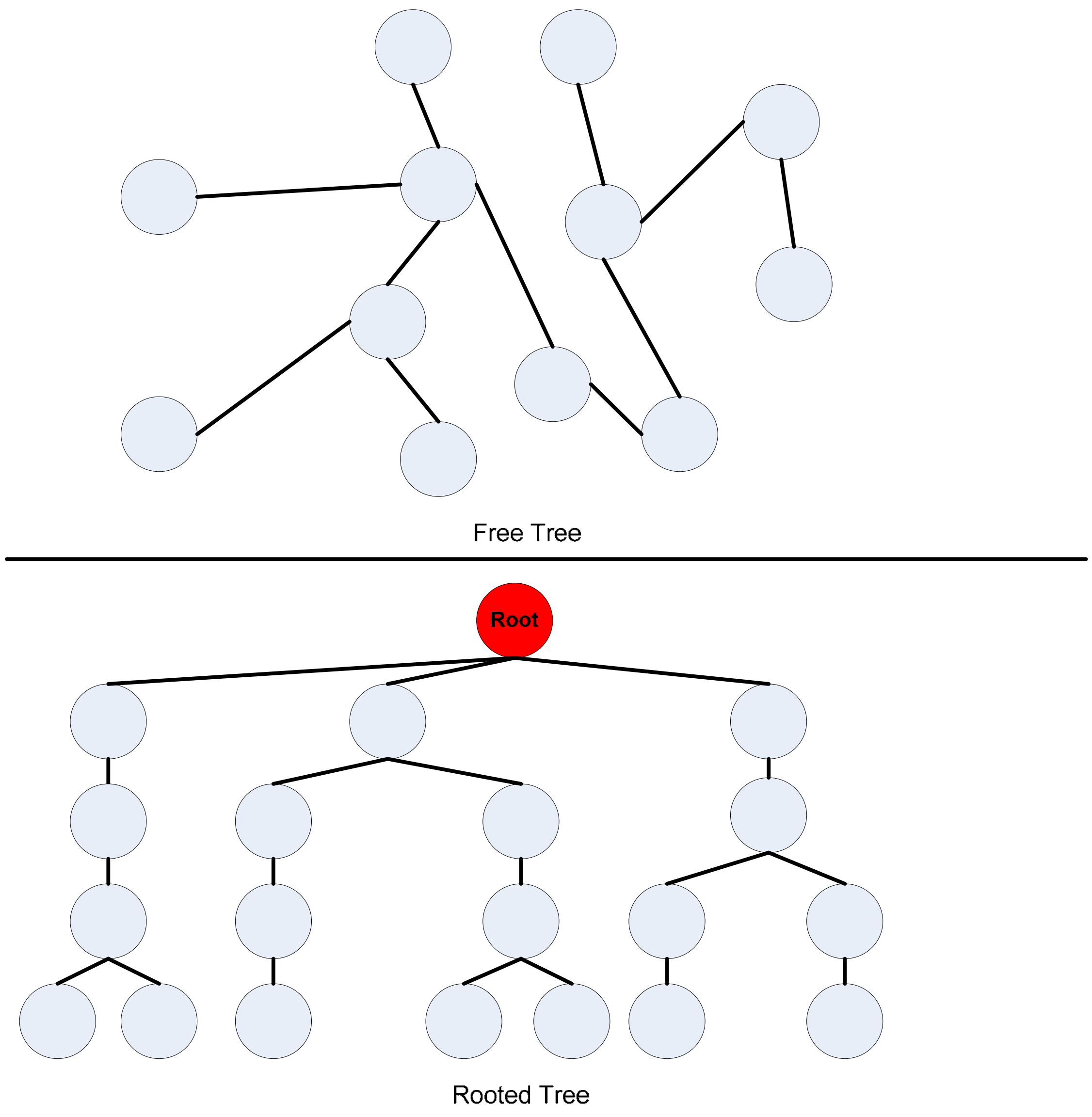


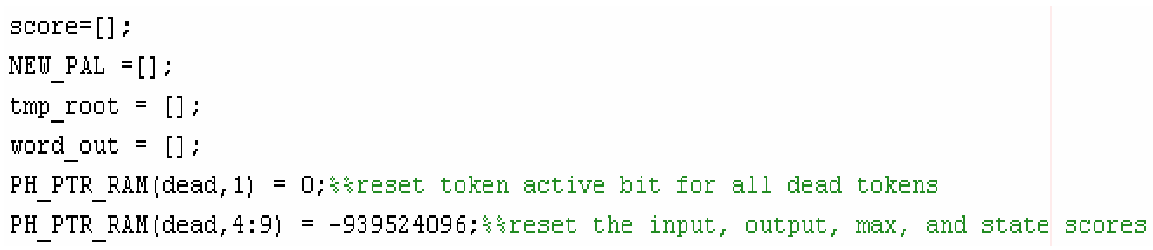
**Figure 48.** Sample Tree Topologies

In even small sized speech recognition systems there are going to be a number of different trees that all need to be maintained while the system is active. The entire set of trees that comprise the search space is often called the *forest* and while some trees may only contain a single branch others may be significantly more complex. Regardless of this notion however, all trees in a given forest must be maintained with the same regularity to ensure the system is not partial to the recognition of some words and not others. When using trees for ASR each of the node in the in the tree represents a particular HMM, described in Chapter 5. As HMMs obtain valid output scores, this information is forwarded along the paths defined by the tree and used as input scores to all subsequent HMMs. By propagating the scores of the HMMs through the system the probability of word seen at the end of a given branch in the tree, or leaf, becomes a function of the probabilities of each of the sounds in the word, helping to provide the user with some sense of the quality of the recognition when a word is observed at the output of the system.

Because most modern ASR systems focus on the recognition of semi-continuous or fully-continuous speech it is necessary to provide a mechanism that allows for the transitions from the leaves of a given tree to the potential roots of all other trees in the forest. If no priority is given to the order in which trees may follow one another then all leaves must connect to all roots and the potential for a search space explosion becomes exceedingly large. To counter this problem some systems limit the amount of time that the user can speak to the system before they must stop and wait for the system to respond [1, 9]. Another possible solution to this problem is to simply limit the number of words in the dictionary effectively forcing the system to stay within a stable range, and this is the implementation we have chosen for our baseline system. The following section of this chapter will explain the MATLAB code used to implement the rooted tree search and describe how scores are propagated inside of a tree and from tree to tree.

## 6.2 MATRIX MATLAB REPRESENTATION

After the PE block has finished pruning and the token FIFOs have been filled the WM may begin the process of updating the forest. As mentioned in Section 5.3 the WM is responsible for both resetting the newly inactive tokens as well as launching new tokens when a given node in the tree has exited. The process of token deactivation is quite simple in the MATLAB code and only require that the current contents of a given memory location be reset to some known constant values. The code for performing the token deactivation as well as frame initialization for the word modeler is shown in Figure 49.

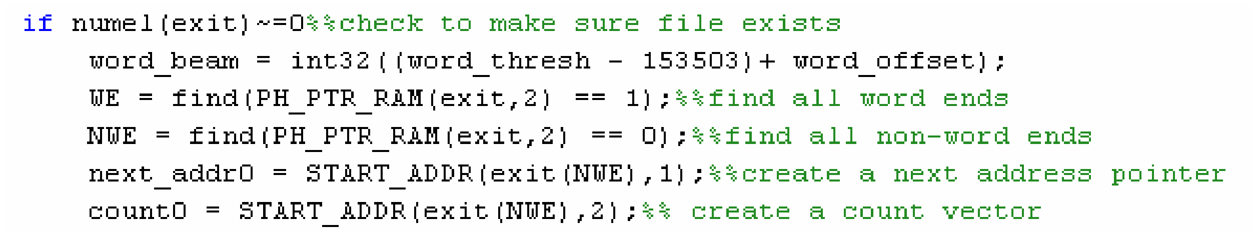


```
score=[];
NEW_PAL =[];
tmp_root = [];
word_out = [];
PH_PTR_RAM(dead,1) = 0;%%reset token active bit for all dead tokens
PH_PTR_RAM(dead,4:9) = -939524096;%%reset the input, output, max, and state scores
```

**Figure 49.** Code for Token Deactivation and Frame Initialization

Figure 48 shows that before the token activation portion of WM can begin a number of vectors must be re-initialized to ensure proper operation.  Because the size of these vectors will vary throughout the operation of the system depending on how many tokens need processed, they must be cleared before every new frame or else erroneous tokens could potentially enter the system.

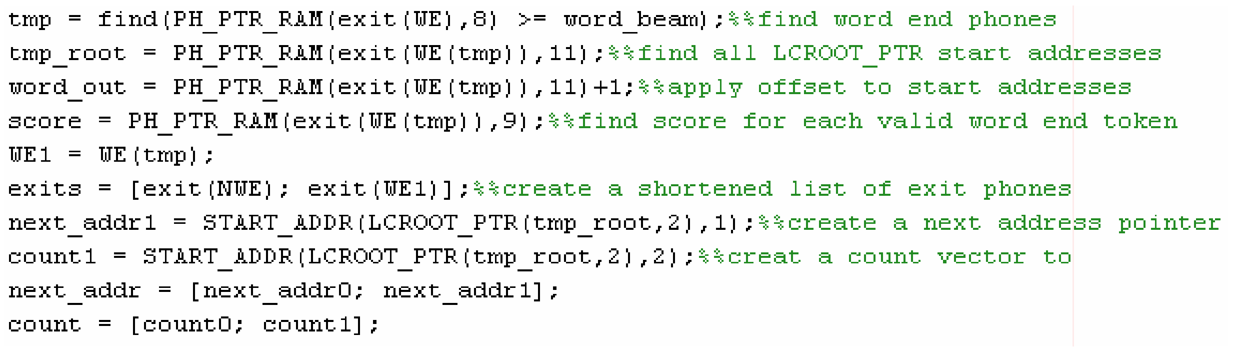
Once the vectors are cleared and any inactive tokens have been reset the WM will next check to make sure there are tokens in the exit cue that need processed.  Since it is possible that no HMMs exited during a given frame it is possible to completely skip the rest of the WM process, making the inclusion of this check a valuable time saver during run-time.  Assuming there are tokens in the cue the next step is to calculate the word beam and to separate the tokens representing leaf nodes from the tokens representing inner-tree nodes.  The inner-tree nodes will each be treated as a short link-list where each element in the list represents a branch from the node.  To process these link-lists a start address and a count needs to be obtained for each node from a predefined data structure referred to in the code as *START_ADDR*.  The code for checking the cue, calculating the beam and obtaining the link-list information is shown in Figure 50.



```
if numel(exit)~=0%%check to make sure file exists
    word_beam = int32((word_thresh - 153503)+ word_offset);
    WE = find(PH_PTR_RAM(exit,2) == 1);%%find all word ends
    NWE = find(PH_PTR_RAM(exit,2) == 0);%%find all non-word ends
    next_addr0 = START_ADDR(exit(NWE),1);%%create a next address pointer
    count0 = START_ADDR(exit(NWE),2);%% create a count vector
```

**Figure 50.** Code for Obtaining Link-List Information

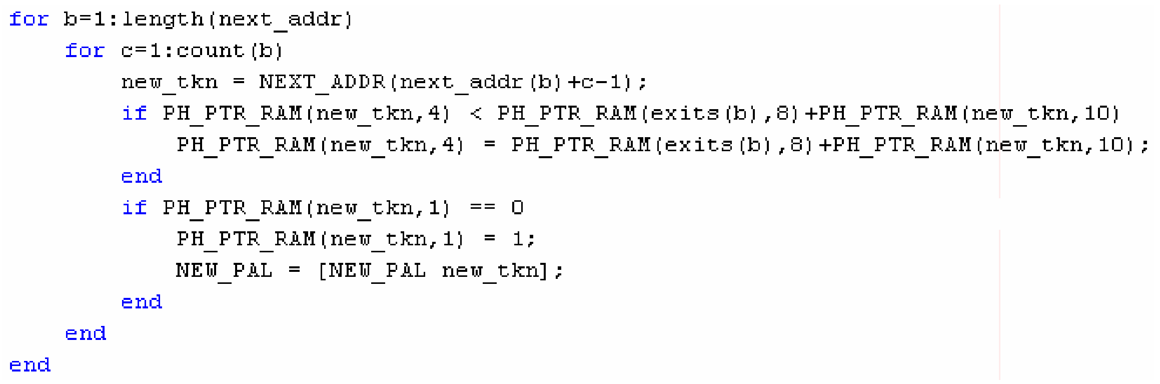
Before beginning to process the link-lists for the inner-tree nodes it is necessary to evaluate the leaf-nodes to see if they pass the word threshold and obtain the information about which root nodes to enter for all leaves above that threshold. Once the leaves above the threshold have been found and the information about which root nodes to transition to has been obtained this information can be concatenated to the information about the inner-tree nodes and they can all be processed the same way. The MATLAB code necessary to perform these operations is shown in Figure 51 with the *next_addr* and *count* vectors representing the final list of work that needs performed by the WM.



```
tmp = find(PH_PTR_RAM(exit(WE),8) >= word_beam);%%find word end phones
tmp_root = PH_PTR_RAM(exit(WE(tmp)),11);%%find all LCROOT_PTR start addresses
word_out = PH_PTR_RAM(exit(WE(tmp)),11)+1;%%apply offset to start addresses
score = PH_PTR_RAM(exit(WE(tmp)),9);%%find score for each valid word end token
WE1 = WE(tmp);
exits = [exit(NWE); exit(WE1)];%%create a shortened list of exit phones
next_addr1 = START_ADDR(LCROOT_PTR(tmp_root,2),1);%%create a next address pointer
count1 = START_ADDR(LCROOT_PTR(tmp_root,2),2);%%creat a count vector to
next_addr = [next_addr0; next_addr1];
count = [count0; count1];
```

**Figure 51.** Code for Obtaining Leaf-Node Propagation Information

With a complete list of the tasks to be completed the WM must next begin to process the link-lists to determine which tokens need launched into the system and which tokens are already active in the system and should be updated but not launched again. Due to the fact that each branch in the tree must be check to ensure that node is not already active before transitioning into it this portion of the code must be executed as a series of nested for-loops, preventing the ability to further vectorize the process. For each branch the current input score of the potential child node must be check against the exit score for the parent node. If the current input score for the child node is better than the exit score for the parent node plus the transition penalty then the score will not be updated, but if this is not the case then the current input score for the child node will be replaced with the value of the exit score plus the transition penalty. The status of the child node must also be checked to ensure that the token is not re-launched into the system if it is already active. If the status bit is set for the given node then it will not be launched into the system, but if the bit is not set then the WM will set it and place the token onto the cue. This process must be repeated for each branch of each node in the *next_addr* vector and the code needed to execute these loops is shown in Figure 52. Once the last node has been processed and all new tokens are in the cue the WM operations are complete for the given frame and the system can wait for the next frame of input data.



```
for b=1:length(next_addr)
    for c=1:count(b)
        new_tkn = NEXT_ADDR(next_addr(b)+c-1);
        if PH_PTR_RAM(new_tkn,4) < PH_PTR_RAM(exits(b),8)+PH_PTR_RAM(new_tkn,10)
            PH_PTR_RAM(new_tkn,4) = PH_PTR_RAM(exits(b),8)+PH_PTR_RAM(new_tkn,10);
        end
        if PH_PTR_RAM(new_tkn,1) == 0
            PH_PTR_RAM(new_tkn,1) = 1;
            NEW_PAL = [NEW_PAL new_tkn];
        end
    end
end
```

**Figure 52.** Code for Main WM Loop

## 6.3 HARDWARE ARCHITECTURE

Similar to the software implementation, the hardware version of WM can be broken down into two major steps; resetting of newly inactive tokens and updating of currently active tokens. Given that the functionality of these two components is distinctly different it was optimal to design two separate components, one for each task. The token deactivator reads data from the DEAD FIFO and resets the scores in the PH RAM, while the token activator reads from the EXIT FIFO and processes the word tree to determine which new tokens need to be placed in the nPAL FIFO. The creation of two separate blocks also minimizes the amount of logic active at any one point in time leading to the potential power savings crucial to ensuring sustainability on a mobile platform. The operations of these blocks are described in Sections 6.3.1 & 6.3.2 respectively. The hardware development presented in these sections represents the work performed by the ECE 2121 Hardware Design Methodologies II course taught by Prof. Raymond Hoare during the spring term of 2005. These hardware cells are based directly off of the MATLAB models described in Section 6.2 and although they do not represent my direct contributions to this thesis they are important to comment on for sake of completeness.

### 6.3.1 Token Deactivator

After the PE block has pruned the active HMMs and placed the appropriate tokens in the DEAD FIFO the word modeler can begin the task of resetting the PH RAM entries corresponding to these tokens. This process is made quite simple by the fact that all PH RAM values need to be reset to the exact same value. This means that to deactivate a token the token must be popped from the DEAD FIFO and used to address the PH RAM. When this address is applied to the RAM the reset constant is then written to the RAM and the process is repeated until the FIFO is empty. This process can be performed in a simple-two stage, POP-WRITE pipeline, and is limited only by the speed of the RAM it is addressing. The values used for token deactivation are given in section 6.2 of this paper and represent the values of the maximum negative integer allowed in the system and the binary zero value used for deactivation of a given token in the database.

### 6.3.2 Token Activator

The token activator portion of the word modeler is noticeably more complicated than deactivator portion and required some amount of research to determine the optimal way to implement the logic. When a HMM is found to have a valid exit score, the word modeler must determine where in the word tree that HMM exists and which HMMs are tied to its exit state. As shown in Figure 53 a word tree can have a large number of branches stemming from one root and mapping these types of topologies into hardware structures is not nearly as simple as implementing them in the software described in section 6.2.

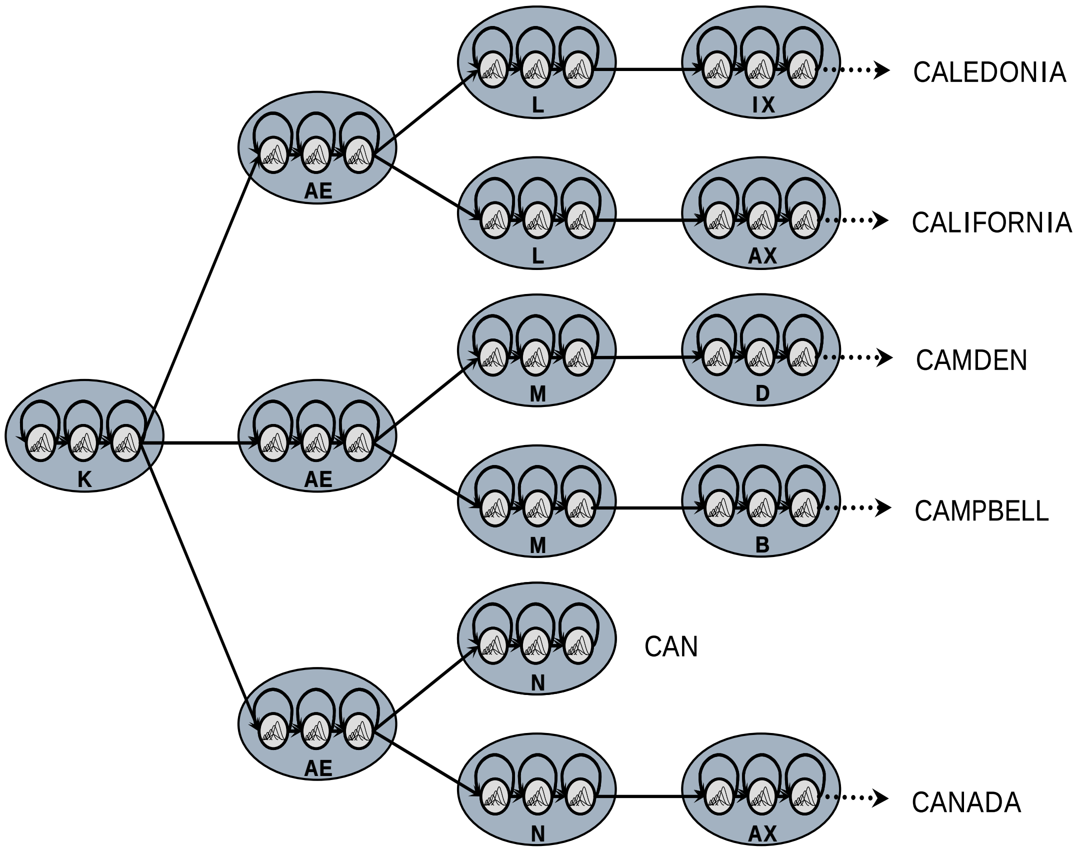


**Figure 53.** Sample Word Tree

Another unique problem in token activation is that while a given HMM may be used multiple times in multiple different word trees such as the *'CH'* sound at the end of *pouch* and *couch*, these two sounds must be represented by completely unique events. This means that while a given dictionary may not need all possible phonemes in a language, it will most likely need multiple instances of some of the phones. Taking this into account we were forced to come up with a way of indexing specific nodes in the search space so that their information could remain in a unique location in the PH RAM. To do this the entire word search space was mapped and each of the nodes given a unique ID. An example of this process is shown in Figure 54.

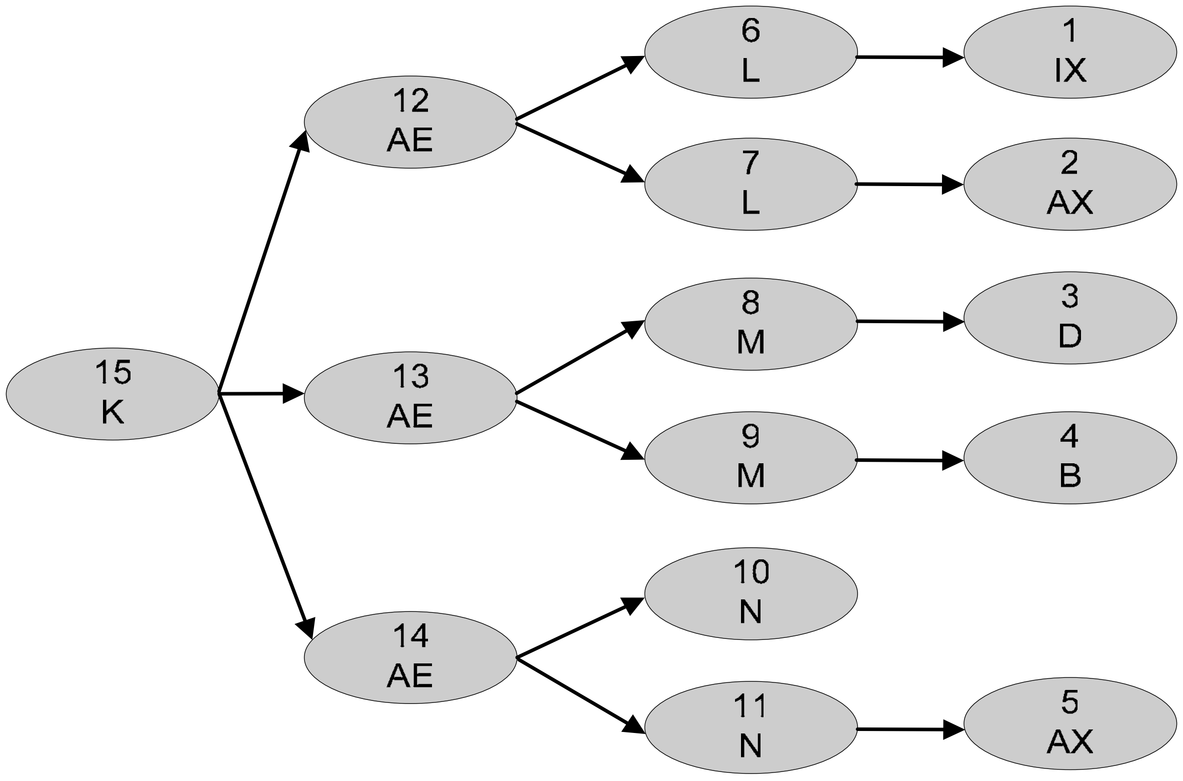


**Figure 54.** Sample Search Space Node Mapping

Based on this mapping scheme, even though nodes 12-14 in Figure 53 all relate to the *AE* phoneme, they all have unique IDs and will be treated separately in search algorithm. The mappings determined in the process relate directly to the HMM IDs stored in the tokens passed between the PE and WM blocks, and define the core of the token passing algorithm as implemented in our system.

Having established our mapping scheme it was next necessary to determine how to approach implementing a tree structure in hardware. Unlike the previous section of the design this portion is less arithmetically intensive and involves searching instead of computational overheads. One of our immediate observations when looking at the data structures was that each node in the search space can be thought of as a short link-list. Evaluating link-lists in hardware is not a new topic of research, so to create a system able to process link-lists of link-lists appeared a straight forward solution to our problem.

Based on our idea of link-lists we then derived the architecture that would handle such a task in an optimal way. During each evaluation a token must be read from the EXIT FIFO and its link-list retrieved. Further the $H_{EXIT}$ score from the exiting HMM must be added to a word penalty and propagated to the $H_0$ score of the HMMs in the link-list. Since the word penalty will be different for each HMM in the link-list is becomes necessary to process the HMMs one at a time until the end of the list is reached. To keep track of the link-list in an efficient manner two ROMs were designed the START ROM and the NEXT ROM. The START ROM is directly addressed by the token in the EXIT FIFO and contains both a starting address in the NEXT ROM for the link-list and a count of how many values are in the list. The NEXT ROM holds all of the HMM IDs necessary to process the link-lists. Figure 55 shows the data-flow for the token activator.

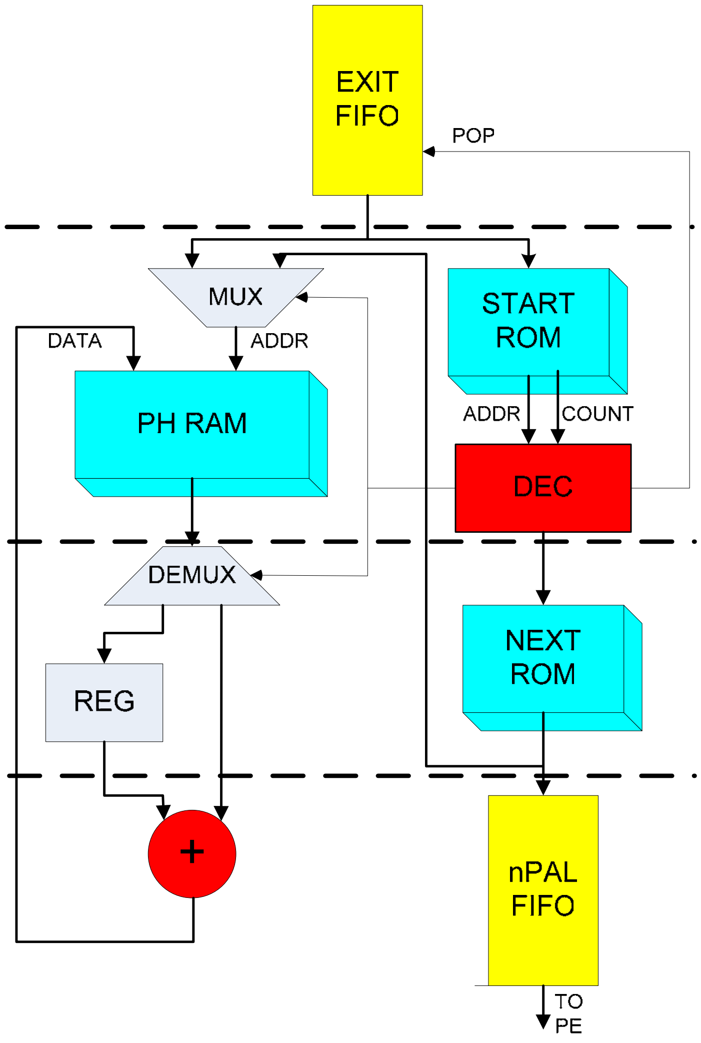


**Figure 55.** Data Flow Graph for Token Activator

When a token is read from the EXIT FIFO a mux control bit is set to determine which token is in control of the PH RAM address. While the count for the link-list is non-zero the bit will remain set but once the final decrement has been completed the bit control bit will switch to allow a new exiting HMM to be read from the PH RAM. This process is repeated for each element in the EXIT FIFO until the FIFO is emptied at which time the system goes idle and awaits the next frame of input data.

## 6.4 HARDWARE PERFORMANCE RESULTS

For the hardware design for WM little time was spent performing thorough analysis due to the fact the less than 5% of the overall execution time of the designed device is spent performing WM. Unlike the work presented for the AM and PE blocks the components needed for the WM design were created in FPGAdvantage 6.1 by the ECE 2121 course participants as noted in section 6.3. Synthesis for this portion of the design was performed in Precision Synthesis and the final place and route was performed in ISE 7.1. Results for synthesis and place-and-route are given in Table 7. As noted at then end of the previous chapter, the RAM allocation for this block is strictly a function of the chosen database and will change as new vocabularies are chosen.

**Table 7.** Summary of Word Model Synthesis Results

| Component | SYN (MHz) | Place-and-Route (MHz) | Slices | 18K RAMs |
|---|---|---|---|---|
| **Token Deactivate** | 377 | 170 | 54 | -- |
| **Token Activate** | 184 | 120 | 160 | 3 BRAMs |
| **WM Block (TOTAL)** | **166** | **129** | **414** | **3 BRAMs** |

# 7.0 PERFORMANCE PROFILES

This work has examined a number of current state-of-the-art technologies as well as presented work on the creation of new systems in both software and hardware. Having considered such an array of designs, a qualitative summary of the performance of these systems is useful in helping to focus in on the pros and cons of each of the aforementioned works. First, a summary of performance results given for the systems in Chapter 2 will be presented to help characterize the current industry. Next, the vectorized MATLAB code will be analyzed, presenting results obtained through use of the MATLAB code profiler. This profile helps to illuminate the strengths and weakness of the MATLAB programming environment by providing the execution times for each line of code in the derived algorithm. Finally, the hardware created for this work will be reviewed to present a concise synopsis of the performance characteristics of the device.

## 7.1 LITERATURE PERFORMANCE REVIEW

The first major class of systems considered was software based systems designed to run on desktop PCs. The four main projects discussed were all presented at DARPA speech recognition systems evaluation throughout the late 90's and stand as strong representatives for the current state of speech recognition technology. All the software systems considered were based on the 48,000 word HUB-4 speech corpus which represents multiple hours of continuous conversational speech. IBM's Via Voice system was shown to achieve recognition rates as high as 85% on the HUB-4 corpus, but available documentation does not give the total execution time for the code. Via Voice used a total 5.7K HMM states grouped into 3-state HMMs and required a total of 170K Gaussian Distributions to process the Acoustic Model [8, 32]. The BYBLOS system from

BBN was able to achieve similar results during the DARPA evaluations and was observed to take 10X real-time to complete the recognition task [9, 34, 35]. In order to help with the recognition process BYBLOS made use of 12K HMM states again clustered into 3-state HMMs and also used a more complex Gaussian database, requiring 768K Gaussian Distributions. SRI's DECIPHER system was also presented around the same time and showed recognition rates between 70% - 80% on the HUB-4 corpus. This system was fundamentally different from other systems at the evaluation in two major ways. Primarily, DECIPHER is gender-dependent, and the user must identify themselves as male or female before using the system. Secondly, this system makes use of the Gaussian Merging-Splitting Algorithm described in [36] to minimize the number of Gaussians necessary for the evaluation. While this architecture is based on Hidden Markov Model evaluations, the states are not uniquely stored as the other systems described. For the male-version, 535 *genone* states were used requiring 68K Gaussian distributions. The female-version was slightly more complex, requiring 569 *genones* and a total of 73K Gaussians. The last of the software systems discussed was the SPHINX 3 engine from Carnegie Mellon University. SPHINX has been able to produce recognition rates over 90% for the HUB-4 corpus running at 10X real-time. The acoustic model for SPHINX-3 contains 6,000 HMM states clustered into 5-state HMMs and makes used of 96K Gaussians. Table 8 summarized the performance results for the software systems presented in Chapter 2.

After considering the current state-of-the-art software systems, some cutting edge hardware systems were discussed to present possible alternative solutions to the real-time recognition problem. The first system considered was developed at the University of Birmingham and was prototyped on a Xilinx Virtex XCV1000 FPGA. Their system operated on the 500 word TIMIT database and while it was able to run at 13 times faster than real-time, it only produced recognition rates between 50% - 60% [46, 47, 48]. The main reason for the lack of accuracy seems to lie in the fact that the hardware only used 1,900 HMM states due to constraints on the amount of memory obtainable on the target device. Unfortunately, while the available research remarks on the development of more advanced versions of the system, no documents were available on such systems as of the publication of this work.

**Table 8.** Summary of Software Performance Results

| SYSTEM | ACCURACY | SPEED | COMPLEXITY of ACOUSTIC MODEL |
|---|---|---|---|
| IBM's Via Voice | 70% - 85% | N/A | 5.7K HMM states<br>170K Gaussians |
| BBN's BYBLOS | 70% - 80% | 10X<br>real-time | 12K HMM states<br>768K Gaussians |
| SRI's DECIPHER | 70% - 80% | N/A | 535/569 *Genones*<br>68K/73K Gaussians |
| CMU's SPHINX 3 | 75% - 90% | 10X<br>real-time | 6K HMM states<br>96K Gaussians |

The last major work considered was a project initiated at the University of California at Berkely in 2002. This project attempted to restructure the speech recognition problem into a parallel processing architecture creating a large number of identical processing units and connecting them via aggregator units that manage the traffic on the busses in the system. While only capable of processing a dictionary of approximately 30 words containing just 84 HMM states, it was able to run in real-time with low-power consumption and still achieve recognition rates between 70% - 80% [44, 45]. According to the designers of the system it does scale well and would be able to run efficiently on larger tasks, but even the current implementation requires the use of 20 Xilinx Virtex XCV2000E FPGAs.

## 7.2 MATRIX MATLAB PERFORMANCE PROFILE

When designing code in the MATLAB environment a different approach must be taken to writing efficient algorithms as would be chosen if a similar design were to be done in C/C++. MATLAB was designed specifically for running, high-complexity mathematical algorithms instead of general purpose processing. Because of this design choice, MATLAB is capable of performing very complex mathematics on large matrices with very little effort but does not perform as well in situations requiring searching or large data accesses. One of the major reasons for this behavior lies in the fact that MATLAB is interpreted and not compiled at run-time. This allows vector operations and other mathematical tasks to be performed very efficiently because they are based on a precompiled library. Branching and looping however must be interpreted and therefore take a significantly larger amount of effort to execute than if they were natively run in C/C++. With this in mind it was determined relevant to use the MATLAB code profiler to analyze the code written for the speech recognition system to observe the true implications of this paradigm. The first experiment performed used the MATLAB code profiler to examine the code written for AM to see where the bulk of the computational load occurred. The pie chart in Figure 56 shows the results of the code profile in terms of total execution time spent in each routine.

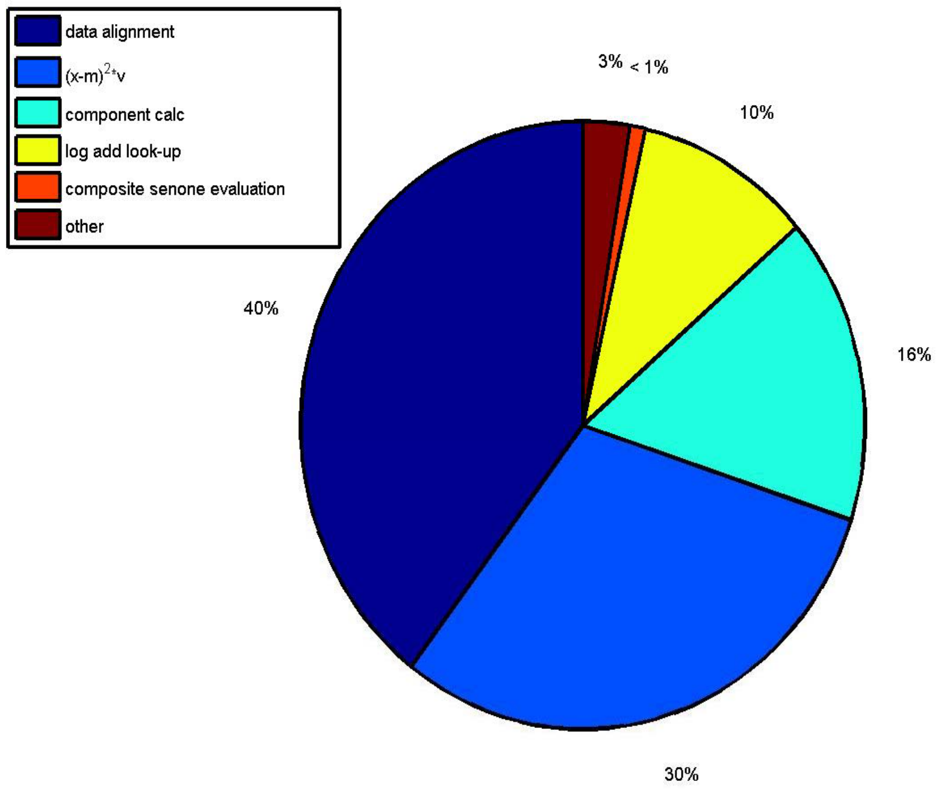


**Figure 56.** Performance analysis of MATLAB Model for Acoustic Modeling

One of the most striking observations in Figure 56 is that 40% of the entire run-time is spent aligning the data necessary to calculate the senones. Before beginning the calculation the incoming feature vector must be replicated 16,000 times in order to gain the most parallelism. This operation however consumes a large portion of the run-time for MATLAB and cannot be reduced. Experiments were conducted to observe the effects of memory pre-allocation, iterative calculation, and dimension manipulation all of which proved inferior to the chosen method. While it is concerning that 40% of the calculation time is spent performing non-mathematical operations this has been determined to be a property of the MATLAB computing environment and something that must be accepted. Of the remaining 60% of the execution time for the MATLAB model, half of this is spent performing the calculation of the Mahalanobis Distance. It has already been shown that this calculation is the most heavily used calculation in the entire system, so observing that it takes the most time fits well with our expectations. The rest of the execution time for the MATLAB code is spent performing the composite senone calculation and the log-add LUT, neither of which showed themselves to be of any major computational significance. On average, the MATLAB code for AM takes 4X real-time to process a frame of data but since this code serves only to exhibit the latent parallelism in the AM calculation and not to drive a functional device, this is of little concern.

After quantifying the AM block the code for PE was profiled to observe where the software was spending most of its time. Using the MATLAB code profiler to examine the work load of PE for 10,000 frames the pie chart in Figure 57 was generated to help visualize the results. As was to be expected the majority of the time spent in PE is taken up by the large amount of data allocation that must be performed before beginning the HMM pipeline. Surprisingly however it is observed that the pruning of the HMMs actually takes more time than the calculation of the HMMs themselves. This relates to the prune block necessitating the use of the *'sort()'* and *'find()'* as discussed in Section 5.3. While these commands are quite powerful when applying a search criterion to an entire matrix, they do consume a large amount of time with respect to the arithmetic operations that MATLAB was originally designed for.

The other portion of the code that consumes a significant amount of the runtime is the storing of the HMM data back into the central database after processing. During this process the *H* values of Equations 23-27 are stored in the database and the input probability for each HMM is reset to *–INF*. Since this requires the updating of a large number of fields in the database the overall access time to the database is increased resulting in even more time lost to data management. Although it is relevant to note the excessive amount of time spent performing data allocation / storage, it is more important to note that the software only required 10 seconds to process 100 seconds of speech input, leading us to conclude that even MATLAB code is capable performing large scale HMM calculations at approximately 10X faster than real-time.

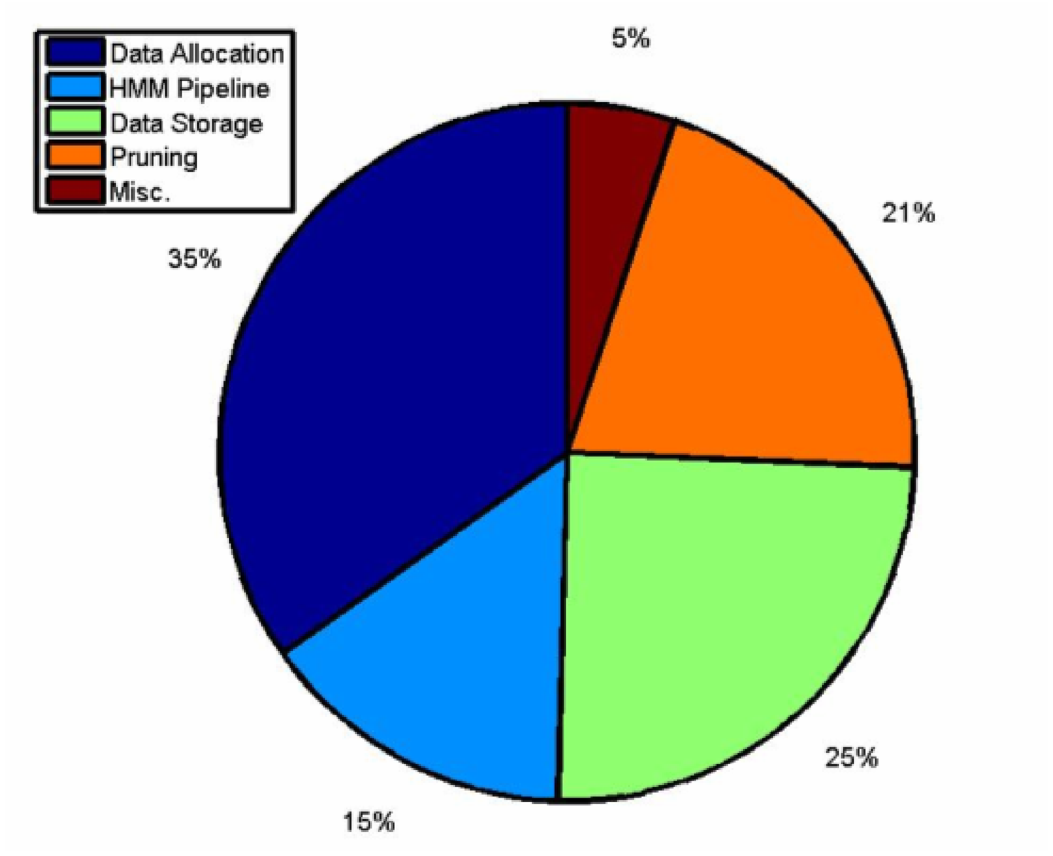


**Figure 57.** Results of Code Profile for PE

Next, in order to properly profile the MATLAB code written for WM, a test frame was chosen in which the WM block was required to perform an average number of tasks on both the dead and exit cues. This frame was then run 10,000 times to observe where the WM process spent the majority of its time. The entire experiment represents the processing of 10 seconds worth of ‘average’ speech input and was executed by the MATLAB routine in only 4.11 seconds. The results of the code profile are shown in a pie chart in Figure 58 with the major sections of code described in Section 6.2 represented by their own slice of the pie.

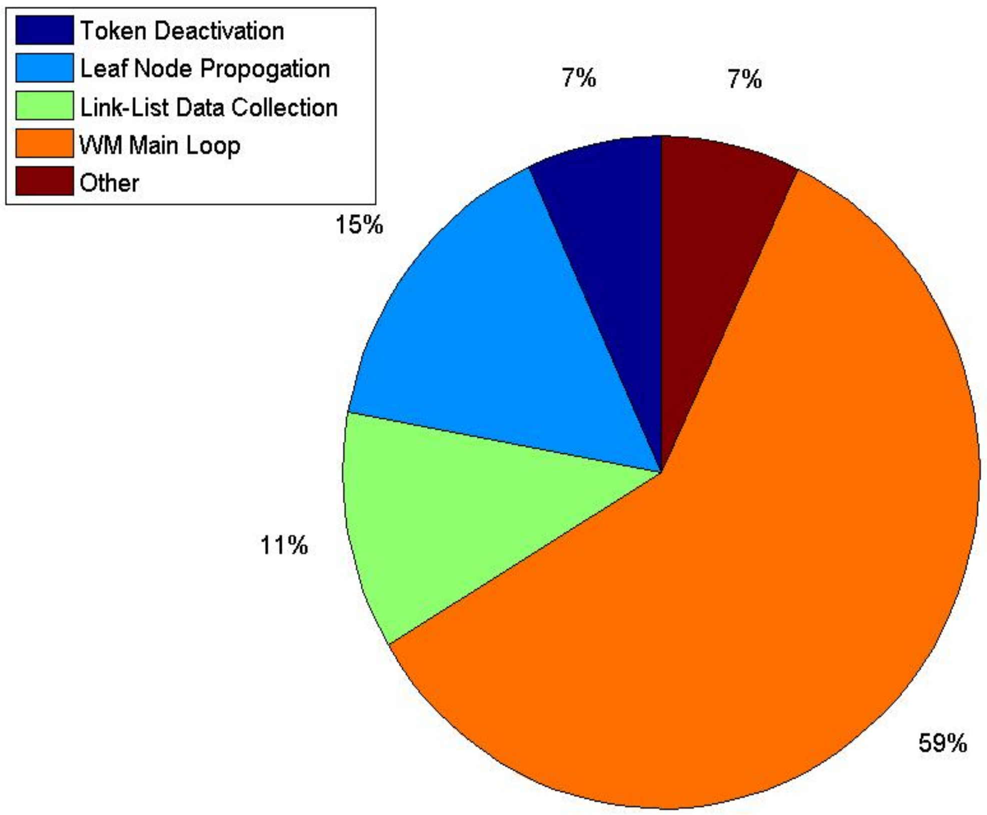


**Figure 58.** Code Profile for WM

As would be expected the main loop of the WM code takes the most significant amount of time both because of the two if-conditionals that it must execute by also because the MATLAB environment was developed for complex mathematical programming. Operations like *for* and *if* statements are not as easily optimized in the MATLAB language as are highly complicated mathematics like those described for the AM design and consequently development of search routine in MATLAB is not entirely desirable. The next largest time consuming block in the code is the propagation of the leaf node tokens to their potential root token counter-parts. Processing the leaf-node propagation requires obtaining information from special ‘word-identification’ source before fetching the link-list data and the additional overhead incurred during this fetch easily account for the amount of time spent performing these operations.

Having looked at the code for the individual stages of the speech recognition process the last observation left was to profile the entire system while it was processing known utterances and observe its performance. Figure 59 shows the results of this profile for 306 frames of speech from the RM1 speech corpus consisting of three different utterances.

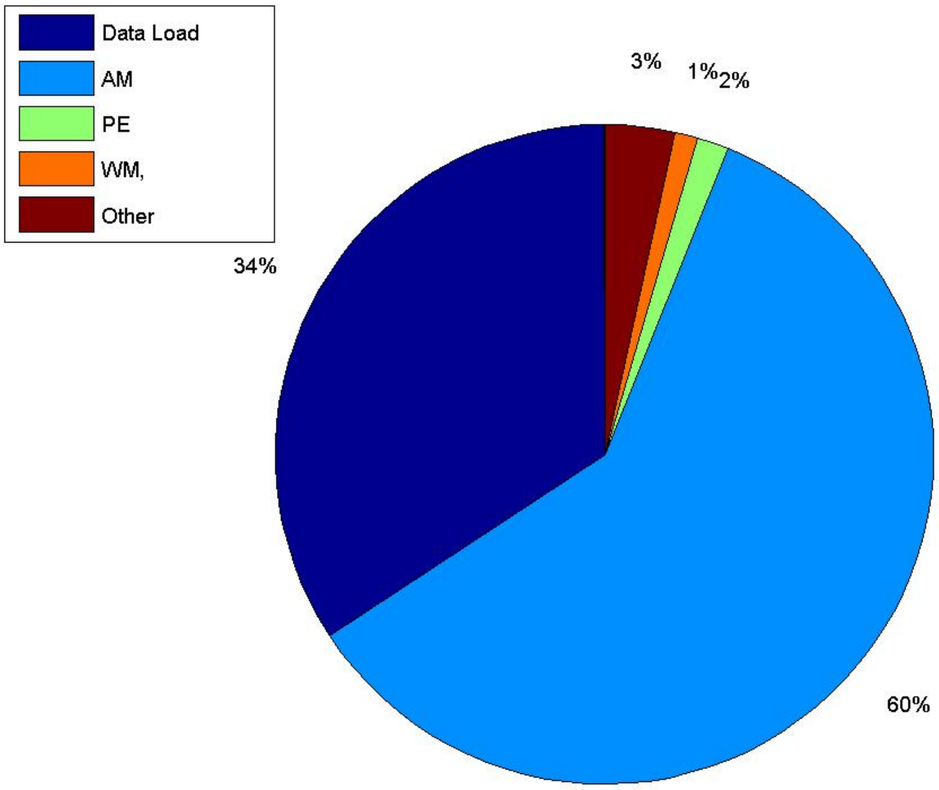


**Figure 59.** Code Profile for Entire MATLAB Code

As expected from our initial observations, the AM portion of the design is by far the most time consuming portion of the code with the data fetch routines coming in a distant second. The sheer volume of calculation that needs performed by AM for every frame of input data dominates this system as well as the original SPHINX model and the derived hardware device. The observation that this portion of the design consumes the lion-share of the execution time for the same algorithm in three different environments confirms the complexity of the calculations and points directly to the portion of the design that needs the most attention in order to ensure all performance goals are met. This profile also helps to show that while during some frame of speech the PE and WM blocks may be very active, on the whole they do not contribute significantly to the total work load of the system over the course of multiple utterances.

The profiling of the MATLAB code and testing of its ability to recognize a number of different utterances as well as sequences of utterances effectively concludes the software development portion of the project. This work illustrated the latent parallelism found in a high-performance, high-bandwidth application, SPHINX 3, and exploited that parallelism to derive compact, vectorized algorithm for performing the same task in the MATLAB environment. This development help to quantify both the specific recognition engine and the MATLAB software environment, while also provided the foundation for the design of custom hardware capable of performing the speech recognition task at speeds sufficient for real-time operation.

## 7.3 HARDWARE PERFORMANCE SUMMARY

The hardware development presented in this work presents a novel processing architecture capable of executing the CSR algorithm at over 100MHz. As discussed in Chapter 3, 100MHz has been determined to be the minimum operating speed for a device to process speech in real-time. This requirement comes from the known input frame rate of 10ms and the proposed maximum cycle count of one million. Preliminary results on the entire operational system have shown a device running at 105MHz on a Vertix-4 SX35 ff668-10 and requiring less than 800,000 cycles to complete all necessary operations. These results clearly show a system able to run at sufficient speeds for real-time recognition as well as maintain an average of 20% down-time during which the engine is inactive.

Aside from being able to recognize human speech in real-time, special attention was paid to ensure that the throughput of the design was maximized via the creation of custom pipelines for each stage of the algorithm. The first major portion of the algorithm, Acoustic Modeling, has been shown to be the most computational intensive part of the problem and much effort was taken to design this block as efficiently as possible. The result is a custom hardware pipeline capable of operating at 125MHz post-place-and-route on a Virtex-4 SX35. This pipeline is completely data-driven and involves no internal state machines to guide the process. By giving the design this flexibility the complexity of the inputs can be varied without needing to reconfigure the design.

During the design of the Phoneme Evaluation stage the large data access problem encountered was effectively reduced through the used of multiple small parallel ROMs and pointer arrays. When processing a Hidden Markov Model a large amount of data must first be retrieved to perform the calculations. While the need for moving such large quantities of data within the design adversely effects the performance of the pipeline, speeds of 111MHz are post-place-and-route are still possible, with the core of the processing unit able to operate as fast as 140MHz post-place-and-route.

The final portion of the design, the Word Modeler, involves the design of a tree-search algorithm in hardware. This work was complete by the ECE 2121 Hardware Design II course taught during the spring term of 2005 by Prof. Raymond Hoare and is presented in this thesis for sake of completeness. The hardware was designed as a large link-list evaluation unit capable of propagation information throughout the tree while also deactivating nodes in the tree and connecting multiple trees for the creation of word strings. The deactivation portion of the hardware is capable of running at 170MHz post-place-and-route but the activation logic can only operate at 120MHz, limiting the overall performance of the Word Modeler but not impacting the overall performance of the system. Table 9 summarized the performance results for the major portions of the hardware development.

**Table 9.** Summary of Hardware Performance Results

| Component | Synthesis (MHz) | Place-and-Route (MHz) | Area | Cycle Count |
|---|---|---|---|---|
| Gaussian Distance Pipeline | 157 | 145 | 6 DSP Tiles, 411 Slices | 10 cycle/gauss |
| Log-Add Look-up | 164 | 150 | 13 BRAMs, 307 Slices | 10 cycle/comp |
| **AM Block Total** | **164** | **125** | **6 DSP Tiles, 30 BRAMS, 1328 Slices** | **162 cycle/senone 640K cycle total** |
| Hidden Markov Model Pipeline | 261 | 140 | 775 Slices | 8 cycle/load 5 cycle/calc |
| Pruner | 277 | 177 | 112 Slices | 4 cycle/HMM |
| **PE Block Total** | **115** | **111** | **84 BRAMs, 1866 Slices** | **22 cycle/HMM** |
| Token Deactivator | 377 | 170 | 54 Slices | 2 cycle/dead HMM |
| Token Activator | 184 | 120 | 3 BRAMs, 160 Slices | 10*branch cycle/active HMM |
| **WM Block TOTAL** | **166** | **129** | **3 BRAMs, 414 Slices** | **22 cycle/dead HMM + 10*branch cycle/active HMM** |

# 8.0 CONCLUSIONS, CONTRIBUTIONS, AND FUTURE DIRECTIONS

## 8.1 CONCLUSIONS

This work has shown the full design of an automatic speech recognition system in to highly-vectorized MATLAB and custom VHDL. By studying an array of existing technologies and analyzing their strengths and weaknesses as well as their computational algorithms, a methodology was derived that would highlight the advantages of many of the systems. Specifically, by basing our work on a well-known industry accepted software package, SPHINX 3 by CMU, it was known that any implementation that could emulate the algorithm accurately would be capable of being used for real-world speech applications. The use of MATLAB and SIMULINK allowed for the development of code capable of operating on very large matrices which served to show the latent parallelism present in the sequential algorithms of SPHINX. Additionally, the MATLAB code provided a compact easy to understand representation of the entire speech recognition process that was then able to be transformed into high-performance hardware devices. The development of the hardware portion of the project expanded on the idea of large matrix-based operations to create high-throughput hardware blocks capable of performing the necessary tasks in real-time. By determining the potential cycle count for the system and then fitting that count to the pre-determined input frame rate, our target operating frequency was chosen to drive the design process. Creating this hard boundary on the clock speed ensured that our hardware would be able to process all events in real-time assuming our cycle budget is not exceeded. Certain portions of the design like the AM block also highlighted the ability to create a completely data-driven pipeline which allows the functionality of the circuit to be re-configured by simply changing the inputs to the pipe.

## 8.2 CONTRIBUTIONS

During the course of this research I have worked on numerous portions of the project requiring various skills. My work on this project began as an undergraduate student working to quantify the functional blocks of the SPHINX 2 system and subsequently performing in-depth precision analysis on the multi-variant Gaussian Distribution calculations performed in the system. This work then led to the functional analysis of the SPHINX 3 algorithm and the observation of any parallelism implied in the code. From this analysis I implemented the major functionality of the algorithm in less than 200 of highly efficient matrix-based MATLAB code. Additionally, I have taken my understanding of the speech recognition algorithm and used it to develop a hardware architecture capable of performing this algorithm and shown its functionality on a Virtex-4 SX35 FPGA device. During this hardware development I created a number of highly efficient hardware devices, including a Hidden Markov Model processing unit and a Multi-Variant Gaussian Distribution based Acoustic Modeling unit both capable of operating at over 100 MHz.

The first major contribution to this thesis is the examination and extraction of the parallelism from speech processing algorithms. Current software systems operate sequentially and with very poor ILP, but examination of the mathematics reveals a process possessing a large amount inherent parallelism. By first showing the parallelism via the equations and then generating MATLAB scripts to illustrate how to take advantage it, a method is presented for performing speech recognition as a highly vectorized process. Basing this work on the recognition of the RM1 speech corpus containing approximately 1,000 words, the MATLAB code shows the potential to perform upto 620K operations independently of each other at the lowest level of the systems, and upto 4K at the higher level of the process. The results of this contribution serve to further the development of systems able to fully exploit the parallelism of the speech recognition process and process naturally spoken speech in real-time.

The second contribution of this work builds on the results of the first contribution to produce a hardware system able to utilize some of the inherent parallelism in order to perform speech recognition in real-time. By focusing on the most computationally intensive portions of the speech recognition algorithm and designing pipelined logic to tackle these operations a system was created able to operate on a 1,000 word dictionary at over 100MHz. The Acoustic Modeling pipeline built for the system operates with a parallelism of 20, much smaller than the potential 620K but sufficient to produce a real-time system. Further, the pipeline created can be replicated numerous times inorder to increase the parallelism if needed. To help enhance the final functionality of the hardware the Acoustic Modeling unit was designed to be completely input driven, possessing no internal finite state machine. By designing the device in this fashion, the complexity of the inputs can be varied as well as the dimensionality of the problem. This variability allows the hardware to be customized to many different environments and languages by simply changing the data located in the external RAMs. These results combined with the use of a token passing scheme between the later stages of the design create a system able to operate with maximum throughput while simultaneously minimizing the amount of active data in the system. The final design is able to operate at over 100MHz and requires less than 850,000 cycles to complete providing ample buffer room around the 1 million cycle budget allotted for the process.

## 8.3 FUTURE DIRECTIONS

Possible future directions for this research include the development of a fully functioning prototype that could be demonstrated in real situational environments. Aside from the fabrication and platform details involved in this task the major complication would be the derivation of full-scale speech models for the system to run on and the subsequent porting of these values to an external memory as opposed to the current internal memory configuration. While marketing the entire system is the most desirable scenario considering the potential for a compact hardware-based speech recognition system, the potential also exists for the packaging of the individual hardware component for use in third-party development of speech processing systems. The functional units designed for both the AM and the PE portion of the design are compact, self-contained units requiring little direction from a global control mechanism, making them ideal devices for release as soft-IP cores.

Completely away from the potential for the hardware designed on this project are the potential directions for the MATLAB code written during this research. The MATLAB code represents a highly-efficient algorithm for performing speech recognition and could be used as both a development platform as well as benchmark for the design of future systems. Having proven the ability to execute medium sized tasks such as the RM1 speech corpus it would also next be desirable to expand the MATLAB model to work on a large sized dictionary such as the HUB-4 corpus.

While the immediate directions for a project for this nature may be simple enough to quantify, the long-term potential for the development of highly-efficient, highly-accurate speech recognition devices is hard to speculate on. As speech recognition begins to permeate our lives and we slowly begin to speak to electronics in the same fashion we speak to each other, the potential benefit of research like this will only gain momentum helping to reach the end goal of a truly interactive electronic world.

# APPENDIX A

## MATLAB CODE

A.1 – RM1_top_level.m

```
%%9-3-05 -- JWS : changed feature_frame matrix to read in value from DSP
%4-5-05 -- JWS: added values to apply global beam offsets to attempt
%%better recognition rates
%%3-27-05 -- JWS: creation of top level code for RM1 began.  this code will
%%cover all aspects of the SPHINX 3 system from Acoustic Modeling through
%%the evaluation of the word model
%%DEFINE THE BEAM OFFSET
valid_offset=-200000;%%-100000%%-200000
exit_offset=-100000;%%-60000%%-100000
word_offset=-5000;%%-5000

%%fid =fopen('Utterances\our data\utterances.txt');
fid =fopen('Utterances\utterances.txt');
set_tmp = textscan(fid,'%s');
set_tmp = set_tmp{1};
fclose(fid);

%%stmp = 1 ==> 'CASUALTY' --> dict_entry 16 ('capabilities' is 8)
%%stmp = 4 ==> 'CAPACITIES' --> dict_entry 11 ('capacity' is 12)
%%stmp = 5 ==> 'CALIFORNIA' --> dict_entry 2 ('caledonia' is 1')
```

```
[RMmeans,constant,RM_mixwt,RMvar_precomp,LUT_logs3_add,mdef,cstates,tmat,START_A
DDR,NEXT_ADDR,LCROOT_PTR,DICT]=data_loader;
for stmp =1% [1 4 5]%1:length(set_tmp);%%loop through for every available test set
   utterance = set_tmp{stmp};
   utt_class = utterance(1:6);
   file_path = strcat('new_test_sets\',utt_class,'\',utterance,'.feat');
   feature_frame = load(file_path);
   senones = zeros(length(RM_mixwt)/8,1);
   PTR_RAM = load('EE2121_test_data\PH_PTR_RAM.mat');%%load the phone pointer RAM
structure
   PTR_RAM = int32(PTR_RAM.PTR_RAM);
   NEW_PAL = [];%%this will be the vector that represents what is in the NEW_PAL and PAL
FIFOs
   PAL = [];
   word_thresh = 0;
   disp(strcat('begining analysis for utterance -->',utterance))
   fi_sscr_rel = load(strcat('EE2121_test_data\',utterance,'.mat'));
   fi_sscr_rel = fi_sscr_rel.fi_sscr_rel;
   for frame =1%:length(feature_frame(:,1))%%loop through for every frame of current file
      senones                                                                          =
acoustic_modeler_comp_senones(feature_frame(frame,:),RMmeans,RMvar_precomp,constant,R
M_mixwt,LUT_logs3_add,cstates);
      if (frame == 1) | ((numel(find(PAL==451))==0) & (numel(find(PAL==451))==0))%%loop
to check for absence of silence token
         NEW_PAL = [NEW_PAL; 451];
         PTR_RAM(451,4) = PTR_RAM(451,10)+word_thresh;
      end
      [valid, dead, exit, word_thresh, PH_PTR_RAM] = HMM_eval_no_comps(PTR_RAM,
senones, tmat, mdef, PAL, NEW_PAL,valid_offset,exit_offset);
      PAL = valid;
```

```
        [PTR_RAM, NEW_PAL, word_out, score]=word_model(PH_PTR_RAM, dead, exit,
word_thresh, START_ADDR, NEXT_ADDR, LCROOT_PTR,word_offset);
        NEW_PAL = NEW_PAL';
        tmp=find(word_out ~=18);%%find all valid words other than <SIL>
        if numel(tmp)~=0%%if there is a valid word during the current frame, output it
            word={DICT{word_out(tmp)} score(tmp)' frame}
        end
    end
end
```

## A.2 – data_loader.m

```
function
[means,constant,mixwt,var,LUT,mdef,cstates,tmat,start_addr,next_addr,lcroot,dict]=data_loader;

disp('Loading Acoustic Modeling data...')
means = load('RM1_data\RM1_mean.txt');
constant = load('RM1_data\RM1_LRD.txt');
mixwt = load('RM1_data\RM1_mixwt.txt');
var=load('RM1_data\RMvar_precomp.txt');
LUT=load('LUT_logs3_add.mat');
LUT=LUT.LUT_logs3_add;

disp('Loading HMM calculation data...')
mdef=load('EE2121_test_data\mdef.mat');%%load the mdef file from sphinx.
mdef=mdef.mdef;%%this is the mdef with the component scores added in
%cstates = load('EE2121_test_data\cstates.mat');%%load the matrix of possible senones for
composites
cstates = load('EE2121_test_data\cstates_old.mat');%%load the matrix of possible senones for
composites
cstates = cstates.cstates;
```

```
tmat = load('RM1_data\TMAT_RM1.txt');%%load the transition matrices

disp('Loading Word Model data...')
START_ADDR =load('EE2121_test_data\word_calc\START_ADDR.mat');          %%stmp = 1
==> 'CASUALTY' --> dict_entry 16 ('capabilities' is 8)
start_addr = int32(START_ADDR.START_ADDR);                                %%stmp = 4 ==>
'CAPACITIES' --> dict_entry 11 ('capacity' is 12)
NEXT_ADDR =load('EE2121_test_data\word_calc\NEXT_ADDR.mat');        %%stmp = 5 ==>
'CALIFORNIA' --> dict_entry 2 ('caledonia' is 1')
next_addr = int32(NEXT_ADDR.NEXT_ADDR);
LCROOT_PTR =load('EE2121_test_data\word_calc\LCROOT_PTR.mat');
lcroot = int32(LCROOT_PTR.data);
DICT = load('EE2121_test_data\DICT.mat');
dict = DICT.dict;
```

A.3 – acoustic_modeler_comp_senones.m

```
function                                   [senones]                                   =
acoutic_modeler_comp_senones(features,means,variances,constant,mix_wt,LUT,cstates)

f = 1/log(1.0003);                  %% constant for senone calculation
mark = 1:8:length(mix_wt);%%pointer to first component of each senone
feat_mat = repmat(features,length(means(:,1)),1);%%replicate input feature stream to allow for
vector ops
partial_calc = ((feat_mat-means).^2).*variances;%%calculate all gaussian probabilities
```

```
component_calc = mix_wt+f.*(constant-sum(partial_calc,2));%% weight all scores and convert
to the logs3 domain
senone_calc=component_calc(mark)';%%place first component of each senone into senone
evaluation matrix
temp = reshape(component_calc,8,1935);%%reshape component score matrix for senone scoring
for d=2:8
   senone_calc = logs3_add_look_up_float(senone_calc(1,:),temp(d,:),LUT);%%log add 2nd
through 8th components for final senone score
end
best = max(senone_calc);%%find best senone for normalization factor
sen = senone_calc-best;
for a=1:length(cstates(:,1))%%calulate the scores for all composite senones
   tmp=find(cstates(a,:)~=0);
   comp_sen(a)=max(sen(cstates(a,tmp)));
end

senones=int32([sen comp_sen]);
```

A.4 – logs3_add_look_up_float.m

```
function [result] = logs3_add(val1,val2,LUT)

% This function is designed to replicate Sphinx's logs3_add() function
% implemented in the logs3.c file.
%%shortened implementation of Kshitij Guptas fn_logs2_add
c0=val1 > val2;
c1=~c0;
d0=val1-val2;   r0=val1;
```

```
d1=val2-val1;   r1=val2;
d0=c0.*d0;      r0=c0.*r0;
d1=c1.*d1;      r1=c1.*r1;
d=d0+d1;        r=r0+r1;
d = int32(d);
d = d+1;
greater = d > 30001;
d(greater)= 30001;
result = r + LUT(d);
```

A.5 – HMM_eval_no_comps.m

```
function [valid, dead, exit, word_thresh, PH_PTR_RAM] = HMM_eval(PTR_RAM, senones,
tmat, mdef, PAL, NEW_PAL,valid_offset,exit_offset)

dead=[];
exit=[];
val_ID = [NEW_PAL; PAL];


%%%%%BEAMS HAVE BEEN CHANGED%%%%%
hmm_beam = -307006;%%any score above this beam can be seen as a valid hmm score
phone_beam = -230254;%%any score above this beam can be seen as a valid exit score
hmm_beam = int32(hmm_beam+valid_offset);%%create valid beam
phone_beam = int32(phone_beam+exit_offset);%%create exit beam
tmat_pointer = mdef(PTR_RAM(val_ID,3)+1,2)*3;%%create a vector of starting points for each
HMMs tmat
tmat11 = int32(tmat(tmat_pointer+1,1));%%gather all necessary tmat values for each element of
val_ID
tmat12 = int32(tmat(tmat_pointer+1,2));
```

```
tmat22 = int32(tmat(tmat_pointer+2,2));
tmat23 = int32(tmat(tmat_pointer+2,3));
tmat33 = int32(tmat(tmat_pointer+3,3));
tmat3e = int32(tmat(tmat_pointer+3,4));
senone1 = senones(mdef(PTR_RAM(val_ID,3),3)+1)';%%gather all necessary senone scores for
each element of val_ID
senone2 = senones(mdef(PTR_RAM(val_ID,3),4)+1)';
senone3 = senones(mdef(PTR_RAM(val_ID,3),5)+1)';
score0=max([PTR_RAM(val_ID,4) PTR_RAM(val_ID,5)+tmat11],[],2);%%evaluate 1st state of
HMM & remember where best was
score0 = score0+senone1;
score1=max([PTR_RAM(val_ID,5)+tmat12      PTR_RAM(val_ID,6)+tmat22],[],2);%%evaluate
2nd state of HMM
score1 = score1+senone2;
score2=max([PTR_RAM(val_ID,6)+tmat23      PTR_RAM(val_ID,7)+tmat33],[],2);%%evaluate
3rd state of HMM
score2=score2+senone3;
scoreE=score2+tmat3e;%%calulate the exit prob for the HMM
PTR_RAM(val_ID,4)=repmat(-939524096,length(val_ID),1);%%reset all active HMMs input
scores to -INF
PTR_RAM(val_ID,9) = max([score0 score1 score2],[],2);%%find best score for current HMM
PTR_RAM(val_ID,5:8)= [score0 score1 score2 scoreE];%%update HMM values for current
HMM
[B_HMM,ID] = sort(PTR_RAM(val_ID,9),'descend');%%sort HMM scores for current frame
dd = find(PTR_RAM(val_ID,9) < B_HMM(1)+hmm_beam);%%find HMMs below the valid
beam
dead=val_ID(dd);
vd = find(PTR_RAM(val_ID,9) >= B_HMM(1)+hmm_beam);%%find HMMs above the valid
beam
valid=val_ID(vd);
PTR_RAM(valid,1) = 1;%%make sure that all valid HMMs have their token active bit set
```

```
et = find(PTR_RAM(val_ID,8) >= B_HMM(1)+phone_beam);%%find HMMs above the exit
beam
exit=val_ID(et);
word_ends = find(PTR_RAM(val_ID,2) == 1);
word_thresh = max(PTR_RAM(val_ID(word_ends),8));%%find the word threshold value to
pass to the word model
PH_PTR_RAM = PTR_RAM;
```

A.6 – word_model.m

```
function [PTR_RAM, NEW_PAL, word_out, score]=word_model(PH_PTR_RAM, dead, exit,
word_thresh, START_ADDR, NEXT_ADDR, LCROOT_PTR,word_offset)
score=[];
NEW_PAL =[];
tmp_root = [];
word_out = [];
PH_PTR_RAM(dead,1) = 0;%%reset token active bit for all dead tokens
PH_PTR_RAM(dead,4:9) = -939524096;%%reset the input, output, max, and state scores for all
dead tokens
if numel(exit)~=0%%check to make sure file exists (certain frames will not have any exiting
phones so this loop is not necessary in those cases)
   word_beam = int32((word_thresh - 153503)+ word_offset);%%create the word beam from the
word threshold
   WE = find(PH_PTR_RAM(exit,2) == 1);%%find all word ends in the valid exit tokens
   NWE = find(PH_PTR_RAM(exit,2) == 0);%%find all non-word ends in the valid exit tokens
   next_addr0 = START_ADDR(exit(NWE),1);%%create a next address pointer
   count0 = START_ADDR(exit(NWE),2);%% create a count vector to indicate how many
jumps are in a given link-list
   tmp = find(PH_PTR_RAM(exit(WE),8) >= word_beam);%%find word end phones with
scores above the threshold
   tmp_root = PH_PTR_RAM(exit(WE(tmp)),11);%%find all LCROOT_PTR start addresses
```

```
  word_out = PH_PTR_RAM(exit(WE(tmp)),11)+1;%%apply offset to start addresses to obtain
dict entry
  score = PH_PTR_RAM(exit(WE(tmp)),9);%%find score for each valid word end token
  WE1 = WE(tmp);
  exits = [exit(NWE); exit(WE1)];%%create a shortened list of exit phones given that bad word
ends have been removed
  next_addr1 = START_ADDR(LCROOT_PTR(tmp_root,2),1);%%create a next address
pointer for all entrances into new trees
  count1 = START_ADDR(LCROOT_PTR(tmp_root,2),2);%%creat a count vector to indicate
number of trees to enter
  next_addr = [next_addr0; next_addr1];
  count = [count0; count1];
  for b=1:length(next_addr)
    for c=1:count(b)
      new_tkn = NEXT_ADDR(next_addr(b)+c-1);
      if                    PH_PTR_RAM(new_tkn,4)                    <
PH_PTR_RAM(exits(b),8)+PH_PTR_RAM(new_tkn,10)%%if out score of current node is
better than in score of branch node
        PH_PTR_RAM(new_tkn,4)                                        =
PH_PTR_RAM(exits(b),8)+PH_PTR_RAM(new_tkn,10);%%update branch node in score
      end
      if PH_PTR_RAM(new_tkn,1) == 0%%if token is not already active, activate it and put it
in the NEW_PAL
        PH_PTR_RAM(new_tkn,1) = 1;
        NEW_PAL = [NEW_PAL new_tkn];
      end
    end
  end
end
PTR_RAM = PH_PTR_RAM;
```

# APPENDIX B

## VHDL CODE

B.1 – AM_calc.vhd

```
----------------------------------------------------------------------------------
-- Company:   University of Pittsburgh
-- Engineer:  Jeffrey W. Schuster
--
-- Create Date:    11:22:02 11/01/05
-- Design Name:
-- Module Name:    AM_top - Behavioral
-- Project Name:
-- Target Device:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

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

entity AM_calc is
   Port ( clk          : in std_logic;
          ce           : in std_logic;
          sclr         : in std_logic;
          new_frame    : in std_logic;
                          feat_data     : in std_logic_vector(31 downto 0);
                          feat_addr     : out std_logic_vector(7 downto 0);
          senone       : out std_logic_vector(31 downto 0);
                          senone_rdy    : out std_logic;
                          senone_wr_addr : out  std_logic_vector(12 downto 0);
                          senone_wr     : out std_logic;
                          max_done      : out std_logic;
                          normalize_done : out std_logic;
          job_done     : out std_logic
                          );
end AM_calc;

architecture Behavioral of AM_calc is

component gaus_top is
   Port (
                          input_stall_in  : in std_logic;
                          clk           : in std_logic;
                          ce            : in std_logic;
                          sclr          : in std_logic;
                  x             : in std_logic_vector(31 downto 0);
```

```
                        x_valid        : in std_logic;
         mw            : in std_logic_vector(31 downto 0);
                        mw_valid       : in std_logic;
         vk            : in std_logic_vector(31 downto 0);
                        vk_valid       : in std_logic;
                        status         : in std_logic_vector(6 downto 0);
                        status_valid   : in std_logic;
         new_frame     : in std_logic;
--                      senone_rd_addr  : in std_logic_vector(12 downto 0);
--                      senone_addr_rdy : in std_logic;
--                      senone_ram_sel  : in std_logic;
                        SENONE         : out std_logic_vector(31 downto 0);
                        SENONE_WR_ADDR  : out std_logic_vector(12 downto 0);
                        SENONE_WR      : out std_logic;
                        SENONE_ready   : out std_logic;
         max_done      : out std_logic;
                        normalize_done  : out std_logic;
                        composite_done  : out std_logic;
                        input_stall    : out std_logic

                        );
end component;

component MW_rom is
         PORT(
                  clk     : in std_logic;
                  addr    : in std_logic_vector(7 downto 0);
                  dout    : out std_logic_vector(31 downto 0)
                  );
end component;
```

```
component VK_rom is
        PORT(
                clk      : in std_logic;
                addr     : in std_logic_vector(7 downto 0);
                dout     : out std_logic_vector(31 downto 0)
                );
end component;

--component X_rom is
--      PORT(
--              clk      : in std_logic;
--              addr     : in std_logic_vector(7 downto 0);
--              dout     : out std_logic_vector(31 downto 0)
--              );
--end component;

component status_rom is
        PORT(
                clk     : in std_logic;
                addr    : in std_logic_vector(7 downto 0);
                dout    : out std_logic_vector(6 downto 0)
                );
end component;

SIGNAL senone_rd_addr  : std_logic_vector(12 downto 0);
SIGNAL senone_addr_rdy : std_logic;
SIGNAL senone_ram_sel  : std_logic;
SIGNAL MW_valid     : std_logic;
SIGNAL VK_valid     : std_logic;
SIGNAL X_valid      : std_logic;
SIGNAL status_valid : std_logic;
SIGNAL MW_addr      : std_logic_vector(7 downto 0);
SIGNAL VK_addr      : std_logic_vector(7 downto 0);
SIGNAL X_addr       : std_logic_vector(7 downto 0);
SIGNAL status_addr  : std_logic_vector(7 downto 0);
SIGNAL mw           : std_logic_vector(31 downto 0);
SIGNAL vk           : std_logic_vector(31 downto 0);
SIGNAL x            : std_logic_vector(31 downto 0);
SIGNAL status       : std_logic_vector(6 downto 0);
SIGNAL done         : std_logic;
SIGNAL input_stall  : std_logic;
SIGNAL max_done_tmp : std_logic;
SIGNAL norm_done    : std_logic;
SIGNAL freeze       : std_logic;
SIGNAL new_frame_p1 : std_logic;
SIGNAL new_frame_p2 : std_logic;
SIGNAL new_frame_p3 : std_logic;
SIGNAL new_frame_p4 : std_logic;
SIGNAL new_frame_p5 : std_logic;
SIGNAL new_frame_p6 : std_logic;
SIGNAL new_frame_p7 : std_logic;
SIGNAL new_frame_p8 : std_logic;
SIGNAL new_frame_p9 : std_logic;
SIGNAL new_frame_p10 : std_logic;
SIGNAL new_frame_p11 : std_logic;
SIGNAL new_frame_p12 : std_logic;
SIGNAL new_frame_p13 : std_logic;
SIGNAL input_stall_internal : std_logic;

begin
senone_rd_addr<=(OTHERS=>'0');
senone_addr_rdy<='0';
```

```
senone_ram_sel<='0';
--job_done<=done;
--max_done<=max_done_tmp;

process (clk, ce, new_frame, sclr, input_stall)
begin
        if(clk = '1' and clk'event)then
                if(sclr = '1')then
                        MW_valid<='0';
                        VK_valid<='0';
                        X_valid<='0';
                        status_valid<='0';
                        MW_addr<=(OTHERS=>'0');
                        VK_addr<=(OTHERS=>'0');
                        X_addr<=(OTHERS=>'0');
                        status_addr<=(OTHERS=>'0');
                        max_done<='0';
                        normalize_done<='0';
                        job_done<='0';
                        input_stall_internal<='0';
                        new_frame_p1<='0';
                        new_frame_p2<='0';
                        new_frame_p3<='0';
                        new_frame_p4<='0';
                        new_frame_p5<='0';
                        new_frame_p6<='0';
                        new_frame_p7<='0';
                        new_frame_p8<='0';
                        new_frame_p9<='0';
                        new_frame_p11<='0';
                        new_frame_p11<='0';
                        new_frame_p12<='0';
                        new_frame_p13<='0';
                elsif (ce = '1')then
                        x<=feat_data;
                        feat_addr<=X_addr;
                        new_frame_p1<=new_frame;
                        new_frame_p2<=new_frame_p1;
                        new_frame_p3<=new_frame_p2;
                        new_frame_p4<=new_frame_p3;
                        new_frame_p5<=new_frame_p4;
                        new_frame_p6<=new_frame_p5;
                        new_frame_p7<=new_frame_p6;
                        new_frame_p8<=new_frame_p7;
                        new_frame_p9<=new_frame_p8;
                        new_frame_p10<=new_frame_p9;
                        new_frame_p11<=new_frame_p10;
                        new_frame_p12<=new_frame_p11;
                        new_frame_p13<=new_frame_p12;
                        input_stall_internal<=input_stall OR new_frame_p13;
                        if(new_frame = '1')then
                                max_done<='0';
                                normalize_done<='0';
                                job_done<='0';
                        end if;
                        if(max_done_tmp = '1')then
                                max_done<=max_done_tmp;
                        end if;
                        if(norm_done = '1')then
                                normalize_done<=norm_done;
                        end if;
                        if(done = '1')then
                job_done<=done;
end if;
if(max_done_tmp = '1')then
        freeze<='1';
        MW_valid<='0';
        VK_valid<='0';
        X_valid<='0';
        status_valid<='0';
        MW_addr<=(OTHERS=>'0');
        VK_addr<=(OTHERS=>'0');
        X_addr<=(OTHERS=>'0');
        status_addr<=(OTHERS=>'0');
elsif(new_frame = '1')then
        freeze <= '0';
        MW_valid<=new_frame;
        VK_valid<=new_frame;
        X_valid<=new_frame;
        status_valid<=new_frame;
        MW_addr<="00000001";
        VK_addr<="00000001";
        X_addr<="00000001";
        status_addr<="00000001";
elsif(freeze = '0')then
        if(input_stall = '0')then
                MW_addr<=MW_addr+1;
                VK_addr<=VK_addr+1;
                X_addr<=X_addr+1;
                status_addr<=status_addr+1;
                MW_valid<='1';
                VK_valid<='1';
                status_valid<='1';
```

```
                    X_valid<='1';
                else
                    MW_valid<='0';
                    VK_valid<='0';
                    status_valid<='0';
                    X_valid<='0';
                end if;
            else
                MW_valid<='0';
                VK_valid<='0';
                X_valid<='0';
                status_valid<='0';
                MW_addr<=(OTHERS=>'0');
                VK_addr<=(OTHERS=>'0');
                X_addr<=(OTHERS=>'0');
            status_addr<=(OTHERS=>'0');
            end if;
        else
            MW_valid<='0';
            VK_valid<='0';
            X_valid<='0';
            status_valid<='0';
            MW_addr<=(OTHERS=>'0');
            VK_addr<=(OTHERS=>'0');
            X_addr<=(OTHERS=>'0');
          status_addr<=(OTHERS=>'0');
            max_done<='0';
            normalize_done<='0';
            job_done<='0';
        end if;
    end if;
```

```
end process;


gaus_calc : gaus_top
        PORT MAP(
                        input_stall_in => input_stall_internal,
                        clk         => clk,
                        ce          => ce,
                  sclr        => sclr,
                        x           => x,
                  x_valid     => x_valid,
        mw          =>mw,
                  mw_valid    =>mw_valid,
        vk          =>vk,
                  vk_valid    =>vk_valid,
                  status      =>status,
                  status_valid =>status_valid,
        new_frame   => new_frame,
--                senone_rd_addr  =>senone_rd_addr,
--                senone_addr_rdy =>senone_addr_rdy,
--                senone_ram_sel  =>senone_ram_sel,
                  SENONE         =>senone,
                  SENONE_ready    =>senone_rdy,
                        senone_wr_addr  => senone_wr_addr,
                        senone_wr       =>senone_wr,
        max_done       => max_done_tmp,
                  normalize_done  => norm_done,
                  composite_done  => done,
                        input_stall     => input_stall
                        );
MW_rom_test : MW_rom
```

```
          PORT MAP(
                              clk => clk,
                              addr => MW_addr,
                              dout => mw
                              );
VK_rom_test : VK_rom
          PORT MAP(
                              clk => clk,
                              addr => VK_addr,
                              dout => vk
                              );
--X_rom_test : X_rom
--        PORT MAP(
--                            clk => clk,
--                            addr => X_addr,
--                            dout => x
--                            );
status_rom_test : status_rom
          PORT MAP(
                              clk => clk,
                              addr => status_addr,
                              dout => status
                              );



end Behavioral;
```

B.2 – gaus_core_top_v2.vhd

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:     Jeffrey W. Schuster
--
-- Create Date:    11:49:39 10/18/05
-- Design Name:
-- Module Name:    gaus_core_top - Behavioral
-- Project Name:
-- Target Device:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
```

```
--library UNISIM;
--use UNISIM.VComponents.all;

entity gaus_top is
   Port (
                         input_stall_in  : in std_logic;
                         clk             : in std_logic;
                         ce              : in std_logic;
                         sclr            : in std_logic;
                 x              : in std_logic_vector(31 downto 0);
                         x_valid         : in std_logic;
        mw             : in std_logic_vector(31 downto 0);
                         mw_valid        : in std_logic;
        vk             : in std_logic_vector(31 downto 0);
                         vk_valid        : in std_logic;
                         status          : in std_logic_vector(6 downto 0);
                         status_valid    : in std_logic;
        new_frame      : in std_logic;
                        -- senone_rd_addr  : in std_logic_vector(12 downto 0);
                         --senone_addr_rdy : in std_logic;
                         --senone_ram_sel  : in std_logic;
                         SENONE          : out std_logic_vector(31 downto 0);
                         SENONE_ready    : out std_logic;
        SENONE_WR_ADDR  : out std_logic_vector(12 downto 0);
                         SENONE_WR       : out std_logic;
                         max_done        : out std_logic;
                         normalize_done  : out std_logic;
                         composite_done  : out std_logic;
                         input_stall     : out std_logic

                         );
```

```
end gaus_top;

architecture Behavioral of gaus_top is

component gaus_top_control_logic
            Port (
                                    clk             : in std_logic;
          sclr            : in std_logic;
                                    new_frame       : in std_logic;
                                    max_done        : in std_logic;
                                    norm_done       : in std_logic;
                                    input_stall_in  : in std_logic;
--                                  input_stall_1   : in std_logic;
--                                  input_stall_2   : in std_logic;
          ---------ROM PINS--------------
                                    x               : in std_logic_vector(31 downto 0);
                                    x_valid         : in std_logic;
          mw              : in std_logic_vector(31 downto 0);
                                    mw_valid        : in std_logic;
                                    vk              : in std_logic_vector(31 downto 0);
                                    vk_valid        : in std_logic;
                                    status          : in std_logic_vector(6 downto 0);
                                    status_valid    : in std_logic;
                                    ------RAW SENONE RAM PINS------
                                    raw_ram_data    : in std_logic_vector(31 downto 0);
                                    raw_data_rdy    : in std_logic;
          -----NORMAL SENONE RAM PINS----
                                    norm_ram_addr   : in std_logic_vector(12 downto 0);
                                    norm_addr_rdy   : in std_logic;
                                    norm_ram_data   : in std_logic_vector(31 downto 0);
                                    norm_ram_wr     : in std_logic;
```

```
                    norm_mux_sel    : in std_logic;
                    ---COMPOSITE SENONE RAM PINS---
                    comp_ram_addr   : in std_logic_vector(12 downto 0);
                    comp_addr_rdy   : in std_logic;
                    comp_ram_data   : in std_logic_vector(31 downto 0);
                    comp_ram_wr     : in std_logic;
                    comp_mux_sel    : in std_logic;
                    --------RAM READ PINS----------
--                  senone_rd_addr  : in std_logic_vector(12 downto 0);
--                  senone_addr_rdy : in std_logic;
--                  senone_ram_sel  : in std_logic;
                    ----------OUTPUT PINS----------
                    x_out           : out std_logic_vector(31 downto 0);
                    mw_out          : out std_logic_vector(31 downto 0);
                    vk_out          : out std_logic_vector(31 downto 0);
                    first_dist_calc : out std_logic;
                    mid_dist_calc   : out std_logic;
                    last_dist_calc  : out std_logic;
                    valid_dist_calc : out std_logic;
                    first_comp_calc : out std_logic;
                    mid_comp_calc   : out std_logic;
                    last_comp_calc  : out std_logic;
                    valid_comp_calc : out std_logic;
                    last_sen        : out std_logic;
                    ram_data_rdy    : out std_logic;
                    ram_addr                : out std_logic_vector(12 downto 0);
                    ram_data_in     : out std_logic_vector(31 downto 0);
                    ram_wr          : out std_logic
                    );
end component;
```

```
component gaus_core_pipe
        port(
                clk        : IN std_logic;
                ce         : IN std_logic;
                sclr       : IN std_logic;
                X          : IN std_logic_vector( 31 downto 0);
                MW          : IN std_logic_vector(31 downto 0);
                VK          : IN std_logic_vector(31 downto 0);
                new_frame  : IN std_logic;
                first_calc : IN std_logic;
                mid_calc   : IN std_logic;
                last_calc        : IN std_logic;
                valid_calc : IN std_logic;
                COMP_SCORE : OUT std_logic_vector(31 downto 0);
                COMP_RDY   : OUT std_logic;
                input_stall: OUT std_logic
--              input_stall_1: out std_logic;
--              input_stall_2: out std_logic
                );
end component;

component log_add_calc
   Port (
                        clk        : IN std_logic;
       ce         : IN std_logic;
       sclr       : IN std_logic;
                        first_comp : IN std_logic;
                        mid_comp      : IN std_logic;
                        last_comp      : IN std_logic;
                        valid_calc : IN std_logic;
       COMP_IN    : IN std_logic_vector(31 downto 0);
       COMP_RDY   : IN std_logic;
       SEN_OUT    : OUT std_logic_vector(31 downto 0);
       SEN_RDY    : OUT std_logic
                        );
end component;

component find_max
        PORT(
                        clk          : IN std_logic;
                        ce           : IN std_logic;
                        sclr         : IN std_logic;
                        new_frame    : IN std_logic;
                        last_senone  : IN std_logic;
                        senone_in    : IN std_logic_vector(31 downto 0);
                        new_senone   : IN std_logic;
                        senone_out   : OUT std_logic_vector(31 downto 0);
                        senone_rdy   : OUT std_logic;
                        best_score   : OUT std_logic_vector(31 downto 0);
                        max_done     : OUT std_logic
                        );
end component;

component raw_senone_ram
        PORT(
                        clk    : IN std_logic;
                        we     : IN std_logic;
                        din    : IN std_logic_vector(31 downto 0);
                        addr   : IN std_logic_vector(12 downto 0);
                        dout   : OUT std_logic_vector(31 downto 0)
                        );
```

```
end component;

component normalizer
          PORT (
                              clk           : in std_logic;
        ce            : in std_logic;
        sclr          : in std_logic;
        start_normalize : in std_logic;
                              last_raw_senone : in std_logic;
                              raw_senone     : in std_logic_vector(31 downto 0);
                              raw_sen_rdy    : in std_logic;
                              best_score     : in std_logic_vector(31 downto 0);
        address        : out std_logic_vector(12 downto 0);
                              address_rdy    : out std_logic;
        norm_senone    : out std_logic_vector(31 downto 0);
        senone_rdy     : out std_logic;
                              normalize_done  : out std_logic;
                              ram_addr_sel   : out std_logic;
                              sen_cnt        : out std_logic_vector(12 downto 0)
                              );
end component;

component composite_senone_calc
          Port (
                              clk             : in std_logic;
        sclr             : in std_logic;
        ce               : in std_logic;
        calc_go          : in std_logic;
        ram_data_in       : in std_logic_vector(31 downto 0);
        ram_data_rdy       : in std_logic;
                              senone_addr_offset : in std_logic_vector (12 downto 0);
        ram_addr          : out std_logic_vector(12 downto 0);
        addr_rdy          : out std_logic;
        ram_wr_en         : out std_logic;
        comp_sen_out      : out std_logic_vector(31 downto 0);
        comp_calc_done    : out std_logic;
                        comp_mux_sel      : out std_logic
                        );
end component;

SIGNAL COMP_OUT       : std_logic_vector(31 downto 0);
SIGNAL COMP_RDY       : std_logic;
SIGNAL SEN_OUT        : std_logic_vector(31 downto 0);
SIGNAL SEN_RDY        : std_logic;
SIGNAL COMP_OUT_REG   : std_logic_vector(31 downto 0);
SIGNAL COMP_RDY_REG   : std_logic;
SIGNAL SEN_OUT_REG    : std_logic_vector(31 downto 0);
SIGNAL SEN_RDY_REG    : std_logic;
SIGNAL SENONE_OUT     : std_logic_vector(31 downto 0);
SIGNAL SENONE_RDY     : std_logic;
SIGNAL BEST_SEN       : std_logic_vector(31 downto 0);
SIGNAL BEST_SEN_REG   : std_logic_vector(31 downto 0);
SIGNAL X_out          : std_logic_vector(31 downto 0);
SIGNAL MW_out         : std_logic_vector(31 downto 0);
SIGNAL VK_out         : std_logic_vector(31 downto 0);
SIGNAL first_dist_calc: std_logic;
SIGNAL mid_dist_calc  : std_logic;
SIGNAL last_dist_calc : std_logic;
SIGNAL valid_dist_calc: std_logic;
SIGNAL first_comp_calc: std_logic;
SIGNAL mid_comp_calc  : std_logic;
SIGNAL last_comp_calc : std_logic;
SIGNAL valid_comp_calc: std_logic;
SIGNAL max_done_tmp   : std_logic;
SIGNAL last_senone    : std_logic;
SIGNAL ram_addr       : std_logic_vector(12 downto 0);
SIGNAL ram_wr         : std_logic;
SIGNAL ram_data_in    : std_logic_vector(31 downto 0);
SIGNAL sen_ram_data   : std_logic_vector(31 downto 0);
SIGNAL sen_data_out   : std_logic_vector(31 downto 0);
SIGNAL ram_data_rdy   : std_logic;
SIGNAL start_normalize: std_logic;
SIGNAL norm_ram_addr  : std_logic_vector(12 downto 0);
SIGNAL norm_addr_rdy  : std_logic;
SIGNAL norm_ram_data  : std_logic_vector(31 downto 0);
SIGNAL norm_ram_wr    : std_logic;
SIGNAL norm_done      : std_logic;
SIGNAL norm_mux_sel   : std_logic;
SIGNAL norm_sen_cnt   : std_logic_vector(12 downto 0);
SIGNAL start_composite: std_logic;
SIGNAL comp_ram_addr  : std_logic_vector(12 downto 0);
SIGNAL comp_addr_rdy  : std_logic;
SIGNAL comp_ram_data  : std_logic_vector(31 downto 0);
SIGNAL comp_ram_wr    : std_logic;
SIGNAL comp_calc_done : std_logic;
SIGNAL comp_mux_sel   : std_logic;
SIGNAL max_done_tmp1  : std_logic;
SIGNAL input_stall_out : std_logic;
SIGNAL input_stall_sig : std_logic;
--SIGNAL input_stall_tmp_1: std_logic;
--SIGNAL input_stall_tmp_2: std_logic;
begin
```

```
process (clk, input_stall_in)
begin
        if(clk = '1' and clk'event)then
                input_stall_sig<=input_stall_in;
                max_done<=max_done_tmp;
                normalize_done<=norm_done;
                composite_done<=comp_calc_done;
                max_done_tmp1<=max_done_tmp;
                start_normalize<=max_done_tmp1;
                start_composite<=norm_done;
                input_stall<=input_stall_out;
                sen_data_out<=ram_data_in;
                senone<=sen_data_out;
                senone_ready<=ram_data_rdy;
                COMP_OUT_REG<=COMP_OUT;
                COMP_RDY_REG<=COMP_RDY;
                SEN_OUT_REG<=SEN_OUT;
                SEN_RDY_REG<=SEN_RDY;
                BEST_SEN_REG<=BEST_SEN;
                SENONE_WR_ADDR<=ram_addr;
                SENONE_WR<=ram_wr;

        end if;
end process;



gaus_core: gaus_core_pipe
        PORT MAP(
                        clk     => clk,
                        ce      => ce,
                        sclr    => sclr,
                              new_frame  => new_frame,
                              first_calc => first_dist_calc,
                              mid_calc   => mid_dist_calc,
                              last_calc  => last_dist_calc,
                              valid_calc => valid_dist_calc,
                              X          => X_out,
                              MW         => MW_out,
                              VK         => VK_out,
                          COMP_SCORE => COMP_OUT,
                              COMP_RDY   => COMP_RDY,
                              input_stall=> input_stall_out
--                            input_stall_1=> input_stall_tmp_1,
--                            input_stall_2=> input_stall_tmp_2
                              );
logadd : log_add_calc
     PORT MAP(
                              clk        => clk,
                              ce         => ce,
                              sclr       => sclr,
                              first_comp => first_comp_calc,
                              mid_comp   => mid_comp_calc,
                              last_comp  => last_comp_calc,
                              valid_calc => valid_comp_calc,
                              COMP_IN    => COMP_OUT_REG,
                              COMP_RDY   => COMP_RDY_REG,
                              SEN_OUT    => SEN_OUT,
                              SEN_RDY    => SEN_RDY
                              );
senone_max : find_max
     PORT MAP(
                              clk          => clk,
                              ce              => ce,
                              sclr            => sclr,
                              new_frame       => new_frame,
                              last_senone     => last_senone,
                              new_senone      => SEN_RDY_REG,
                              senone_in       => SEN_OUT_REG,
                              senone_out      => SENONE_OUT,
                              senone_rdy      => SENONE_RDY,
                              best_score      => BEST_SEN,
                              max_done        => max_done_tmp
                              );
--raw_sen_ram : raw_senone_ram
--      PORT MAP(
--                      clk     => clk,
--                      we      => ram_wr,
--                      addr    => ram_addr,
--                      din     => ram_data_in,
--                      dout    => sen_ram_data
--                      );
normalize : normalizer
        PORT MAP(
                              clk             => clk,
                              ce              => ce,
                              sclr            => sclr,
                              start_normalize => start_normalize,
                              last_raw_senone => last_senone,
                              raw_senone      => sen_ram_data,
                              raw_sen_rdy     => ram_data_rdy,
                              best_score          => BEST_SEN_REG,
                              address             => norm_ram_addr,
                              address_rdy     => norm_addr_rdy,
                                        norm_senone      => norm_ram_data,
                                        senone_rdy        => norm_ram_wr,
                                        normalize_done          => norm_done,
                                        ram_addr_sel   => norm_mux_sel,
                                        sen_cnt        => norm_sen_cnt
                                        );
composite_calc : composite_senone_calc
        PORT MAP(
                        clk              => clk,
        sclr             => sclr,
        ce               => ce,
        calc_go          => start_composite,
        ram_data_in       => sen_ram_data,
        ram_data_rdy       =>   ram_data_rdy,
                                senone_addr_offset => norm_sen_cnt,
        ram_addr          =>      comp_ram_addr,
        addr_rdy          =>      comp_addr_rdy,
                                ram_wr_en         => comp_ram_wr,
                                comp_sen_out      => comp_ram_data,
        comp_calc_done     => comp_calc_done,
                                comp_mux_sel      => comp_mux_sel
                                );
control_logic : gaus_top_control_logic
        PORT MAP(
                                clk           => clk,
        sclr          => sclr,
                                new_frame      => new_frame,
                                        max_done       => max_done_tmp,
                                        norm_done      => norm_done,
                                        input_stall_in  => input_stall_sig,
--                                      input_stall_1     => input_stall_tmp_1,
--                    input_stall_2   => input_stall_tmp_2,
     x             => x,
                 x_valid        => x_valid,
     mw            => mw,
                 mw_valid       => mw_valid,
                 vk             => vk,
                 vk_valid       => vk_valid,
                 status         => status,
                    status_valid   => status_valid,
                    raw_ram_data   => SENONE_OUT,
                 raw_data_rdy   => SENONE_RDY,
--                  senone_rd_addr  => senone_rd_addr,
--                  senone_addr_rdy => senone_addr_rdy,
--                  senone_ram_sel  => senone_ram_sel,
     norm_ram_addr   => norm_ram_addr,
                 norm_addr_rdy   => norm_addr_rdy,
                 norm_ram_data   => norm_ram_data,
                 norm_ram_wr     => norm_ram_wr,
                 norm_mux_sel    => norm_mux_sel,
                 comp_ram_addr   => comp_ram_addr,
                 comp_addr_rdy   => comp_addr_rdy,
                 comp_ram_data   => comp_ram_data,
                 comp_ram_wr     => comp_ram_wr,
                 comp_mux_sel    => comp_mux_sel,
                    x_out          => X_out,
                    mw_out         => MW_out,
                    vk_out         => VK_out,
                 first_dist_calc => first_dist_calc,
                 mid_dist_calc   => mid_dist_calc,
                 last_dist_calc  => last_dist_calc,
                 valid_dist_calc => valid_dist_calc,
                 first_comp_calc => first_comp_calc,
                 mid_comp_calc   => mid_comp_calc,
                 last_comp_calc  => last_comp_calc,
                 valid_comp_calc => valid_comp_calc,
                    last_sen        => last_senone,
                 ram_data_rdy    => ram_data_rdy,
                 ram_addr                    => ram_addr,
                 ram_data_in     => ram_data_in,
                 ram_wr          => ram_wr
                 );

end Behavioral;
```

B.3 – composite_senone_calc.vhd

```
--------------------------------------------------------------------------------
-- Company:
-- Engineer:       Jeffrey W. Schuster
--
-- Create Date:    19:47:25 10/12/05
-- Design Name:
-- Module Name:    composite_senone_calc - Behavioral
-- Project Name:
-- Target Device:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

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

entity composite_senone_calc is
   Port ( clk : in std_logic;
          sclr : in std_logic;
          ce : in std_logic;
          calc_go : in std_logic;
          ram_data_in : in std_logic_vector(31 downto 0);
          ram_data_rdy : in std_logic;
                              senone_addr_offset : in std_logic_vector (12 downto 0);
          ram_addr : out std_logic_vector(12 downto 0);
          addr_rdy : out std_logic;
          ram_wr_en : out std_logic;
          comp_sen_out : out std_logic_vector(31 downto 0);
          comp_calc_done : out std_logic;
                              comp_mux_sel : out std_logic
                              );
end composite_senone_calc;

architecture Behavioral of composite_senone_calc is
```

```
component count_rom
        PORT(
                        clk     : IN std_logic;
                        addr  : IN std_logic_vector(5 downto 0);
                        dout  : OUT std_logic_vector(4 downto 0)
                        );
end component;

component state_rom
        PORT(
                        clk     : IN std_logic;
                        addr  : IN std_logic_vector(8 downto 0);
                        dout  : OUT std_logic_vector(11 downto 0)
                        );
end component;

SIGNAL count_addr     : std_logic_vector(5 downto 0);
SIGNAL state_addr     : std_logic_vector(8 downto 0);
SIGNAL state_count    : std_logic_vector(4 downto 0);
SIGNAL senone_addr    : std_logic_vector(11 downto 0);

SIGNAL calc_done_p1     : std_logic;
SIGNAL calc_done_p2     : std_logic;
SIGNAL calc_done_p3     : std_logic;
SIGNAL calc_done_p4     : std_logic;
SIGNAL calc_done_p5     : std_logic;
SIGNAL calc_done_p6     : std_logic;
SIGNAL calc_done_p7     : std_logic;
SIGNAL calc_done_p8     : std_logic;
SIGNAL calc_done_p9     : std_logic;
SIGNAL count          : std_logic_vector(5 downto 0);
SIGNAL new_offset     : std_logic;
SIGNAL new_offset_reg : std_logic;
SIGNAL composite_done : std_logic;
SIGNAL new_offset_rdy : std_logic;
SIGNAL offset         : std_logic_vector(4 downto 0);
SIGNAL state_id_tmp   : std_logic_vector(8 downto 0);
SIGNAL list_done      : std_logic;
SIGNAL new_id         : std_logic;
SIGNAL fetch_done     : std_logic;
SIGNAL fetch_done_reg : std_logic;
SIGNAL ram_addr_tmp   : std_logic_vector(12 downto 0);
SIGNAL comp_done_reg  : std_logic;
SIGNAL comp_scr_reg   : std_logic_vector(31 downto 0);
SIGNAL new_senone     : std_logic_vector(31 downto 0);
SIGNAL last_comp      : std_logic;
SIGNAL last_comp_reg  : std_logic;
SIGNAL write_addr     : std_logic_vector(12 downto 0);
SIGNAL write_addr_tmp : std_logic_vector(12 downto 0);
SIGNAL freeze         : std_logic;
SIGNAL comp_mux       : std_logic;
signal pipe_freeze    : std_logic;
signal calc_go_reg    : std_logic;
signal ram_data_rdy_reg : std_logic;
signal ram_data_rdy_reg1 : std_logic;
signal first_addr_rdy : std_logic;
signal new_senone_reg : std_logic_vector(31 downto 0);
signal fetch_done_reg1 : std_logic;
signal fetch_done_reg2 : std_logic;
signal new_offset_rdy_reg : std_logic;
```

```
attribute syn_ramstyle : string;
attribute syn_ramstyle of senone_addr : signal is "block_ram";
attribute syn_ramstyle of state_count : signal is "block_ram";

begin



process(clk,sclr,ce,calc_go)
begin
if(clk = '1' and clk'event)then
        if(sclr = '1')then
                count<=(OTHERS=>'0');
                new_offset<='0';
                new_offset_reg<='0';
                new_offset_rdy<='0';
                list_done<='0';
                new_id<='0';
                state_id_tmp<=(OTHERS=>'0');
                comp_scr_reg<=(OTHERS=>'0');
                offset<=(OTHERS=>'0');
                composite_done<='0';
                comp_done_reg<='0';
                fetch_done_reg<='0';
                fetch_done<='0';
                state_addr<=(OTHERS=>'0');
                count_addr<=(OTHERS=>'0');
                new_senone<=(OTHERS=>'0');
                ram_addr_tmp<=(OTHERS=>'0');
                ram_addr<=(OTHERS=>'0');
                ram_wr_en<='0';
                addr_rdy<='0';
                write_addr<=(OTHERS=>'0');
                write_addr_tmp<=(OTHERS=>'0');
                last_comp<='0';
                last_comp_reg<='0';
                calc_done_p1<='0';
                calc_done_p2<='0';
                calc_done_p3<='0';
                calc_done_p4<='0';
                calc_done_p5<='0';
                calc_done_p6<='0';
                calc_done_p7<='0';
                calc_done_p8<='0';
                calc_done_p9<='0';
                freeze<='0';
                comp_mux<='0';
                pipe_freeze<='1';
                calc_go_reg<='0';
                ram_data_rdy_reg<='0';
                ram_data_rdy_reg1<='0';
                first_addr_rdy<='0';
                new_senone_reg<=(OTHERS=>'0');
                fetch_done_reg1<='0';
                fetch_done_reg2<='0';
                new_offset_rdy_reg<='0';
        elsif(ce = '1')then
                if(calc_go = '1')then
                        pipe_freeze<='0';
                elsif(calc_done_p9 = '1')then
                        pipe_freeze<='1';
                end if;
```

```
calc_go_reg<=calc_go;
if(pipe_freeze = '0')then
        if(calc_go_reg = '1')then
                count<="000000";
                new_offset_rdy<='1';
                new_offset<='0';
                new_offset_reg<='0';
                write_addr_tmp<=senone_addr_offset;
                comp_mux<='1';
                first_addr_rdy<='1';
        elsif(composite_done = '1')then
                count<=count+1;
                new_offset<='1';
                new_offset_rdy<='0';
                new_offset_reg<='0';
                write_addr_tmp<=write_addr_tmp+1;
                first_addr_rdy<='0';
        elsif(calc_done_p9 = '1')then
                count<=(OTHERS=>'0');
                new_offset_rdy<='0';
                new_offset<='0';
                new_offset_reg<='0';
                comp_mux<='0';
                first_addr_rdy<='0';
        else
                new_offset_reg<=new_offset;
                new_offset_rdy<=new_offset_reg;
                new_offset<='0';
                first_addr_rdy<='0';
        end if;
        write_addr<=write_addr_tmp;
```

```
count_addr<=count;
if (new_offset_rdy = '1')then
        offset<=state_count;---1; CHANGED : JWS : 11-03-05
end if;
if(calc_done_p9 = '1')then
        state_id_tmp<=(OTHERS=>'0');
elsif(fetch_done = '1')then
        if(freeze = '0')then
                offset<=offset-1;
                if(offset > "00001")then
                        state_id_tmp<=state_id_tmp+1;
                        list_done<='0';
                        new_id<='1';
                elsif(offset = "00001")then
                        state_id_tmp<=state_id_tmp+1;
                        list_done<='1';
                        new_id<='1';
                else
                        list_done<='0';
                        new_id<='0';
                end if;
        else
                list_done<='0';
                new_id<='0';
        end if;
else
        list_done<='0';
        new_id<='0';
end if;
if(list_done = '1')then
        freeze<='1';
```

```
elsif(new_offset_rdy = '1')then
        freeze<='0';
end if;
composite_done<=list_done;
last_comp<=list_done;
last_comp_reg<=last_comp;
ram_data_rdy_reg<=ram_data_rdy;
ram_data_rdy_reg1<=ram_data_rdy_reg;
fetch_done<=ram_data_rdy_reg1;
new_senone<=ram_data_in;

--new_senone_reg<=new_senone;
fetch_done_reg1<=fetch_done;
fetch_done_reg2<=fetch_done_reg1;
new_offset_rdy_reg<=new_offset_rdy;

state_addr<=state_id_tmp;
ram_addr_tmp<="00" & senone_addr(10 downto 0);
addr_rdy<=new_id ;--or first_addr_rdy; CHANGED : JWS : 11-03-05
if(new_offset_rdy_reg = '1')then
        comp_scr_reg<=(OTHERS=>'0');
elsif(fetch_done_reg2= '1')then
        if(new_senone > comp_scr_reg)then
                comp_scr_reg<=new_senone;
        end if;
end if;
if(last_comp_reg = '1')then
        ram_wr_en<='1';
        comp_sen_out<=comp_scr_reg;
        ram_addr<=write_addr;
else
```

```
                ram_addr<=ram_addr_tmp;
                ram_wr_en<='0';
                comp_sen_out<=(OTHERS=>'0');
            end if;
            calc_done_p1<=senone_addr(11);
            calc_done_p2<=calc_done_p1;
            calc_done_p3<=calc_done_p2;
            calc_done_p4<=calc_done_p3;
            calc_done_p5<=calc_done_p4;
            calc_done_p6<=calc_done_p5;
            calc_done_p7<=calc_done_p6;
            calc_done_p8<=calc_done_p7;
            calc_done_p9<=calc_done_p8;
            comp_calc_done<=calc_done_p9;

            comp_mux_sel<=comp_mux;
        else
            ram_wr_en<='0';
            comp_calc_done<='0';
            comp_mux_sel<='0';
            composite_done<='0';
            freeze<='0';
            fetch_done<='0';
            first_addr_rdy<='0';
            comp_sen_out<=(OTHERS=>'0');
        end if;
    end if;
end if;
end process;
```

```
composite_count_rom : count_rom
        PORT MAP(
                        clk     => clk,
                        addr  => count_addr,
                        dout    => state_count
                        );
composite_state_rom : state_rom
        PORT MAP(
                        clk     => clk,
                        addr    => state_addr,
                        dout  => senone_addr
                        );



end Behavioral;
```

B.4 – find_max.vhd

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:                    Jeffrey W. Schuster
--
-- Create Date:    14:51:30 10/09/05
-- Design Name:
-- Module Name:    find_max - Behavioral
-- Project Name:
-- Target Device:
-- Tool versions:
```

```
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

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

entity find_max is
   Port (
                              senone_in : in std_logic_vector(31 downto 0);
        new_senone : in std_logic;
                              last_senone : in std_logic;
        clk : in std_logic;
        ce : in std_logic;
        sclr : in std_logic;
                              new_frame : in std_logic;
        senone_out : out std_logic_vector(31 downto 0);
        senone_rdy : out std_logic;
        best_score : out std_logic_vector(31 downto 0);
```

```
                                max_done   : out std_logic
                                );

end find_max;

architecture Behavioral of find_max is

SIGNAL senone_tmp   : std_logic_vector(31 downto 0);
SIGNAL sen_rdy_tmp  : std_logic;
SIGNAL current_best : std_logic_vector(31 downto 0);
SIGNAL tmp_rdy      : std_logic;
SIGNAL last_sen_tmp : std_logic;
SIGNAL last_sen_tmp1: std_logic;


begin




process(clk)
begin
if(clk = '1' and clk'event)then
        if(sclr = '1')then
                senone_tmp<=(OTHERS=>'0');
                sen_rdy_tmp<='0';
                current_best<=(OTHERS=>'0');
                best_score<=(OTHERS=>'0');
                senone_out<=(OTHERS=>'0');
                senone_rdy<='0';
                tmp_rdy<='0';
                last_sen_tmp<='0';
        elsif(ce = '1')then
                last_sen_tmp<=last_senone;
                if(new_frame = '1')then
                                current_best<=(OTHERS=>'0');--this will need changed to -INF
for signed numbers
                end if;
                if (new_senone = '1')then
                        senone_tmp<=senone_in;
                        tmp_rdy<='1';
                else
                        tmp_rdy<='0';
                end if;
                if(tmp_rdy = '1')then
                        if(senone_tmp > current_best)then
                                current_best<=senone_tmp;
                        end if;
                end if;
                sen_rdy_tmp<=tmp_rdy;
        else
                current_best<=(OTHERS=>'0');
                sen_rdy_tmp<='0';
                last_sen_tmp<='0';
        end if;
        last_sen_tmp1<=last_sen_tmp;
        max_done<=last_sen_tmp1 AND sen_rdy_tmp;
        best_score<=current_best;
        senone_out<=senone_tmp;
        senone_rdy<=sen_rdy_tmp;
end if;
end process;
end Behavioral;
```

B.5 – gaus_dist_full_pipe_V2_FINAL.vhd

```
-- 10-05-05 -- JWS : Created pipe-line to perform gaussian probability evaluation and senone
calculation

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_signed.all;


entity gaus_core_pipe is
        port(
          clk       : IN  std_logic;
                ce        : IN  std_logic;
                sclr      : IN  std_logic;
                new_frame  : IN  std_logic;
                first_calc : IN  std_logic;
                last_calc  : IN  std_logic;
                mid_calc   : IN  std_logic;
                valid_calc : IN  std_logic;
                X          : IN  std_logic_vector(31 downto 0);
                MW         : IN  std_logic_vector(31 downto 0);
                VK         : IN  std_logic_vector(31 downto 0);
                COMP_SCORE : OUT std_logic_vector(31 downto 0);
                COMP_RDY   : OUT std_logic;
                input_stall: OUT std_logic
--              input_stall_2: OUT std_logic
```

```
                    );
end gaus_core_pipe;

architecture struct of gaus_core_pipe is

SIGNAL DIFF            : std_logic_vector(31 downto 0);
SIGNAL DIFF_reg        : std_logic_vector(31 downto 0);
SIGNAL DIFF_valid      : std_logic;
SIGNAL DIFF_valid_reg  : std_logic;
SIGNAL DIFF_valid_reg1 : std_logic;
SIGNAL SQUARE          : std_logic_vector(63 downto 0);
SIGNAL SQUARE_valid    : std_logic;
SIGNAL SQUARE_valid_reg: std_logic;
SIGNAL SCALE                   : std_logic_vector(63 downto 0);
SIGNAL SCALE_valid     : std_logic;
SIGNAL SCALE_valid_reg : std_logic;
SIGNAL SUM_OUT         : std_logic_vector(31 downto 0);
SIGNAL COMP_OUT        : std_logic_vector(31 downto 0);
SIGNAL COMP_valid      : std_logic;
SIGNAL sub_in1         : std_logic_vector(31 downto 0);
SIGNAL sub_in2         : std_logic_vector(31 downto 0);
SIGNAL scal_in1        : std_logic_vector(31 downto 0);
SIGNAL scal_in2        : std_logic_vector(31 downto 0);
SIGNAL mul_in1         : std_logic_vector(31 downto 0);
SIGNAL mul_in2         : std_logic_vector(31 downto 0);
SIGNAL mul_in1_reg     : std_logic_vector(31 downto 0);
SIGNAL add_in1         : std_logic_vector(31 downto 0);
SIGNAL add_in2         : std_logic_vector(31 downto 0);
SIGNAL sum_in1         : std_logic_vector(31 downto 0);
SIGNAL sum_in2         : std_logic_vector(31 downto 0);
SIGNAL REG_S1          : std_logic_vector(31 downto 0);
SIGNAL REG_S2         : std_logic_vector(31 downto 0);
SIGNAL REG_S3         : std_logic_vector(31 downto 0);
SIGNAL REG_S4         : std_logic_vector(31 downto 0);
SIGNAL REG_S5         : std_logic_vector(31 downto 0);
SIGNAL MW_p1          : std_logic_vector(31 downto 0);
SIGNAL MW_p2          : std_logic_vector(31 downto 0);
SIGNAL MW_p3          : std_logic_vector(31 downto 0);
SIGNAL MW_p4          : std_logic_vector(31 downto 0);
SIGNAL MW_p5          : std_logic_vector(31 downto 0);
SIGNAL MW_p6          : std_logic_vector(31 downto 0);
SIGNAL MW_p7          : std_logic_vector(31 downto 0);
SIGNAL MW_p8          : std_logic_vector(31 downto 0);
SIGNAL VK_p1         : std_logic_vector(31 downto 0);
SIGNAL VK_p2         : std_logic_vector(31 downto 0);
SIGNAL VK_p3         : std_logic_vector(31 downto 0);
SIGNAL VK_p4         : std_logic_vector(31 downto 0);
SIGNAL VK_p5         : std_logic_vector(31 downto 0);
SIGNAL VK_p6         : std_logic_vector(31 downto 0);
SIGNAL VK_p7         : std_logic_vector(31 downto 0);
SIGNAL VK_p8         : std_logic_vector(31 downto 0);
SIGNAL first_calc_p1   : std_logic;
SIGNAL last_calc_p1    : std_logic;
SIGNAL first_calc_p2   : std_logic;
SIGNAL last_calc_p2    : std_logic;
SIGNAL first_calc_p3   : std_logic;
SIGNAL last_calc_p3    : std_logic;
SIGNAL first_calc_p4   : std_logic;
SIGNAL last_calc_p4    : std_logic;
SIGNAL first_calc_p5   : std_logic;
SIGNAL last_calc_p5    : std_logic;
SIGNAL first_calc_p6   : std_logic;
SIGNAL last_calc_p6    : std_logic;
SIGNAL first_calc_p7   : std_logic;
SIGNAL last_calc_p7    : std_logic;
SIGNAL first_calc_p8   : std_logic;
SIGNAL last_calc_p8    : std_logic;
SIGNAL last_calc_p9    : std_logic;
SIGNAL last_calc_p10   : std_logic;
SIGNAL last_calc_p11   : std_logic;
SIGNAL last_calc_p12   : std_logic;
SIGNAL last_calc_p13   : std_logic;
SIGNAL last_calc_p14   : std_logic;
SIGNAL last_calc_p15   : std_logic;
SIGNAL valid_calc_reg  : std_logic;
SIGNAL out_data_reg1   : std_logic_vector(31 downto 0);
SIGNAL out_rdy_reg1    : std_logic;
SIGNAL out_data_reg2   : std_logic_vector(31 downto 0);
SIGNAL out_rdy_reg2    : std_logic;
SIGNAL out_data_reg3   : std_logic_vector(31 downto 0);
SIGNAL out_rdy_reg3    : std_logic;
SIGNAL new_frame_p1    : std_logic;
SIGNAL new_frame_p2    : std_logic;
SIGNAL new_frame_p3    : std_logic;
SIGNAL new_frame_p4    : std_logic;
SIGNAL new_frame_p5    : std_logic;
SIGNAL new_frame_p6    : std_logic;
SIGNAL new_frame_p7    : std_logic;
SIGNAL new_frame_p8    : std_logic;
SIGNAL new_frame_p9    : std_logic;
SIGNAL new_frame_p10   : std_logic;
SIGNAL new_frame_p11   : std_logic;
SIGNAL new_frame_p12   : std_logic;
SIGNAL new_frame_p13   : std_logic;





attribute syn_dspstyle : string;
attribute syn_dspstyle of DIFF : signal is "dsp48";
attribute syn_dspstyle of SQUARE : signal is "dsp48";
attribute syn_dspstyle of SCALE : signal is "dsp48";
attribute syn_dspstyle of COMP_OUT : signal is "dsp48";
attribute syn_dspstyle of SUM_OUT : signal is "dsp48";
--constant F : std_logic_vector(31 downto 0) := "00000000000011010000010111010101";
  constant F : std_logic_vector(31 downto 0) := "00000000000000000000000000001000";

begin
process (clk,last_calc_p7,new_frame_p13)
begin
        if (clk'event and clk = '1') then
                if (sclr = '1') then
                        sub_in1<=(OTHERS=>'0');
                        sub_in2<=(OTHERS=>'0');
                        scal_in1<=(OTHERS=>'0');
                        scal_in2<=(OTHERS=>'0');
                        add_in1<=(OTHERS=>'0');
                        add_in2<=(OTHERS=>'0');
                        sum_in1<=(OTHERS=>'0');
                        sum_in2<=(OTHERS=>'0');
                        mul_in1<=(OTHERS=>'0');
                        mul_in2<=(OTHERS=>'0');
                        mul_in1_reg<=(OTHERS=>'0');
```

```
REG_S1<=(OTHERS=>'0');
REG_S2<=(OTHERS=>'0');
REG_S3<=(OTHERS=>'0');
REG_S4<=(OTHERS=>'0');
REG_S5<=(OTHERS=>'0');
DIFF<=(OTHERS=>'0');
DIFF_reg<=(OTHERS=>'0');
SQUARE<=(OTHERS=>'0');
SCALE<=(OTHERS=>'0');
SUM_OUT<=(OTHERS=>'0');
COMP_OUT<=(OTHERS=>'0');
MW_p1<=(OTHERS=>'0');
MW_p2<=(OTHERS=>'0');
MW_p3<=(OTHERS=>'0');
MW_p4<=(OTHERS=>'0');
MW_p5<=(OTHERS=>'0');
MW_p6<=(OTHERS=>'0');
MW_p7<=(OTHERS=>'0');
MW_p8<=(OTHERS=>'0');
VK_p1<=(OTHERS=>'0');
VK_p2<=(OTHERS=>'0');
VK_p3<=(OTHERS=>'0');
VK_p4<=(OTHERS=>'0');
VK_p5<=(OTHERS=>'0');
VK_p6<=(OTHERS=>'0');
VK_p7<=(OTHERS=>'0');
VK_p8<=(OTHERS=>'0');
DIFF_valid<='0';
DIFF_valid_reg<='0';
DIFF_valid_reg1<='0';
SQUARE_valid<='0';
```

```
SQUARE_valid_reg<='0';
SCALE_valid<='0';
SCALE_valid_reg<='0';
COMP_valid<='0';
COMP_RDY<='0';
first_calc_p1<='0';
last_calc_p1<='0';
first_calc_p2<='0';
last_calc_p2<='0';
first_calc_p3<='0';
last_calc_p3<='0';
first_calc_p4<='0';
last_calc_p4<='0';
first_calc_p5<='0';
last_calc_p5<='0';
first_calc_p6<='0';
last_calc_p6<='0';
first_calc_p7<='0';
last_calc_p7<='0';
first_calc_p8<='0';
last_calc_p8<='0';
last_calc_p9<='0';
last_calc_p10<='0';
last_calc_p11<='0';
last_calc_p12<='0';
last_calc_p13<='0';
last_calc_p14<='0';
last_calc_p15<='0';
valid_calc_reg<='0';
out_data_reg1<=(OTHERS=>'0');
out_rdy_reg1<='0';
```

```
                    out_data_reg2<=(OTHERS=>'0');
                    out_rdy_reg2<='0';
                    out_data_reg3<=(OTHERS=>'0');
                    out_rdy_reg3<='0';
                    input_stall<='1';
--                  input_stall_2<='1';
                    new_frame_p1<='0';
                    new_frame_p2<='0';
                    new_frame_p3<='0';
                    new_frame_p4<='0';
                    new_frame_p5<='0';
                    new_frame_p6<='0';
                    new_frame_p7<='0';
                    new_frame_p8<='0';
                    new_frame_p9<='0';
                    new_frame_p10<='0';
                    new_frame_p11<='0';
                    new_frame_p12<='0';
                    new_frame_p13<='0';
                elsif (ce = '1') then
                    if (last_calc_p8 = '1')then
                        sub_in1<=VK_p8;
                        sub_in2<=SUM_OUT(31 downto 0);
                        REG_S1<=MW_p8;
                    else
                        sub_in1<=X;
                        sub_in2<=MW;
                        REG_S1<=VK;
                    end if;
                    input_stall<=last_calc_p6;--changed JWS 12-07-05 was _p6
--                  input_stall_2<=new_frame_p13;
```

```
valid_calc_reg<=valid_calc;
--------STAGE 1----------
DIFF_valid<=valid_calc_reg or last_calc_p9;
if (valid_calc_reg = '1' or last_calc_p9 = '1' )then
        DIFF<=sub_in1-sub_in2;
end if;
--------STAGE 2----------
DIFF_reg<=DIFF;
DIFF_valid_reg<=DIFF_valid;
if(last_calc_p10 = '1')then
        mul_in1_reg<=F;
else
        mul_in1_reg<=DIFF;
end if;
--------STAGE 3----------
mul_in1<=mul_in1_reg;
mul_in2<=DIFF_reg;
DIFF_valid_reg1<=DIFF_valid_reg;
SQUARE_valid<=DIFF_valid_reg1;
if(DIFF_valid_reg1 = '1')then
        SQUARE<=mul_in1*mul_in2;
end if;
--------STAGE 4----------
SQUARE_valid_reg<=SQUARE_valid;
if(last_calc_p13 = '1')then
        add_in1<=SQUARE(31 downto 0);
        add_in2<=REG_S5;
        scal_in1<=SQUARE(31 downto 0);
        scal_in2<=REG_S5;
else
        scal_in1<=SQUARE(31 downto 0);
        scal_in2<=REG_S5;
end if;
SCALE_valid<=SQUARE_valid ;
if(SQUARE_valid = '1')then
        SCALE<=scal_in1*scal_in2;
end if;
--------STAGE 5----------
COMP_valid<=SQUARE_valid_reg;
if(SQUARE_valid_reg = '1')then
        COMP_OUT<=add_in1+add_in2;
end if;
if(last_calc_p15 = '1')then
        sum_in1<=MW_p6;
else
        sum_in1<=SCALE(31 downto 0);
end if;
-------STAGE 6----------
scale_valid_reg<=scale_valid;
if (SCALE_valid_reg = '1')then
        if (first_calc_p8 ='1')then
                SUM_OUT<=sum_in1;
        else
                SUM_OUT<=sum_in1+SUM_OUT;
        end if;
end if;
-------REGISTERS--------
out_data_reg1<=COMP_OUT;
out_rdy_reg1<=COMP_valid AND last_calc_p15;
out_data_reg2<=out_data_reg1;
out_rdy_reg2<=out_rdy_reg1;
out_data_reg3<=out_data_reg2;
```

```
out_rdy_reg3<=out_rdy_reg2;
COMP_SCORE<=out_data_reg3;
COMP_RDY<=out_rdy_reg3;
new_frame_p1<=new_frame;
new_frame_p2<=new_frame_p1;
new_frame_p3<=new_frame_p2;
new_frame_p4<=new_frame_p3;
new_frame_p5<=new_frame_p4;
new_frame_p6<=new_frame_p5;
new_frame_p7<=new_frame_p6;
new_frame_p8<=new_frame_p7;
new_frame_p9<=new_frame_p8;
new_frame_p10<=new_frame_p9;
new_frame_p11<=new_frame_p10;
new_frame_p12<=new_frame_p11;
new_frame_p13<=new_frame_p12;
REG_S2<=REG_S1;
REG_S3<=REG_S2;
REG_S4<=REG_S3;
REG_S5<=REG_S4;
last_calc_p1<=last_calc;
first_calc_p1<=first_calc;
last_calc_p2<=last_calc_p1;
first_calc_p2<=first_calc_p1;
last_calc_p3<=last_calc_p2;
first_calc_p3<=first_calc_p2;
last_calc_p4<=last_calc_p3;
first_calc_p4<=first_calc_p3;
last_calc_p5<=last_calc_p4;
first_calc_p5<=first_calc_p4;
first_calc_p6<=first_calc_p5;
```

```
                last_calc_p6<=last_calc_p5;
                first_calc_p7<=first_calc_p6;
                last_calc_p7<=last_calc_p6;
                first_calc_p8<=first_calc_p7;
                last_calc_p8<=last_calc_p7;
                last_calc_p9<=last_calc_p8;
                last_calc_p10<=last_calc_p9;
                last_calc_p11<=last_calc_p10;
                last_calc_p12<=last_calc_p11;
                last_calc_p13<=last_calc_p12;
                last_calc_p14<=last_calc_p13;
                last_calc_p15<=last_calc_p14;
                MW_p1<=MW;
                MW_p2<=MW_p1;
                MW_p3<=MW_p2;
                MW_p4<=MW_p3;
                MW_p5<=MW_p4;
                MW_p6<=MW_p5;
                MW_p7<=MW_p6;
                MW_p8<=MW_p7;
                VK_p1<=VK;
                VK_p2<=VK_p1;
                VK_p3<=VK_p2;
                VK_p4<=VK_p3;
                VK_p5<=VK_p4;
                VK_p6<=VK_p5;
                VK_p7<=VK_p6;
                VK_p8<=VK_p7;

        end if;
end if;
end process;

end struct;
```

B.6 – log_add_calc_V1_FINAL.vhd

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:     Jeff Schuster
--
-- Create Date:    13:31:07 10/06/05
-- Design Name:
-- Module Name:    log_add_calc - Behavioral
-- Project Name:
-- Target Device:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
```

```
--library UNISIM;
--use UNISIM.VComponents.all;

entity log_add_calc is
  Port ( clk       : in std_logic;
       ce        : in std_logic;
       sclr      : in std_logic;
                         valid_calc : in std_logic;
                         first_comp : in std_logic;
                         mid_comp   : in std_logic;
                         last_comp  : in std_logic;
       COMP_IN    : in std_logic_vector(31 downto 0);
       COMP_RDY   : in std_logic;
       SEN_OUT    : out std_logic_vector(31 downto 0);
       SEN_RDY    : out std_logic);
end log_add_calc;

architecture Behavioral of log_add_calc is

SIGNAL NEW_COMP       : std_logic_vector(31 downto 0);
SIGNAL SEN_TMP        : std_logic_vector(31 downto 0);
SIGNAL SEN_TMP1     : std_logic_vector(31 downto 0);
SIGNAL SEN_TMP_REG    : std_logic_vector(31 downto 0);
SIGNAL SEN_TMP1_REG   : std_logic_vector(31 downto 0);
SIGNAL SEN_RDY_TMP    : std_logic;
SIGNAL SEN_RDY_TMP1   : std_logic;
SIGNAL SEN_RDY_TMP2   : std_logic;
SIGNAL SEN_RDY_TMP3   : std_logic;
SIGNAL SEN_RDY_TMP4   : std_logic;
SIGNAL SEN_RDY_TMP5   : std_logic;
SIGNAL SEN_RDY_TMP6   : std_logic;
SIGNAL SEN_RDY_TMP7   : std_logic;
SIGNAL SEN_RDY_TMP8   : std_logic;
SIGNAL SEN_RDY_TMP9   : std_logic;
SIGNAL SEN_RDY_TMP10  : std_logic;
SIGNAL SEN_RDY_TMP11  : std_logic;
SIGNAL SEN_RDY_TMP12  : std_logic;
SIGNAL first_comp_reg : std_logic;
SIGNAL mid_comp_reg        : std_logic;
SIGNAL last_comp_reg  : std_logic;
SIGNAL first_comp_reg1: std_logic;
SIGNAL mid_comp_reg1  : std_logic;
SIGNAL last_comp_reg1 : std_logic;
SIGNAL ADDR_SUB       : std_logic_vector(31 downto 0);
SIGNAL ADDR_SUB_TMP   : std_logic_vector(31 downto 0);
SIGNAL BIG            : std_logic_vector(31 downto 0);
SIGNAL LUT_ADDR       : std_logic_vector(14 downto 0);
SIGNAL LUT_DATA_REG   : std_logic_vector(11 downto 0);
SIGNAL LUT_DATA_LONG  : std_logic_vector(31 downto 0);
SIGNAL RES_TMP        : std_logic_vector(31 downto 0);
SIGNAL RES            : std_logic_vector(31 downto 0);
SIGNAL tick           : std_logic;
SIGNAL ADDRESS1       : std_logic_vector(14 downto 0);
SIGNAL ADDRESS2       : std_logic_vector(14 downto 0);
SIGNAL ADDRESS3       : std_logic_vector(14 downto 0);
SIGNAL ADDRESS4       : std_logic_vector(14 downto 0);
SIGNAL ADDRESS5       : std_logic_vector(14 downto 0);
SIGNAL LUT_DATA1      : std_logic_vector (11 downto 0);
SIGNAL LUT_DATA2      : std_logic_vector (11 downto 0);
SIGNAL LUT_DATA3      : std_logic_vector (11 downto 0);
SIGNAL LUT_DATA4      : std_logic_vector (11 downto 0);
SIGNAL LUT_DATA5      : std_logic_vector (11 downto 0);
SIGNAL chk            :  std_logic_vector(2 downto 0);
SIGNAL comp_rdy_reg   : std_logic;
SIGNAL valid_calc_reg : std_logic;
signal addr_chk       : std_logic_vector(2 downto 0);
SIGNAL BIG_REG1       : std_logic_vector(31 downto 0);
SIGNAL BIG_REG2       : std_logic_vector(31 downto 0);
SIGNAL BIG_REG3       : std_logic_vector(31 downto 0);
SIGNAL tick_reg       : std_logic;
SIGNAL TEST_REG1      : std_logic_vector(31 downto 0);
SIGNAL TEST_REG2      : std_logic_vector(31 downto 0);
SIGNAL fcomp_reg      : std_logic;
SIGNAL lcomp_reg      : std_logic;

attribute syn_ramstyle : string;
attribute syn_ramstyle of LUT_DATA1 : signal is "block_ram";
attribute syn_ramstyle of LUT_DATA2 : signal is "block_ram";
attribute syn_ramstyle of LUT_DATA3 : signal is "block_ram";
attribute syn_ramstyle of LUT_DATA4 : signal is "block_ram";
attribute syn_ramstyle of LUT_DATA5 : signal is "block_ram";

type s1_array is array (0 to 4095) of bit_vector (11 downto 0);
type s2_array is array (0 to 4095) of bit_vector (11 downto 0);
type s3_array is array (0 to 4095) of bit_vector (11 downto 0);
type s4_array is array (0 to 4095) of bit_vector (11 downto 0);
type s5_array is array (0 to 4095) of bit_vector (11 downto 0);

   -- Internal signal declarations
constant sbox1 : s1_array :=
(
REMOVED FOR SIZE CONSIDERATION
);
```

```
constant sbox2 : s2_array :=
(
REMOVED FOR SIZE CONSIDERATION
);
constant sbox3 : s3_array :=
(
REMOVED FOR SIZE CONSIDERATION
);
constant sbox4 : s4_array :=
(
REMOVED FOR SIZE CONSIDERATION
);
constant sbox5 : s5_array :=
(
REMOVED FOR SIZE CONSIDERATION
);

begin

process(clk,ce,sclr,comp_rdy)

begin
        if(clk ='1' and clk'event)then
                if(sclr = '1')then
                        address1<=(OTHERS=>'0');
                        address2<=(OTHERS=>'0');
                        address3<=(OTHERS=>'0');
                        address4<=(OTHERS=>'0');
                        address5<=(OTHERS=>'0');
                        SEN_OUT<=(OTHERS=>'0');
                        SEN_RDY<='0';
```

```
SEN_TMP<=(OTHERS=>'0');
SEN_TMP1<=(OTHERS=>'0');
SEN_TMP_REG<=(OTHERS=>'0');
SEN_TMP1_REG<=(OTHERS=>'0');
SEN_RDY_TMP<='0';
SEN_RDY_TMP1<='0';
SEN_RDY_TMP2<='0';
SEN_RDY_TMP3<='0';
SEN_RDY_TMP4<='0';
SEN_RDY_TMP5<='0';
SEN_RDY_TMP6<='0';
SEN_RDY_TMP7<='0';
SEN_RDY_TMP8<='0';
SEN_RDY_TMP9<='0';
SEN_RDY_TMP10<='0';
SEN_RDY_TMP11<='0';
SEN_RDY_TMP12<='0';
first_comp_reg<='0';
mid_comp_reg<='0';
last_comp_reg<='0';
first_comp_reg1<='0';
mid_comp_reg1<='0';
last_comp_reg1<='0';
RES_TMP<=(OTHERS=>'0');
LUT_DATA_REG<=(OTHERS=>'0');
LUT_DATA_LONG<=(OTHERS=>'0');
ADDR_SUB<=(OTHERS=>'0');
ADDR_SUB_TMP<=(OTHERS=>'0');
BIG<=(OTHERS=>'0');
NEW_COMP<=(OTHERS=>'0');
tick<='0';
```

```
          chk<="000";
          comp_rdy_reg<='0';
          valid_calc_reg<='0';
          addr_chk<="000";
          BIG_REG1<=(OTHERS=>'0');
          BIG_REG2<=(OTHERS=>'0');
          BIG_REG3<=(OTHERS=>'0');
          tick_reg<='0';
          TEST_REG1<=(OTHERS=>'0');
          TEST_REG2<=(OTHERS=>'0');
          fcomp_reg<='0';
          lcomp_reg<='0';

elsif (ce = '1')then
          addr_chk<=LUT_ADDR(14 downto 12);
          case addr_chk is
                    when "000"=>
                              address1<="000" & LUT_ADDR(11 downto 0);
                              address2<=(OTHERS=>'0');
                              address3<=(OTHERS=>'0');
                              address4<=(OTHERS=>'0');
                              address5<=(OTHERS=>'0');
                              chk<="000";
                    when "001"=>
                              address1<=(OTHERS=>'0');
                              address2<="000" & LUT_ADDR(11 downto 0);
                              address3<=(OTHERS=>'0');
                              address4<=(OTHERS=>'0');
                              address5<=(OTHERS=>'0');
                              chk<="001";
                    when "010"=>
                    address1<=(OTHERS=>'0');
                    address2<=(OTHERS=>'0');
                    address3<="000" & LUT_ADDR(11 downto 0);
                    address4<=(OTHERS=>'0');
                    address5<=(OTHERS=>'0');
                    chk<="010";
          when "011"=>
                    address1<=(OTHERS=>'0');
                    address2<=(OTHERS=>'0');
                    address3<=(OTHERS=>'0');
                    address4<="000" & LUT_ADDR(11 downto 0);
                    address5<=(OTHERS=>'0');
                    chk<="011";
          when others=>
                    address1<=(OTHERS=>'0');
                    address2<=(OTHERS=>'0');
                    address3<=(OTHERS=>'0');
                    address4<=(OTHERS=>'0');
                    address5<="000" & LUT_ADDR(11 downto 0);
                    chk<="100";
end case;
LUT_DATA1<=To_StdLogicVector(sbox1(conv_integer(address1)));
LUT_DATA2<=To_StdLogicVector(sbox2(conv_integer(address2)));
LUT_DATA3<=To_StdLogicVector(sbox3(conv_integer(address3)));
LUT_DATA4<=To_StdLogicVector(sbox4(conv_integer(address4)));
LUT_DATA5<=To_StdLogicVector(sbox5(conv_integer(address5)));
case chk is
          when "000" =>
                              LUT_DATA_REG <=LUT_DATA1;
          when "001" =>
                              LUT_DATA_REG <=LUT_DATA2;
```

```
        when "010" =>
                LUT_DATA_REG <=LUT_DATA3;
        when "011" =>
                LUT_DATA_REG <=LUT_DATA4;
        when "100" =>
                LUT_DATA_REG <=LUT_DATA5;
        when others =>
                LUT_DATA_REG <=(OTHERS=>'0');
end case;
comp_rdy_reg<=comp_rdy;
valid_calc_reg<=valid_calc;
first_comp_reg1<=first_comp;
mid_comp_reg1<=mid_comp;
last_comp_reg1<=last_comp;
first_comp_reg<=first_comp_reg1;
mid_comp_reg<=mid_comp_reg1;
last_comp_reg<=last_comp_reg1;
if(comp_rdy_reg = '1' or valid_calc_reg = '1')then
        NEW_COMP<=COMP_IN;
        tick<='1';
else
        tick<='0';
end if;
if (tick = '1')then
        if(first_comp_reg = '1')then
                SEN_TMP<=NEW_COMP;
                SEN_TMP1<="11001000000000000000000000000000";
        else
                SEN_TMP<=NEW_COMP;
                SEN_TMP1<=RES_TMP;
        end if;
end if;
if(SEN_TMP >= SEN_TMP1)then
        ADDR_SUB<=SEN_TMP - SEN_TMP1;
        BIG<=SEN_TMP;
else
        ADDR_SUB<=SEN_TMP1 - SEN_TMP;
        BIG<=SEN_TMP1;
end if;
SEN_RDY_TMP1<=tick AND last_comp_reg;
if (ADDR_SUB > "0100111111111111")then
        LUT_ADDR<="100111111111111";
else
        LUT_ADDR<=ADDR_SUB(14 downto 0);
end if;

SEN_RDY_TMP2<=SEN_RDY_TMP1;
SEN_RDY_TMP3<=SEN_RDY_TMP2;
SEN_RDY_TMP4<=SEN_RDY_TMP3;
SEN_RDY_TMP5<=SEN_RDY_TMP4;
SEN_RDY_TMP6<=SEN_RDY_TMP5;
SEN_RDY_TMP7<=SEN_RDY_TMP6;
SEN_RDY_TMP8<=SEN_RDY_TMP7;
SEN_RDY_TMP9<=SEN_RDY_TMP8;
SEN_RDY_TMP10<=SEN_RDY_TMP9;
SEN_RDY_TMP11<=SEN_RDY_TMP10;
SEN_RDY_TMP12<=SEN_RDY_TMP11;
BIG_REG1<=BIG;
BIG_REG2<=BIG_REG1;
BIG_REG3<=BIG_REG2;
LUT_DATA_LONG<="00000000000000000000" & LUT_DATA_REG;
RES_TMP<= LUT_DATA_LONG+BIG_REG3;
```

```
            RES<=RES_TMP;
            SEN_OUT<=RES;
            SEN_RDY<=SEN_RDY_TMP11;
    else
            address1<=(OTHERS=>'0');
            address2<=(OTHERS=>'0');
            address3<=(OTHERS=>'0');
            address4<=(OTHERS=>'0');
            address5<=(OTHERS=>'0');
            SEN_OUT<=(OTHERS=>'0');
            SEN_RDY<='0';
            SEN_TMP<=(OTHERS=>'0');
            SEN_TMP1<=(OTHERS=>'0');
            SEN_TMP_REG<=(OTHERS=>'0');
            SEN_TMP1_REG<=(OTHERS=>'0');
            SEN_RDY_TMP<='0';
            SEN_RDY_TMP1<='0';
            SEN_RDY_TMP2<='0';
            SEN_RDY_TMP3<='0';
            SEN_RDY_TMP4<='0';
            SEN_RDY_TMP5<='0';
            SEN_RDY_TMP6<='0';
            SEN_RDY_TMP7<='0';
            SEN_RDY_TMP8<='0';
            SEN_RDY_TMP9<='0';
            SEN_RDY_TMP10<='0';
            SEN_RDY_TMP11<='0';
            first_comp_reg<='0';
            mid_comp_reg<='0';
            last_comp_reg<='0';
            first_comp_reg1<='0';
```

```
                mid_comp_reg1<='0';
                last_comp_reg1<='0';
                comp_rdy_reg<='0';
                valid_calc_reg<='0';
                LUT_ADDR<=(OTHERS=>'0');
                LUT_DATA_LONG<=(OTHERS=>'0');
                ADDR_SUB<=(OTHERS=>'0');
                ADDR_SUB_TMP<=(OTHERS=>'0');
                RES<=(OTHERS=>'0');
                RES_TMP<=(OTHERS=>'0');
                LUT_DATA_REG<=(OTHERS=>'0');
                BIG<=(OTHERS=>'0');
                NEW_COMP<=(OTHERS=>'0');
                tick<='0';

            end if;
        end if;
end process;

end Behavioral;
```

B.7 – normalizer.vdh

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:            Jeffrey W. Schuster
--
-- Create Date:    13:30:33 10/11/05
-- Design Name:
-- Module Name:    normalizer - Behavioral
-- Project Name:
```

```
-- Target Device:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

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

entity normalizer is
   Port (
                              clk            : in std_logic;
         ce             : in std_logic;
         sclr           : in std_logic;
         start_normalize : in std_logic;
                              last_raw_senone : in std_logic;
                              raw_senone      : in std_logic_vector(31 downto 0);
                              raw_sen_rdy     : in std_logic;
                              best_score      : in std_logic_vector(31 downto 0);
```

```
        address         : out std_logic_vector(12 downto 0);
                          address_rdy     : out std_logic;
                    norm_senone     : out std_logic_vector(31 downto 0);
        senone_rdy      : out std_logic;
                          normalize_done  : out std_logic;
                          ram_addr_sel    : out std_logic;
                          sen_cnt         : out std_logic_vector(12 downto 0)
                          );
end normalizer;

architecture Behavioral of normalizer is

SIGNAL RES_TMP        : std_logic_vector(31 downto 0);
SIGNAL RES_TMP1       : std_logic_vector(31 downto 0);
SIGNAL RAW_TMP        : std_logic_vector(31 downto 0);
SIGNAL RES_RDY        : std_logic;
SIGNAL RES_RDY1       : std_logic;
SIGNAL RES_RDY2       : std_logic;
SIGNAL COUNT1         : std_logic_vector(12 downto 0);
SIGNAL COUNT2         : std_logic_vector(12 downto 0);
SIGNAL COUNT3         : std_logic_vector(12 downto 0);
SIGNAL COUNT4         : std_logic_vector(12 downto 0);
SIGNAL norm_done_reg  : std_logic;
SIGNAL norm_done_reg1 : std_logic;
SIGNAL norm_done_reg2 : std_logic;
SIGNAL norm_done_reg3 : std_logic;
SIGNAL addr_sel       : std_logic;
SIGNAL addr_sel1      : std_logic;
SIGNAL addr_sel2      : std_logic;
SIGNAL addr_sel3      : std_logic;
SIGNAL addr_rdy       : std_logic;
SIGNAL addr_rdy1      : std_logic;
SIGNAL addr_rdy2      : std_logic;
SIGNAL addr_rdy3      : std_logic;
SIGNAL pipe_freeze    : std_logic;
signal raw_rdy_reg    : std_logic;



begin

process(clk,sclr,ce,last_raw_senone,raw_sen_rdy)
begin

if(clk = '1' and clk'event)then
        if (sclr = '1')then
                count1<=(OTHERS=>'0');
                count2<=(OTHERS=>'0');
                count3<=(OTHERS=>'0');
                count4<=(OTHERS=>'0');
                RES_TMP<=(OTHERS=>'0');
                RES_TMP1<=(OTHERS=>'0');
                RAW_TMP<=(OTHERS=>'0');
                RES_RDY<='0';
                RES_RDY1<='0';
                RES_RDY2<='0';
          norm_done_reg<='0';
                norm_done_reg1<='0';
                norm_done_reg2<='0';
                norm_done_reg3<='0';
                addr_sel<='0';
                addr_sel1<='0';
                addr_sel2<='0';
                addr_sel3<='0';
                ram_addr_sel<='0';
                senone_rdy<='0';
                norm_senone<=(OTHERS=>'0');
                normalize_done<='0';
                address_rdy<='0';
                addr_rdy<='0';
                addr_rdy1<='0';
                addr_rdy2<='0';
                addr_rdy3<='0';
                pipe_freeze<='1';
                sen_cnt<=(OTHERS=>'0');
                raw_rdy_reg<='0';
        elsif (start_normalize = '1')then
                count1<=(OTHERS=>'0');
                count2<=(OTHERS=>'0');
                count3<=(OTHERS=>'0');
                count4<=(OTHERS=>'0');
                addr_sel<='1';
                addr_sel1<='1';
                addr_sel2<='1';
                addr_sel3<='1';
                address_rdy<='0';
                addr_rdy<='1';
                addr_rdy1<='0';
                addr_rdy2<='0';
                addr_rdy3<='0';--changed 10-20-05 JWS : was <='1';
                pipe_freeze<='0';
        elsif(ce = '1')then
                raw_rdy_reg<=raw_sen_rdy;
```

```
        if(norm_done_reg3 = '1')then
                pipe_freeze<='1';
        end if;
if(pipe_freeze = '0')then
                if (raw_rdy_reg = '1')then
                        count1<=count1+1;
                        raw_tmp<=raw_senone;
                        res_rdy<=raw_rdy_reg;
                        addr_rdy<='1';
                        if (last_raw_senone = '1')then
                                norm_done_reg<=last_raw_senone;
                                addr_sel<='0';
                        else
                                norm_done_reg<='0';
                        end if;
                else
                        res_rdy<='0';
                        addr_rdy<='0';
                end if;
                res_tmp<=best_score-raw_tmp;

                addr_rdy1<=addr_rdy;
                addr_rdy2<=addr_rdy1;
                addr_rdy3<=addr_rdy2;
                address_rdy<=addr_rdy3;
                count2<=count1;
                count3<=count2;
                count4<=count3;
                norm_done_reg1<=norm_done_reg;
                norm_done_reg2<=norm_done_reg1;
                addr_sel1<=addr_sel;
        addr_sel2<=addr_sel1;
        addr_sel3<=addr_sel2;
        res_tmp1<=res_tmp;
        res_rdy1<=res_rdy;
        res_rdy2<=res_rdy1;
        norm_senone<=res_tmp1;
        senone_rdy<=res_rdy2;
        normalize_done<=norm_done_reg3;
        norm_done_reg3<=norm_done_reg2;
        ram_addr_sel<=addr_sel3;
        sen_cnt<=count4;
        if (res_rdy2 = '1')then
                address<=count4;
        else
                address<=count1;
        end if;
else
        address_rdy<='0';
        address<=(OTHERS=>'0');
        RES_TMP<=(OTHERS=>'0');
        RAW_TMP<=(OTHERS=>'0');
        RES_RDY<='0';
        RES_TMP1<=(OTHERS=>'0');
        RES_RDY1<='0';
        RES_RDY2<='0';
        norm_done_reg<='0';
        norm_done_reg1<='0';
        norm_done_reg2<='0';
        norm_done_reg3<='0';
        normalize_done<='0';
end if;
```

```
          else
                    RES_TMP<=(OTHERS=>'0');
                    RAW_TMP<=(OTHERS=>'0');
                    RES_RDY<='0';
                    RES_TMP1<=(OTHERS=>'0');
                    RES_RDY1<='0';
                    RES_RDY2<='0';
                    norm_done_reg<='0';
                    norm_done_reg1<='0';
                    norm_done_reg2<='0';
                    norm_done_reg3<='0';
                    normalize_done<='0';
          end if;
end if;
end process;

end Behavioral;
```

B.8 – hmm_top_struct.vhd

```
-- VHDL Entity phoneme_evaluator_lib.hmm_top.symbol
--
-- Created:          Jeffrey W. Schuster
--          by - Speech Research.UNKNOWN (SPEECH1)
--          at - 13:51:33 01/19/2006
--
-- Generated by Mentor Graphics' HDL Designer(TM) 2003.1 (Build 399)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
```

```
ENTITY hmm_top IS
  PORT(
    PH_WRD_SEL    : IN    std_logic;
    clk           : IN    std_logic;
    dead_rd       : IN    std_logic;
    global_en     : IN    std_logic;
    nPAL_data     : IN    std_logic_VECTOR (9 DOWNTO 0);
    nPAL_wr       : IN    std_logic;
    phone_en      : IN    std_logic;
    phone_start   : IN    std_logic;
    senone_data_in : IN    std_logic_vector (31 DOWNTO 0);
    senone_mux_sel : IN    std_logic;
    senone_wr_addr : IN    std_logic_vector (12 DOWNTO 0);
    senone_wr_en   : IN    std_logic;
    sinit         : IN    std_logic;
    valid_rd      : IN    std_logic;
    word_addr_rdy  : IN    std_logic;
    word_data     : IN    std_logic_vector (192 DOWNTO 0);
    word_rd_addr   : IN    std_logic_vector (8 DOWNTO 0);
    word_wr       : IN    std_logic;
    word_wr_addr   : IN    std_logic_vector (8 DOWNTO 0);
    dead_empty    : OUT   std_logic;
    dead_full     : OUT   std_logic;
    dead_out      : OUT   std_logic_VECTOR (9 DOWNTO 0);
    phone_done    : OUT   std_logic;
    valid_empty   : OUT   std_logic;
    valid_full    : OUT   std_logic;
    valid_out     : OUT   std_logic_VECTOR (9 DOWNTO 0);
    word_beam     : OUT   std_logic_vector (31 DOWNTO 0)
--     word_data_out  : OUT   std_logic_vector (252 DOWNTO 0)
```

```
  );

-- Declarations

END hmm_top ;

--
-- VHDL Architecture phoneme_evaluator_lib.hmm_top.struct
--
-- Created:
--          by - Speech Research.UNKNOWN (SPEECH1)
--          at - 13:51:34 01/19/2006
--
-- Generated by Mentor Graphics' HDL Designer(TM) 2003.1 (Build 399)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE IEEE.STD_LOGIC_SIGNED.all;
USE ieee.STD_LOGIC_UNSIGNED.all;



ARCHITECTURE struct OF hmm_top IS

  -- Architecture declarations
-- Non hierarchical state machine declarations
TYPE CSM1_STATE_TYPE IS (
    idle,
    npal,
    pal,
      prune,
      init
   );

-- State vector declaration
ATTRIBUTE state_vector : string;
ATTRIBUTE state_vector OF struct : ARCHITECTURE IS "csm1_current_state" ;


-- Declare current and next state signals
SIGNAL csm1_current_state : CSM1_STATE_TYPE ;
SIGNAL csm1_next_state : CSM1_STATE_TYPE ;


   -- Internal signal declarations
   SIGNAL addr_rdy         : std_logic;
   SIGNAL data_in          : std_logic_vector(9 DOWNTO 0);
   SIGNAL data_ready       : std_logic;
   SIGNAL dead             : std_logic;
   SIGNAL din              : std_logic_VECTOR(9 DOWNTO 0);
   SIGNAL dout             : std_logic_VECTOR(9 DOWNTO 0);
   SIGNAL dout1            : std_logic_VECTOR(9 DOWNTO 0);
   SIGNAL enable           : std_logic;
   SIGNAL end_of_phase     : std_logic;
   SIGNAL exit_beam        : std_logic_vector(31 DOWNTO 0);
   SIGNAL fifo_sel         : std_logic;
   SIGNAL first_calc       : std_logic;
   SIGNAL hmm_calc_en      : std_logic;
   SIGNAL hmm_rdy          : std_logic;
   SIGNAL new_data         : std_logic;
   SIGNAL norm_wr          : std_logic;
   SIGNAL npal_empty       : std_logic;
   SIGNAL npal_full        : std_logic;
   SIGNAL npal_rd          : std_logic;
   SIGNAL pal_empty        : std_logic;
   SIGNAL pal_full         : std_logic;
   SIGNAL pal_rd           : std_logic;
   SIGNAL pal_val          : std_logic;
   SIGNAL pal_wr           : std_logic;
   SIGNAL phone_data_out   : std_logic_vector(252 DOWNTO 0);
   SIGNAL prune_en         : std_logic;
   SIGNAL ptr_ram_data_out : std_logic_vector(192 DOWNTO 0);
   SIGNAL tkn_rdy          : std_logic;
           SIGNAL tkn_rdy_tmp      : std_logic;
   SIGNAL token            : std_logic_vector(8 DOWNTO 0);
   SIGNAL token_out        : std_logic_vector(8 DOWNTO 0);
   SIGNAL token_pop        : std_logic;
   SIGNAL token_to_fifo    : std_logic_vector(9 DOWNTO 0);
   SIGNAL valid_beam       : std_logic_vector(31 DOWNTO 0);
   SIGNAL wr_en            : std_logic;
           SIGNAL token_reg        : std_logic_vector(8 downto 0);
           SIGNAL phn_wr           : std_logic;
           SIGNAL word_data_out_tmp: std_logic_vector(252 downto 0);
           SIGNAL end_of_phase_reg : std_logic;


   -- ModuleWare signal declarations(v1.0) for instance 'I5' of 'mux'
   SIGNAL mw_I5din0 : std_logic_vector(9 DOWNTO 0);
   SIGNAL mw_I5din1 : std_logic_vector(9 DOWNTO 0);

   -- Component Declarations
   COMPONENT PH_PTR_RAM
```

```
PORT (
  PH_WRD_SEL     : IN     std_logic;
  addr_rdy       : IN     std_logic;
  clk            : IN     std_logic;
  init           : IN     std_logic;
  phone_data     : IN     std_logic_vector (192 DOWNTO 0);
  phone_rd_addr  : IN     std_logic_vector (8 DOWNTO 0);
  phone_wr       : IN     std_logic;
  phone_wr_addr  : IN     std_logic_vector (8 DOWNTO 0);
  word_data      : IN     std_logic_vector (192 DOWNTO 0);
  word_rd_addr   : IN     std_logic_vector (8 DOWNTO 0);
  word_wr        : IN     std_logic;
  word_wr_addr   : IN     std_logic_vector (8 DOWNTO 0);
  new_data       : OUT    std_logic;
  phone_data_out : OUT    std_logic_vector (252 DOWNTO 0);
  word_data_out  : OUT    std_logic_vector (252 DOWNTO 0)
);
END COMPONENT;
COMPONENT dp_fifo
PORT (
  clk   : IN     std_logic;
  din   : IN     std_logic_VECTOR (9 DOWNTO 0);
  rd_en : IN     std_logic;
  sinit : IN     std_logic;
  wr_en : IN     std_logic;
  dout  : OUT    std_logic_VECTOR (9 DOWNTO 0);
  empty : OUT    std_logic;
  full  : OUT    std_logic
);
END COMPONENT;
COMPONENT hmm_pipe_w_mdef
```

```
PORT (
  ce             : IN     std_logic;
  clk            : IN     std_logic;
  new_frame      : IN     std_logic;
  new_input_data  : IN     std_logic;
  ptr_ram_data_in : IN     std_logic_vector (252 DOWNTO 0);
  sclr           : IN     std_logic;
  senone_data_in  : IN     std_logic_vector (31 DOWNTO 0);
  senone_mux_sel  : IN     std_logic;
  senone_wr_addr  : IN     std_logic_vector (12 DOWNTO 0);
  senone_wr_en    : IN     std_logic;
  exit_beam      : OUT    std_logic_vector (31 DOWNTO 0);
  hmm_rdy        : OUT    std_logic;
  ptr_ram_data_out : OUT    std_logic_vector (192 DOWNTO 0);
  valid_beam     : OUT    std_logic_vector (31 DOWNTO 0);
  word_beam      : OUT    std_logic_vector (31 DOWNTO 0)
);
END COMPONENT;
COMPONENT prune_block
PORT (
  clk        : IN     std_logic;
  e_beam     : IN     std_logic_vector (31 DOWNTO 0);
  hmm_data    : IN     std_logic_vector (252 DOWNTO 0);
  prune_enable : IN     std_logic;
  ram_data_rdy : IN     std_logic;
  reset      : IN     std_logic;
  v_beam     : IN     std_logic_vector (31 DOWNTO 0);
  data_ready  : OUT    std_logic;
  dead       : OUT    std_logic;
  pal        : OUT    std_logic;
  pal_val     : OUT    std_logic
  );
  END COMPONENT;

  -- Optional embedded configurations
  -- pragma synthesis_off
  FOR ALL : PH_PTR_RAM USE ENTITY work.PH_PTR_RAM;
  FOR ALL : dp_fifo USE ENTITY work.dp_fifo;
  FOR ALL : hmm_pipe_w_mdef USE ENTITY work.hmm_pipe_w_mdef;
  FOR ALL : prune_block USE ENTITY work.prune_block;
  -- pragma synthesis_on


BEGIN
  -- Architecture concurrent statements
  -- HDL Embedded Block 1 eb1
  -- Non hierarchical state machine
  ---------------------------------------------------------------------------
  csm1_clocked : PROCESS(
    clk
  )
  ---------------------------------------------------------------------------
  BEGIN
    IF (clk'EVENT AND clk = '1') THEN
      IF (enable = '1') THEN
        IF (sinit = '1') THEN
          csm1_current_state <= idle;
          -- Reset Values
        ELSE
          csm1_current_state <= csm1_next_state;
          -- Default Assignment To Internals
        END IF;
      END IF;
    END IF;

  END PROCESS csm1_clocked;

  ---------------------------------------------------------------------------
  csm1_nextstate : PROCESS (
    csm1_current_state,
    data_ready,
    end_of_phase,
    hmm_rdy,
    npal_empty,
    phone_start
  )
  ---------------------------------------------------------------------------
  BEGIN
    CASE csm1_current_state IS
    WHEN idle =>
      IF (phone_start = '1') THEN
        csm1_next_state <= init;
      ELSE
        csm1_next_state <= idle;
      END IF;
    WHEN npal =>
      IF (hmm_rdy = '0') THEN
        csm1_next_state <= npal;
      ELSIF (hmm_rdy = '1' AND npal_empty = '0') THEN
        csm1_next_state <= npal;
      ELSIF (hmm_rdy = '1' AND npal_empty = '1') THEN
        csm1_next_state <= pal;
      ELSE
        csm1_next_state <= npal;
      END IF;
    WHEN pal =>
      IF (hmm_rdy = '0') THEN
        csm1_next_state <= pal;
      ELSIF (hmm_rdy = '1'  AND end_of_phase = '0') THEN
        csm1_next_state <= pal;
      ELSIF (hmm_rdy = '1' AND end_of_phase = '1') THEN
        csm1_next_state <= prune;
      ELSE
        csm1_next_state <= pal;
      END IF;
    WHEN prune =>
      IF (data_ready = '0') THEN
        csm1_next_state <= prune;
      ELSIF (data_ready = '1' AND end_of_phase = '0') THEN
        csm1_next_state <= prune;
      ELSIF (data_ready = '1' AND end_of_phase = '1') THEN
        csm1_next_state <= idle;
      ELSE
        csm1_next_state <= prune;
      END IF;
    WHEN init =>
      IF (npal_empty = '0') THEN
        csm1_next_state <= npal;
      ELSIF (npal_empty = '1') THEN
        csm1_next_state <= pal;
      ELSE
        csm1_next_state <= init;
      END IF;
      WHEN OTHERS =>
        csm1_next_state <= idle;
      END CASE;

    END PROCESS csm1_nextstate;

    ---------------------------------------------------------------------------
    csm1_output : PROCESS (
      csm1_current_state,
      data_ready,
      end_of_phase,
      hmm_rdy,
      npal_empty,
      phone_start
    )
    ---------------------------------------------------------------------------
    BEGIN
      -- Default Assignment
      fifo_sel <= '0';
      first_calc <= '0';
      hmm_calc_en <= '0';
      phone_done <= '0';
      prune_en <= '0';
      token_pop <= '0';
      -- Default Assignment To Internals

      -- Combined Actions
      CASE csm1_current_state IS
      WHEN idle =>
        IF (phone_start = '1') THEN
          first_calc<='1';
          fifo_sel<='0';
        ELSE
          prune_en<='0';
          hmm_calc_en<='0';
          phone_done<='0';
        END IF;
      WHEN npal =>
        IF (hmm_rdy = '0') THEN
          token_pop<='0';
          hmm_calc_en<='1';
          fifo_sel<='1';
        ELSIF (hmm_rdy = '1' AND npal_empty = '0') THEN
          hmm_calc_en<='1';
          token_pop<='1';
          fifo_sel<='1';
        ELSIF (hmm_rdy = '1' AND npal_empty = '1') THEN
          hmm_calc_en<='1';
          token_pop<='1';
          fifo_sel<='0';
        END IF;
      WHEN pal =>
        IF (hmm_rdy = '0') THEN
          hmm_calc_en<='1';
          token_pop<='0';
          fifo_sel<='0';
        ELSIF (hmm_rdy = '1'  AND end_of_phase = '0') THEN
          hmm_calc_en<='1';
          token_pop<='1';
          fifo_sel<='0';
        ELSIF (hmm_rdy = '1' AND end_of_phase = '1') THEN
          hmm_calc_en<='0';
```

```
        prune_en<='1';
        fifo_sel<='0';
        token_pop<='1';
                                    first_calc<='1';--added 1-20-06 -- JWS

      END IF;
    WHEN prune =>
      IF (data_ready = '0') THEN
        prune_en<='1';
        fifo_sel<='0';
        token_pop<='0';
      ELSIF (data_ready = '1' AND end_of_phase = '0') THEN
        prune_en<='1';
        fifo_sel<='0';
        token_pop<='1';
      ELSIF (data_ready = '1' AND end_of_phase = '1') THEN
        prune_en<='0';
        fifo_sel<='0';
        token_pop<='0';
        phone_done<='1';
      END IF;
    WHEN init =>
      IF (npal_empty = '0') THEN
        hmm_calc_en<='1';
        fifo_sel<='1';
        token_pop<='1';
        first_calc<='0';
      ELSIF (npal_empty = '1') THEN
        hmm_calc_en<='1';
        token_pop<='1';
        fifo_sel<='0';
        first_calc<='0';
      END IF;
    WHEN OTHERS =>
      NULL;
    END CASE;

  END PROCESS csm1_output;

  -- Concurrent Statements




  -- HDL Embedded Text Block 2 eb2
  -- eb2 2
  end_of_phase<=data_in(9);
  token<=data_in(8 downto 0);
        process(clk,sinit)
        begin
                if(clk = '1' and clk'event)then
                        if(sinit = '1')then
                                    token_reg<=(OTHERS=>'0');
                                    token_out<=(OTHERS=>'0');
                                    end_of_phase_reg<='0';
                        else
                                    token_reg<=token;
                                    token_out<=token_reg;
                                    end_of_phase_reg<=end_of_phase;
                        end if;
                end if;
        end process;
```

```
-- HDL Embedded Text Block 3 eb3
-- eb3 3
process(fifo_sel,sinit,clk,token_pop)
begin
        if(clk = '1' and clk'event)then
                if(sinit = '1')then
                        pal_rd<='0';
                        npal_rd<='0';
                elsif(fifo_sel = '0')then
                        pal_rd<=token_pop;
                        npal_rd<='0';
                elsif(fifo_sel = '1')then
                        pal_rd<='0';
                        npal_rd<=token_pop;
                else
                        pal_rd<='0';
                        npal_rd<='0';
                end if;
        end if;
end process;

-- HDL Embedded Text Block 4 eb4
-- eb4 4
process(clk,sinit,token_pop)
  variable tmp : std_logic;


  begin
    if(clk = '1' and clk'event)then
      if(sinit = '1')then
        tkn_rdy_tmp<='0';
                                            tkn_rdy<='0';
      else
        tkn_rdy_tmp<=token_pop;
                                   tkn_rdy<=tkn_rdy_tmp;
      end if;
  end if;

end process;

-- HDL Embedded Text Block 5 eb5
-- eb5 5
process(clk,sinit)
begin
if(clk = '1' and clk'event)then
  if(data_ready = '1')then
      token_to_fifo<='0' & token_out;
  end if;
end if;
end process;

-- HDL Embedded Text Block 6 eb6
-- eb6 6
process(first_calc,clk,norm_wr,sinit)
  begin
                      if(clk = '1' and clk'event)then
                              if(sinit = '1')then
                                   wr_en<='0';
                                   din<=(OTHERS=>'0');
                              elsif(first_calc = '1')then
              din<="1111111111";
              wr_en<=first_calc;
      elsif(norm_wr = '1')then
```

```
                din<='0' & token_out;
                wr_en<=norm_wr;
                        else
                            din<=(OTHERS=>'0');
                            wr_en<='0';
                        end if;
     end if;
  end process;


-- ModuleWare code(v1.0) for instance 'I5' of 'mux'
I5combo: PROCESS(mw_I5din0, mw_I5din1, fifo_sel)
VARIABLE dtemp : std_logic_vector(9 DOWNTO 0);
BEGIN
  CASE fifo_sel IS
  WHEN '0'|'L' => dtemp := mw_I5din0;
  WHEN '1'|'H' => dtemp := mw_I5din1;
  WHEN OTHERS => dtemp := (OTHERS => 'X');
  END CASE;
  data_in <= dtemp;
END PROCESS I5combo;
mw_I5din0 <= dout;
mw_I5din1 <= dout1;

enable <= global_en AND phone_en;
norm_wr <= hmm_rdy OR pal_wr OR pal_val;
addr_rdy <= word_addr_rdy OR tkn_rdy;
      phn_wr<= hmm_rdy AND not(end_of_phase_reg);

-- Instance port mappings.
I3 : PH_PTR_RAM
```

```
  PORT MAP (
    PH_WRD_SEL     => PH_WRD_SEL,
    addr_rdy       => addr_rdy,
    clk            => clk,
    init           => sinit,
    phone_data     => ptr_ram_data_out,
    phone_rd_addr  => token,
    phone_wr_addr  => token,
    phone_wr       => phn_wr,
    word_data      => word_data,
    word_rd_addr   => word_rd_addr,
    word_wr_addr   => word_wr_addr,
    word_wr        => word_wr,
    phone_data_out => phone_data_out,
    word_data_out  => word_data_out_tmp,
    new_data       => new_data
  );
I2 : dp_fifo
  PORT MAP (
    clk   => clk,
    sinit => sinit,
    din   => din,
    wr_en => wr_en,
    rd_en => pal_rd,
    dout  => dout,
    full  => pal_full,
    empty => pal_empty
  );
I4 : dp_fifo
  PORT MAP (
    clk   => clk,
    sinit => sinit,
    din   => nPAL_data,
    wr_en => nPAL_wr,
    rd_en => npal_rd,
    dout  => dout1,
    full  => npal_full,
    empty => npal_empty
  );
I7 : dp_fifo
  PORT MAP (
    clk   => clk,
    sinit => sinit,
    din   => token_to_fifo,
    wr_en => pal_val,
    rd_en => valid_rd,
    dout  => valid_out,
    full  => valid_full,
    empty => valid_empty
  );
I8 : dp_fifo
  PORT MAP (
    clk   => clk,
    sinit => sinit,
    din   => token_to_fifo,
    wr_en => dead,
    rd_en => dead_rd,
    dout  => dead_out,
    full  => dead_full,
    empty => dead_empty
  );
I0 : hmm_pipe_w_mdef
```

```
  PORT MAP (
    clk            => clk,
    ce             => hmm_calc_en,
    sclr           => sinit,
    new_frame       => phone_start,
    new_input_data   => new_data,
    senone_data_in   => senone_data_in,
    senone_wr_addr   => senone_wr_addr,
    senone_wr_en     => senone_wr_en,
    senone_mux_sel   => senone_mux_sel,
    ptr_ram_data_in  => phone_data_out,
    ptr_ram_data_out => ptr_ram_data_out,
    hmm_rdy          => hmm_rdy,
    exit_beam        => exit_beam,
    valid_beam       => valid_beam,
    word_beam        => word_beam
  );
I1 : prune_block
  PORT MAP (
    clk         => clk,
    reset        => sinit,
    prune_enable => prune_en,
    e_beam       => exit_beam,
    v_beam       => valid_beam,
    ram_data_rdy => new_data,
    hmm_data     => phone_data_out,
    data_ready   => data_ready,
    dead         => dead,
    pal          => pal_wr,
    pal_val      => pal_val
  );
```

```
END struct;
```

B.9 – hmm_pipe_w_mdef.vhd

```
----------------------------------------------------------------------------
-- Company:
-- Engineer:  Jeffrey W. Schuster
--
-- Create Date:    09:46:36 11/14/05
-- Design Name:
-- Module Name:    hmm_pipe_w_mdef - Behavioral
-- Project Name:
-- Target Device:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
```

```
--library UNISIM;
--use UNISIM.VComponents.all;

entity hmm_pipe_w_mdef is
   Port ( clk             : in std_logic;
         ce              : in std_logic;
         sclr            : in std_logic;
         new_frame        : in std_logic;
         new_input_data   : in std_logic;
                             senone_data_in   : in std_logic_vector(31 downto 0);
                             senone_wr_addr   : in std_logic_vector(12 downto 0);
                             senone_wr_en     : in std_logic;
                             senone_mux_sel   : in std_logic;
         ptr_ram_data_in  : in std_logic_vector(252 downto 0);
         ptr_ram_data_out : out std_logic_vector(192 downto 0);
         hmm_rdy          : out std_logic;
         exit_beam        : out std_logic_vector(31 downto 0);
         valid_beam       : out std_logic_vector(31 downto 0);
         word_beam        : out std_logic_vector(31 downto 0)
                            );
end hmm_pipe_w_mdef;


architecture Behavioral of hmm_pipe_w_mdef is

component hmm_calc
          PORT(
                    H0_prev    : IN     std_logic_vector (31 DOWNTO 0);
     H1_prev    : IN     std_logic_vector (31 DOWNTO 0);
     H2_prev    : IN     std_logic_vector (31 DOWNTO 0);
     Input_Prob : IN     std_logic_vector (31 DOWNTO 0);
     clk        : IN     std_logic;
     ce         : IN     std_logic;
     sclr       : IN     std_logic;
                 word_end   : IN     std_logic;
                 inputs_rdy : IN     std_logic;
                 new_frame  : IN     std_logic;
     senone_in0 : IN     std_logic_vector (31 DOWNTO 0);
     senone_in1 : IN     std_logic_vector (31 DOWNTO 0);
     senone_in2 : IN     std_logic_vector (31 DOWNTO 0);
     tmat00     : IN     std_logic_vector (31 DOWNTO 0);
     tmat01     : IN     std_logic_vector (31 DOWNTO 0);
     tmat11     : IN     std_logic_vector (31 DOWNTO 0);
     tmat12     : IN     std_logic_vector (31 DOWNTO 0);
     tmat22     : IN     std_logic_vector (31 DOWNTO 0);
     tmat2E     : IN     std_logic_vector (31 DOWNTO 0);
                 InProb_out : OUT    std_logic_vector (32 downto 0);
     H0Out      : OUT    std_logic_vector (31 DOWNTO 0);
     H1Out      : OUT    std_logic_vector (31 DOWNTO 0);
     H2Out      : OUT    std_logic_vector (31 DOWNTO 0);
     HMM_best   : OUT    std_logic_vector (31 DOWNTO 0);
     HMM_exit   : OUT    std_logic_vector (31 DOWNTO 0);
                 exit_beam  : OUT    std_logic_vector (31 downto 0);
                 valid_beam : OUT    std_logic_vector (31 downto 0);
                 word_beam  : OUT    std_logic_vector (31 downto 0);
     data_valid : OUT    std_logic
   );
END component ;
component mdef_rom
         PORT(
                 PH_ID       : IN     std_logic_VECTOR (12 DOWNTO 0);
                 new_id      : IN     std_logic;
```

```
    clk          : IN     std_logic;
    enable       : IN     std_logic;
    reset        : IN     std_logic;
    senone_addr1 : OUT    std_logic_vector (10 DOWNTO 0);
    senone_addr2 : OUT    std_logic_vector (10 DOWNTO 0);
    senone_addr3 : OUT    std_logic_vector (10 DOWNTO 0);
                sen_addr_rdy : OUT    std_logic;
    tmat00       : OUT    std_logic_vector (31 DOWNTO 0);
    tmat01       : OUT    std_logic_vector (31 DOWNTO 0);
    tmat11       : OUT    std_logic_vector (31 DOWNTO 0);
    tmat12       : OUT    std_logic_vector (31 DOWNTO 0);
    tmat22       : OUT    std_logic_vector (31 DOWNTO 0);
    tmat2E       : OUT    std_logic_vector (31 DOWNTO 0);
                tmat_scr_rdy : OUT    std_logic
 );
END component ;
component raw_senone_ram
        PORT(
                clk     : IN std_logic;
                we      : IN std_logic;
                din     : IN std_logic_vector(31 downto 0);
                addr    : IN std_logic_vector(12 downto 0);
                dout    : OUT std_logic_vector(31 downto 0)
                    );
end component;

FOR ALL : raw_senone_ram USE ENTITY work.raw_senone_ram;
FOR ALL : mdef_rom USE ENTITY work.mdef_rom;
FOR ALL : hmm_calc USE ENTITY work.hmm_calc;

SIGNAL ptr_ram_data_tmp   : std_logic_vector(192 downto 0);
SIGNAL ptr_ram_data_rdy   : std_logic;
SIGNAL senone_addr_rdy    : std_logic;
SIGNAL senone_data_rdy1   : std_logic;
SIGNAL H0                 : std_logic_vector(31 downto 0);
SIGNAL H1                 : std_logic_vector(31 downto 0);
SIGNAL H2                 : std_logic_vector(31 downto 0);
SIGNAL H0_out             : std_logic_vector(31 downto 0);
SIGNAL H1_out             : std_logic_vector(31 downto 0);
SIGNAL H2_out             : std_logic_vector(31 downto 0);
SIGNAL HMMbest            : std_logic_vector(31 downto 0);
SIGNAL HMMexit            : std_logic_vector(31 downto 0);
SIGNAL INPROB             : std_logic_vector(31 downto 0);
SIGNAL InPrb_out          : std_logic_vector(32 downto 0);
SIGNAL WRD_END            : std_logic;
SIGNAL PH_ID              : std_logic_vector(12 downto 0);
SIGNAL tmat_rdy_reg       : std_logic;
SIGNAL tmat00_reg         : std_logic_vector(31 downto 0);
SIGNAL tmat01_reg         : std_logic_vector(31 downto 0);
SIGNAL tmat11_reg         : std_logic_vector(31 downto 0);
SIGNAL tmat12_reg         : std_logic_vector(31 downto 0);
SIGNAL tmat22_reg         : std_logic_vector(31 downto 0);
SIGNAL tmat2E_reg         : std_logic_vector(31 downto 0);
SIGNAL tmat_rdy           : std_logic;
SIGNAL tmat00_out         : std_logic_vector(31 downto 0);
SIGNAL tmat01_out         : std_logic_vector(31 downto 0);
SIGNAL tmat11_out         : std_logic_vector(31 downto 0);
SIGNAL tmat12_out         : std_logic_vector(31 downto 0);
SIGNAL tmat22_out         : std_logic_vector(31 downto 0);
SIGNAL tmat2E_out         : std_logic_vector(31 downto 0);
SIGNAL input_rdy          : std_logic;
SIGNAL senone_ram_addr1   : std_logic_vector(12 downto 0);
SIGNAL senone_ram_addr2   : std_logic_vector(12 downto 0);
SIGNAL senone_ram_addr3   : std_logic_vector(12 downto 0);
SIGNAL senone_ram_data1   : std_logic_vector(31 downto 0);
SIGNAL senone_ram_data2   : std_logic_vector(31 downto 0);
SIGNAL senone_ram_data3   : std_logic_vector(31 downto 0);
SIGNAL senone_ram_wr1     : std_logic;
SIGNAL senone_ram_wr2     : std_logic;
SIGNAL senone_ram_wr3     : std_logic;
SIGNAL sen_ram_data_out1  : std_logic_vector(31 downto 0);
SIGNAL sen_ram_data_out2  : std_logic_vector(31 downto 0);
SIGNAL sen_ram_data_out3  : std_logic_vector(31 downto 0);
SIGNAL HMM_done           : std_logic;
SIGNAL exit_reg           : std_logic_vector(31 downto 0);
SIGNAL valid_reg          : std_logic_vector(31 downto 0);
SIGNAL word_reg           : std_logic_vector(31 downto 0);
SIGNAL new_ph_id          : std_logic;
SIGNAL sen_addr1          : std_logic_vector(10 downto 0);
SIGNAL sen_addr2          : std_logic_vector(10 downto 0);
SIGNAL sen_addr3          : std_logic_vector(10 downto 0);
SIGNAL sen_addr_rdy       : std_logic;



begin


process(clk,ce,sclr)
begin
        if(clk =  '1' and clk'event)then
                if(sclr = '1')then
                        ptr_ram_data_tmp<=(others=>'0');
                        ptr_ram_data_rdy<='0';
```

```
			senone_addr_rdy<='0';
			senone_data_rdy1<='0';
			H0<=(OTHERS=>'0');
			H1<=(OTHERS=>'0');
			H2<=(OTHERS=>'0');
			INPROB<=(OTHERS=>'0');
			WRD_END<='0';
			PH_ID<=(OTHERS=>'0');
			tmat00_reg<=(OTHERS=>'0');
			tmat01_reg<=(OTHERS=>'0');
			tmat11_reg<=(OTHERS=>'0');
			tmat12_reg<=(OTHERS=>'0');
			tmat22_reg<=(OTHERS=>'0');
			tmat2E_reg<=(OTHERS=>'0');
			tmat_rdy_reg<='0';
			input_rdy<='0';
			hmm_rdy<='0';
		elsif(ce = '1')then
			if(new_input_data = '1')then
				INPROB<=ptr_ram_data_in(251 downto 220);
				H0<=ptr_ram_data_in(219 downto 188);
				H1<=ptr_ram_data_in(187 downto 156);
				H2<=ptr_ram_data_in(155 downto 124);
				PH_ID<=ptr_ram_data_in(24 downto 12);
				WRD_END<=ptr_ram_data_in(0);
				new_ph_id<=new_input_data;
			else
				new_ph_id<='0';
			end if;
			senone_data_rdy1<=sen_addr_rdy;
			tmat00_reg<=tmat00_out;
```

```
				tmat01_reg<=tmat01_out;
				tmat11_reg<=tmat11_out;
				tmat12_reg<=tmat12_out;
				tmat22_reg<=tmat22_out;
				tmat2E_reg<=tmat2E_out;
				input_rdy<=senone_data_rdy1 AND tmat_rdy;
				ptr_ram_data_rdy<=hmm_done;
				ptr_ram_data_tmp(192 downto 160)<=InPrb_out;
				ptr_ram_data_tmp(159 downto 128)<=H0_out;
				ptr_ram_data_tmp(127 downto 96)<=H1_out;
				ptr_ram_data_tmp(95 downto 64)<=H2_out;
				ptr_ram_data_tmp(63 downto 32)<=HMMexit;
				ptr_ram_data_tmp(31 downto 0)<=HMMbest;
				ptr_ram_data_out<=ptr_ram_data_tmp;
				hmm_rdy<=ptr_ram_data_rdy;
				word_beam<=word_reg;
				valid_beam<=valid_reg;
				exit_beam<=exit_reg;
			else
				hmm_rdy<='0';
			end if;
		end if;
end process;

process(senone_mux_sel,clk,sclr)
begin
		if(clk = '1' and clk'event)then
			if(sclr = '1')then
				senone_ram_addr1<=(others=>'0');
				senone_ram_addr2<=(others=>'0');
				senone_ram_addr3<=(others=>'0');
```

```
            senone_ram_data1<=(others=>'0');
            senone_ram_data2<=(others=>'0');
            senone_ram_data3<=(others=>'0');
            senone_ram_wr1<='0';
            senone_ram_wr2<='0';
            senone_ram_wr3<='0';
    elsif(senone_mux_sel = '0')then
            senone_ram_addr1<=senone_wr_addr;
            senone_ram_wr1<=senone_wr_en;
            senone_ram_data1<=senone_data_in;
            senone_ram_addr2<=senone_wr_addr;
            senone_ram_wr2<=senone_wr_en;
            senone_ram_data2<=senone_data_in;
            senone_ram_addr3<=senone_wr_addr;
            senone_ram_wr3<=senone_wr_en;
            senone_ram_data3<=senone_data_in;
    elsif(senone_mux_sel = '1')then
            senone_ram_addr1<="00" & sen_addr1;
            senone_ram_addr2<="00" & sen_addr2;
            senone_ram_addr3<="00" & sen_addr3;
            senone_ram_data1<=(others=>'0');
            senone_ram_data2<=(others=>'0');
            senone_ram_data3<=(others=>'0');
            senone_ram_wr1<='0';
            senone_ram_wr2<='0';
            senone_ram_wr3<='0';
    else
            senone_ram_addr1<=(OTHERS=>'0');
            senone_ram_addr2<=(OTHERS=>'0');
            senone_ram_addr3<=(OTHERS=>'0');
            senone_ram_data1<=(OTHERS=>'0');
```

```
                    senone_ram_data2<=(OTHERS=>'0');
                    senone_ram_data3<=(OTHERS=>'0');
                    senone_ram_wr1<='0';
                    senone_ram_wr2<='0';
                    senone_ram_wr3<='0';
                end if;
        end if;
end process;
hmm_pipe : hmm_calc
        PORT MAP(
                        clk        => clk,
                        ce         => ce,
                        sclr       => sclr,
                        new_frame  => new_frame,
                        H0_prev    => H0,
                H1_prev    => H1,
                    H2_prev    => H2,
                Input_Prob => INPROB,
                        word_end   => WRD_END,
                        inputs_rdy => input_rdy,
                  senone_in0 => sen_ram_data_out1,
                    senone_in1 => sen_ram_data_out2,
                senone_in2 => sen_ram_data_out3,
                tmat00     => tmat00_reg,
                tmat01     => tmat01_reg,
                tmat11     => tmat11_reg,
                tmat12     => tmat12_reg,
                tmat22     => tmat22_reg,
                tmat2E     => tmat2E_reg,
                        InProb_out => InPrb_out,
                H0Out      => H0_out,
                    H1Out      => H1_out,
                    H2Out      => H2_out,
                    HMM_best   => HMMbest,
                    HMM_exit   => HMMexit,
                                        exit_beam  => exit_reg,
                                        valid_beam => valid_reg,
                                        word_beam  => word_reg,
                              data_valid => hmm_done
                                        );
mdef : mdef_rom
          PORT MAP(
                                        clk          => clk,
                                        enable       => ce,
                                        reset        => sclr,
                                        PH_ID        => PH_ID,
                                        new_id       => new_ph_id,
                                        senone_addr1 => sen_addr1,
                    senone_addr2 => sen_addr2,
                    senone_addr3 => sen_addr3,
                                        sen_addr_rdy => sen_addr_rdy,
                    tmat00      => tmat00_out,
                    tmat01      => tmat01_out,
                    tmat11      => tmat11_out,
                    tmat12      => tmat12_out,
                    tmat22      => tmat22_out,
                    tmat2E      => tmat2E_out,
                                        tmat_scr_rdy => tmat_rdy
                                        );
sen_ram1 : raw_senone_ram
          PORT MAP(
                                        clk   => clk,
                                        we                  => senone_ram_wr1,
                                        din       => senone_ram_data1,
                                        addr      => senone_ram_addr1,
                                        dout      => sen_ram_data_out1
                                        );
sen_ram2 : raw_senone_ram
        PORT MAP(
                                        clk   => clk,
                                        we                  => senone_ram_wr2,
                                        din       => senone_ram_data2,
                                        addr      => senone_ram_addr2,
                                        dout      => sen_ram_data_out2
                                        );
sen_ram3 : raw_senone_ram
        PORT MAP(
                                        clk   => clk,
                                        we                  => senone_ram_wr3,
                                        din       => senone_ram_data3,
                                        addr      => senone_ram_addr3,
                                        dout      => sen_ram_data_out3
                                        );

end Behavioral;
```

B.10 – hmm_calc_vNEW.vhd

```
-- VHDL Entity Phone_lib.hmm_calc.symbol
--
-- Created:
--          by - Jeffrey W. Schuster
--          on - 11-12-05
```

```
--
-- Generated by Mentor Graphics' HDL Designer(TM) 2003.1 (Build 399)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.STD_LOGIC_SIGNED.all;


ENTITY hmm_calc IS
  PORT(
    H0_prev    : IN     std_logic_vector (31 DOWNTO 0);
    H1_prev    : IN     std_logic_vector (31 DOWNTO 0);
    H2_prev    : IN     std_logic_vector (31 DOWNTO 0);
    Input_Prob : IN     std_logic_vector (31 DOWNTO 0);
    clk        : IN     std_logic;
    ce         : IN     std_logic;
    sclr       : IN     std_logic;
                word_end   : IN     std_logic;
                inputs_rdy : IN     std_logic;
                new_frame  : IN     std_logic;
    senone_in0 : IN     std_logic_vector (31 DOWNTO 0);
    senone_in1 : IN     std_logic_vector (31 DOWNTO 0);
    senone_in2 : IN     std_logic_vector (31 DOWNTO 0);
    tmat00     : IN     std_logic_vector (31 DOWNTO 0);
    tmat01     : IN     std_logic_vector (31 DOWNTO 0);
    tmat11     : IN     std_logic_vector (31 DOWNTO 0);
    tmat12     : IN     std_logic_vector (31 DOWNTO 0);
    tmat22     : IN     std_logic_vector (31 DOWNTO 0);
    tmat2E     : IN     std_logic_vector (31 DOWNTO 0);
                InProb_out : OUT    std_logic_vector (32 downto 0);
    H0Out      : OUT    std_logic_vector (31 DOWNTO 0);
    H1Out      : OUT    std_logic_vector (31 DOWNTO 0);
    H2Out      : OUT    std_logic_vector (31 DOWNTO 0);
    HMM_best   : OUT    std_logic_vector (31 DOWNTO 0);
    HMM_exit   : OUT    std_logic_vector (31 DOWNTO 0);
                exit_beam  : OUT    std_logic_vector (31 downto 0);
                valid_beam : OUT    std_logic_vector (31 downto 0);
                word_beam  : OUT    std_logic_vector (31 downto 0);
    data_valid : OUT    std_logic
  );

-- Declarations

END hmm_calc ;

ARCHITECTURE struct OF hmm_calc IS

   -- Architecture declarations

   -- Internal signal declarations
        SIGNAL comp1     : std_logic_vector(31 downto 0);
        SIGNAL comp2     : std_logic_vector(31 downto 0);
        SIGNAL comp3     : std_logic_vector(31 downto 0);
        SIGNAL comp4     : std_logic_vector(31 downto 0);
        SIGNAL comp5     : std_logic_vector(31 downto 0);
        SIGNAL comp6     : std_logic_vector(31 downto 0);
        SIGNAL state0    : std_logic_vector(31 downto 0);
        SIGNAL state1    : std_logic_vector(31 downto 0);
        SIGNAL state2    : std_logic_vector(31 downto 0);
        SIGNAL state0_reg    : std_logic_vector(31 downto 0);
        SIGNAL state1_reg    : std_logic_vector(31 downto 0);
        SIGNAL state2_reg    : std_logic_vector(31 downto 0);
        SIGNAL state0_reg1    : std_logic_vector(31 downto 0);
        SIGNAL state1_reg1    : std_logic_vector(31 downto 0);
        SIGNAL state2_reg1    : std_logic_vector(31 downto 0);
        SIGNAL senone0_reg    : std_logic_vector(31 downto 0);
        SIGNAL senone1_reg    : std_logic_vector(31 downto 0);
        SIGNAL senone2_reg    : std_logic_vector(31 downto 0);
        SIGNAL senone0_reg1   : std_logic_vector(31 downto 0);
        SIGNAL senone1_reg1   : std_logic_vector(31 downto 0);
        SIGNAL senone2_reg1   : std_logic_vector(31 downto 0);
        SIGNAL bcomp1         : std_logic_vector(31 downto 0);
        SIGNAL bcomp2         : std_logic_vector(31 downto 0);
        SIGNAL bcomp3         : std_logic_vector(31 downto 0);
        SIGNAL bstate         : std_logic_vector(31 downto 0);
        SIGNAL bstate1        : std_logic_vector(31 downto 0);
        SIGNAL tmat_e_reg     : std_logic_vector(31 downto 0);
        SIGNAL we_reg         : std_logic;
        SIGNAL tmat_e_reg1    : std_logic_vector(31 downto 0);
        SIGNAL we_reg1        : std_logic;
        SIGNAL tmat_e_reg2    : std_logic_vector(31 downto 0);
        SIGNAL we_reg2        : std_logic;
        SIGNAL we_reg3        : std_logic;
        SIGNAL we_reg4        : std_logic;
        SIGNAL ebeam          : std_logic_vector(31 downto 0);
        SIGNAL vbeam          : std_logic_vector(31 downto 0);
        SIGNAL wbeam          : std_logic_vector(31 downto 0);
        SIGNAL exit_scr_reg   : std_logic_vector(31 downto 0);
        SIGNAL exit_scr_reg1  : std_logic_vector(31 downto 0);
        SIGNAL comp_go        : std_logic;
        SIGNAL comp_rdy       : std_logic;
        SIGNAL comp1_rdy      : std_logic;
```

```
        SIGNAL comp2_rdy      : std_logic;
        SIGNAL beam_comp_go   : std_logic;
        SIGNAL H0out_reg      : std_logic_vector(31 downto 0);
        SIGNAL H1out_reg      : std_logic_vector(31 downto 0);
        SIGNAL H2out_reg      : std_logic_vector(31 downto 0);
        SIGNAL HMM_best_reg   : std_logic_vector(31 downto 0);
        SIGNAL HMM_exit_reg   : std_logic_vector(31 downto 0);
        SIGNAL HMM_rdy        : std_logic;


--      SIGNAL exit_beam : std_logic_vector(31 downto 0);
--      SIGNAL valid_beam: std_logic_vector(31 downto 0);
--      SIGNAL word_beam : std_logic_vector(31 downto 0);

        constant input_reset : std_logic_vector(32 downto 0) := (OTHERS =>'1');
--      constant    valid_offset:    std_logic_vector(31    downto    0)    :=
"00000000000000000000000000000010";
--      constant    exit_offset    :    std_logic_vector(31    downto    0)    :=
"00000000000000000000000000000011";
--      constant    word_offset    :    std_logic_vector(31    downto    0)    :=
"00000000000000000000000000000001";


  -- Component Declarations


BEGIN
  -- Architecture concurrent statements
  -- HDL Embedded Text Block 1 eb1
  -- eb1 1
process(clk,sclr,ce,word_end)
```

```
begin
        if(clk = '1' and clk'event)then
                if(sclr = '1')then
                        comp1<=(others=>'0');
                        comp2<=(others=>'0');
                        comp3<=(others=>'0');
                        comp4<=(others=>'0');
                        comp5<=(others=>'0');
                        comp6<=(others=>'0');
                        state0<=(others=>'0');
                        state1<=(others=>'0');
                        state2<=(others=>'0');
                        senone0_reg<=(others=>'0');
                        senone1_reg<=(others=>'0');
                        senone2_reg<=(others=>'0');
                        tmat_e_reg<=(others=>'0');
                        we_reg<='0';
                        senone0_reg1<=(others=>'0');
                        senone1_reg1<=(others=>'0');
                        senone2_reg1<=(others=>'0');
                        tmat_e_reg1<=(others=>'0');
                        we_reg1<='0';
                        bcomp1<=(others=>'0');
                        bcomp2<=(others=>'0');
                        bcomp3<=(others=>'0');
                        bstate<=(others=>'0');
                        bstate1<=(others=>'0');
                        state0_reg<=(others=>'0');
                        state1_reg<=(others=>'0');
                        state2_reg<=(others=>'0');
                        state0_reg1<=(others=>'0');
```

```
          state1_reg1<=(others=>'0');
          state2_reg1<=(others=>'0');
          tmat_e_reg2<=(others=>'0');
          we_reg2<='0';
          we_reg3<='0';
          we_reg4<='0';
          ebeam<=(others=>'0');
          vbeam<=(others=>'0');
          wbeam<=(others=>'0');
          exit_scr_reg<=(others=>'0');
          exit_scr_reg1<=(others=>'0');
          comp_go<='0';
          comp_rdy<='0';
          comp1_rdy<='0';
          comp2_rdy<='0';
          beam_comp_go<='0';
          H0out_reg<=(others=>'0');
          H1out_reg<=(others=>'0');
          H2out_reg<=(others=>'0');
          HMM_best_reg<=(others=>'0');
          HMM_exit_reg<=(others=>'0');
          HMM_rdy<='0';
          data_valid<='0';
     elsif(ce = '1')then
          if(new_frame = '1')then
               vbeam<=(others=>'0');
               ebeam<=(others=>'0');
               wbeam<=(others=>'0');
          end if;
          if(inputs_rdy = '1')then
               comp1<=Input_prob;
```

```
        comp2<=H0_prev+tmat00;
        comp3<=H0_prev+tmat01;
        comp4<=H1_prev+tmat11;
        comp5<=H1_prev+tmat12;
        comp6<=H2_prev+tmat22;
        tmat_e_reg<=tmat2e;
        senone0_reg<=senone_in0;
        senone1_reg<=senone_in1;
        senone2_reg<=senone_in2;
        we_reg<=word_end;
        comp_go<=inputs_rdy;
else
        comp_go<='0';
end if;
if(comp_go<='1')then
        if(comp1 > comp2)then
                bcomp1<=comp1;
        else
                bcomp1<=comp2;
        end if;
        if(comp3 > comp4)then
                bcomp2<=comp3;
        else
                bcomp2<=comp4;
        end if;
        if(comp5 > comp6)then
                bcomp3<=comp5;
        else
                bcomp3<=comp6;
        end if;
        tmat_e_reg1<=tmat_e_reg;
```

```
		we_reg1<=we_reg;
		senone0_reg1<=senone0_reg;
		senone1_reg1<=senone1_reg;
		senone2_reg1<=senone2_reg;
		comp_rdy<=comp_go;
	else
		comp_rdy<='0';
	end if;
	if(comp_rdy = '1')then
		state0<=bcomp1+senone0_reg1;
		state1<=bcomp2+senone1_reg1;
		state2<=bcomp3+senone2_reg1;
		tmat_e_reg2<=tmat_e_reg1;
		we_reg2<=we_reg1;
		comp1_rdy<=comp_rdy;
	else
		comp1_rdy<='0';
	end if;
	if(comp1_rdy = '1')then
		if(state0 > state1)then
			bstate1<=state0;
		else
			bstate1<=state1;
		end if;
		state0_reg<=state0;
		state1_reg<=state1;
		state2_reg<=state2;
		exit_scr_reg<= state2+tmat_e_reg2;
		we_reg3<=we_reg2;
		comp2_rdy<=comp1_rdy;
	else
```

```
        comp2_rdy<='0';
end if;
if (comp2_rdy = '1')then
        if(bstate1 > state2_reg)then
                bstate<=bstate1;
        else
                bstate<=state2_reg;
        end if;
        state0_reg1<=state0_reg;
        state1_reg1<=state1_reg;
        state2_reg1<=state2_reg;
        exit_scr_reg1<=exit_scr_reg;
        we_reg4<=we_reg3;
        beam_comp_go<=comp2_rdy;
else
        beam_comp_go<='0';
end if;
if(beam_comp_go = '1')then
        if(bstate > vbeam)then
                vbeam<=bstate;
        end if;
        if(exit_scr_reg > ebeam)then
                ebeam<=exit_scr_reg;
        end if;
        if(we_reg4 = '1')then
                if(bstate > wbeam)then
                        wbeam<=bstate;
                end if;
        end if;
        H0out_reg<=state0_reg1;
        H1out_reg<=state1_reg1;
```

```
                    H2out_reg<=state2_reg1;
                    HMM_exit_reg<=exit_scr_reg1;
                    HMM_best_reg<=bstate;
                    HMM_rdy<=beam_comp_go;
                else
                    HMM_rdy<='0';
                end if;
                InProb_out<=input_reset;
                H0out<=H0out_reg;
                H1out<=H1out_reg;
                H2out<=H2out_reg;
                HMM_best<=HMM_best_reg;
                HMM_exit<=HMM_exit_reg;
                exit_beam<=ebeam-330254;
                valid_beam<=vbeam-507006;
                word_beam<=wbeam;
                data_valid<=HMM_rdy;
            else
            end if;
        end if;
end process;

-- Instance port mappings.
END struct;
```

B.11 – mdef_rom_struct.vhd

```
-- VHDL Entity Phone_lib.mdef_rom.symbol
--
-- Created:
--          by - Jeffrey W. Schuster
--         on - 11-11-05
--
-- Generated by Mentor Graphics' HDL Designer(TM) 2003.1 (Build 399)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;

ENTITY mdef_rom IS
  PORT(
    PH_ID       : IN     std_logic_VECTOR (12 DOWNTO 0);
                new_id       : IN     std_logic;
    clk         : IN     std_logic;
    enable      : IN     std_logic;
    reset       : IN     std_logic;
    senone_addr1 : OUT    std_logic_vector (10 DOWNTO 0);
    senone_addr2 : OUT    std_logic_vector (10 DOWNTO 0);
    senone_addr3 : OUT    std_logic_vector (10 DOWNTO 0);
                sen_addr_rdy : OUT    std_logic;
    tmat00      : OUT    std_logic_vector (31 DOWNTO 0);
    tmat01      : OUT    std_logic_vector (31 DOWNTO 0);
    tmat11      : OUT    std_logic_vector (31 DOWNTO 0);
    tmat12      : OUT    std_logic_vector (31 DOWNTO 0);
    tmat22      : OUT    std_logic_vector (31 DOWNTO 0);
    tmat2E      : OUT    std_logic_vector (31 DOWNTO 0);
                tmat_scr_rdy : OUT    std_logic
  );

-- Declarations
```

```
END mdef_rom ;


ARCHITECTURE struct OF mdef_rom IS

   -- Architecture declarations

   -- Internal signal declarations
   SIGNAL SPO   : std_logic_VECTOR(31 DOWNTO 0);
   SIGNAL SPO1  : std_logic_VECTOR(31 DOWNTO 0);
   SIGNAL SPO2  : std_logic_VECTOR(31 DOWNTO 0);
   SIGNAL SPO3  : std_logic_VECTOR(31 DOWNTO 0);
   SIGNAL SPO4  : std_logic_VECTOR(31 DOWNTO 0);
   SIGNAL SPO5  : std_logic_VECTOR(31 DOWNTO 0);
   SIGNAL dout0  : std_logic_VECTOR(5 DOWNTO 0);
   SIGNAL dout1 : std_logic_VECTOR(10 DOWNTO 0);
   SIGNAL dout2 : std_logic_VECTOR(10 DOWNTO 0);
   SIGNAL dout3 : std_logic_VECTOR(10 DOWNTO 0);
         SIGNAL new_sen_addr : std_logic;
         SIGNAL new_tmat_scr1: std_logic;
         SIGNAL new_tmat_scr2: std_logic;

         attribute syn_ramstyle : string;
         attribute syn_ramstyle of dout0 : signal is "block_ram";
         attribute syn_ramstyle of dout1 : signal is "block_ram";
         attribute syn_ramstyle of dout2 : signal is "block_ram";
         attribute syn_ramstyle of dout3 : signal is "block_ram";
         attribute syn_ramstyle of SPO  : signal is "block_ram";
         attribute syn_ramstyle of SPO1 : signal is "block_ram";
         attribute syn_ramstyle of SPO2 : signal is "block_ram";
         attribute syn_ramstyle of SPO3 : signal is "block_ram";
```

```
        attribute syn_ramstyle of SPO4 : signal is "block_ram";
        attribute syn_ramstyle of SPO5 : signal is "block_ram";

-- Component Declarations


COMPONENT sen1_addr_rom
PORT (
  addr : IN     std_logic_VECTOR (12 DOWNTO 0);
  clk  : IN     std_logic;
  dout : OUT    std_logic_VECTOR (10 DOWNTO 0)
);
END COMPONENT;
COMPONENT sen2_addr_rom
PORT (
  addr : IN     std_logic_VECTOR (12 DOWNTO 0);
  clk  : IN     std_logic;
  dout : OUT    std_logic_VECTOR (10 DOWNTO 0)
);
END COMPONENT;
COMPONENT sen3_addr_rom
PORT (
  addr : IN     std_logic_VECTOR (12 DOWNTO 0);
  clk  : IN     std_logic;
  dout : OUT    std_logic_VECTOR (10 DOWNTO 0)
);
END COMPONENT;
COMPONENT t0_hmm_1_rom
PORT (
  addr   : IN     std_logic_VECTOR (5 DOWNTO 0);
        clk      : IN     std_logic;
  dout : OUT    std_logic_VECTOR (31 DOWNTO 0)
);
END COMPONENT;
COMPONENT t0_hmm_2_rom
PORT (
  addr   : IN     std_logic_VECTOR (5 DOWNTO 0);
        clk      : IN     std_logic;
  dout : OUT    std_logic_VECTOR (31 DOWNTO 0)
);
END COMPONENT;
COMPONENT t1_hmm_0_rom
PORT (
  addr   : IN     std_logic_VECTOR (5 DOWNTO 0);
        clk      : IN     std_logic;
  dout : OUT    std_logic_VECTOR (31 DOWNTO 0)
);
END COMPONENT;
COMPONENT t1_hmm_1_rom
PORT (
  addr   : IN     std_logic_VECTOR (5 DOWNTO 0);
        clk      : IN     std_logic;
  dout : OUT    std_logic_VECTOR (31 DOWNTO 0)
);
END COMPONENT;
COMPONENT t1_hmm_2_rom
PORT (
  addr   : IN     std_logic_VECTOR (5 DOWNTO 0);
        clk      : IN     std_logic;
  dout : OUT    std_logic_VECTOR (31 DOWNTO 0)
);
END COMPONENT;
```

```
COMPONENT t_exit_rom
PORT (
  addr   : IN     std_logic_VECTOR (5 DOWNTO 0);
        clk      : IN     std_logic;
  dout : OUT    std_logic_VECTOR (31 DOWNTO 0)
);
END COMPONENT;
COMPONENT tmat_id_rom
PORT (
  addr : IN     std_logic_VECTOR (12 DOWNTO 0);
  clk  : IN     std_logic;
  dout : OUT    std_logic_VECTOR (5 DOWNTO 0)
);
END COMPONENT;

-- Optional embedded configurations
-- pragma synthesis_off


FOR ALL : sen1_addr_rom USE ENTITY work.sen1_addr_rom;
FOR ALL : sen2_addr_rom USE ENTITY work.sen2_addr_rom;
FOR ALL : sen3_addr_rom USE ENTITY work.sen3_addr_rom;
FOR ALL : t0_hmm_1_rom USE ENTITY work.t0_hmm_1_rom;
FOR ALL : t0_hmm_2_rom USE ENTITY work.t0_hmm_2_rom;
FOR ALL : t1_hmm_0_rom USE ENTITY work.t1_hmm_0_rom;
FOR ALL : t1_hmm_1_rom USE ENTITY work.t1_hmm_1_rom;
FOR ALL : t1_hmm_2_rom USE ENTITY work.t1_hmm_2_rom;
FOR ALL : t_exit_rom USE ENTITY work.t_exit_rom;
FOR ALL : tmat_id_rom USE ENTITY work.tmat_id_rom;
-- pragma synthesis_on
```

```
BEGIN
  -- Architecture concurrent statements

  process(clk,reset,enable)
    begin
      if (clk = '1' and clk'event)then
                    if(reset = '1')then
                              tmat00<=(OTHERS=>'0');
                              tmat01<=(OTHERS=>'0');
                              tmat11<=(OTHERS=>'0');
                              tmat12<=(OTHERS=>'0');
                              tmat22<=(OTHERS=>'0');
                              tmat2e<=(OTHERS=>'0');
                              senone_addr1<="00000000000";
          senone_addr2<="00000000000";
          senone_addr3<="00000000000";
                              new_sen_addr<='0';
                              new_tmat_scr1<='0';
                              new_tmat_scr2<='0';
                    elsif(enable ='1')then
                              tmat00<=SPO;
                              tmat01<=SPO1;
                              tmat11<=SPO2;
                              tmat12<=SPO3;
                              tmat22<=SPO4;
                              tmat2e<=SPO5;
          senone_addr1<=dout1;
          senone_addr2<=dout2;
          senone_addr3<=dout3;
                   if(new_id = '1')then
```

```
                              new_sen_addr<=new_id;
                                        new_tmat_scr1<=new_id;
                                   else
                                        new_sen_addr<='0';
                                        new_tmat_scr1<='0';
                                   end if;
                                   sen_addr_rdy<=new_sen_addr;
                                   new_tmat_scr2<=new_tmat_scr1;
                                   tmat_scr_rdy<=new_tmat_scr2;
                              else
                                   sen_addr_rdy<='0';
                                   tmat_scr_rdy<='0';
                              end if;
   end if;
end process;

 -- Instance port mappings.
I0 : tmat_id_rom
  PORT MAP (
    addr => PH_ID,
    clk  => clk,
    dout => dout0
  );
I1 : sen1_addr_rom
  PORT MAP (
    addr => PH_ID,
    clk  => clk,
    dout => dout1
  );
I2 : sen2_addr_rom
  PORT MAP (
    addr => PH_ID,
    clk  => clk,
    dout => dout2
  );
I3 : sen3_addr_rom
  PORT MAP (
    addr => PH_ID,
    clk  => clk,
    dout => dout3
  );
I4 : t0_hmm_1_rom
  PORT MAP (
    addr   => dout0,
    dout => SPO1,
        clk =>clk
  );
I5 : t0_hmm_2_rom
  PORT MAP (
    addr   => dout0,
    dout => SPO3,
        clk => clk
  );
I6 : t1_hmm_0_rom
  PORT MAP (
    addr   => dout0,
    dout => SPO,
        clk => clk
  );
I7 : t1_hmm_1_rom
  PORT MAP (
    addr   => dout0,
        dout => SPO2,
            clk => clk
      );
   I8 : t1_hmm_2_rom
      PORT MAP (
        addr   => dout0,
        dout => SPO4,
            clk => clk
      );
   I9 : t_exit_rom
      PORT MAP (
        addr   => dout0,
        dout => SPO5,
            clk => clk
      );


END struct;
```

B.12 – ph_ptr_ram_NEW_struct.vhd

```
-- VHDL Entity PH_PTR_RAM_V2
--
-- Created:
--          by - Jeffrey W. Schuster
--          at - 11:05:05 AM 1-16-06
--
--
--
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.STD_LOGIC_UNSIGNED.all;

ENTITY PH_PTR_RAM IS
  PORT(
            PH_WRD_SEL    : IN     std_logic;
    addr_rdy      : IN     std_logic;
    clk           : IN     std_logic;
    init          : IN     std_logic;
   phone_data    : IN     std_logic_vector (192 DOWNTO 0);
    phone_rd_addr : IN     std_logic_vector (8 DOWNTO 0);
    phone_wr_addr : IN     std_logic_vector (8 DOWNTO 0);
                  phone_wr                 : IN      std_logic;
  word_data     : IN     std_logic_vector (192 DOWNTO 0);
    word_rd_addr  : IN     std_logic_vector (8 DOWNTO 0);
    word_wr_addr  : IN     std_logic_vector (8 DOWNTO 0);
    word_wr       : IN     std_logic;
    phone_data_out: OUT    std_logic_vector (252 DOWNTO 0);
                  word_data_out : OUT    std_logic_vector (252 DOWNTO 0);
    new_data      : OUT    std_logic
  );

-- Declarations

END PH_PTR_RAM ;


ARCHITECTURE struct OF PH_PTR_RAM IS
```

```
        SIGNAL  data_rdy_tmp  : std_logic;
        SIGNAL  phone_addr    : std_logic_vector(8 downto 0);
        SIGNAL  word_addr     : std_logic_vector(8 downto 0);
        SIGNAL  rd_addr       : std_logic_vector(8 downto 0);
        SIGNAL  phone_data_tmp: std_logic_vector(192 downto 0);
        SIGNAL  word_data_tmp : std_logic_vector(192 downto 0);
        SIGNAL  data          : std_logic_vector(59 downto 0);
--      SIGNAL  data_rdy_tmp1 : std_logic;


  COMPONENT data_ram
  PORT (
    addra : IN     std_logic_VECTOR (8 DOWNTO 0);
    addrb : IN     std_logic_VECTOR (8 DOWNTO 0);
--              sinita: IN     std_logic;
--              sinitb: IN     std_logic;
    clka  : IN     std_logic;
    clkb  : IN     std_logic;
    dina  : IN     std_logic_VECTOR (192 DOWNTO 0);
                dinb  : IN     std_logic_vector (192 downto 0);
    wea   : IN     std_logic;
                web   : IN     std_logic;
                douta : OUT    std_logic_VECTOR (192 DOWNTO 0);
    doutb : OUT    std_logic_VECTOR (192 DOWNTO 0)
  );
  END COMPONENT;
  COMPONENT ptr_rom
  PORT (
    addr : IN     std_logic_VECTOR (8 DOWNTO 0);
    clk  : IN     std_logic;
    dout : OUT    std_logic_VECTOR (59 DOWNTO 0)
  );
  END COMPONENT;

  FOR ALL : data_ram USE ENTITY work.data_ram;
  FOR ALL : ptr_rom USE ENTITY work.ptr_rom;
  -- pragma synthesis_on


BEGIN


        process(init,clk)
        begin
        if(clk  = '1' and clk'event)then
                if(init = '1')then
                        data_rdy_tmp<='0';
--                      data_rdy_tmp1<='0';
                else
                        data_rdy_tmp<=addr_rdy;
--                      data_rdy_tmp<=data_rdy_tmp1;
                end if;
        new_data<=data_rdy_tmp;
        end if;
        end process;

        process(word_wr,clk,init)
        begin
--      CASE word_wr IS
--              WHEN '0'|'L' => word_addr<=word_rd_addr;
--              WHEN '1'|'H' => word_addr<=word_wr_addr;
--              WHEN OTHERS => word_addr <= (OTHERS => 'X');
--   END CASE;
        if(clk = '1' and clk'event)then
                if(init = '1' )then
                        word_addr<=(OTHERS=>'0');
                elsif(word_wr = '1')then
                        word_addr<=word_wr_addr;
                elsif(word_wr = '0')then
                        word_addr<=word_rd_addr;
                else
                        word_addr<=(OTHERS=>'0');
                end if;
        end if;

        end process;

        process(phone_wr,clk,init)
        begin
--      CASE phone_wr IS
--              WHEN '0'|'L' => phone_addr<=phone_rd_addr;
--              WHEN '1'|'H' => phone_addr<=phone_wr_addr;
--              WHEN OTHERS => phone_addr <= (OTHERS => 'X');
-- END CASE;
        if(clk = '1' and clk'event)then
                if(init = '1')then
                        phone_addr<=(OTHERS=>'0');
                elsif(phone_wr = '1')then
                        phone_addr<=phone_wr_addr;
                elsif(phone_wr = '0')then
                        phone_addr<=phone_rd_addr;
                else
                        phone_addr<=(OTHERS=>'0');
                end if;
        end if;
        end process;

        phone_data_out<=phone_data_tmp & data;
        word_data_out <=word_data_tmp  & data;

        process(PH_WRD_SEL,clk,init)
        begin
--      CASE PH_WRD_SEL IS
--              WHEN '0'|'L' => rd_addr<=phone_rd_addr;
--              WHEN '1'|'H' => rd_addr<=word_rd_addr;
--              WHEN OTHERS => rd_addr <= (OTHERS => 'X');
--  END CASE;
        if(clk = '1' and clk'event)then
                if(init = '1')then
                        rd_addr<=(OTHERS=>'0');
                elsif(PH_WRD_SEL = '1')then
                        rd_addr<=word_rd_addr;
                elsif(PH_WRD_SEL = '0')then
                        rd_addr<=phone_rd_addr;
                else
                        rd_addr<=(OTHERS=>'0');
                end if;
        end if;
        end process;

 -- Instance port mappings.
 I0 : data_ram
    PORT MAP (
      addra => phone_addr,
      addrb => word_addr,
--                        sinita=> init,
--                        sinitb=> init,
      clka  => clk,
      clkb  => clk,
      dina  => phone_data,
                          dinb  => word_data,
                          douta => phone_data_tmp,
      doutb => word_data_tmp,
      wea   => phone_wr,
                          web   => word_wr
    );
  I1 : ptr_rom
    PORT MAP (
      addr => rd_addr,
      clk  => clk,
      dout => data
    );

END struct;
```

B.13 – new_pruner.vhd

```
-- VHDL Entity prune_block
--
-- Created:
--        by - Jeffrey W. Schuster
--        at - 1:08PM 1-6-06
--
--
--
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.STD_LOGIC_UNSIGNED.all;

ENTITY prune_block IS
  PORT(
    clk          : IN     std_logic;
                  reset        : IN     std_logic;
    prune_enable : IN     std_logic;
--                end_of_phase : IN     std_logic;
    e_beam       : IN     std_logic_vector (31 DOWNTO 0);
                  v_beam       : IN     std_logic_vector (31 DOWNTO 0);
    ram_data_rdy : IN     std_logic;
    hmm_data     : IN     std_logic_vector (252 DOWNTO 0);
    data_ready   : OUT    std_logic;
    dead         : OUT    std_logic;
    pal          : OUT    std_logic;
    pal_val      : OUT    std_logic
--     prune_done   : OUT    std_logic
    );

-- Declarations

END prune_block ;



ARCHITECTURE struct OF prune_block IS


        SIGNAL data_rdy_tmp1   : std_logic;
        SIGNAL data_rdy_tmp2   : std_logic;
        SIGNAL prune_go        : std_logic;
        SIGNAL hmm_exit_tmp    : std_logic_vector(31 downto 0);
        SIGNAL hmm_best_tmp    : std_logic_vector(31 downto 0);
        SIGNAL e_beam_tmp      : std_logic_vector(31 downto 0);
        SIGNAL v_beam_tmp      : std_logic_vector(31 downto 0);
        SIGNAL e_chk           : std_logic;
  SIGNAL v_chk           : std_logic;
        SIGNAL prune_go_reg    : std_logic;
        SIGNAL prune_go_reg1   : std_logic;

BEGIN

        process(clk,reset,prune_enable,ram_data_rdy)
        begin
        if(clk = '1' and clk'event)then
                if(reset = '1')then
                        data_ready<='0';
                        data_rdy_tmp1<='0';
                        data_rdy_tmp2<='0';
                        prune_go<='0';
                        hmm_exit_tmp<=(OTHERS=>'0');
                        hmm_best_tmp<=(OTHERS=>'0');
                        e_beam_tmp<=(OTHERS=>'0');
                        v_beam_tmp<=(OTHERS=>'0');
                        prune_go_reg<='0';
                        prune_go_reg1<='0';
                elsif(prune_enable = '1')then
                        e_beam_tmp<=e_beam;
                        v_beam_tmp<=v_beam;
                        if(ram_data_rdy = '1')then
                                hmm_best_tmp<=hmm_data(91 downto 60);
                                hmm_exit_tmp<=hmm_data(123 downto 92);
                                prune_go<=ram_data_rdy;
                        else
                                prune_go<='0';
                        end if;
                        prune_go_reg<=prune_go;
                        prune_go_reg1<=prune_go_reg;
                        data_rdy_tmp1<=ram_data_rdy;
                        data_rdy_tmp2<=data_rdy_tmp1;
                        data_ready<=data_rdy_tmp2;
                else
                        prune_go_reg<='0';
                        prune_go_reg1<='0';
                end if;
        end if;
        end process;

         process(clk,prune_go_reg)

     begin
       if (clk = '1' and clk'event)then
                        if(prune_go_reg = '1')then
        if( hmm_best_tmp >= v_beam_tmp ) then
                v_chk <= '1';
        else
                v_chk <= '0';
        end if;
                                if( hmm_exit_tmp >= e_beam_tmp ) then
                e_chk <= '1';
        else
                e_chk <= '0';
```

```
        end if;
                        else
                                v_chk<='0';
                                e_chk<='0';
                        end if;
      end if;
    end process;


process(clk,reset,prune_go_reg1)
begin
        if(clk = '1' and clk'event)then
                if(reset = '1')then
                        pal<='0';
                        pal_val<='0';
                        dead<='0';
                elsif(prune_enable = '1')then
                        if(prune_go_reg1 = '1')then
                                dead<=not(e_chk) AND not(v_chk);
                                pal<=v_chk OR e_chk;
                                pal_val<= e_chk;
                        else
                                pal<='0';
                                pal_val<='0';
                                dead<='0';
                        end if;
                elsif(prune_enable = '0')then
                        pal<='0';
                        pal_val<='0';
                        dead<='0';
                else
```

```
                        pal<='0';
                        pal_val<='0';
                        dead<='0';
                end if;
        end if;
  end process;

END struct;
```

# BIBLIOGRAPHY

[1] M. Ravishankar, *et al*, “The 1999 CMU 10X Real Time Broadcast News Transcription System”, *Proc. DARPA Workshop on Automatic Transcription of Broadcast News*, Washington DC, May 2000

[2] F. Jelinek, Statistical Methods for Speech Recognition, Massachusetts: The MIT Press, 2001.

[3] C. Lai, S.-L. Lu, & Q. Zhao, “Performance Analysis of Speech Recognition Software”, *Proc. Fifth Workshop on Computer Architecture Evaluation using Commercial Workloads*, Feb. 2002.

[4] M.J.F.Gales, K.M.Knill, and S.J.Young, “Use of State Based Gaussian Selection in Large Vocabulary Continuous Speech Recognition using HMMs,” *IEEE Trans. on Speech and Audio Processing*, vol. 7: 152-161, 1999.

[5] A. Davis, Z. Fang, & B. K. Mathew, “A Gaussian Probability Accelerator for SPHINX 3”, University of Utah #UUCS-03-02, Nov 18, 2002.

[6] K.K. Agaram, S.W. Keckler, D. Burger, “Characterizing the SPHINX Speech Recognition System”, University of Texas at Austin, Department of Computer Sciences, Technical Report TR2001-18, January 2001.

[7] A. Sankar, *et al,* “Development of SRI’s 1997 Broadcast News Transcription System”, Speech Technology & Research Laboratory, SRI International, 1998.

[8] S. Chen, *et al,* “IBM’s LVCSR System for Transcription of Broadcast News used in the 1997 HUB4 English Evaluation”, *Proc. DARPA Speech Research Workshop,* Lansdowne, VA, Feb. 1998.

[9] F. Kubala, *et al,* “The 1996 BBN Byblos HUB-4 Transcription System”, *Proc. DARPA 1997 Speech Recognition Workshop”*, Feb. 1997.

[10] C. Becchetti, L.P. Ricotti, Speech Recognition – Theory & C++ Implemtation, New York: John Wiley & Sons, 1999.

[11] E.L. Bocchieri, & J.G. Wilpon, “Discriminative Feature Selection for Speech Recognition”, *Computer Speech and Language 7, pages 229-246.* AT&T Bell Laboratories, 1993.

[12] L. Rabiner, B.H. Juang, Fundamentals of Speech Recognition, New Jersey: Prentice Hall Signal Processing Series, 1993.

[13] J.M. Huerta & R.M. Stern, “Speech Recognition from GSM Codec Parameters”, Carnegie Mellon University, School of Computer Science. *Proc. ICSLP 1998.*

[14] O. Deshmukh, *et al,* “Acoustic-Phonetic Speech Parameters for Speaker-Independent Speech Recognition”, University of Maryland, Department of Electrical & Computer Engineering, 1998.

[15] M.J. Hunt, “Spectral Signal Processing for ASR”, Dragon Systems UK Research and Development Ltd., 2000.

[16] M. Ravishankar, *Efficient Algorithms for Speech Recognition*, Ph.D. Thesis, Carnegie Mellon University, May 1996, Tech Report CMU-CS-96-143.

[17] K.F. Lee, *et al.*, “An Overview of the SPHINX Speech Recognition System”, *IEEE Transactions on Acoustics, Speech, and Signal Processing,* Vol 38, No. 1, January 1990.

[18] K.K. Agarm, S.W. Keckler, D. Burger, “A Characterization of Speech Recognition on Modern Computer Systems”, University of Texas at Austin, Department of Computer Sciences, 2001.

[19] H. P. Combrinck & E. C. Botha, “On the Mel-scaled Cepstrum”, University of Pretoria, Department of Electrical and Electronic Engineering, 1996.

[20] E.L. Bocchieri, “Vector Quantization for the Efficient Computation of Continuous Density Likelihoods,” *Proc. IEEE Int. Conf. Acoust., Speech & Sig. Processing*, April 1993, pp. 692-695.

[21] J. Fonollosa, “Fast Vector Quantization based on Subcodebook Selection and its Applications to Speech Processing”, TALP Research Center, Unversidad Politenica de Cataluna.

[22] L. Kai-Fu, Automatic Speech Recognition : The Development of the SPHINX System, Massachusetts, Kluwer Academic Publishing, 1989.

[23] D.B. Paul. “An Investigation of Gaussian Shortlists”, *Proc. IEEE workshop on Automatic Speech Recognition and Understanding,* 1999.

[24] K. Knill, M. Gales, & S. Young, “Use of Gaussian Selection in Large Vocabulary Continuous Speech Recognition using HMMs”, *Proc. ICLSP 96*, pp. 470-473, Philadelphia PA, Oct. 1996.

[25] M. Ravishankar, R. Bisiani, and E. Thayer, “Sub-vector Clustering to Improve Memory and Speed Performance of Acoustic Likelihood Computation,” *Proc. Eurospeech*, 1997.

[26] J. Nouza, “Feature Selection Methods for Hidden Markov Model-Based Speech Recognition,” *Proc. International. Conference on Pattern Recognition*, 1996, vol. 2, pp. 186-190.

[27] X. Li & J. Blimes, “Feature Pruning in Likelihood Evaluation of HMM-Based Speech Recognition”, Unversity of Washington, 2003.

[28] X. Huang, F. Alleva, *et al.*, “An Overview of the SPHINX-II Speech Recognition System”, Carnegie Mellon University, School of Computer Science, 1994.

[29] J. Pylkkonen, “An Efficient One-Pass Decoder for Finnish Large Vocabulary Continuous Speech Recognition”, Helsinki University of Technology, Neural Networks Research Centre, 2004

[30] X. Yifang, Z. Jinjie, *et al.*, “Robust Recognition of Noisy Speech Using Speech Enhancement”, Tsinghua University, Department of Electronic Engineering.

[31] S. Kim, S. Nedevschi, & R. K. Patra, “Hardware Speech Recognition in Low Cost, Low Power Devices”, University of California at Berkley, Computer Science Division, 2003.

[32] X. F. Guo, *et, al.*, “IBM’s LVCSR System for Transcription of Broadcast News used in the 1997 HUB4 Mandarin Chinese Evaluation”, *Proc. of 1998 Broadcast News Transcription and Understanding Workshop*, 1998.

[33] L. Nguen, S. Matsoukas, *et al.*, “ The 1999 BBN BYBLOS 10xRT Broadcast News Transcription System”, BBN Technologies, 2000.

[34] R. Schwartz, *et al.*,” The BBN BYBLOS Continuous Speech Recognition System”, BBN Technologies, 1996.

[35] S. Colbath & F. Kubala, “ Rough’n’Ready: A Meeting Recorder and Browser”, BBN Technologies, 1998.

[36] A. Sankar, “Experiments with a Gaussian Merging-Splitting Algorithm for HMM training for Speech Recognition”, SRI International, 1997.

[37] P. Placeway, *et al.*, “The 1996 HUB-4 SPHINX-3 System”, Carnegie Mellon University, School of Computer Science, 1997.

[38] K. Seymore, *et al.*, “The 1997 CMU SPHINX-3 English Broadcast News Transcription System”, Carnegie Mellon University, School of Computer Science, 1998.

[39] K. Manley, “Pathfinding: From A* to LPA”, University of Minnesota, Computer Science Seminar, Spring 2003.

[40] P. Lamere, *et al.*, “Design of the SPHINX-4 Decoder”, Carnegie Mellon University, School of Computer Science, 2004.

[41] R. Singh, *et al.*, "Classification with Free Energy at Raised Temperatures", in *EUROSPEECH 2003*, 2003.

[42] Sensory Inc., RSC-4128 Preliminary Product Brief 80-0225-B, 2002.

[43] Sensory Inc., RSC-4128 Speech Recognition Processor Data Sheet, P/N 80-0206-M, 2005.

[44] S. Kim, *et al.*, "Hardware Speech Recognition for User Interfaces in Low Cost, Low Power Devices", University of California at Berkley, Computer Sciences Division, 2003.

[45] S. Kim, *et al.*, "Hardware Speech Recognition in Low Cost, Low Power Devices", University of California at Berkley, Computer Sciences Division, 2003.

[46] S. J. Melinkoff, S. F. Quigley, & M. J. Russell, "Speech Recognition on an FPGA Using Discrete and Continuous Hidden Markov Models", University of Birmingham, Department of Electronic, Electrical and Computer Engineering, 2001.

[47] S. J. Melinkoff, S. F. Quigley, & M. J. Russell, "Implementing a Simple Continuous Speech Recognition System on an FPGA", *Proc. FCCM'02*, 2002.

[48] S. J. Melinkoff, S. F. Quigley, & M. J. Russell, "Performing Speech Recognition on Multiple Parallel Files Using Continuous Hidden Markov Models on an FPGA", University of Birmingham, Department of Electronic, Electrical and Computer Engineering, 2002.

[49] J. L. Hennessy & D. A. Patterson, Computer Architecture A Quantitative Approach, San Francisco, Morgan Kaufmann Publishers, 1990.

[50] J. G. Holm, "Evaluation of some Superscalar and VLIW Processor Designs", M.S. Thesis, University of Illinois at Urbana-Champaign, 1992.

[51] R. Hoare, *et al.*, "An FPGA-Based VLIW Processor with Custom Hardware Execution", *Proc. FPGA '05*, 2005.

[52] J. H. Stokes, "SIMD Architectures", Tuesday March 21, 2000. http://arstechnica.com/articles/paedia/cpu/simd.ars

[53] N. Blachford, "Cell Architecture Explained", 2005. http://www.blachford.info/computer/Cells/Cell0.html

[54] "Stretch's Software-Configurable Processor Embeds Programmable Logic Within the Processor to Accelerate Performance, Cut Development Time", April 26, 2004. http://www.analogzone.com/netp0426a.htm

[55] Stretch Inc., S5000 Technical Overview, http://www.stretchinc.com/products_overview.php

[56] J. L. Devore, Probability and Statistics for Engineering and the Sciences, Pacific Grove, CA: Duxbury, 2000.

[57] T. Schultz, Course notes from CMU LTI 11-751

[58] T. H. Cormen, C. E. Leiserson, R. L. Rivest, Introduction to Algorithms, The MIT Press, Cambridge, MA, 1990.

[59] R. Hoare, K. Gupta, J. Schuster, **"Speech Silicon: A data-driven SoC for Performing Hidden Markov Model based Speech Recognition,"** *Proc. HPEC 2005,* Lincoln Labs, MIT, Boston, MA. 2005.

[60] R. Hoare, J. Schuster, K. Gupta, "Speech Silicon: A Real-Time Continuous Speech Recognition SoC on a 90nm FPGA," *Proc. DAC 2005*, Anaheim, CA. 2005.

[61] J. Schuster, R. Hoare, K. Gupta, "A Hardware Based Acoustic Modeling Pipeline for Hidden Markov Model based Speech Recognition," *Proc. RAW 2006*, Rhodes Island, Greece. 2006

[62] J. Schuster, R. Hoare, K. Gupta, A. Jones , **"FPGA Implementation of a Real-Time Hidden Markov Model Based Speech Processing System,"** *Eurasip Journal on Embedded Systems.* Publication Pending.

[63] B. Mathew, A. Davis, Z. Fang, "A Low-Power Accelerator for the SPHINX 3 Speech Recognition System," *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, San Jose, CA, 2003.

[64] CMU Sphinx, http://cmusphinx.sourceforge.net/html/cmusphinx.php

[65] Linguistic Data Consortium, http://www.ldc.upenn.edu/

[66] S.J. Young, N.H. Russel, J.H.S. Thornton, "Token Passing: a Simple Conceptual Model for Connected Speech Recognition Systems", Cambridge University Engineering Department, Cambridge, U.K., July 1989.

# The CMU Sphinx Group Open Source Speech Recognition Engines

Speech at CMU | Sphinx at SourceForge

Introduction

**General Documentation**

- Sphinx tutorial
- Sphinx version comparison
- Resources to build a system
- Sphinx manual
- See also some external links

**Detailed Documentation**

- PocketSphinx
- Sphinx-2
- Sphinx-3
- Sphinx-4
- SphinxTrain

**Developers' Notes**

- Sphinx-3 TWiki
- Sphinx-4 TWiki

Download

## Welcome to the CMU Sphinx project page!

The Sphinx Group at Carnegie Mellon University is committed to releasing the long-time, DARPA-funded Sphinx projects widely, in order to stimulate the creation of speech-using tools and applications, and to advance the state of the art both directly in speech recognition, as well as in related areas including dialog systems and speech synthesis.

The Sphinx Group has been supported for many years by funding from the Defense Advanced Research Projects Agency, and the recognition engines to be released are those that the group used for the various DARPA projects and their respective evaluations.

Recent support for the project also include Telefónica I & D, Sun Microsystems, and Mitsubishi Electric Research Labs.

The licensing terms for the Sphinx engines and tools are derived from BSD, and based, in particular, upon the license for the Apache web server. There is no restriction against commercial use or redistribution. (License terms for CMU Sphinx)

The packages that the CMU Sphinx Group is releasing are a set of reasonably mature, world-class speech components that provide a basic level of technology to anyone interested in creating speech-using applications without the once-prohibitive initial investment cost in research and development; the same components are open to peer review by all researchers in the field, and are used for linguistic research as well.

*Note however that Sphinx is not a final product. Those with a certain level of expertise can achieve great results with the versions of Sphinx available here, but a naive user will certainly need further help. In other words, the software available here is not meant for users with no experience in speech, but for expert users.*

**instructions/links**

- **PocketSphinx**
- **Sphinx-2**
- **Sphinx-3**
- **Sphinx-4**
- **SphinxTrain**
- **Utilities**

---

**Latest News**

**Switch to Subversion at April 18**
2006-04-03 21:06
Read More »

**Sphinx 3.6 Release Candidate I Released**
2006-03-30 11:14
Read More »

**Sphinx2 0.6 released**
2005-10-13 14:20
Read More »

Site news archive »

---

**External links**

Notice: if you have comments about the links below, please contact the authors directly.

- **How to use a SR system (and other related**

This site will be the canonical location for the release of the Sphinx trainers, recognizers, acoustic and language models, and documentation.

## Try a System

If you'd like to have a chance to try out  an application that uses CMU Sphinx, try one of these.

- Roomline, a system that handles conference room reservations within CMU. You can reach it at the toll-free number 1-877-CMU-PLAN (1-877-268-7526) or at +1 412 268 1084.

  *Note that your call will be recorded for development purposes and may be shared with other researchers. We don't have a policy set up yet for placing such recordings into a publicly availably database, and so there is no guarantee that this data will become publicly available -- though we're motivated to set that up in the future.*

- Let's Go, a spoken dialog system for the general public. The Let's Go! project is working in the domain of bus information, providing information such as schedules and route information for the city of Pittsburgh's Port Authority Transit (PAT) buses. You can interact with a version of this system right now by calling 412-268-3526 (requires some knowledge of Pittsburgh's transit system).

## Bug Tracking and Discussion Groups

There are fora for bug tracking and discussions on the SourceForge site, also. Please go there for help, questions, to report bugs, and to see the latest work. The work is currently pre-version 1.0, so there is a lot yet to be done.

## Platforms

- GNU/Linux, Unix variants, and Windows NT or later

**material)**
- **Speech aware applications**

---

This page is maintained by E. Gouvêa (egouvea+cmusphinx@cs.cmu.edu)
CMUSphinx is a project within the Sphinx Group at Carnegie Mellon

**What's New! What's Free!**
**New LDC Online Membership Option!** - LDC is pleased to announce a reduced cost alternative to accessing LDC Online resources
**MIXER 3** - talk on the phone in your native language and make up to $270.
**Language Resource and Evaluation Conference 2006** - as in previous years, LDC is pleased to contribute to the organization and promotion of LREC.
**New LDC Online!** - improved LDC Online services are now available.
**Fisher Spanish** - talk on the phone in Spanish for free and make up to $200!
**New Membership Option!** - LDC is pleased to announce the new Subscription Membership. Click above for more info!
**CallFriend2** - make a free phone call to a friend or family member with a chance to win $500!
**What's New Archive**

**New Corpora**
**CSLU: Spelled and Spoken Words** ~transcribed speech from over 3000 speakers
**Korean Propbanks** ~semantic annotation containing over 30K annotated predicate tokens
**Speech Controlled Computing** ~supports the development of ASR applications in the domain of voice control for the home
**ACE 2005 Multilingual Training Corpus** ~Arabic, Chinese, and English data annotated for entities, relations, and events
**Levantine Arabic QT Training Data Set 5, Speech and Transcripts** ~250 hours of transcribed telephone speech
**Arabic Gigaword Second Edition** ~text from five Arabic news sources
**CSLU Voices** ~twelve speakers reading phonetically rich sentences

**New Corpora Archive**

**Employment at the LDC**

**ACL Anthology** ~ A Digital Archive of

## Linguistic Data Consortium

The Linguistic Data Consortium supports language-related education, research and technology development by creating and sharing linguistic resources: data, tools and standards.

LDC is supported in part by grant IRI-9528587 from the Information and Intelligent Systems division and grant 9982201 from the Human Computer Interaction Program of the National Science Foundation. LDC's corpus creation efforts are powered in part by Academic Equipment Grant 7826-990 237-US from Sun Microsystems.

Research Papers in Computational Linguistics, hosted at the LDC

Contact ldc@ldc.upenn.edu
Last modified: Thursday, 30-Mar-2006 17:39:36 EST