DESIGN & IMPLEMENTATION OF A CO-PROCESSOR FOR EMBEDDED, REAL-TIME, SPEAKER-INDEPENDENT, CONTINUOUS SPEECH RECOGNITION SYSTEM-ON-A-CHIP

by

Kshitij Gupta

B.E., Osmania University, 2002

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

2005

UNIVERSITY OF PITTSBURGH

SCHOOL OF ENGINEERING

This thesis was presented

by

Kshitij Gupta

It was defended on

December 2, 2005

and approved by

Dr. Alex K. Jones, Assistant Professor, Department of Electrical & Computer Engineering

Dr. Steven P. Levitan, Professor, Department of Electrical & Computer Engineering

Thesis Advisor: Dr. Raymond R. Hoare, Assistant Professor, Department of Electrical & Computer Engineering

Copyright © by Kshitij Gupta

2005

DESIGN & IMPLEMENTATION OF A CO-PROCESSOR FOR EMBEDDED, REAL-TIME, SPEAKER-INDEPENDENT, CONTINUOUS SPEECH RECOGNITION SYSTEM-ON-A-CHIP

Kshitij Gupta, M.S.

University of Pittsburgh, 2005

This thesis aims to break the myth that multi-GHz machines are required for processing speaker-independent, continuous speech recognition based on full models performing full-precision computations in real-time. Through the design of a custom hardware architecture this research shows that 100 MHz is sufficient to process a 1,000 word dictionary in real-time. The design and implementation of the architecture is discussed in this thesis. It is shown that this implementation requires limited hardware resources and therefore can be incorporated as a dedicated speech recognition co-processor.

The system comprises of three major blocks corresponding to Acoustic, Phonetic and Word Modeling. For maximum performance, each of the blocks has been implemented in a highly pipelined manner, thereby enabling the computation of several quantities simultaneously. Further, fewer computations implies lower power consumption. To achieve this, optimizations at every stage of the computations have been made by incorporating feedback which enables the computation of only active data at any given time instant. For ensuring a scalable implementation, a dynamic memory allocation scheme has also been incorporated which helps manage the internal memory.

Amongst the three blocks, Acoustic Modeling contributes between 55-95% towards the overall computations performed by the system. Therefore special attention was paid onto the computations in Acoustic Modeling and a new computation reduction technique, *bestN*, is proposed. This technique addresses both the bandwidth requirement and the complexity of the computations. It is shown that for little loss in relative accuracy, only 8-bit integer addition operations are required while traditional systems need numerous 32-bit multiply and add operations. This technique also helps address the bandwidth requirement of the system by requiring 1/8th the bandwidth of traditional methods and for the same bus width, an 8x speedup in performance can be achieved.

TABLE OF CONTENTS

PREFACE XIII
1.0 INTRODUCTION 1
1.1 MOTIVATION 1
1.1.1 The Problem 2
1.1.2 The Solution 4
1.2 OVERVIEW 5
1.2.1 Basic Definition 6
1.2.2 Task-level Overview 8
1.2.2.1 Computationally Un-optimized Brute-force Approach 9
1.2.2.2 Computationally Optimized Approach using Feedback 11
1.3 PERFORMANCE SPECIFICATIONS 13
1.3.1 Performance Characterization of Speech Algorithms 13
1.3.2 System Specifications 14
1.4 CONTRIBUTIONS & ORGANIZATION OF THE THESIS 15
1.4.1 Major Contributions 15
1.4.2 Organization of the Thesis 16
2.0 LITERATURE & ALGORITHM OVERVIEW 19
2.1 LITERATURE OVERVIEW 19
2.2 THEORY & ALGORITHM OVERVIEW 21
2.2.1 Word Block 23
2.2.2 Phone Block 25
2.2.2.1 Phone Score Calculations 26
2.2.2.2 Phone Pruning 27
2.2.2.3 Context-Dependent Phones 29

2.2.3 Acoustic Modeling Block 29
2.2.3.1 Senone Scoring 29
2.2.3.2 Composite Senone Scoring 31
3.0 SYSTEM DESIGN 34
3.1 INTRODUCTION 34
3.2 DESIGN METHODOLOGY 34
3.2.1 System Requirements 34
3.2.2 Timing and Resource Requirements 35
3.3 SYSTEM ARCHITECTURE 36
3.3.1 Top-Level Block Partitioning 37
3.3.2 High Level Phase Partitioning 38
3.3.3 Detailed System Architecture and Data Flow 40
3.3.3.1 Feedback using Active Lists 40
3.3.3.2 System Data Flow 41
3.4 SUMMARY 44
4.0 ACOUSTIC MODELING 45
4.1 INTRODUCTION 45
4.2 BRUTE-FORCE ACOUSTIC MODELING 51
4.2.1 Gaussian Distance Block 51
4.2.1.1 Bit-precision Analysis for Fixed-point Computations 52
4.2.2 Log-add Block 57
4.2.3 Senone/Composite Senone Scoring Block 61
4.3 SUBVECTOR QUANTIZATION 62
4.3.1 Introduction 62
4.3.2 Motivation 63
4.3.3 Sub-Vector Quantization 64
4.3.4 Implementation 66
4.4 TOP-LEVEL 67
5.0 THE “BESTN” COMPUTATION REDUCTION TECHNIQUE FOR AM 70
5.1 MOTIVATION 70
5.1.1 Ideal Characteristics of Computation Reduction Techniques 70

5.2 HINDSIGHT OBSERVATION 71
5.3 THE "BESTN" METHOD 74
5.4 IMPLEMENTATION ASPECTS 79
5.5 TEST RESULTS 82
5.6 CONCLUSION 84
6.0 PHONEME & WORD MODELING 86
6.1 INTRODUCTION 86
6.2 PHONE BLOCK 86
6.2.1 Introduction 86
6.2.2 Phone Block Top-Level 88
6.2.3 Phone Feedback: Generation of Senone Active List (SAL) 89
6.2.4 Phone Calculation Block 91
6.2.5 Prune Block 93
6.3 WORD BLOCK 95
6.3.1 Introduction 95
6.3.2 Word Phone De-activation Phase 98
6.3.3 Word Phone Propagation Phase 98
7.0 SYSTEM INTEGRATION, TESTING, AREA AND PERFORMANCE 102
7.1 INTRODUCTION 102
7.2 IMPLEMENTATION 102
7.2.1 Introduction 102
7.2.2 FPGA Prototyping Environment 103
7.2.3 Highly efficient, fully pipelined design 104
7.3 DESIGN ENVIRONMENT 105
7.3.1 Xilinx System Generator 105
7.3.2 Xilinx ISE 106
7.4 SYSTEM SET-UP, TESTING & RESULTS 107
7.4.1 System Set-up 107
7.4.1.1 Compute Cycles 108
7.4.1.2 Memory 108
7.4.2 Testing 109

7.4.3 Simulation Results **110**

7.5 AREA & PERFORMANCE **112**

7.5.1 Area **113**

7.5.1.1 DSP Blocks **113**

7.5.1.2 RAM Blocks **114**

7.5.1.3 Logic Slices **114**

7.5.1.4 Design Floorplan on Xilinx's Virtex4 SX-35 .... **114**

7.5.2 Performance **116**

7.5.3 Reduction in Computation using Feedback **116**

7.6 SUMMARY **117**

8.0 CONCLUSIONS **118**

8.1 MAJOR CONTRIBUTIONS **119**

8.2 CONCLUSIONS **119**

8.3 FUTURE DIRECTIONS **120**

APPENDIX A **121**

BIBLIOGRAPHY **172**

LIST OF TABLES

Table 1.1: Numbner of Compute Cycles for 3 different Speech Corpouses 14

Table 3.1: Timing and resource requirements for the entire system... 35

Table 4.1: Number of operations per second required for Acoustic Modeling 48

Table 4.2: Summary of minimum and maximum data ranges for each stage of the Gaussian Distance computation with the corresponding number of "integer" bits required to represent the ranges accurately 53

Table 4.3: Final bit-precision breakup for representing the inputs and the intermediate outputs of Acoustic Modeling.. 55

Table 5.1: Percentage Error in Senone Scores of the three setups w.r.t. the original scores......... 77

Table 5.2: Ballpark of Senone Scores on quantizing for N for different levels............................ 80

Table 5.3: the Word Error Rate Results for 3 setups of bestN 83

Table 7.1: Detailed Resource and Timing Requirements of the System 107

Table 7.2: Area and Performance of the entire System as implemented on the Virtex4 SX-35 FPGA 113

Table 7.3: Table showing the number of Phones and Senones active for 3 test-utterances 117

LIST OF FIGURES

Figure 1.1: Conceptual view of Automatic Speech Recognition System 2

Figure 1.2: A simple 3-state Hidden Markov Model 7

Figure 1.3: Conceptual view of an Automatic Speech Recognition System with the tree major blocks, namely, AM, PHN, WRD Blocks along with the major data-structures associated with each of them 8

Figure 1.4: Conceptual Block Diagram depicting Brute-force Approach 9

Figure 1.5: Pseudo-code representation of the Brute-force Approach 10

Figure 1.6: Conceptual Block Diagram depicting a Computationally Optimized Approach using Feedback 12

Figure 1.7: Pseudo-code representation for Computationally Optimized Approach..................... 12

Figure 2.1: Conceptual view of a Automatic Speech Recognition System with AM, PHN, WRD Blocks 22

Figure 2.2: A 8-word Sample Dictionary with its Phonetic Representation................................. 23

Figure 2.3: Word Tree-structure corresponding to the 8-word Sample Dictionary....................... 24

Figure 2.4: A Simple 3-state HMM Model.. 25

Figure 2.5: Phone State-tracing using Viterbi Algorithm 26

Figure 2.6: A 3-Dimensional view of the Acoustic Modeling Database 31

Figure 2.7: Cross-word Phone Modeling using Composite Senones 32

Figure 2.8: 3-state HMM for a cross-word Phone 32

Figure 3.1: Top-Level System Block Diagram.. 37

Figure 3.2: Conceptual Phase Partitioning of the System.. 39

Figure 3.3: Detailed Top-Level System Block Diagram with data flow information 40

Figure 3.4: Detailed System Data Flow 42

Figure 4.1: Pseudo-code representation of Brute-force Acoustic Modeling Calculations 49

Figure 4.2: Top-level Block Diagram for brute-force Acoustic Modeling.................................... 50

Figure 4.3: Pseudo-code for Gaussian Distance Calculation 52
Figure 4.4: A Graph showing average percentage error in Senone Scores per frame over all frames in the sample utterance 56
Figure 4.5: A Histogram of Percentage Error in Senone Scores for the Sample Utterance 57
Figure 4.6: Monotonically decreasing value of the Pre-computed Log-add with increase in the difference of the inputs 59
Figure 4.7: Pseudo-code for Log-add Calculation 60
Figure 4.8: Pseudo-code for Normalized Senone score Calculation 61
Figure 4.9: Pseudo-code for Composite Senone score Calculation 62
Figure 4.10: Senone Scores as a function of Component Score 64
Figure 4.11: A 3-Dimensional Representation of the SubVQ Database 65
Figure 4.12: Pseudo-code for Sub-Vector Quantization Calculation 66
Figure 4.13: Top-level Block Diagram of Acoustic Modeling with SubVQ along with the phase breakup 67
Figure 4.14: High-level Pseudo-code representation of Acoustic Modeling Block (with SubVQ)68
Figure 5.1: Gaussian Probability superimposed onto GAUS_DIST calculation for a single dimension 72
Figure 5.2: Distance Metric Superimposed on Probability graph for a single dimension 73
Figure 5.3: A 3D graph of N's for all Coefficients over all Components of 1 Senone 74
Figure 5.4: A 3D graph of N's for all Coefficients over all Components of 5 Senones 75
Figure 5.5: Summation of N's of individual dimensions of a Component 76
Figure 5.6: Selection of "top" Components based on ascending order of N's in a Senone 76
Figure 5.7: Average Percentage Error of Senone Scores per frame for all frames of sample utterance 78
Figure 5.8: Implementation technique for bestN in a practical system 81
Figure 5.9: Pseudo-code representation of the bestN implementation 82
Figure 5.10: Word Error Rates for 3 sets of experiements with varying use of the number of Components 84
Figure 6.1: 3-state HMM Structure 87
Figure 6.2: Top-level Block Diagram representing the various blocks associated with Phone Block Calculations 88
Figure 6.3: Top-level Block Diagram with Phase information for SAL Generation 89
Figure 6.4: Pseudo-code for the Generation of the Senone Active List 91

Figure 6.5: Block Diagram representation of computations in PHN_CALC 92
Figure 6.6: Top-level Block Diagram with Phase information for PHN_CALC 93
Figure 6.7: Pseudo-code for the Phone Prune Phase 94
Figure 6.8: Top-level Block Diagram with Phase information for PHN_PRN 94
Figure 6.9: 8-word sample dictionary tree structure.. 95
Figure 6.10: Top-level Block Diagram representing the various blocks associated with Word Block Calculations 96
Figure 6.11: Top-level Block Diagram depicting the Dead Phase 98
Figure 6.12: Top-level Block Diagram with Phase information for WRD_NXT_PHN phase (1)99
Figure 6.13: Top-level Block Diagram with Phase information for WRD_NXT_PHN phase (2)100
Figure 7.1: Prototyping Environment Top-Level Diagram 104
Figure 7.2: Top-Level System Block Diagram.. 107
Figure 7.3: Simulation Result for System Controller going through the various phases 110
Figure 7.4: Simulation Result for SAL Generator (the active phones correspond to the SIL phone) 111
Figure 7.5: Simulation Result showing AM Controller going through the 4 phases................... 111
Figure 7.6: Simulation and MATLAB results for Phone Scores 112
Figure 7.7: Floorplan for the entire Speech Recognition System on a Virtex4 SX-35 115

PREFACE

This thesis is dedicated to…

BB, my parents, and all my well-wishers

1.0 INTRODUCTION

1.1 MOTIVATION

Evolution of life as we know it today has been a function of several factors, one of the key ones being *communication*. The ability to quickly and efficiently communicate, interact and exchange ideas/experiences has helped us, humans, to progress much quicker than any other life-form. What sets humans apart from most of the other life forms is the ability to combine *all* 5 sensory perceptions in our day-to-day lives. Amongst them, speech, with its ability to convey information precisely and with minimal effort has been one of the major drivers of human progress as we know it.

With the increase in scientific knowledge over the past few decades, tremendous progress in the field of information technology has been achieved. Progress on the information technology front has directly helped humans to progress further and ensure their well-being. The development in information technology is revolutionizing virtually every aspect of human life: from disease diagnosis and cure, to entertainment and communication.

These technological enhancements have made the world today a small place where several things can be done at the mere touch of a button. What if the most important aspect of human communication, speech, could be integrated with these devices that are helping us achieve so many things in our day-to-day lives? Realizing the tremendous benefit to the community at large, this research focuses on exploring ways to enable seamless integration of speech enabled devices.

1.1.1 The Problem

Although several years of research has gone into the development of speech recognition, the progress has been rather slow. This is a result of several limiting factors amongst which recognition accuracy is one of the most important. The ability of machines to mimic the human auditory perceptory organs and the decoding process taking place in the brain has been a challenge, especially when it comes to the recognition of natural, irregular speech [1].

These challenges are a direct result of the tremendous variability in the speech signal. This variability is mostly a function of a few factors: difference in dialects, gender, age, environment and the target application. Every time one or more of these variables are changed, a new set of conditions needs to be addressed. This leads to non-reproduceable results and therefore requires further research into each of the specific aspects created by the change in variables.

To date, however, state-of-the-art recognition systems have been able to overcome some of these issues. Specifically, recognition systems that are centered around command and control-based applications provide accuracies in excess of 90% for speaker independent systems with medium sized dictionaries [2].

Despite the satisfactory accuracy rate achieved for such applications, speech recognition is still yet to penetrate our day-to-day lives in a big way. Whether it be the setting of functions for the washing machine or switching of channels on the television or to typing short text/email messages on the mobile devices like PDAs and cell phones, we still need to use our hands and are required to press buttons in order to perform these operations. These are a few examples of medium vocabulary, command and control type applications that we come across in our every day lives.

The problem stems from the fact that speech recognition is computationally intensive requiring several million floating-point operations per second. Most speech recognition systems in use today are predominantly software based [12,13]. They utilize the computational and memory resources provided by General Purpose Processors (GPP) that are built on architectures to support a wide range of applications.

Such general architectures however, because of limited Arithmetic Logic Units (ALU) prove to be insufficient for computationally intensive applications like Speech Recognition. The

number of ALUs is not the only problem. The problem lies with the very nature of GPPs which are based on cache-based architectures for speed of operation. While such architectures work well for applications in general, speech, with tremendous variability requires access to large amounts of non-sequential data. Cache sizes in most processors available today, especially those catering towards embedded applications, is very limited, the order being 10s of KBs only [3]. Therefore, accessing 10's of MB of speech data using 10's of kB of on-chip cache results in several cache misses thereby leading to pipeline stalls. Hence, several extra processor compute cycles are needed to process the data.

Further, since several peripherals and applications running on a device need access to a common processor, bus-based communication is required. This requires the synchronization of all elements connected to the bus by making use of bus transaction protocols thereby incurring several cycles of additional cycle overhead. For example, a Microblaze based interface to DDR memory using the On-Chip Peripheral Bus (OPB) requires 4 cycles for read and 5 cycles for write operations to account for bus arbitration in addition to memory access latencies which can range between 3-4 cycles. This in turn implies decreasing the overall performance of memory read/write operations by almost 50%.

Because of these inefficiencies, it is not surprising that Speech Recognition systems execute less than 1 Instructions Per Cycle (IPC) [4,5] on such GPPs making speech recognition process slower than real-time [6]. Considering some of the regular computations that are performed in these systems, this result is counter-intuitive.

To counter these effects, implementers have two options. They could either make use of processors with higher clock-rate to account for processor idle time caused on account of pipeline stalls and bus arbitration overheads, or, they could re-design the processor that caters to the specific requirements of the application. Since implementers developing software based systems are dependent on the underlying processor architecture, they tend to take the former option. This approach results in the need for devices with multi-GHz processors.

Since multi-GHz machines is not always practical, implementers are forced to make some compromises. The reduction in the bit-precision and making use of coarser models so as to decrease the data size are two favored approaches. While this helps in making the system practically deployable, the loss in computational precision in most cases leads to a degraded performance (in terms of accuracy) and in decreasing the robustness of the system (from speaker

independent to speaker dependent and from continuous to discrete speech). All in all, the user experience suffers.

1.1.2 The Solution

On analyzing these aspects, it is clear that GPP-based speech recognition needs to be done away with. The solution lies with the development of dedicated hardware architecture with bit-level optimizations that can cater to the heavy computational requirements of speech recognition algorithms. This hardware could act as a speech co-processor and could be directly interfaced with existing commercially available GPPs.

Designing a dedicated architecture would not only allow for optimizing the available resources required by the application, but also allow for the creation of dedicated data-paths thereby eliminating significant bus transaction overhead. It would also allow for performing bit-level optimizations and enable the design of a power-efficient system. All in all, such an approach would provide the ability to process speech in real-time without sacrificing the necessary bit-precision.

The design and implementation of such an architecture is the focus of this thesis. It is shown that a highly efficient design running at less than 100 MHz is sufficient for processing a 1,000 word Command and Control based application and can be implemented using 93,000 gates implying a small silicon-footprint, thereby enabling it to be incorporated into existing systems as a dedicated speech recognition co-processor.

Rather than the traditional theoretical approach most theses related to speech recognition tend to take, a more computation based approach is followed in this thesis. The description of the algorithms/computations is done in a way so as to enable in understanding some of the design considerations made on the various blocks and subsequently used in the implementation phase.

1.2 OVERVIEW

Speech Recognition is essentially a search problem since it deals with finding the most "probable" words/sequences of words from among a set of pre-defined words in the system dictionary. A conceptual representation of a speech recognition system is shown in Figure 1.1.

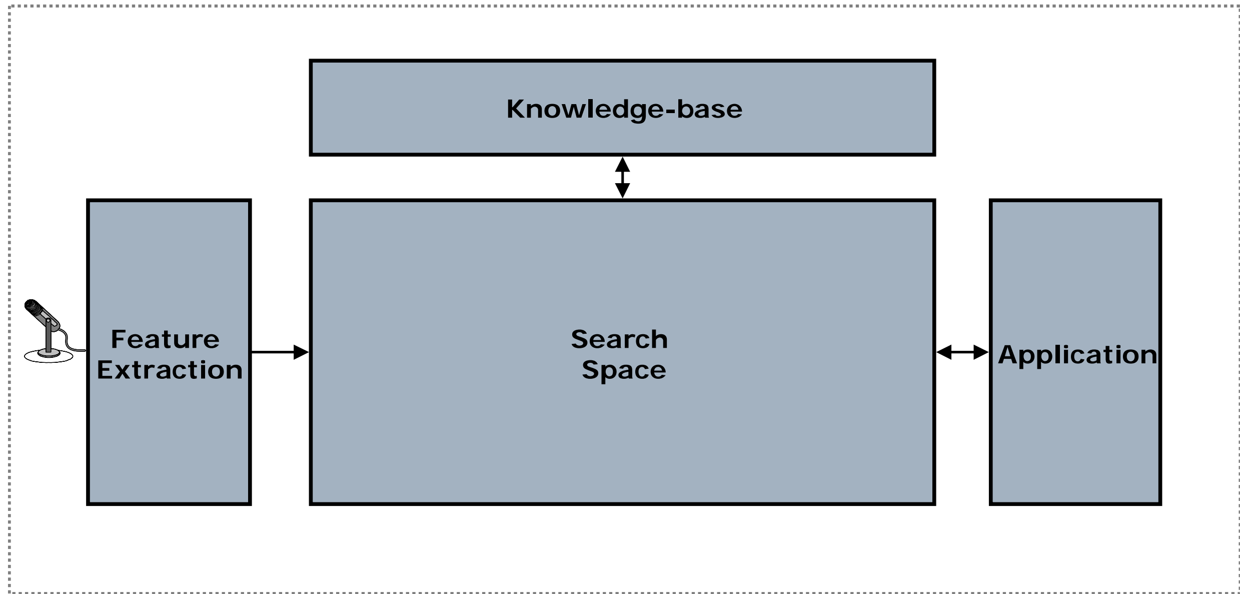


Figure 1.1 Conceptual view of Automatic Speech Recognition System

The spoken words are first processed by the Speech Recognizers Front-end (FE). As its name suggests, this block is responsible for obtaining meaningful information from the input speech samples by extracting essential speech features. A 39-dimensional *feature vector* with each dimension corresponding to frequency characteristic in different frequency bands is output from FE to Search Space at 10ms intervals. Each 10ms interval is known as a *frame*. The Search Space along with the trained statistical models contained in the knowledgebase, produces probability of observing pre-defined words for the given speech input. The computation in the Search Space accounts for almost 95-99% of the overall computations in the entire system.

Whenever the probability of a word in the Search Space crosses a certain threshold, the word along with its probability is forwarded to the Application for further processing. The application then parses the information from several possible observed words over several frames and picks the most promising one determined by its probability. This is the basic idea of how machine based automatic speech recognition is performed.

Since the majority of the computational work-load is accounted by the Search Space, data processing in the Feature Extraction and Application blocks can be thought as pre- and post-processing operations respectively. Therefore, this thesis focuses completely on the computations performed within the Search Space.

1.2.1 Basic Definition

As for a more formal definition of speech recognition, it deals with finding the most probable word/sequence of words given a particular set of speech samples (observation) and can be represented by the expression $P(O/W)$ [7]. This however implies finding the probability of all possible words in the word database for the given samples of speech, which is virtually impossible.

Hence, the problem needs to be reframed. For this purpose, using basic probability and statistic algorithm, Baye's rule [8], the above expression can be re-written as,

$$P(W/O) = \left[\frac{P(O/W) * P(W)}{P(O)} \right] \quad \text{Eq. 1.1}$$

According to this equation, Speech Recognition could be seen as the probability of finding the word given the observed speech samples, $P(W/O)$, and can be found by evaluating the probability of observing the given speech samples for a *given word* $P(O/W)$, times the probability of observing the word itself, $P(W)$, over the probability of the observation, $P(O)$. The probability of observation, $P(O)$, is assumed to be completely random and therefore does not provide useful information and hence, can be ignored altogether.

However, one slight modification needs to be made to the above equation. Since the aim of speech recognition is to find the most likely word, a more accurate representation of speech recognition is given in Eq. 1.2. From this equation it can be seen that the aim is to find the word, *W*, which maximizes the overall probability for a given set of speech samples, *O*.

$$\begin{aligned} W_{best} &= \arg\max_W P(W/O) \\ &= \arg\max_W \left[P(O/W) * P(W) \right] \end{aligned} \quad \text{Eq. 1.2}$$

From a computation stand-point, this equation consists of the computation of two main quantities, the word (right hand term, $P(W)$) and sub-word units, *phonemes*, that make up the

word (left hand term, $P(O/W)$). The probability of the word is obtained during the training process and is computed as the frequency of occurrence of the given word over the probability of all possible words in the training setup. For better performance in sentence recognition, state-of-the-art recognition systems use higher order probabilities by keeping context of previously spoken words. However, since this information is pre-computed, the task of obtaining $P(W)$ is completely based on looking up appropriate values from the database and hence does not pose a computational bottleneck.

The probability of the sub-word units, phonemes (also referred to as *phones*), is based on the input feature stream. Modeling of phones essentially deals with pattern matching whereby the frequency characteristic of the phonetic sound is compared with the input speech. There are two main approaches of modeling phones: Hidden Markov Models (HMM) [16] and Neural Networks. Most state-of-the-art recognition systems today use HMM-based models. One such system, Sphinx3 [6] developed at Carnegie Mellon University, is used as the basis of this research.

Sphinx3 is a HMM-based state-of-the-art, speaker independent, large vocabulary, continuous speech recognition system developed at CMU. Since the aim of this research was to design an architecture, all statistical models used were derived from Sphinx 3.3. This research focuses on a 1,000 word Command & Control dictionary based on the RM1 Speech Corpus [9] which relates to commands spoken by naval officers in a Resource Management task.

Sphinx uses a simple 3-state HMM Model, each state corresponding to sub-phonetic, *acoustic* sounds, modeling the beginning, middle and end of the phone (Figure 1.2). Each state has a Gaussian distribution and two possible transitions associated with it. The process of recognition deals with traversing the states and is described in detail in Chapter 2.

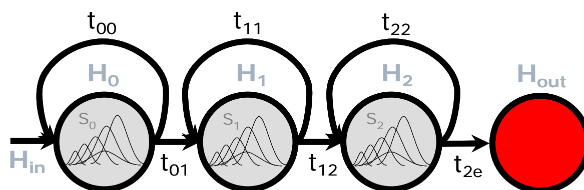


Figure 1.2: A simple 3-state Hidden Markov Model

Since the feature vector output of the Feature Extraction phase is a 39-dimensional quantity, each Gaussian shown in Figure 1.2 too is a 39-dimensional quantity. In speech recognition systems, there can be as many as 100k such multi-dimensional Gaussians requiring

several hundreds of million floating-point operations per second. Since the evaluation of Gaussian probabilities is not only a computationally intensive task, but also very different in nature when compared to the core computations related to state-tracing that take place in the evaluation of a HMM, the two computations can be thought to be performed in two separate, Acoustic Modeling and Phone Modeling Blocks respectively.

Based on this information, the conceptual apeech recognition system presented in Figure 1.1 can be re-drawn to include the major blocks according to the different sounds, Word, Phonetic and Acoustic sounds each of them model. Hence the system can be viewed as a combination of the Word (WRD), Phone (PHN) and Acoustic Modeling (AM) Blocks, which have unique data-structures associated with each of them. This is represented as ROMs in the *Knowledgebase* in Figure 1.3.

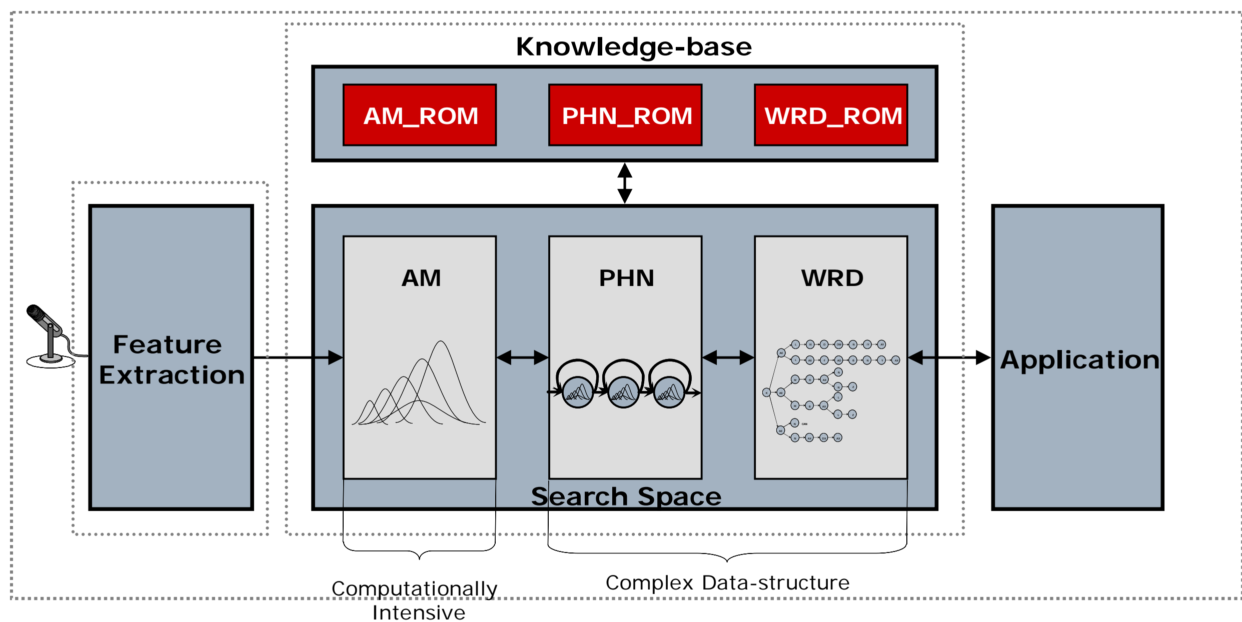


Figure 1.3: Conceptual view of an Automatic Speech Recognition System with the tree major blocks, namely, AM, PHN, WRD Blocks along with the major data-structures associated with each of them

1.2.2 Task-level Overview

In this section, the order of computations is presented. Each computation is treated as a task and hence, the description of the system is presented at a task level. First, a computationally un-optimized, brute-force approach is described whereby all computations are performed in a left

to right order of Figure 1.3 (Section 1.2.2.1). Such an approach however requires the computation of all quantities in each data-structure. But at any given time, based on processing the first few frames of speech, only a limited number of words can be hypothesized. Therefore, data corresponding to only the most likely spoken words need to be evaluated.

Based on this, the ability to compute only those quantities in the individual data-structures that correspond to the likely list of hypothesized words can help reduce the number of computations significantly. This is done by the incorporation of feedback from every stage (Section 1.2.2.2). The incorporation of feedback into the architecture in an efficient manner is one of the major contributions of this thesis.

1.2.2.1 Computationally Un-optimized Brute-force Approach

For a brute-force approach, the system can be thought to follow a left to right approach of Figure 1.3. The processing of each frame begins from at the Feature Extraction block, goes through the evaluations in the Search Space and ends at the Application. A Block Diagram representing this flow is shown in Figure 1.4 and a corresponding pseudo-code representation of the major computations being performed in each of the blocks is shown in Figure 1.5.

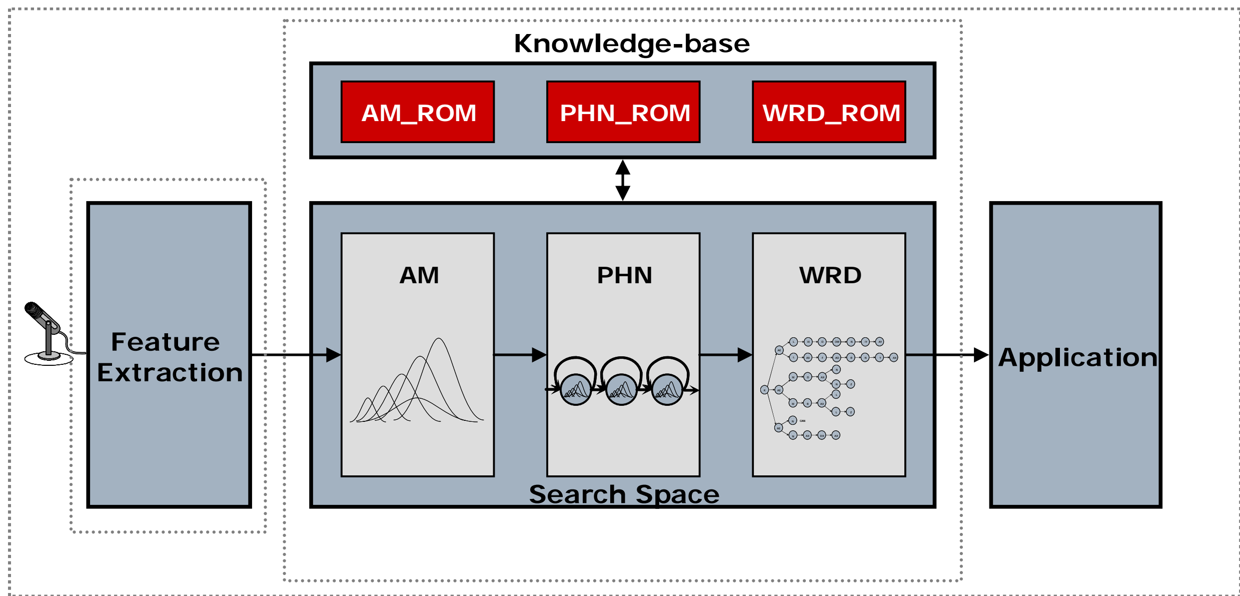


Figure 1.4: Conceptual Block Diagram depicting Brute-force Approach

The processing begins with the Analog-to-Digital sampling of input speech. A frequency response of the speech samples is obtained using a 512-point FFT. The frequency response is

filtered and the output from each of the filters is de-correlated resulting in a 39-dimensional feature vector.

```
for frame
    FEATURE EXTRACTION [FE]:
        INPUT: Speech Samples
        A-to-D Sampling from Microphone()
        512-point FFT()
        Filter and obtain de-correlated features()
        OUTPUT: 39-dimensional feature vector
    ACOUSTIC MODELING [AM]:
        INPUT: 39-dimensional feature vector
        Gaussian probability evaluation()
        Normalizing probabilities w.r.t. the BEST()
        OUTPUT: Normalized Gaussian probabilities
    PHONE MODELING [PHN]:
        INPUT: Normalized Gaussian probabilities
        Phone score evaluation()
        Prune Phones based on their scores()
        OUTPUT: Phone scores + List of pruned phones Phones
    WORD MODELING [WRD]:
        INPUT: Phone scores + List of pruned phones
        Reset scores for non-promising phones()
        Propagate Phones from current to next in the word()
        OUTPUT: List of possible words with their probabilities
    APPLICATION [APP]:
        Pick most likely word()
end frame
```

Figure 1.5: Pseudo-code representation of the Brute-force Approach

In the Search Space, first, the Acoustic Modeling Block uses the 39-dimensional feature vector to compute Gaussian probabilities with respect to the mean and variance pairs in the database (AM_ROM). The probabilities for *all* Gaussians in the data-structure are computed for the input feature vector. The probabilities are then normalized with the best probability for the current frame and the set of normalized scores are forwarded to the Phone Block. There are two major computations performed in the Phone Block. The first relates to the computation of the Phone Scores themselves based on moving through the 3-state structure of Figure 1.2 while the second one relates to pruning of phones that are not above a certain threshold.

The phone scores along with the list of pruned data are forwarded to the Word Block which essentially consists of a word to phone mapping. This mapping is maintained as a tree structure as part of the WRD_ROM. The Word Block essentially deals with traversing this tree structure from the beginning to the end of a word. Stepping through the tree is based on Phone

scores forwarded by the Phone Block. Succeeding Phones are assigned their scores based on the score of their predecessors. This management is done by the Word Block.

Further, un-promising Phones which the Phone Block determines can be pruned out are reset and paths corresponding to such words are stopped from further processing at the Word level. Based on the scores, a list of all possible words that can be hypothesized is generated and forwarded to the Application for post-processing. This is the sequence of operations when following a brute-force approach.

1.2.2.2 Computationally Optimized Approach using Feedback

While implementation of a system based on the brute-force approach requires little synchronization between the individual blocks thereby enabling the implementation of a less complex system, this approach has one major drawback. Since *all* quantities in the individual databases are calculated, it implies the computation of even those quantities that do not correspond to words that are very different sounding. This in turn implies a significant overhead on the computations performed. Hence, for a system to be practically deployable, especially for systems with tight power constraints, the number of computations needs to be reduced.

One way of achieving this is to keep track of "active" data. Since at any given point in time only a subset of words can be hypothesized, extensive data management needs to be incorporated into the system enabling it to keep track of these "active" words and their associated data-structures. From an implementation stand-point, this implies the incorporation of a *feedback* mechanism that helps in keeping track of "active" data based on information form preceding stages.

For the sake of clarity, Figure 1.4 has been re-drawn to show the feedback in Figure 1.6. Although from a high-level, the feedback has been depicted as one block, in reality it corresponds to a list of active data for each of the individual blocks in the Search Space.

Having incorporated feedback into the system, the sequence of operation of the tasks needs to be re-ordered. For this reason, the pseudo-code of Figure 1.5 has been re-ordered into Figure 1.7. While the underlying computations are the same, the order in which they are evaluated has been changed. Further, two steps relating to the computation of active lists from Words to Phones and Phones to Gaussians has been incorporated into the flow.

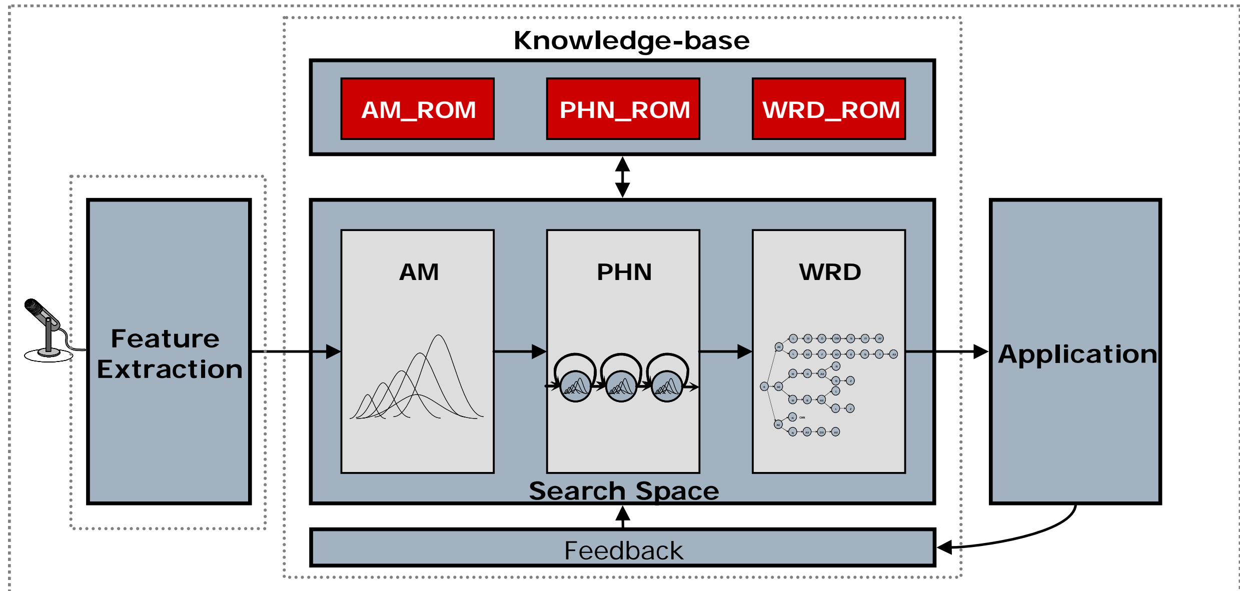


Figure 1.6: Conceptual Block Diagram depicting a Computationally Optimized Approach using Feedback

```
for frame
    FEATURE EXTRACTION [FE]:
        INPUT: Speech Samples
        A-to-D Sampling from Microphone()
        512-point FFT()
        Filter and obtain de-correlated features()
        OUTPUT: 39-dimensional feature vector
    WORD MODELING [WRD]:
        INPUT: Phone scores + List of pruned phones
        Reset scores for non-promising phones()
        Propagate Phones from current to next in the word()
        Generate list of active Phones()
        OUTPUT: List of possible words + List of active Phones
    GENERATE LIST OF ACTIVE SENONES from active Phone list()
    ACOUSTIC MODELING [AM]:
        INPUT: 39-dimensional feature vector
        Gaussian probability evaluation for active Senones()
        Normalizing probabilities w.r.t. the BEST()
        OUTPUT: Normalized Gaussian probabilities
    PHONE MODELING [PHN]:
        INPUT: Normalized Gaussian probabilities
        Phone score evaluation for active Phones()
        Prune Phones based on their scores()
        OUTPUT: Phone scores + List of pruned Phones
    APPLICATION [APP]:
        Pick most likely word()
end frame
```

Figure 1.7: Pseudo-code representation for Computationally Optimized Approach

Since a word is the biggest sound unit, the processing starts from the Word Block. On performing the computations, a list of active phones corresponding to active words is generated. Since the computation of Phone scores requires the computation of Gaussian probabilities, the next step is the generation of a list of active Gaussians that correspond to the active Phones. Only the Gaussians included in this list are computed. After this, Phone scores for all active phones are computed and pruned as appropriate. The same steps are repeated for every frame.

1.3 PERFORMANCE SPECIFICATIONS

1.3.1 Performance Characterization of Speech Algorithms

State-of-the-art Speech Recognition systems have been profiled and it have been found that Feature Extraction phase consumes less than 1% [10] of the overall compute cycles. This stems from the fact that the most dominant computation relates to the computation of a 512-point FFT to obtain the necessary frequency response of the system followed by obtaining the energy of the signal in different frequency bands requiring multiply and addition operations. Therefore, it was concluded that since feature extraction is not a computational bottleneck, dedicated hardware resources need not be allocated. For this reason, Feature Extraction can be implemented on a standard floating-point DSP and hence is not a focus of this thesis.

Acoustic Modeling on the other hand takes between 55-95% [10,11] of the compute cycles. All computations in AM are devoted towards the computation of multi-dimensional Gaussian probabilities. It was observed that one of the major problems associated with this block was the shear number of values that need to be computed. This in turn requires accessing of several millions of mean/variance pairs for every Gaussian distribution thereby making this a memory bandwidth intensive task. Further, because large amounts of data need to be processed, processors with small caches perform poorly. From these observations, it was concluded that a hardware resources would significantly help address the issues that prove to be a performance limiter.

Finally, Word Modeling has been found to take between 45-5% [10] of the total compute cycles. As described earlier, since Word Modeling essentially deals with the look-up of pre-

computed word probabilities for different sequences of words several 10's of 100's of MBytes of data, a majority of the compute cycles can be attributed to delays in memory access.

1.3.2 System Specifications

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 Athalon processor [10]. The models for a 64k word task are 3 times larger than that for a 1,000 word Command & Control task. 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.

For real-time operation, since feature frames are generated in 10ms intervals, the processing of every frame should be completed within 10ms before the next frame is available for processing. Designing the system on this constraint would ensure that the system would process speech in real-time. Based on open-source Sphinx models [38,39], Table 1.1 was created for three different recognition tasks: digit recognition, command & control, and continuous speech.

Table 1.1: Numbner of Compute Cycles for 3 different Speech Corpouses

Speech Corpus	Dictionary Size (words)	# of Gaussians	# of Gaussian Evals per 10ms (1,000)
Continuous Digits [TI Digits]	12	4816	192.64
Command & Control [RM 1]	1,000	15480	619.2
Continuos Speech [HUB-4]	64,000	49152	1966.08

The table shows the number of "compute cycles" required for the computation of all Gaussians for different tasks assuming a fully pipelined design. It can be seen that assuming a one-cycle latency in access memory, the RM1 task would require 620k compute cycles while HUB4 would require 2M cycles.

Given that 100 frames are generated every second, this implies that 62M and 200M compute cycles are required for real-time operation of the RM1 and HUB4 tasks. This implies the need for designs running at 62 MHz and 200 MHz respectively.

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 should be sufficient to process data for every frame for RM1 task. Therefore, based on the assumption that 1 million operations need to be completed every 10ms; 100 million operations would be performed in 1 second. This in turn implies a clock speech of 100 MHz. Based on this calculation, the target frequency of 100 MHz was set for the system to operate in real-time. Further, given that most mobile processors run around the 100 MHz range [3], a 100 MHz clock would make it ideally suited for being incorporated as a dedicated speech co-processor.

1.4 CONTRIBUTIONS & ORGANIZATION OF THE THESIS

1.4.1 Major Contributions

The major contributions of this thesis are as follows:

- Designed and implemented a scalable, fully pipelined custom hardware architecture for real-time, speaker-independent continuous speech recognition.
- Using dedicated FIFOs, incorporated feedback into the architecture in an efficient way from every stage of the design so as to enable the computation of only the active data, thereby minimizing the number of computations to be performed.
- Incorporated dynamic memory management into the architecture for maintaining all active data at the Word and Phone level, thereby allowing for the implementation of a scalable system.
- Converted single-precision floating-point computations into 32-bit custom fixed-point computations for Gaussian probability evaluation in Acoustic Modeling with an average of 10^{-3} % loss in accuracy.

- ♦ Explored a computation reduction technique in Acoustic Modeling from a fully hardware implementation perspective. This lead to proposing the "*bestN*" technique that allows for a bandwidth reduction by a factor of 8 and reduces the computation operations into 8-bit integer addition as opposed to several 32-bit multiply & add/subtract operations with 0.1% degradation in word recognition accuracy when compared to the baseline Sphinx 3 system on a 1,000 word Command & Control task.

1.4.2 Organization of the Thesis

The remainder of this thesis is organized as follows. The following chapter discusses some of the previous work done towards the creation of dedicated hardware for speech recognition. This is followed by a brief description of the basic computations and data-structures corresponding to the tree major blocks discussed above. The terms and terminologies introduced in this chapter are used throughout the remainder of this thesis.

Chapter 3 is devoted to the description of the system architecture. Based on timing and resource requirements, first, the design methodology followed for the design of the system is presented. After this, the system architecture is discussed in detail. Initially, a conceptual system is described with the major data-structures. Block and phase partitioning considerations that influenced the design process are discussed in detail. It was concluded that non-overlapping phases had certain distinct advantages and hence, the entire system was designed to ensure that at no point in time two blocks would access the same resources. Further, it is shown that an efficient mechanism for information and data exchange was achieved by using dedicated FIFOs and shared memories.

Finally, a detailed discussion of the top-level blocks in the system is presented along with data-flow information. The data-flow presented includes feedback of a list of active data from predecessor blocks thereby enabling the computation of only active data. The incorporation of feedback into the system results in a huge savings in the number of overall computations for the entire system.

In Chapters 4-6 a description of the three major blocks in the system is discussed. Since the Gaussian probability evaluations in the Acoustic Modeling block account for a majority of the computations performed in the entire system, special attention was paid to the computations

in this block. For this reason, Chapters 4 and 5 are devoted for Acoustic Modeling while Chapter 6 is devoted for the Phone and Word Modeling blocks.

In Chapter 4 a detailed description of the computations performed in Acoustic Modeling is discussed. Since most software-based state-of-the-art recognition systems perform floating point computations, it was first necessary to convert these computations into fixed-point with minimal loss in accuracy. For this, a thorough analysis of the bit-precision using MATLAB was done. It is shown that 32-bit fixed-point operations can be used with 10^{-3} % loss in accuracy over floating-point computations.

Since a brute-force approach for the computations is very expensive, it was important to incorporate computation reduction a technique which would help in decreasing the number of computations. For this, the Sub-Vector Quantization (SubVQ) technique employed in Sphinx was chosen. Implementation details of the hardware design of this technique are provided. Finally, with the use of a top-level block diagram, the integration of SubVQ into brute-force Acoustic Modeling is discussed. The various phases are discussed in detail.

Chapter 5 is devoted to a completely new theoretical technique, *bestN*, proposed in this thesis for helping reduce the number of computations significantly. While the SubVQ technique helps reduce the computations by 60-70%, it imposes a significant overhead thereby reducing the overall gain in reduction of the number of computations. For this reason, a new technique, which represents the input as a function of multiplies of standard deviation away from the mean of a Gaussian is proposed.

It is shown that for a 0.1% decrease in the word recognition accuracy over the baseline performance of 3% Word Error Rate for a RM1 dictionary, significant computation savings can be achieved. By requiring only 8-bit addition operations, instead of 32-bit multiply/add operations, the bandwidth and computation complexity can be reduced significantly. Further, since this technique is based on looking-up pre-computed values at run-time, it is ideally suited for hardware-based implementation.

In Chapter 6, the implementation details for the Phone and Word Modeling blocks is provided in detail. Since the computations in these two blocks are relatively more data-movement intensive, data-flow is explained through the use of block diagrams. A description of the generation of the feedback from the Phone to the Acoustic Modeling block is presented.

Finally, the working of the Word block is described along with a dynamic memory management scheme that allows for the management of active data corresponding to active phones.

The results obtained on implementing the system based on the architecture described in Chapter 3 are discussed in detail in Chpater 7. It is shown that the system was successfully implemented and tested in simulation for a 15-word test dictionary. For testing purposes, a MATAB model of the computations was made. It was ensured that the core computations in the AM and PHN blocks were accurated with Sphinx's computations. Then, the MATLAB models were used as the reference to test and debug the computations and the entire system.

It is shown that for a change in the word dictionary, since the only data-structure that would to be modified is the word database, a full 1,000-word RM1 dictionary can be implemented using the current design. Further, it is shown that the system would perform in real-time at 100 MHz.

It is also shown that the system can be implemented using 93,000 equivalent gates thereby making it a good candidate for a co-processor based implementation. The benefits of incorporating feedback are also discussed and a huge reduction in the number of overall computations is shown for the 15-word test dictionary.

In conclusion, it is shown that by implementing a fully pipelined design using dedicated FIFOs for information exchange, shared memories for data exchange, and processing data by non-overlapping phases, a highly efficient system has been designed and implemented.

2.0 LITERATURE & ALGORITHM OVERVIEW

2.1 LITERATURE OVERVIEW

Given the vast applications of recognition of speech by machines, speech recognition has been a hot topic of research for the past 50 years. Until the past decade, focus tended to be more towards signal processing issues. Today, the effort is more focused towards artificial intelligence issues. Specifically, understanding and mimicking the process the human brain goes through when it interprets speech. This exploration has been greatly helped with the enhancement in computational power providing researchers the ability to create and analyze data quickly and efficiently. Developers use software for the design, development and testing of algorithms with different setups because software provides the necessary flexibility and ease of use.

However, the software based approach is not just limited to research and development. The final system that is deployed in the field also tends to be software based. Most, if not all research and commercial state-of-the-art systems available today are software based. Whether it be commercial system slike the Dragon Naturally Speaking [12], IBM's ViaVoice [13]; or research systems like Sphinx [6], developed at Carnegie Mellon University, HTK from University of Cambridge [14], they are all software-based systems.

Very limited work has been done to-date towards the development of dedicated hardware for speech recognition [10,24-26]. This is a direct result of the speech community's focus towards the development of algorithms using software development and implementing them on General Purpose Processors available in the market.

This approach while helpful in research and development of the system, has the major drawback of being based on General Purpose Processors built for applications in general and the architectural limitations posed by them. Hence, as described in Chapter 1, computationally intensive applications speech recognition have a poor performance running several times slower

than real-time. Part of this trend can be attributed to be a direct result of the fact that development of dedicated hardware is a time-consuming and expensive process. However, once done, the overall performance of the system can be much higher resulting in rapid wide-spread deployment of these systems.

Further, because of the lack of process technology until a copule of years back, the transisitor packing desnsity of 130um and higher is significantly less. This means that any dedicated hardware design would consurme a greater percentage of the overall chip resources thereby making it difficult for such designs to be incorporated pratically into commercial systems. For this reason, systems with very limited processing power have been proposed.

Ananthraman, et al [24] was one of the first people to implement dedicated hardware for speech recognition. They implemented a custom multiprocessor architecture for improving Viterbi beam search. This work however was done in 1986 and therefore, used simple statistical models when compared to todays state-of-the-art systems available today.

Binu Mathew et. al [10] proposed a Gaussian co-processor devoted to the computation of the computationally intensive Gaussian probabilities. To address the large memory bandwidth requirement and high cache miss rate in AM, they proposed the computation of several speech frames simultaneously so that rather than iterating over the same data (mean/variance pairs) for successive frames, the data would be accessed once and the computation for several frames of speech computed together. While this would help decrease the amount of memory accesses made, the system would need to perform all Gaussian computations because of non-availablity of feedback from previous stages thereby making it extremely computtionally intensive for embedded devices.

They also performed an extensive analysis on the cache size required for processing data in real-time and conclude that considering a 14 MB data size, an 8 MB cache would be required to get satisfactory performance. Further, on hand optimizing the implementation, they could almost double the IPC of Gaussian calculations from 0.59 to 1.1.

Melnikoff, et. al [25] proposed using multiple FPGAs for performing speech recognition. They implemented the computations that take place in a HMM which were made of continous gaussian distributions. While this was a big step forward, the overall system performance was basaed on 49 mono-phones and 634 bi-phones as compared to Sphinx’s 30k tri-phones. Sergui et. al [26] proposed a hardware implementation approach that could process 10’s of words.

Today, the only dedicated chips for speech processing are available from Sensory Inc. [15]. They have two families of chips RSC-4x and SC-6x dedicated for this purpose. The RSC-4x family [16] is a 8-bit speech optimized microcontroller dedicated towards speech recognition, speaker verification, and speech synthesis for use in low cost consumer applications like toys. The SC-6x family [17] is dedicated towards high quality speech synthesis. These chips have the ability to recognize up to 60 words.

Given that these chips are based on a 8-bit microcontroller architecture with a 12.32 MIPS DSP processor, it is highly unlikely that the system is robust. It is good for cheap applications like toys where the task is simple, but for more complex tasks of speaker-independent continuous recognition, a chip providing the ability to perform full blown computations and the necessary memory management support is needed.

With advancements in fabrication technology presently using 90nm processes and with 65nm processes on the horizon, the ability to pack more transistors is increasing significantly. This in turn implies that the necessary computational resources can be packed into the chip without a significant increase in the silicon area required to implement the design thereby keeping the cost of the chip at a minimum.

Keeping this in mind, the design of one such system is the focus of this thesis. The design and implementation of this architecture is discussed in the following chapters. It is shown that the entire design based on full-precision computations can be implemented using 93,000 equivalent gates using 90nm technology. This implies that the design has a small foot-print and can be incorporated as a dedicated co-processor into exisiting systems without signifiicant increase in the cost of the system.

2.2 THEORY & ALGORITHM OVERVIEW

In this section the necessary theory, mathematical equations and terminologies are introduced that will help understand the architectural design and implementation issues of the system covered in subsequent chapters. To obtain a working knowledge of the system, a simple top-level system representation with the major blocks is shown in Figure 2.1. As mentioned in Chapter 1, the computations have been broken up into three major blocks each relating to

Acoustic, Phonetic and Word Modeling. The figure includes a view of the basic data structures that are part of the each of the blocks.

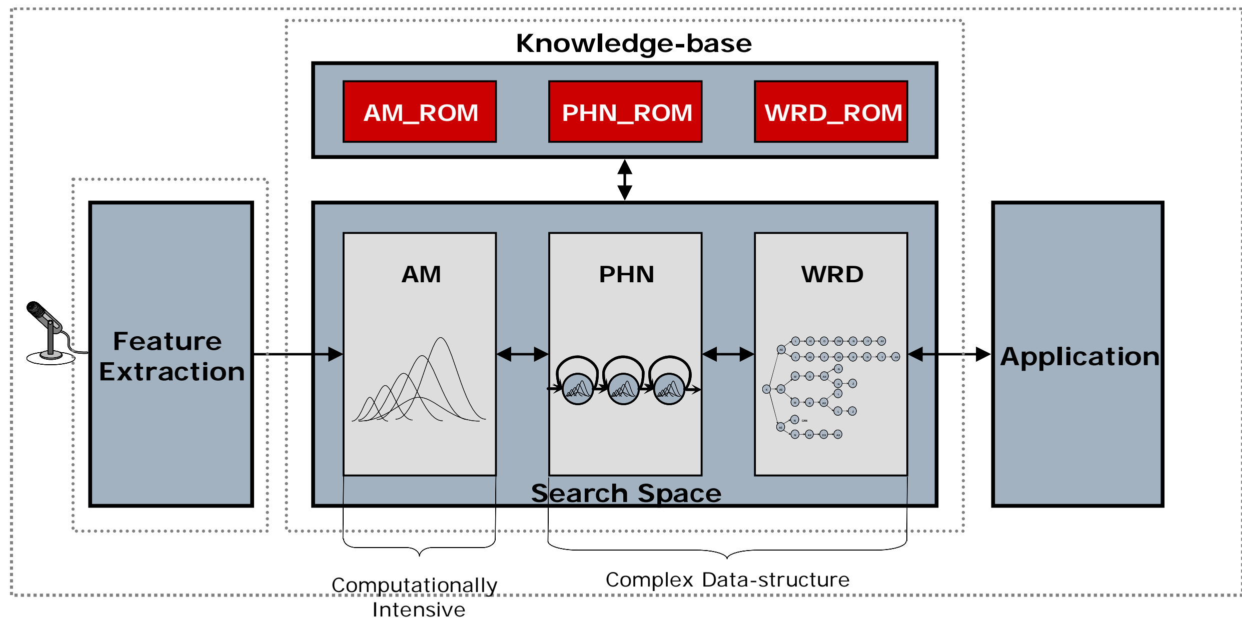


Figure 2.1: Conceptual view of a Automatic Speech Recognition System with AM, PHN, WRD Blocks

As stated in Chapter 1, Acoustic Modeling (AM) deals with the evaluation of Gaussian Probabilities and therefore its database is entirely composed of mean/variance pairs (AM_ROM). Phone Modeling deals with the evaluation of a statistical model towards deciding the entry and exit of individual phones from one to the next. As shown in the figure, it essentially deals with state-tracing (3-states for this research). The phone database (PHN_ROM) consists of this state information for each phone. Finally, words are represented as a sequence of phones and hence this sequence is maintained in the word database (WRD_ROM).

The following three sub-sections are devoted to each of these three blocks and a brief description of the underlying data structures and computations performed. The necessary terminologies are also introduced for each of the blocks and will be used throughout the remainder of the thesis.

The description of each of the blocks is described in a top-down manner (right to left of the search space in Figure 2.1) whereby the biggest sound units, Words, are described first. This is followed by sub-word units called Phonemes (Phones) followed by sub-phonetic units comprising of a mixture of multi-dimensional Gaussian distributions which make up the Acoustic Modeling is described.

2.2.1 Word Block

As can be imagined, words are the biggest sound units of any language. As stated in Eq. 1.2, Automatic speech recognition deals with searching for which word amongst a pre-defined set of words was spoken for a given sequence of speech samples. The pre-defined set of words is said to part of a *dictionary* of words which the system is trained to recognize.

Typical dictionary sizes range from 10 to 64k. 10 words can be used for trivial tasks such as digit recognition, while a relatively more complex task would be command and control based, having dictionaries anywhere in the range of 300 to 1,000 words. The recognition process deals with recognizing specific sequences of spoken word utterances. The most challenging task is that of 64k words dealing with the recognition of naturally spoken speech which tends to be irregular in nature.

This thesis focuses on a 1,000 word Command & Control based application. For this, a standard Speech Corpus, Resource Management 1 (RM1) [9], dealing with naval commands is used. All statistical models used are based on this corpus and extracted from Sphinx 3.3.

A word can be thought to be a sequence of basic sounds units called *phonemes*. Every phoneme has a unique frequency characteristic associated with it. The English language is made up 40 such phonemes. To make the recognition process easier, machine based speech recognition also represents the words in the dictionary as a function of sequence of phones. A sample 8-word dictionary depicting this is shown in Figure 2.2.

WORD	PRONUNCIATION
CALEDONIA	K AE L IX D OW N IY AX
CALIFORNIA	K AE L AX F AO R N Y AX
CAMDEN	K AE M D AX N
CAMDEN'S	K AE M D AX N Z
CAMPBELL	K AE M B AX L
CAMPBELL'S	K AE M B AX L Z
CAN	K AE N
CANADA	K AE N AX DX AX

Figure 2.2: A 8-word Sample Dictionary with its Phonetic Representation

Hence, in finding the most likely word that was spoken, the recognition process involves traversing through each of these sequences of phones and picking the one with the highest probability. However, it can be seen from the above figure that the number of total phones that need to be evaluated is fairly large. Typically, a word is comprises of a sequence 5-6 phones on average. This implies a total of in excess of 300k individual states for a 64k dictionary.

The calculation of such large amounts of data is not practically possible and therefore optimizations need to be made. One way is to combine as many similar states as possible. Since similar sounding words tend to be made up of the same phone sequences, it is a standard practice to combine the beginning of such words [18]. This results in a tree structure shown as in Figure 2.3 for the 8-word sample dictionary of Figure 2.2.

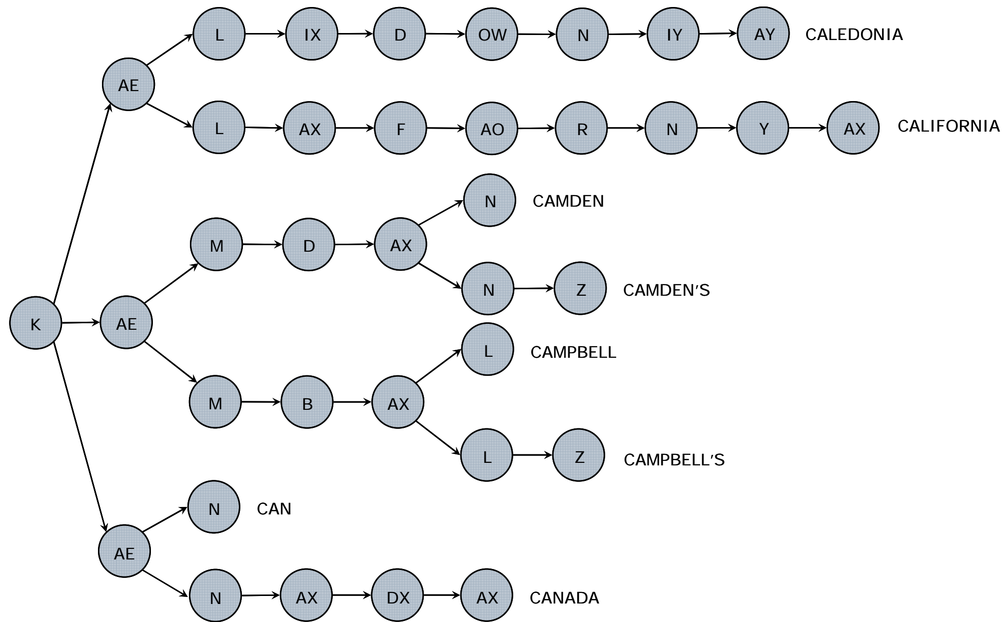


Figure 2.3: Word Tree-structure corresponding to the 8-word Sample Dictionary

The partitioning of the tree is done is shown in Figure 2.2. Since the models used are based on recognition of continuous speech, phones are treated to be similar only if the previous, current and the following phones are the same. For this reason, although the 2nd phone in the entire dictionary is /AE/, because of different right phones they are considered different and therefore represented as different branches in the tree. The full RM1 dictionary comprises of approximately 49k phones (circles) shown in Figure 2.3.

From an implementation perspective, a tree structure while reducing computations poses a huge data management challenge. In software, this tree structure is implemented as a linked-list of linked-lists with extensive use of pointers. Details of how this structure is mapped into hardware are discussed in Chapter 6. Further, since for any given word only the branch corresponding to that leaf needs to be active, a data management of active words needs to be done. This is done as part of the Word Block.

2.2.2 Phone Block

Phonemes (or phones) are the basic building blocks of any language. Every phone has a unique frequency characteristic associated with it. Not just frequency, since every speaker has his/her own speaking rate, this leads to a fair amount of variability in the duration of the utterance of each phone from speaker to speaker. Typically, a phonetic unit is spoken for 5-7 speech frames (50 to 70ms). To account for both frequency and time variability, Hidden Markov Model (HMM) [7,19] based statistical models are used for representing phonetic data.

To provide better resolution of the frequency characteristics, every phone can be broken up into states. Each state consists of *acoustic*, sub-phonetic, information representing the being, middle and end frequency characteristics of the phone. There are several HMM topologies in use for different applications. Sphinx uses a simple 3-state Bakis topology. A figure depicting this is shown in Figure 2.4. Apart from the 3-states of the HMM, the Figure also shows a final, dummy exit state in red. This dummy state has no statistical significance. Each state in the HMM model is associated with two things – a statistical distribution of data and the timing information related to staying in the current state and to move to the next one.

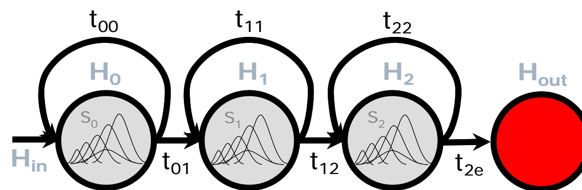


Figure 2.4: A Simple 3-state HMM Model

To account for variability across speaker, age, gender and speaking rates, statistical models are used to represent the distribution of data. Therefore, as shown in the figure, each state

is represented as Gaussian distribution. Details relating to this acoustic model *(S_0, S_1, S_2)* are provided in the following section. The timing information for each state consists of the probability of transitioning from the current state to the next and is referred to as *transition probability*. Only two types of transitions are possible – either staying in the current state itself, or moving on to the next state represented by transition probabilities $t_{(n)(n)}$ and $t_{(n)(n+1)}$ respectively, for a given current state, *n*. Therefore, HMM state H_n is represented by the state distribution S_n and transition probabilities $t_{(n)(n)}$ and $t_{(n)(n+1)}$.

There are two main computations involved at the phonetic level. The first deals with the computation of finding the probability of observing the input frames for a given phone model. The second deals with the computation of whether the phone can de-activated if the probability falls below a given threshold and whether the out score of the phone is good enough for the phone to have transitioned to the next phone.

2.2.2.1 Phone Score Calculations

The phone calculation (referred to as PHN_CALC) mainly deals with traversing the phone from the begin to the exit state. The process can be represented as a two dimensional trellis (Figure 2.5) with time progression on the x-axis and possible phone transitions on the y-axis. The 3-state HMM is shown in the figure for reference depicting the possible state transitions from the various states.

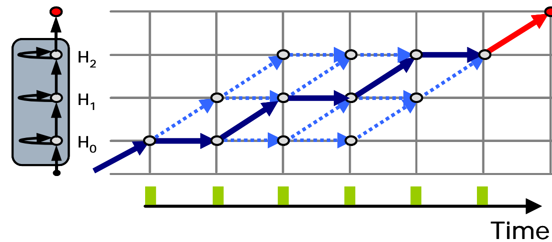


Figure 2.5: Phone State-tracing using Viterbi Algorithm

This figure shows all possible paths that can be taken over 6 sample frames. The paths shown with dotted lines represent potential paths that can be taken while the solid line path shows the actual path taken in this example. The red path indicates the time instant when the phone score is greater than the set threshold and therefore successfully exit the current phone onto the next one.

The actual computation relates to the computation of all scores of all possible paths. But this is extremely expensive since with every increment of time, t, the number of previous paths increases exponentially. Therefore, to simplify the computations, at any given time instant whenever two paths meet, Viterbi algorithm [20] is applied. This way only the "best" path at that time instant is taken and the other one is ignored. Based on this, the following computations are made to obtain the HMM scores for a given phone per frame:

$$\mathrm{H}_2(\mathrm{t}) = \mathrm{MAX}[\mathrm{H}_1(\mathrm{t-1}) + \mathrm{t}_{12}, \mathrm{H}_2(\mathrm{t-1}) + \mathrm{t}_{22}] + S_2(t) \qquad \text{Eq. 2.1}$$

$$\mathrm{H}_1(\mathrm{t}) = \mathrm{MAX}[\mathrm{H}_0(\mathrm{t-1}) + \mathrm{t}_{01}, \mathrm{H}_1(\mathrm{t-1}) + \mathrm{t}_{11}] + S_1(t) \qquad \text{Eq. 2.2}$$

$$\mathrm{H}_0(\mathrm{t}) = \mathrm{MAX}[\mathrm{H}_{\mathrm{in}}(\mathrm{t}), \mathrm{H}_0(\mathrm{t-1}) + \mathrm{t}_{00}] + S_0(t) \qquad \text{Eq. 2.3}$$

$$\mathrm{H}_{\mathrm{out}}(\mathrm{t}) = \mathrm{H}_2(\mathrm{t}) + \mathrm{t}_{2\mathrm{e}} \qquad \text{Eq. 2.4}$$

where;

- ♦ t and $t\text{-}1$ represent the current and previous frames respectively,
- ♦ $H_n(t), H_{n-1}(t-1)$ represent the HMM state score for the current and previous frame of state n,
- ♦ $H_{in}(t), H_{out}(t)$ represent the exit score of the previous phone and the exit score of the current phone respectively,
- ♦ t_{00}, t_{01}, t_{11}, t_{12}, t_{22}, t_{2e} represent the transition probabilities from one state to the next (the final state, dummy state is represented as state *e*), and
- ♦ $S_n(t)$ represents the Senone Scores of state *n* for the current frame.

These are the core calculations that need to take in the Phone Calculation block. Since previous HMM scores for each state is required, sufficient memory needs to be allocated so as to store all the scores. Finally, all computations consist of 32-bit integer addition, compare, addition operations. Because of limited data dependency, the evaluation of each of the HMM states, H_n can be executed in parallel.

2.2.2.2 Phone Pruning

Having performed the Phone score calculation, depending on the scores phones can be pruned or propagated. These computations are described in this section. For pruning/propagation of a phone, three thresholds relating to de-activating phones, propagating phones and propagating words, HMM_TH, PHN_TH, WRD_TH respectively need to be computed. To

account for the large variability in speech because of varying speaker and environmental conditions, relative rather than absolute scores are used when performing pruning and propagation operations. By performing *beam pruning* [21,22], any value lying within a predefined *beam* from the *best score* for that frame is considered to have passed the *threshold*.

$$THRESHOLD = BEST_SCORE + BEAM \quad \text{Eq. 2.5}$$

Prune if, $SCORE < THRESHOLD$ Eq 2.6

The calculation of HMM_TH relates to finding whether the phone scores have gone so low that this phone cannot be the likely phone being spoken. Once the score goes below this threshold the phone can be de-activated implying that the branch corresponding to this phone will not be processed further. This way, only promising phones are kept active in the system thereby helping keep the number of computations required to a manageable size. The threshold used to prune phones this way is referred to as HMM_TH. The computation of this threshold is given by

$$HMM_TH = B_HMM + HMM_BEAM \quad \text{Eq. 2.7}$$

where, *B_HMM* is the "best" HMM Phone score, H_n, of all states over all active phones and HMM_BEAM is a pre-defined constant. B_HMM can be represented as $MAX[H_{best}(t)]$, where $H_{best}(t) = MAX[H_0(t), H_1(t), H_2(t)]$ over all active phones in the current frame.

For finding if the active phone can be considered to have successfully propagated into the next phone, PHN_TH is calculated and is given by

$$PHN_TH = B_HMM + PHN_BEAM \quad \text{Eq. 2.8}$$

It can be seen that the computation of HMM_TH and PHN_TH differs only in the BEAM applied to them.

The last threshold that is calculated relates to the propagation of the valid word exits. For determining whether a word has indeed successfully exited, the final phone that makes a word, word-exit phone, is checked against the WRD_TH which is given by

$$WRD_TH = B_HMM_wrd + WRD_BEAM \quad \text{Eq. 2.9}$$

where, B_HMM_wrd represents the best HMM Phone scores for all active word-exit phones in the current frame.

2.2.2.3 Context-Dependent Phones

Although in English there are 40 phones, it has been found that continuous speech suffers from co-articulatory effects whereby the adjacent phonetic sounds have no clear boundaries and therefore the frequency representation of such sounds cannot be accurately represented by just 40 basic phones. Hence, it is necessary to consider the context in which the phone is being uttered.

For this reason, taking both the left and right context contexts gives a more accurate representation of continuous speech and therefore gives a superior performance [23]. This leads to the presence of *context-dependent* phones. These are also referred to as *tri-phones*, because of the left and right contexts in addition to the basic phone.

For superior performance, Sphinx makes use of such context-dependent phones. While the inclusion of such phones does enhance the overall performance of the system, given the total number of possible different left and right phonetic contexts, the number of phones in the database is significantly more than 40 basic phones.

In total, there are 30k tri-phones for the RM1 Corpus. The only characteristic that differs one from the other is the statistical distribution of the individual states themselves. Therefore, only those phones are retained which have unique state data. This results in obtaining only 5605 phones with unique sequences of acoustic state data. These phones with the state ID information form the major data structures for the PHN_ROM.

2.2.3 Acoustic Modeling Block

2.2.3.1 Senone Scoring

Acoustic Modeling, as the name suggests deals with sub-phonetic, *acoustic* sounds. As stated in the previous section, phones are modeled as begin, middle and end states where each state has a distribution associated with it. The distribution of these acoustic sounds is found to be accurately represented by a Gaussian distribution. The computation of these Gaussian probabilities is handled as part of the Acoustic Modeling block. The data distribution of each HMM-state is referred to as a *Senone* in Sphinx terminology.

However, because of tremendous variability in speech corresponding to speakers with different age, gender, dialect and speaker rate, the data is very widely distributed. Representing data with such wide data distributions by a single Gaussian distribution results in undesired

smoothening of data. To account for this, instead of using a single Gaussian, multiple, or a *mixture* of Gaussians are used to represent the data more accurately. In Sphinx terminology, each Gaussian is referred to as a *Component*.

Hence, a Senone can be thought of being composed of a mixture of several Components, each with its own set of mean/variance pairs. Typically between 2 to 64 components are used in research systems. Depending on the frequency of occurrence of trained data and how many samples correspond to each Component, a weighting factor is assigned to each component. This weighting factor is known as the *Mixture Weight*. The final Senone score is represented as a summation of all weighted Component scores, and can be given by the following equation.

$$Senone_Score_s = \sum_{c=1}^{C} \left[Mixture_Weight_{s,c} * Component_Score_{s,c} \right] \quad \text{Eq. 2.10}$$

Since the feature vector output from Feature Extraction is a 39-dimensional vector, the data representing the various dimensions also needs to be 39-dimensional. From probability statistics [8] multi-dimensional Gaussian probability for Senone, s, Component, c, can be given by,

$$Component_Score_{s,c} = \frac{1}{\sqrt{(2\pi)^D \left|\sigma_{s,c}^2\right|}} e^{-\sum_{d=1}^{D} \left[\frac{(x_d - \mu_{s,c,d})^2}{2\sigma_{s,c,d}^2} \right]} \quad \text{Eq. 2.11}$$

where;

x: Input

μ: Mean

σ: Variance

$\left|\sigma_{s,c}^2\right| = \sigma_{s,c,1}^2 * \sigma_{s,c,2}^2 * \ldots\ldots \sigma_{s,c,D}^2$

The RM1 statistical models of Sphinx used for this research consists of 1935 Senones, S, each comprising of 8 Components, C, consisting of 39 dimensions, D. Since the computations at this stage are completely related to Gaussian probability evaluations, the entire AM_ROM comprises of only mean/variance pairs. The data can be thought of being 3-dimensional and can be conceptually visualized as shown in Figure 2.6.

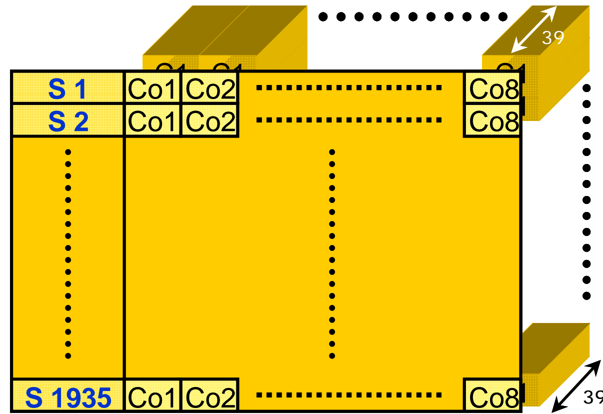


Figure 2.6: A 3-Dimensional view of the Acoustic Modeling Database

There are 1935 rows corresponding to each Senone which is in turn made up of 8 Components each of which is a 39 dimensional Gaussian (represented by the 3rd dimension). Given this data size, there are a total of 603,720 mean/variance pairs in the system. All operations in software are 32-bit floating point operations. The total number of operations performed is approximately 2 million per 10ms frame and therefore requires 200 million floating-point operations per second for a simple 1000 word RM1 command and control word dictionary.

2.2.3.2 Composite Senone Scoring

While a single Senone is sufficient in representing the state information for phones with-in a word, cross-words present a new challenge. Since at the end of a word, several possible words can be spoken, the complexity of the problem while using context-dependent phones increases significantly. The problem stems from the fact that in cross-word utterances for a given left-context, the number of possible base-contexts and the right-contexts can lead to a huge permutation and combination.

Figure 2.7 illustrates one such situation. Assuming that the word CALEDONIA has been uttered, there are three next possible phones corresponding to the three different right-contexts possible for this example dictionary. To account for this situation, researchers try to simplify the problem by representing this data structure by a single phone, with a slight difference.

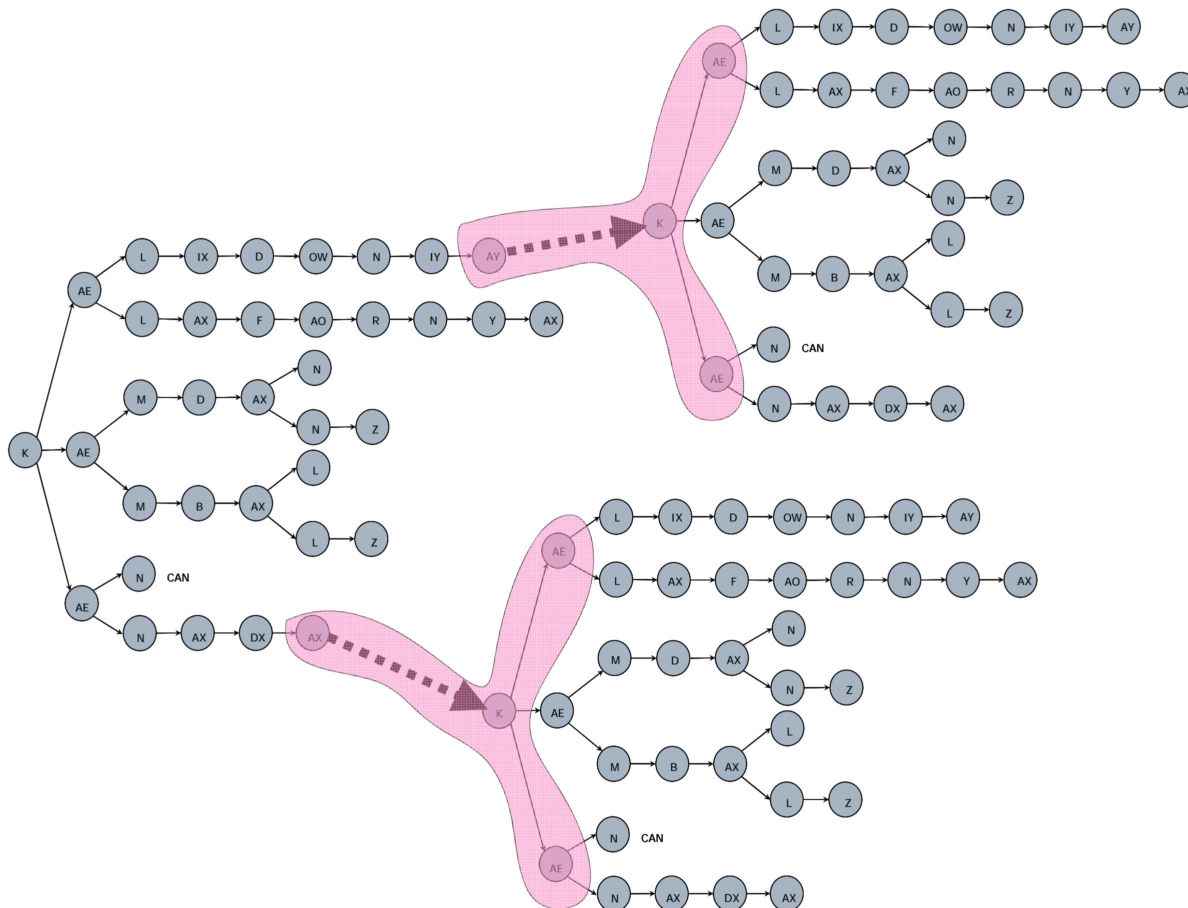


Figure 2.7: Cross-word Phone Modeling using Composite Senones

The HMM states instead of being represented by a single Senone distribution is considered to be composed of all the Senone distributions that are possible in different context positions. Since the HMM state is now “composed” of several individual Senones, this gives rise to a new data structure referred to as *Composite State* or *Composite Senone* in Sphinx terminology,. Hence, phones corresponding to cross-word pronunciations can be represented as shown in the figure below.

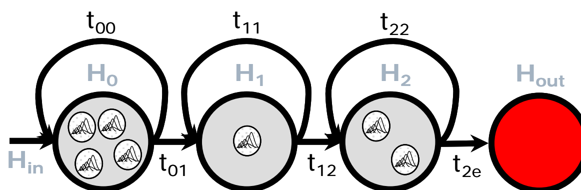


Figure 2.8: 3-state HMM for a cross-word Phone

Except for the states, the basic structure of these phones remains the same. It is useful to note that the number of Senoenes that makeup a Composite Senone is variable depending on the dictionary. Since a Composite Senone comprises of Senones, the evaluation of the Composite Senone is dependent on the evaluation of the Senone Scores themselves. The Score of a Composite Senone is the score of the best scoring Senone of that Composite State.

3.0 SYSTEM DESIGN

3.1 INTRODUCTION

Having both the necessary knowledge of the theory of various algorithms employed in Speech Recognition and a thorough understanding of the Sphinx 3 software along with its data structures, System level design work could be commenced. This chapter is fully devoted to the Design and Development process of the System Architecture.

In the following section, a summary of the Design Methodology followed during the development of the system is presented. This includes laying down the system requirements followed by a timing and memory size & bandwidth analysis of the system. TAfter this, system level design work could be commenced. A simple top-level block diagram of the system is described in Section 3.3.1. This is followed by basic phase partitioning details in Section 3.3.2. Some of the considerations towards the phase partitioning are described in detail. Following this, the system architecture with detailed top-level and detailed system flow diagrams are presented in Section 3.3.3. The description of the system is limited to the top-level. Implementation details of the individual blocks are provided in subsequent chapters.

3.2 DESIGN METHODOLOGY

3.2.1 System Requirements

During the conceptual phase of the project, two major requirements were set. Firstly, the entire system be able to process all data in real-time at a 100 MHz. A 100 MHz clock frequency implies a 10ns clock period. Given that speech frames are computed every 10ms intervals, this

implies that there are 1 million cycles available given a 10ns clock period. In other words, the requirement can be restated as follows: For any given 10ms frame of speech, all computations should be completed within 1 million cycles, assuming a 10ns clock period.

Secondly, the system would be implemented and demonstrated on a prototyping platform. Hence, at every stage of the design process, it was ensured that these two requirements would be met. To ensure this, a detailed analysis of the algorithms was made and detailed area, memory and cycle-budget analysis was done.

3.2.2 Timing and Resource Requirements

The first step in the design process was to analyze the computational resource requirements. A detailed analysis of all computations at every stage of the system based on the algorithms and data-structures presented in Section 2.2 was performed. A table summarizing the number of math and comparator units, bandwidth requirements and the number of cycles required by each of the stages is shown below.

Table 3.1: Timing and resource requirements for the entire system

		AM	PHN	WRD	TOTAL
Math Units	Add	6	9	1	**16**
	Multiply	3	-	-	**3**
Comparator Units		3	6	2	**11**
# of cyles [per 10ms frame]		603,720	8,192	102,400	**714,312**
Memory Bandwidth [MB/sec]		495	-	5	**-**

The number of cycles presented in this table is based on the assumption that all computations are completely pipelined. Therefore, the number of total operations performed in each of the blocks is more than that presented above. 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. Hence, to account for the loss in parallelism, the computation latency (estimated at a worst case of 10 cycles) has been accounted into the projected cycles required by the WRD Block.

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, that a maximum of 4000 phones were simultaneously active. Based on this analysis a worst case estimate of the number of cycles required for the computation is presented in the table.

In the PHN Block since two iterations corresponding to phone calculation and phone prune/propagation are done, the number of cycles required by PHN is equal to twice that of 4k. The number of computations in the WRD Block presented are based on a worst-case assumption that at any given time only half of the phones will propagate. Coupled with this, an average branching factor of 5 is assumed along with a 10-cycle pipeline latency. Based on these assumptions, the total number of cycles required is almost equivalent to 100k cycles.

From the table it is evident where lies the computational bottleneck of the system. Acoustic Modeling, with 603k Gaussians requires an equivalent number of cycles to complete the computations at this stage. As stated in Chapter 1, Acoustic Modeling accounts for 55-95% of the computations in any modern Speech Recognition System. In the above system as well, almost 85% of the number of computation cycles are devoted to the AM Block. Because of the sheer number of values that need to be evaluated, 603k 32-bit mean/variance pairs per 10ms frame, 500 MBytes/sec of memory bandwidth is required by this block. Hence, it is essential to incorporate computation reduction techniques that significantly reduce the number of computations. The best way of achieving this is to incorporate feedback that enables the system to calculate only those Senones that correspond to active phones. Further, it can be seen that the WRD Block is the least computationally intensive block with just 1 adder and 2 comparators.

3.3 SYSTEM ARCHITECTURE

The two major parts of the system architecture, namely Block and Phase partitioning is presented in the following two sub-sections. The description is based on the “conceptual” system outlining some of the decisions made during the architecture design process. Having basic knowledge of the functioning of the system, a much more detailed description is presented in Section 3.3.3 with clear description of the various blocks at the top-level and the sub-phases.

3.3.1 Top-Level Block Partitioning

From the characterization of Speech Recognition algorithms it is clear that there are three different kinds of computations in Speech Recognition. Each of these computations provides its own challenges. From Table 3.1, while Acoustic Modeling is both computationally and bandwidth intensive requiring several million floating-point operations per second, Phonetic Modeling is relatively less computationally expensive with fewer evaluations but needing several math units and Word Modeling comprises of tracing a tree structure. Therefore, based on the characterization of each of these computations, the system has been partitioned into three main blocks; AM, PHN and WRD.

A Top-Level Block Diagram of the system showing the major blocks with the basic data flow details is shown in Figure 3.1. Apart from the three major blocks AM, PHN, WRD, the figure also shows the major memory data structures namely SENONE_RAM and PHN_WRD_RAM. This along with the SAL_GEN Block and FIFOs shows the basic data movement information. The dotted lines in the diagram serve for showing the clear demarcation of the blocks.

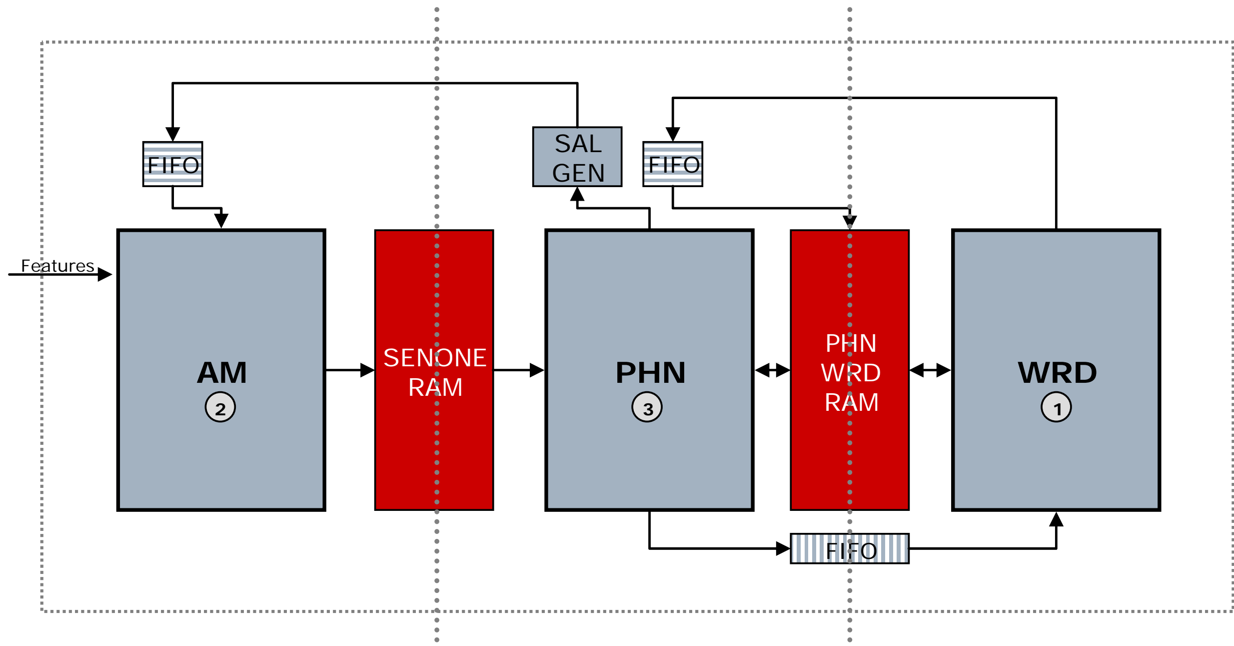


Figure 3.1: Top-Level System Block Diagram

From Figure 3.1, it is clear that the SENONE_RAM and PHN_WRD_RAM data is shared between AM-PHN and PHN-WRD respectively. SEONOE_RAM is a single port memory whereby Senone scores computed by AM are written into it which is then accessed by PHN for the computation of Phone Scores. PHN_WRD_RAM is the other major data structure to which both PHN and WRD read and write into. PHN_WRD_RAM is essentially a phone data structure that contains all the phone scores, H_{in}, H_0, H_1, H_2, H_{out} and H_{best}, the PHN_ID and minor word information relating to whether this phone is a word-ending phone (WRD_END) and the Word ID (WID) for every active phone.

The overall system flow is maintained by using dedicated FIFOs. This is the only form of communication available for the individual blocks to communicate with other blocks. All FIFOs consist of only address pointers into the various data structures while all data sharing is achieved by sharing the SENONE and PHN_WRD RAMs. At the PHN and WRD level, these address pointers are referred to as Tokens (TKN). At the AM level these pointers are IDs for the values that need to be evaluated by that block which are generated by the SAL_GEN Block.

3.3.2 High Level Phase Partitioning

A conceptual phase partitioning is shown in Figure 4.2 below. The first phase deals with the activation of the Phones that are generated by the WRD Block. Since Senone score are required for the computation of Phone scores, AM Block is processed which is finally followed by the PHN Block whereby Phones are scored and pruned/propagated as appropriate. This is how the overall system works.

Further, it can be seen that the phases are sequential in nature. This is because while partitioning the different phases special care was taken to ensure that there was no overlap. Although such an approach has a down side because of lost parallelism which would otherwise be useful in decreasing the total number of cycles required to complete the necessary operations every frame, there are certain key benefits from such an approach.

Firstly, since every block is using some sort of feedback from its preceding block, it is necessary to ensure that all data has been processed by the current stage fully. This helps ensure that the assimilation of data corresponding to active data-structures takes place accurately and without the need for un-necessary increase in system complexity.

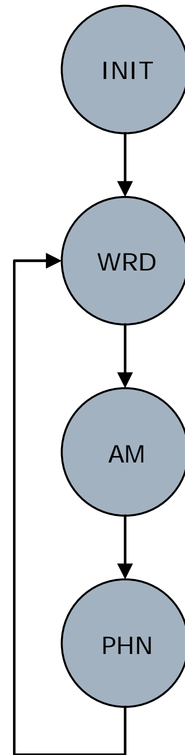


Figure 3.2: Conceptual Phase Partitioning of the System

Secondly, there are no race conditions between the various blocks requiring the use of common resources. Each block has dedicated access to the other block(s) as necessary for performing the desired computation at every stage. This not only helps eliminate any pipe-line stalls that might result due to resource-sharing, but also helps in not requiring arbitration towards which block has access to which resource at what point in time.

From a design and implementation standpoint, phase overlapping would not only add more overhead, but would also make the design and debug process much more complicated for little gain in reducing the number of cycles required for the processing of each frame. Therefore, by eliminating overlap, a highly efficient, pipelined system with maximum throughput and performance can be implemented.

Further, non-overlapping phases imply that computation resources can be shared amongst the various blocks. Given that both the AM and PHN Blocks require 9 Math units each, resource sharing can help reduce the total number of Math resources required by the system into half if the computations by these two blocks could be separated. This not only allows for the device to have a smaller silicon foot-print thereby enabling it to be cheaper and more affordable, but would

also allow for a more practical implementation, especially towards incorporating the design as a co-processor to standard GPPs already available in the market thereby making them part of commercial systems.

3.3.3 Detailed System Architecture and Data Flow

Having understood the working of the system conceptually, detailed system level diagrams depicting the flow by breaking up the computations into sub-phases were created. This section summarizes the various phases and described from a top level perspective. A detailed top-level block diagram with all the major data structures as shown in Figure 3.1 along with addition data structures (FIFOs) is shown in Figure 3.3 so as describe the entire data flow. For this, the figure also contains details of the overall system flow and can be correlated with Figure 3.4 to gain a better understanding of system.

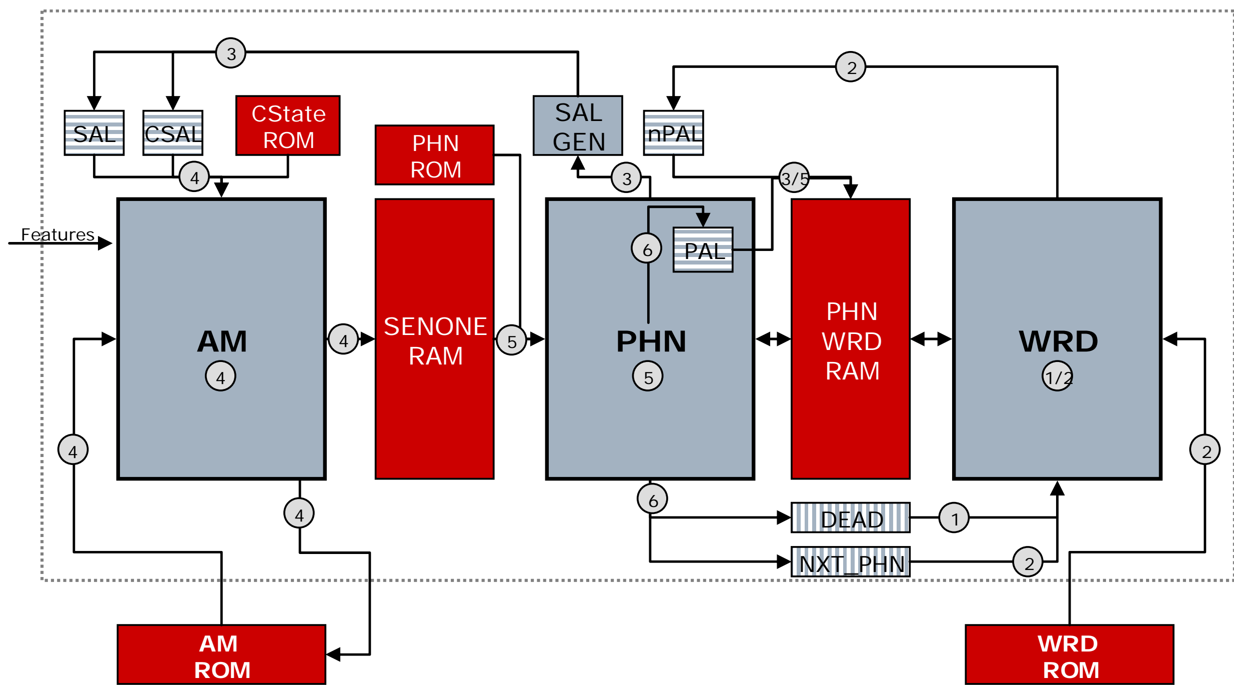


Figure 3.3: Detailed Top-Level System Block Diagram with data flow information

3.3.3.1 Feedback using Active Lists

Given that the total number of computations required for computing "all" data structures simultaneously is not practically feasible, the system needs to keep track of the "active" data. As

described in Section 3.3.1, since all information exchange between the blocks is done using FIFOs, address locations of active data is stored in the FIFO.

Since the data in the FIFOs corresponds to a list of "active" data in the system, this is referred to as Active List (AL). Depending on the quantity in question, pre-fixes are added to it – Senone (S), Composite Senone (CS), Phone (P) resulting in SAL, CSAL, PAL. It is important to note that the use of AL (based on feedback from previous stages) does not affect the accuracy of the system in any way. This is a direct result of the fact that in-active data is not processed at all.

The number of cycles required by AM in Table 3.1 is based on a brute-force approach. Since this approach might lead to a violation of the 1 Million cycle budget for 10ms of speech, given that access to external memory need not have a 1-cycle latency, it was necessary to use feedback using SAL and CSAL. This feedback helps reduce the number of Senone computations by as much as 50% on average. Not only does this AL allow in ensuring that the real-time requirement is met, but it also has helps in decreasing the power consumption of the design significantly.

3.3.3.2 System Data Flow

A detailed system flow diagram outlining the major phases is shown in Figure 3.4. The first step is the initialization of all the memory structures. Once completed, the system can start processing data.

The first data processing step is to clear the memory locations corresponding to the Phones that are that need to be de-activated or in other words, are dead. This deals with resetting memory locations in PHN_WRD_RAM pointed by TKNs in the DEAD_FIFO. After all TKNs are processed, propagation of Phones corresponding to TKNs in the NXT_PHN_FIFO is performed. The propagation deals with traversing the word tree structure from the exiting Phone to the next one(s) in the tree. For this operation, data is read from the PHN_WRD_RAM and WRD_ROM and locations corresponding to active Phones are updated in the PHN_WRD_RAM. If this phone was inactive in the previous frame, then it needs to be specified as an active one for the PHN Block to process it. This is done by passing the TKN to the new PAL FIFO (nPAL_FIFO). The processing of all TKNs in the NXT_PHN_FIFO indicates the completion of the computations at the WRD Block.

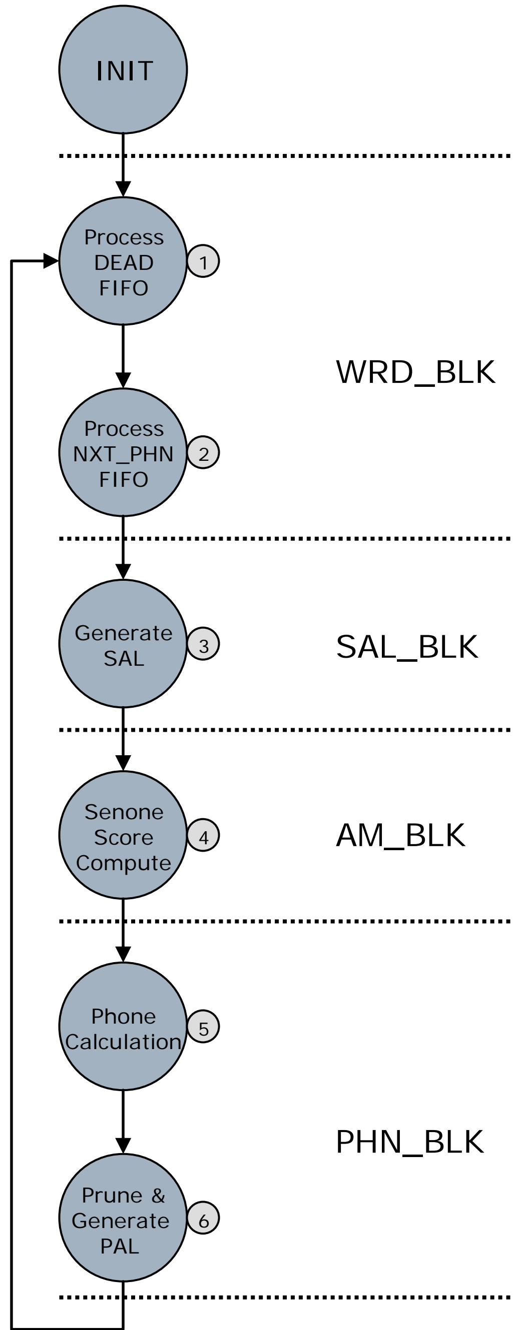


Figure 3.4: Detailed System Data Flow

Having obtained all the "active" phones in the current frame, the next step can be performed. This deals with the calculation of Phone Scores, which are in turn dependent on the Senone Scores. Therefore, Senone Scores need to be evaluated first. But as stated earlier, not all Senones need to be evaluated. Only those corresponding to the active Phones need to be evaluated.

For this, the Senone and Composite-Senone Active Lists, SAL and CSAL are generated. This is done by using the SAL_GEN Block. The SAL_GEN Block reads all the active Phones from the nPAL and PAL FIFOs and looks up the senones corresponding to the PHN_IDs of each phone from the PHN_ROM. The data is then sent to the SAL_GEN block which assimilates the active Senones and pushes them into the SAL and CSAL FIFOs.

With this, the Senone Score evaluation can proceed for the current frame. Since the presence of SAL can imply non-contiguous active Senones, SAL entries are used to generate appropriate addresses into the AM_ROM. This data along with the features from the FE phase are combined to compute the Senone Scores. The scores are finally written into the SENONE_RAM.

Once the Senone scores have been computed, Phones can be evaluated. The first step is the computation of the Phone scores. This deals with popping one TKN at a time from the nPAL_FIFO and later the PAL_FIFO. Data from the corresponding locations in PHN_WRD_RAM is used for accessing the PHN_ROM giving data pointers into SENONE_RAM. This data is combined with the scores for the previous frame and the Phone scores are computed. These scores are finally written back into locations from where they were read from in the PHN_WRD_RAM and "all" TKNs (from both nPAL and PAL FIFOs) are fed back into the PAL_FIFO.

Once the computation of all entries in the nPAL and PAL FIFOs are completed, the final phase corresponding to pruning and propagating can commence. The scores for each Phone is checked with the thresholds presented in Section 2.2.2. If $H_{best}(t)$ is greater than the threshold, the phone remains active and this TKN is sent into the PAL_FIFO, else into the DEAD_FIFO so as to be de-activated at the beginning of processing of the next frame. If the $H_{out}(t)$ of the Phone passes the threshold it can be propagated further for which it is sent to the NXT_PHN_FIFO. On processing all the TKNs in the PAL_FIFO, the processing for the current speech frame can be said to be completed. The same sequence of operations is repeated over all following frames.

3.4 SUMMARY

Having good knowledge of the computations taking place in each of the blocks along with the knowledge of the system architecture and the processing flow, implementation details are provided in the Chapters 4-6. Contrary to the system flow presented in Figure 3.2, the organization of the chapters is as follows: AM, followed by PHN and in turn followed by the WRD Block. This has been done so as to focus on the more computationally intensive blocks in descending order of computational complexity. Chapters 4 and 5 deals with the AM Block while Chapter 6 deals with the PHN and WRD Blocks. Finally, Chapter 7 provides the necessary integration, testing and area, performance numbers of the system implemented.

In Chapter 4, a brute-force implementation followed by a computation reduction technique, SubVQ, is described. To overcome some of the limitations imposed by present computation reduction techniques further analysis was done and a new technique called "bestN" is proposed. This technique helps reduce the bandwidth by a factor of 8 and requires 8-bit additions for a 0.1% degradation if word recognition accuracy. The bestN technique is described in Chapter 5.

In Chapter 6, the PHN and WRD Blocks are described. Apart from the basic functioning of the individual blocks according to the information presented in Section 2.2, the generation of the SAL/CSAL feedbacks using the SAL_GEN Block is described along with the incorporation of a dynamic memory allocation scheme that allows for managing the active phones. Both these implementations are critical to the system whereby the feedback mechanism enables a huge reduction in computation and power savings. The dynamic memory allocation mechanism allows for the implementation of a scalable system.

4.0 ACOUSTIC MODELING

4.1 INTRODUCTION

As described in Section 1.2.2, Acoustic Modeling contributes over $2/3^{rd}$ of the computational requirements of modern Speech Recognition Systems. The computations are dominated by the evaluation of multi-dimensional Gaussian probabilities towards the evaluation of Senone scores given by the following equation.

$$Senone_Score_s = \sum_{c=1}^{8}\left[Mix_Wt_{s,c} * Component_Score_{s,c}\right] \qquad \text{Eq. 4.1}$$

where each Component is a 39-dimensional Gaussian represented by,

$$Component_Score_{s,c} = \frac{1}{\sqrt{(2\pi)^{39}\left|\sigma^2_{s,c,d}\right|}} e^{-\sum_{d=1}^{39}\left[\frac{(x_d-\mu_{s,c,d})^2}{2\sigma^2_{s,c,d}}\right]} \qquad \text{Eq. 4.2}$$

The Senone score is an accumulation of the weighted Gaussian probability of the Components that make up that Senone. Considering that a total of 1935 Senones, each having 8 Components of 39-dimensions make up the database, the sheer number of iterations that need to be performed in the inner-most loop is enormous. Further, these operations require several multiply and add/subtract operations typically done in floating-point.

Further, exponentials also needs to be evaluated. In computation terms, the calculation of an exponential is several orders of magnitude more intensive when compared to even floating-point operations. This stems from that fact that most General Purpose Processors possess dedicated on-chip circuitry in the ALU for basic add/multiply operations while logarithm calculations, being less common, are not directly supported by hardware.

The calculation of an exponential on the other hand can either be calculated using an approximation of the Taylor Series expansion or by using look up tables. Both these techniques

are very expensive. While a Taylor series expansion requires several multiply and add operations [27], look up tables require huge amounts of storage space. The required memory space is directly proportional to the dynamic range of the values in question and the level of quantization required for the error induced to be below the desired threshold.

To address these issues, certain transforms need to be applied. One way of eliminating the exponential operation is to perform the computations in the log-domain. Applying logarithmic identities, an exponential with the base 'e' can be reduced to the computation of just the exponent part [28]. If the logarithmic properties are extended to the entire equation, then this would enable in reducing multiply operations into addition operations. In effect, the transformation of computations into the log-domain would result in considerably reducing the amount of computational power required to perform the calculations.

Hence, all the computations in speech recognition are performed in the log-domain. This conversion in the domain leads to the evaluation of *log-likelihoods* rather than *probabilities*. Taking $\log_e$ on both sides, Eq. 4.1 can be re-written as,

$$\log_e\left[Senone_Score_s\right] = \log_e\left(\sum_{c=1}^{8}\left[Mix_Wt_{s,c} * Component_Score_{s,c}\right]\right) \qquad \text{Eq. 4.3}$$

Taking the log inside the summation, we get,

$$\log_e\left[Senone_Score_s\right] = \sum_{c=1}^{8}\left[\log_e\left(Mix_Wt_{s,c} * Component_Score_{s,c}\right)\right] \qquad \text{Eq. 4.4}$$

$$\log_e\left[Senone_Score_s\right] = LOG\sum_{c=1}^{8}\left[\log_e\left(Mix_Wt_{s,c}\right) + \log_e\left(Component_Score_{s,c}\right)\right] \qquad \text{Eq. 4.5}$$

where, $LOG\sum_{c=1}^{8}$ represents the summation of values which are in the log-domain. This is strange looking representation is known as "log-add". More details on the log-add are provided in Section 4.2.2.

Now substituting for *Component_Score* in the above equation, we get

$$\log_e\left[Senone_Score_s\right] = LOG\sum_{c=1}^{8}\left[\log_e\left(Mix_Wt_{s,c}\right) + \log_e\left(\frac{1}{\sqrt{(2\pi)^{39}\left|\sigma^2_{s,c,d}\right|}}\, e^{-\sum_{d=1}^{39}\left[\frac{\left(x_d-\mu_{s,c,d}\right)^2}{2\sigma^2_{s,c,d}}\right]}\right)\right] \qquad \text{Eq. 4.6}$$

$$\log_e[Senone_Score_s] = LOG\sum_{c=1}^{8}\left[\log_e(Mix_Wt_{s,c}) + \log_e\left(\frac{1}{\sqrt{(2\pi)^{39}|\sigma^2_{s,c,d}|}}\right) + \left(-\sum_{d=1}^{39}\left[\frac{(x_d - \mu_{s,c,d})^2}{2\sigma^2_{s,c,d}}\right]\right)\right] \quad \text{Eq. 4.7}$$

Since the scores obtained have a very small dynamic range, the calculations are scaled by a constant factor so as to aid in differentiating the scores. For this, extensive research was done during the development of Sphinx at CMU. They arrived at a scaling factor of 3333.83 (related to $\log_{s3} e$, where $s3$ = 1.0003, the Sphinx-log-base).

To account for the scaling factor, both sides of Equation 4.7 need to be multiplied with $\log_{s3} e$. On applying the logarithmic identities and simplifying the equation, the log-base of the values changes to $s3$ and the final resultant equation is shown below.

$$Senone_Scr_s = LOG\sum_{c=1}^{8}[W_{s,c} + f * GAUS_DIST] \quad \text{Eq. 4.8}$$

where,

- $Senone_Scr_s = \log_{s3}[Senone_Score_s]$
- $W_{s,c} = \log_{s3}(Mix_Wt_{s,c}) = f * \log_e(Mix_Wt_{s,c})$
- $f = \log_{s3} e = 3333.83$
- $K_{s,c} = \log_e\left(\frac{1}{\sqrt{(2\pi)^{39}|\sigma^2_{s,c,d}|}}\right)$
- $\sigma'^2_{s,c,d} = \frac{1}{2\sigma^2_{s,c,d}}$
- $GAUS_DIST = K_{s,c} - \sum_{d=1}^{39}\left[(x_d - \mu_{s,c,d})^2 * \sigma'^2_{s,c,d}\right]$ Eq. 4.9

On closer observation it can be seen that Eq. 4.9 deals with the computation of the Gaussian Likelihood, Gaussian probability in the log-domain. It basically deals with the computation of the distance of the observed input, x_d, from the mean, $\mu_{s,c,d}$ scaled by the inverted variance, $\sigma'_{s,c,d}$. for a given Seonone, s, Component, c, and Dimension, d. The only variable in this equation is x_d, the input feature frame from the Feature Extraction phase.

The remainder of the calculation deals with the scaling of the Gaussian Likelihood followed by the addition of the weight of this Component amongst the Components making up the Senone giving Weighted-Component score. This is further accumulated over all Components

of the Senone to give the final Senone Score. This is the core computation that takes place at the Acoustic Modeling phase.

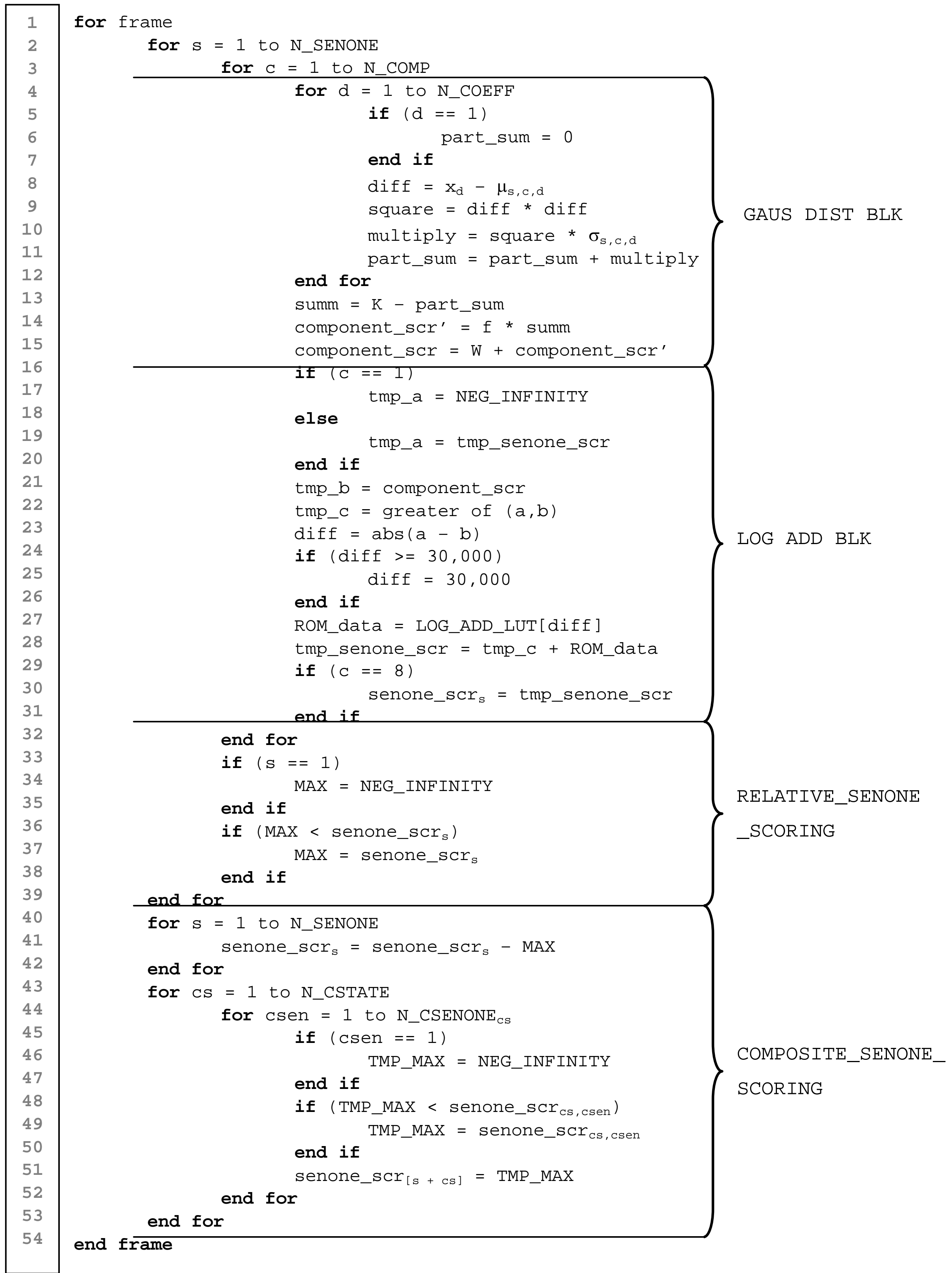
From Eq. 4.9 it can be seen that the Senone Score calculation, requires 2-multiply and 2-add/subtract operations. This is computationally intensive requiring several hundreds of millions of operations per second. In most state-of-the-art recognition systems, these computations are performed in floating-point. A table presenting the number of operations for various data sizes is presented below.

Table 4.1: Number of operations per second required for Acoustic Modeling

Speech Corpus	Dictionary Size (words)	# of Senones	# of Components	# of Gaussian Evals per 10ms (1,000)	# of ops per second (Million)
Continuous Digits [TI Digits]	12	602	4816	192.64	77.056
Command & Control [RM 1]	1,000	1,935	15480	619.2	247.68
Continuos Speech [HUB-4]	64,000	6,144	49152	1966.08	786.432

However, additional processing is required to account for the tremendous variability in speech. For this, “relative” rather than “absolute” scores give a more accurate representation of the observed values. For this reason, a running MAX of all Senone scores is computed every frame. All Senone Scores are normalized w.r.t. this MAX value. As a result of this normalization, the largest Senone Score obtained is zero. All others are some negative value.

The concept of “Composite Senones” was introduced in Section 2.2.3.2. Since the end result is a Senone Score, the input to the next phase, the computation of this quantity is implemented as part of the final phase of AM. A pseudo-code representation of the entire arithmetic of Eq. 4.8 including log-adding and relative Senone scoring of the AM Block is shown below:




```
1   for frame
2       for s = 1 to N_SENONE
3           for c = 1 to N_COMP
4               for d = 1 to N_COEFF
5                   if (d == 1)
6                       part_sum = 0
7                   end if
8                   diff = x_d − μ_s,c,d
9                   square = diff * diff
10                  multiply = square * σ_s,c,d
11                  part_sum = part_sum + multiply
12              end for
13              summ = K − part_sum
14              component_scr' = f * summ
15              component_scr = W + component_scr'
16              if (c == 1)
17                  tmp_a = NEG_INFINITY
18              else
19                  tmp_a = tmp_senone_scr
20              end if
21              tmp_b = component_scr
22              tmp_c = greater of (a,b)
23              diff = abs(a − b)
24              if (diff >= 30,000)
25                  diff = 30,000
26              end if
27              ROM_data = LOG_ADD_LUT[diff]
28              tmp_senone_scr = tmp_c + ROM_data
29              if (c == 8)
30                  senone_scr_s = tmp_senone_scr
31              end if
32          end for
33          if (s == 1)
34              MAX = NEG_INFINITY
35          end if
36          if (MAX < senone_scr_s)
37              MAX = senone_scr_s
38          end if
39      end for
40      for s = 1 to N_SENONE
41          senone_scr_s = senone_scr_s − MAX
42      end for
43      for cs = 1 to N_CSTATE
44          for csen = 1 to N_CSENONE_cs
45              if (csen == 1)
46                  TMP_MAX = NEG_INFINITY
47              end if
48              if (TMP_MAX < senone_scr_cs,csen)
49                  TMP_MAX = senone_scr_cs,csen
50              end if
51              senone_scr_[s + cs] = TMP_MAX
52          end for
53      end for
54  end frame
```

Figure 4.1: Pseudo-code representation of Brute-force Acoustic Modeling Calculations

Based on the overall computations that need to be performed, the hardware was partitioned in a way so as to aid in the implementation of a highly efficient, fully pipelined architecture. From the pseudo-code, the overall computation at this phase can be characterized as follows:

- Lines 4 to 15 deal with the computation of Gaussian Likelihood Scores. This computation is extremely math intensive: GAUS_DIST_BLK
- Lines 16 to 32 mainly deal with a large memory look-up related to the addition of values in the log-domain: LOG_ADD_BLK
- Lines 33 to 42 deal with the normalization of the Senone Scores with the BEST for the current frame and lines 43 to 53 deal with the computation of Composite Senone Scores. As seen from the pseudo-code, given the similarity in the computations in these two sections, the computations are combined into a single SSCR_CSSCR_BLK.

Based on this partitioning, a top-level block diagram is shown below.

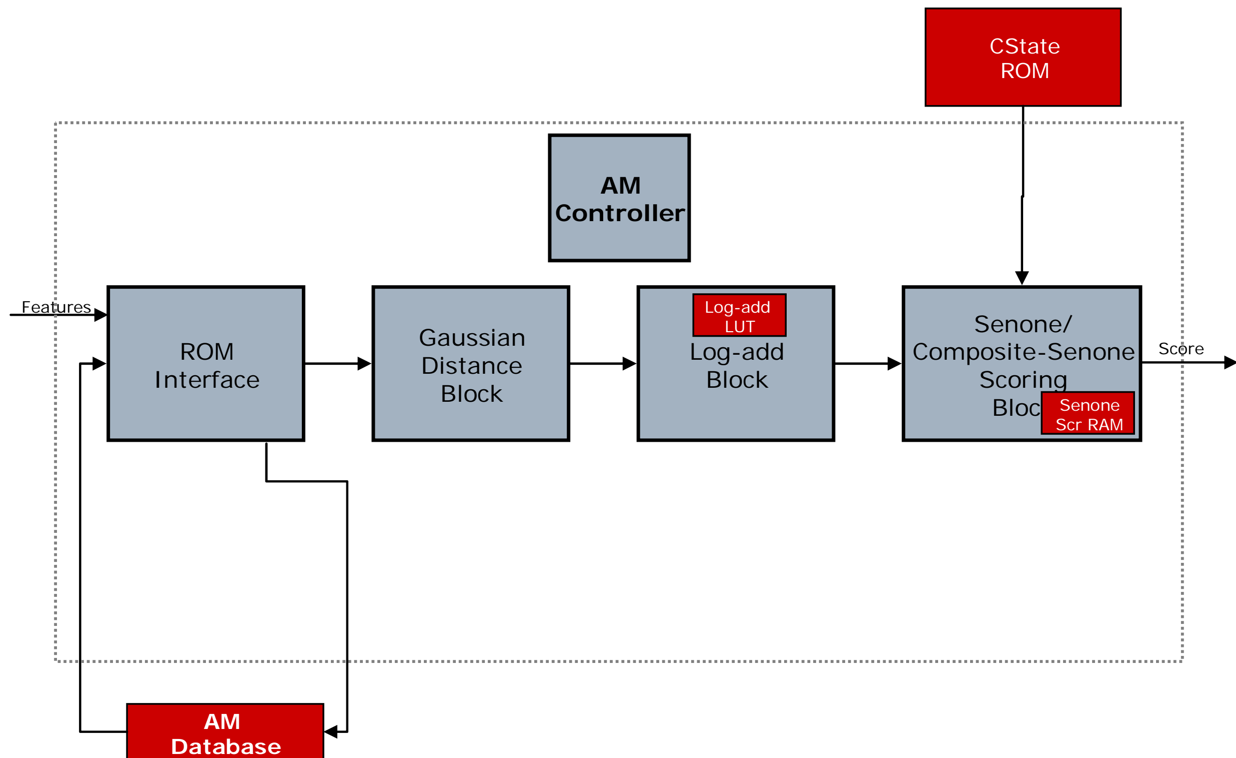


Figure 4.2: Top-level Block Diagram for brute-force Acoustic Modeling

Given the broad overview of the operations that needs to be performed in Acoustic Modeling, the following section focuses on the architectural and implementation aspects. For the sake of clarity, first, a computationally un-optimized “brute-force” implementation based on the above pseudo-code is described. Such an implementation however computes “all” the values present in the AM Database requiring several hundreds of Million floating point operations per second.

Both in terms of real-time operation and power consumption, a system with such large number of computations cannot be afforded and hence cannot be part of any practical system. Therefore techniques that allow for reduction in the number of these Gaussian Computations need to be incorporated into the design.

In Section 4.3, a popular coarse-grain/fine-grain approach is presented. This approach, known as *Sub-Vector Quantization* is also part of Sphinx 3.3. After presenting the basic idea, design considerations along with some implementation details are discussed. Finally, in Section 4.4, a Top-level description of Acoustic Modeling is presented. The integration of the various blocks and the various phases is shown. A description of the working of each of the blocks is presented.

4.2 BRUTE-FORCE ACOUSTIC MODELING

4.2.1 Gaussian Distance Block

The Gaussian Distance Block, GAUS_DIST_BLK deals with the computation of Gaussian Likelihoods and is extremely math intensive. From the pseudo-code it can also be seen that this computation forms the “inner-most” loop. Hence, for any system to have superior performance, it is critical to optimize the functioning of this block to the greatest extent possible.

```
1   for frame
2          for s = 1 to N_SENONE
3                 for c = 1 to N_COMP
4                        /* GAUS_DIST() */
5                        for d = 1 to N_COEFF
6                               if (d == 1)
7                                      part_sum = 0
8                               end if
9                               diff = x_d – μ_s,c,d
10                              square = diff * diff
11                              multiply = square * σ_s,c,d
12                              part_sum = part_sum + multiply
13                       end for
14                       summ = K – part_sum
15                       component_scr' = f * summ
.                        component_scr = W + component_scr'
.                        LOG_ADD()
.                end for
.         end for
.         RELATIVE_SENONE_SCORE()
.         COMPOSITE_SENONE_SCORE()
.  end frame
```

Figure 4.3: Pseudo-code for Gaussian Distance Calculation

4.2.1.1 Bit-precision Analysis for Fixed-point Computations

For this, the first hurdle was the bit-precision of the computations themselves. For maximum accuracy, Sphinx3 stores data in single-precision floating-point format and performs double-precision floating-point computations. For embedded devices with limited computational and battery resources, it is critical to explore the possibility of converting these computations from double-precision to single-precision and further to fixed-point with minimal loss in accuracy, if possible.

The MATLAB programming environment was chosen for this exploration. This stems from the fact that MATLAB is a tool that is designed towards addressing the needs of the computationally and algorithmically intensive industry. It not only allows easy data management with the use of matrix-based data-structures as opposed to pointer-based data-structures in languages like C, but it also provides certain libraries with in-built functions in the form of Toolboxes that further ease the exploration process. One such toolbox, the Fixed-point Toolbox [29] in MATLAB 7 (Release 14) [30], a library with in-built functions for performing fixed-point math made it an ideal choice for the required analysis.

The first analysis that was required was the difference in values obtained when moving from double-precision to single-precision floating-point operations. On simple observation, it was noted that the gain in precision obtained was negligible and hence it was concluded that in the worst-case, single-precision floating point operations could be performed.

Having done this, the next step was to explore the possibility of converting floating-point operations into fixed-point operations. For this analysis, the first step was to identify the range of values in the AM Database. For this, 80 sentence utterances from the RM1 Corpus were processed and the range of values at each stage of the computation (lines 8, 9, 11, 13, 15 and 30 of the pseudo-code) were obtained. The observations are presented in the table below.

Table 4.2: Summary of minimum and maximum data ranges for each stage of the Gaussian Distance computation with the corresponding number of "integer" bits required to represent the ranges accurately

	EXPRESSION	MIN	MAX	Bit-precision
DIFF	$(x - \mu)$	-28.68	27.89	1 | 5 | x
SQUARE	$(x - \mu)^2$	0	822.54	0 | 10 | x
PART_SUM	$(x - \mu)^2 * Var$	0	9,087.40	0 | 14 | x
SUMM	*Summation*	0	13,390.00	0 | 14 | x
COMPNENTT_SCR	*Weighted Comp*	-44,464,464.00	84,397.00	1 | 28 | 0
SENONE_SCR	*Senone Scr*	-6,231,617.00	84,398.00	1 | 25 | 0
FEAT	*Input Feautres*	-15.85	18.22	1 | 5 | x
MEAN	*Mean*	-9.67	12.83	1 | 4 | x
VAR_PRECOMP	*Percomputed Var*	0.01	3,866.70	0 | 12 | x
LRD	*Const [K]*	-2.11	70.89	1 | 7 | x
MIXWT	*Mixture Weight [w]*	-14,051.00	-3,958.00	1 | 14 | 0

The table shows the minimum and maximum signed values in the AM Database along with those at the individual stages of the computation. These values were used to create a table of the "minimum" bit-precision required to safely represent the integer part. This is shown in the "Bit-precision" column in the table where each value is represented as "Signed or unsigned | # of integer-bits | # of fraction-bits".

This representation is similar to that of the Fixed-point Toolbox, with the exception of the fact that the second field represents the number of integer-bits as opposed to total number of bits so as to provide a more intuitive representation of the individual values. The *'x'* in the fraction-

bits field represents an unknown quantity about how many fraction bits are required to keep the percentage error at a minimum.

Upon making this table, a couple of observations were made.

1. Considering the large range of the Component and Senone Scores, the fractional part can be safely ignored without inducing any error. This is a direct result from the multiplication of the *summ* with the $\log_{s3} e$ scaling factor of 3333.83.
2. The dynamic range of *diff, square, part_sum* and *summ* quantities, at least on the integer-bits, is very small, when compared to the range provided by using floating point numbers. Therefore, depending on the impact of the number of fraction-bits required, floating-point calculations might not be necessary.

The next step was to identify the level of fractional-bit precision required at each stage of the computation. First, a very lenient approach was taken whereby the computations were allowed to take the maximum possible bit-precision. Such an approach would also help in identifying small bugs in the MATLAB code written for this exploration, if any. Based on these precision values, extensive tests were run to ensure that there was negligible loss in accuracy. Upon ensuring a bug-free test setup based on the results obtained, the reduction in the number of fractional bits could be explored further.

Initially, the quantization of the fractional-bits was done aggressively. This approach resulted in percentage errors greater than those that were acceptable. Therefore, rather than employing a completely trial and error based methodology, the number of bits at each stage of the operation was fixed. For this, the precision required to represent the input data was taken as the queue.

Based on this approach, a few preliminary trial and error experiments were run to obtain the bit-widths. As a result of these experiments, it was observed that 32-bits of precision on the input values resulted in negligible loss in accuracy. Fixing the bit-widths of the input values then helped in focusing completely on the precision required towards the representation of the values at the individual stages.

After running a few experiments, fixed-precision values for each stage of the computation were obtained. Detailed final results of the analysis conducted are shown below. The bit-widths of the combination of the values are shown below.

Table 4.3: Final bit-precision breakup for representing the inputs and the intermediate outputs of Acoustic Modeling

INPUTS	Bit-precision	TOTAL Bits	OUTPUTS	Bit-precision	TOTAL Bits
FEAT	1 | 6 | 25	32	DIFF	1 | 6 | 25	32
MEAN	1 | 6 | 25	32	SQUARE	0 | 14 | 18	32
VAR_PRECOMP	0 | 12 | 20	32	PART_DIST	0 | 18 | 14	32
LRD	1 | 17 | 14	32	SUMM	1 | 17 | 14	32
MIXWT	1 | 32 | 0	32	CSCR	1 | 32 | 0	32
f	1 | 14 | 18	32	SSCR	1 | 32 | 0	32

As can be seen, the number of fractional bits varies significantly at some of the stages. This was necessary because of the varying dynamic ranges of computations at each of these blocks. On a side note, the fixed-point implementation doesn't perform rounding. Instead, "truncation" math was used at the gain in precision on account of rounding is negligible. Further, implementing rounding in the FPGA would add overhead and lead to a degradation in the performance of the system.

During the entire analysis process, two metrics were of particular interest. Since the Senone Scores are the final outputs of this block, the percentage error in the floating-point versus fixed-point Senone Scores would be a very important indication of how well the bit-precision values were calculated. A graph depicting this for a sample utterance is shown below. This utterance consists of over 500 frames corresponding to 5 seconds of speech requiring the calculation of almost 1 billion Senone Scores.

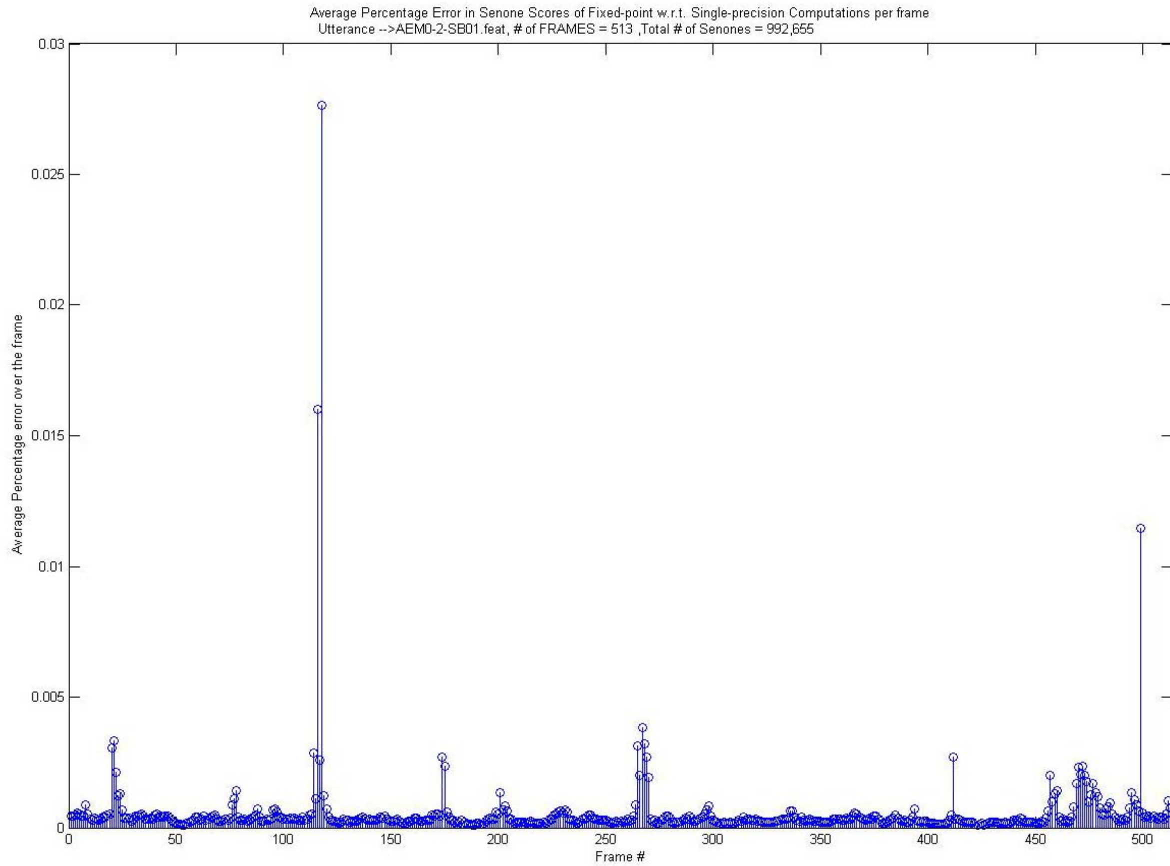


Figure 4.4: A Graph showing average percentage error in Senone Scores per frame over all frames in the sample utterance

The average percentage error per Senone over the entire utterance was of the order of 10^{-3}. This is a clear indication that the conversion into fixed-point math has been done successfully.

Another interesting metric is the distribution of the percentage errors obtained. This would give a good indication of the worst-case percentage error. In previous analysis, some percentage errors were found to be in the several thousands. On a closer observation, it was found that a majority of these cases were because of the proximity of the Senone Score around zero. Because of this, when the floating-point and fixed-point values are close to zero (below 1), then there can be a huge percentage error in “absolute” terms. But the significance of this “absolute” difference is insignificant considering the large range of Senone Scores that are obtained.

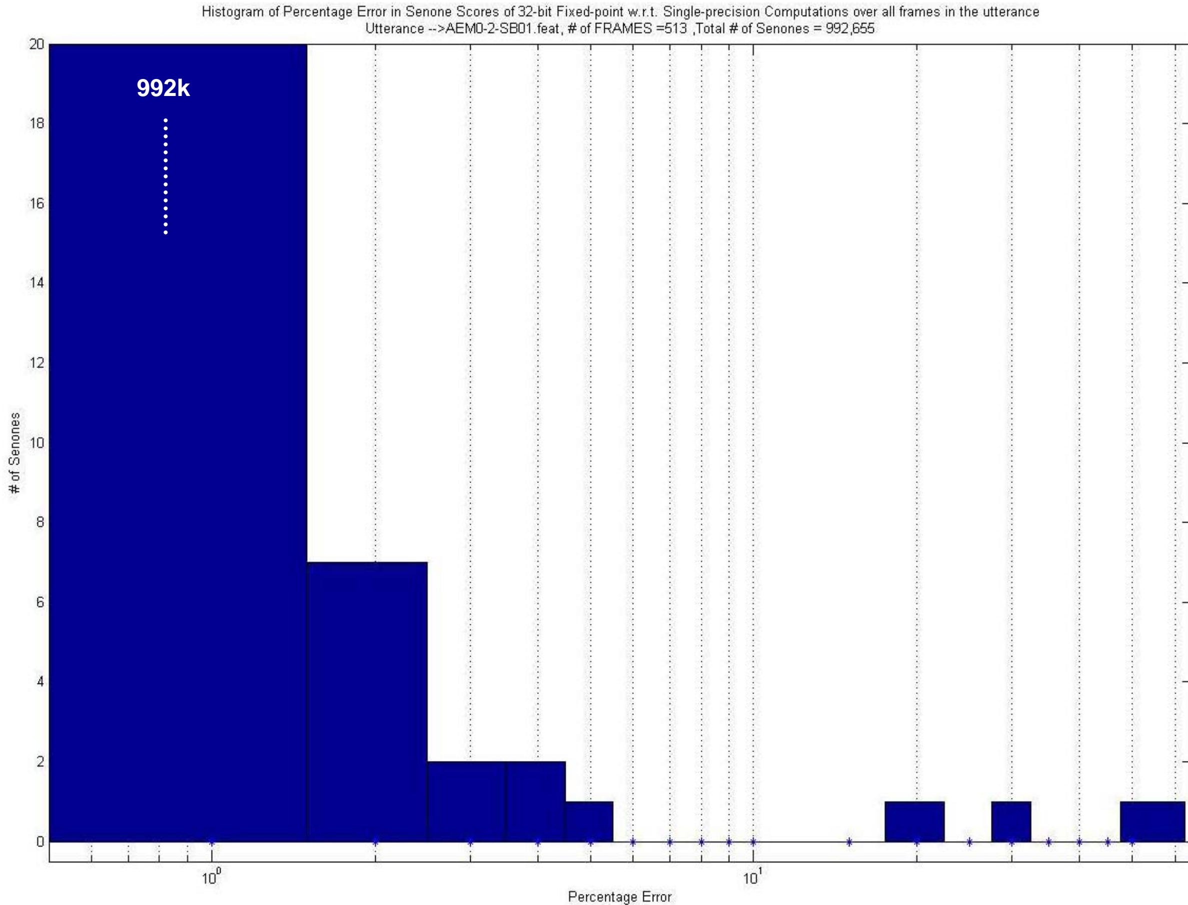


Figure 4.5: A Histogram of Percentage Error in Senone Scores for the Sample Utterance

The histogram shows that 99.99 % of the computations lie with-in 1% error. Further, there are only 3 values that are greater than 20%. This further indicates that all computations using fixed-precision are very close to the floating-point scores.

Having successfully converted the Gaussian distance evaluations from double-precision floating-point operations into fixed-precision, with custom precision values at every stage of the computation, a quantity that could be mapped reasonably well into the FPGA architecture was obtained. With these results, the design and implementation process was commenced.

4.2.2 Log-add Block

Upon obtaining the Component Score from the GAUS_DIST_BLK, 8 of these scores need to be accumulated to obtain the Senone Score. As a direct result of the conversion of the

computations into the log-domain whereby log-likelihoods instead of probabilities are being calculated as shown in Eq. 4.8, the accumulation is no more a straight-forward addition of Components, instead, it involves the addition of values in the log-domain represented by $LOG\sum_{c=1}^{8}$ .This operation is known as "Log-add", LOG_ADD_BLK. The following derivation shows one of the techniques of how the log-add operation can be performed in the most optimum manner.

Suppose that $P = Q + R$.

Calculate z such that, $z = \log_B P$, $x = \log_B Q$ and $y = \log_B R$.

Applying log-identities, the above values can be represented in the natural domain as $P = B^z$, $Q = B^x$ and $R = B^y$ respectively

Substituting them in the original equation, we get

$$B^z = B^x + B^y \quad \text{Eq. 4.9}$$

$$= B^x\left[1 + B^{y-x}\right] \quad \text{Eq. 4.10}$$

Taking $\log_B$ on both sides and applying log-identities,

$$z = \log_B\left(B^x\left[1 + B^{y-x}\right]\right) \quad \text{Eq. 4.11}$$

Taking the $\log_B$ inside the parenthesis, the equation can finally be reduced to

$$z = x + \log_B\left[1 + B^{y-x}\right] \quad \text{Eq. 4.12}$$

Re-writing,

$$z = x + \log_B\left[1 + B^{-(x-y)}\right] \quad \text{Eq. 4.13}$$

Once again, the computation of a logarithm value is encountered. On closer observation however it can be seen that the only variable for the calculation of the logarithm is the $(x - y)$ value. Therefore, one possible solution to avoid the computation of the logarithm during the operation of the system is that of "pre-computing" for all possible x and y values.

This now poses another challenge given that the number of combinations of x and y values can be very large. However, if the result of $(x - y)$ can be kept as a positive quantity, then as the difference between x, y increases, $B^{-(x-y)}$ decreases. Beyond a certain value, the result from the exponent calculation approaches 0 thereby reducing $\log_B\left[1 + B^{-(x-y)}\right]$ to 0

(since $\log_B[1+0]=0$). For B = *s3*, Sphinx log-base (=1.0003), $\log_B\left[1+B^{-(x-y)}\right]$ and ranges between 2311.33 and 0.912 for *(x - y)* for a difference 0 through 30k. A plot depicting this is shown below.

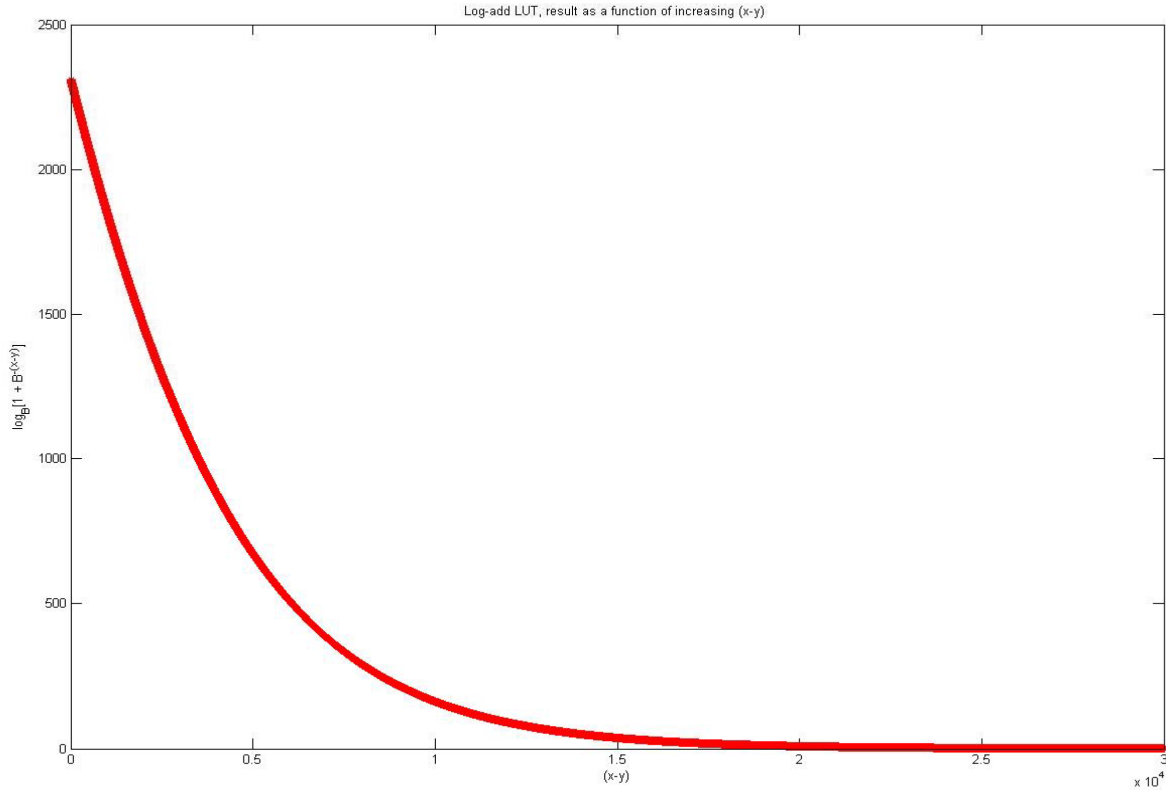


Figure 4.6: Monotonically decreasing value of the Pre-computed Log-add with increase in the difference of the inputs

From the figure, it can be seen that $\log_B\left[1+B^{-(x-y)}\right]$ is a monotonically decreasing function of the difference *(x - y)*. In Sphinx, data is stored as 32-bit floating-point values. This pre-computed data is known as the Log-add lookup table (abbreviated as Log-add LUT).

Given the limited dynamic range of the pre-computed values, and the impact of the fractional precision having negligible if not any impact on the accuracy (given that Senone Scores are represented in integer form), it was concluded that the fractional-bits could be ignored altogether. By doing this, the number of required bits was reduced to 12-bits per log-add LUT entry. Therefore, the amount of memory required to store the log-add LUT has been reduced by almost 2/3rd's.

Depending on the prototyping environment and the type of resources available, the Log-add LUT can either be placed on-chip or off-chip. Off-chip implementation would be the preferred choice if a two-memory implementation was available. The AM Database could be in one and the log-add LUT in the other. This would help ensure that the GAUS_DIST_BLK pipe is being fed with data in an un-interrupted manner. To minimize design complexity, the log-add LUT has been implemented using on-chip BRAM resources.

By taking the look-up approach, the log-add of two values can be computed by simply calculating the difference of $(x - y)$, followed by a look-up table corresponding to this difference and finally adding the look-up value with x. A pseudo-code representation of this calculation is shown below.

```
 .   for frame
 .       for s = 1 to N_SENONE
 .           for c = 1 to N_COMP
 .               GAUS_DIST()
16               /* LOG_ADD() */
17               if (c == 1)
18                   tmp_a = NEG_INFINITY
19               else
20                   tmp_a = tmp_senone_scr
21               end if
22               tmp_b = component_scr
23               tmp_c = greater of (a,b)
24               diff = abs(a - b)
25               if (diff >= 30,000)
26                   diff = 30,000
27               end if
28               ROM_data = LOG_ADD_LUT[diff]
29               tmp_senone_scr = tmp_c + ROM_data
30               if (c == 8)
31                   senone_scr_s = tmp_senone_scr
32               end if
 .           end for
 .       end for
 .       RELATIVE_SENONE_SCORE()
 .       COMPOSITE_SENONE_SCORE()
 .   end frame
```

Figure 4.7: Pseudo-code for Log-add Calculation

Whenever the first component of a Senone is being evaluated, the first Component is log-added with NEG_INFINITY, a very large negative value. In the natural domain, this would mean addition of the Component Score with 0. The computation of $(x - y)$ is a bit tricky in that

not only does the difference needs to be computed, but the greater of the two values also needs to be found. Since the difference directly acts as the address of the log-add LUT, absolute difference is taken. The greater of the two values is required to obtain which of the two values is x in Eq. 4.13. Further, since the size of the log-add LUT is limited, values for which $(x - y)$ is greater than 30k, the size of the log-add LUT, thresholding of the difference is done. Based on these computations the LOG_ADD_BLK was designed.

4.2.3 Senone/Composite Senone Scoring Block

As described earlier, there is a fair amount of variability in speech. To account for this, "relative" rather than "absolute" scores are used as the final Senone scores. For this, the best Senone score over all Senones is found and Senone scores are normalized w.r.t. the best score for a given frame. Normalization of the scores therefore results in the best score of 0 and all other scores being some negative quantity. The normalization not only gives a "relative" measure of the scores, but it also helps in limiting the dynamic range of the values. A pseudo-code representation of the computation is shown below.

```
 .   for frame
 .         for s = 1 to N_SENONE
 .               GAUS_DIST()
 .               LOG_ADD()
33               /* RELATIVE_SENONE_SCORE() */
34               if (s == 1)
35                     MAX = NEG_INFINITY
36               end if
37               if (MAX < senone_scr_s)
38                     MAX = senone_scr_s
39               end if
40         end for
41         for s = 1 to N_SENONE
42               senone_scr_s = senone_scr_s - MAX
 .         end for
 .         COMPOSITE_SENONE_SCORE()
 .   end frame
```

Figure 4.8: Pseudo-code for Normalized Senone score Calculation

As described earlier, each composite state is composed of several Senones. The number of Senones per Composite state varies. For a given Composite state, the Composite Senone score

is the “best” Senone Score for all Senones making up that Composite state. This is shown in the pseudo-code below.

```
 .   for frame
 .          GAUS_DIST()
 .          LOG_ADD()
 .          RELATIVE_SENONE_SCORE()
43          /* COMPOSITE_SENONE_SCORE() */
44          for cs = 1 to N_CSTATE
45                 for csen = 1 to N_CSENONE_cs
46                        if (csen == 1)
47                               TMP_MAX = NEG_INFINITY
48                        end if
49                        if (TMP_MAX < senone_scr_cs,csen)
50                               TMP_MAX = senone_scr_cs,csen
51                        end if
52                        senone_scr_[s + cs] = TMP_MAX
53                 end for
54          end for
 .   end frame
```

Figure 4.9: Pseudo-code for Composite Senone score Calculation

From an implementation perspective, computation in the SSCR_BLK and CSSCR_BLK is very similar. They both deal with the computation of a running MAX and the subtraction for calculating the normalized values (in the case of Senone score evaluation). Further, there is no overlap in the timing of the required resources by each of these blocks. Hence, an optimization was performed and the computation of these two blocks was clubbed into one block, SSCR_CSSCR_BLK.

4.3 SUBVECTOR QUANTIZATION

4.3.1 Introduction

Given that the number of computations required by a Brute-force system is very substantial, computation reduction techniques are implemented in practical systems. Due to the significance of decreasing the computational load on the system, computation reduction techniques have been a hot topic of research. These reduction techniques have tended to focus on

two major aspects. The reduction in the dimensionality of the Gaussians [31,32], and by reducing the number of Components that need evaluation in a Senone.

The latter is more popular and results in a two-pass, coarse-grain/fine-grain approach. The first pass is made on a quantized set of the model from which the Components that are likely to contribute most to the Senone score for each Senone are selected. These selected Components are evaluated in the full model. Since the quantized model is a cruder representation of the data, fewer Components are required to represent the data space that is represented by the original, full models. Not only this, since a handful of Components are being evaluated, typically 2 or 3 Components per Senone on an average, the savings over calculating the full-model is a gain of greater than 66% savings. This however does not include the overhead of the coarse-grain approach itself.

One of the most popular techniques using this coarse-grain/fine-grain approach is the Sub-Vector Quantization approach (SubVQ). It was introduced by Bocecheiri [33] and refined by Ravishankar [34,35]. It has been employed in the Sphinx3 system as a technique to reduce computations. Details of this technique are provided in the following section.

4.3.2 Motivation

One of the major motivations behind the SubVQ approach is based on a very interesting observation. According to this observation, after extensive analysis, over 89% of the Senone score was found to be contributed by the “best” scoring Components of any Senone. An illustration of this is shown in the graph below.

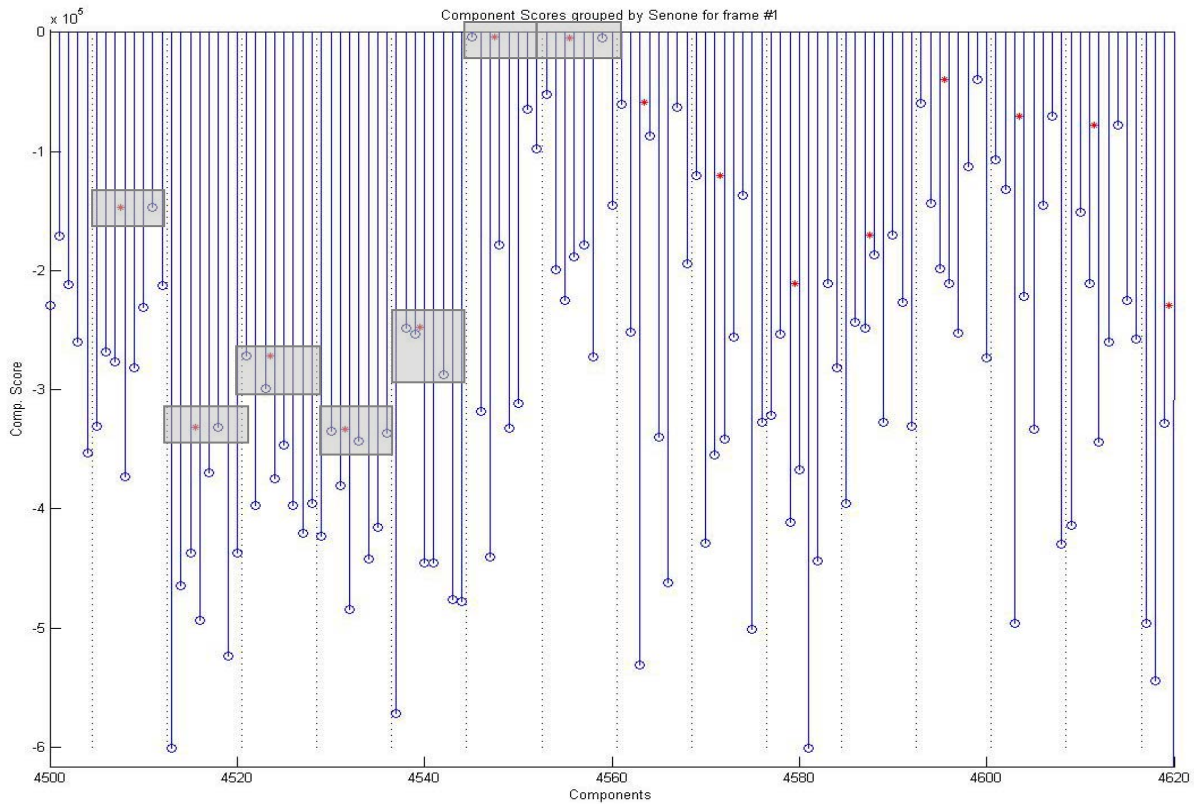


Figure 4.10: Senone Scores as a function of Component Score

This graph shows a stem plot of Component scores for a small section of Components for one frame of speech. The Components that makeup the corresponding Senones are demarcated by dotted lines. The Senone scores for each Senone are superimposed on this graph and represented by the red ‘*’. From the graph it is clear that the final Senone score tends to be closest to the best scoring Component Score(s). The graph is on a negative scale and hence the smaller the magnitude, the larger the value.

Based on this observation, techniques that can help find the most likely “best” or “top” scoring Components can be utilized to reduce the number of computations that go towards the calculation of Senone scores. SubVQ is one such technique.

4.3.3 Sub-Vector Quantization

Sub-Vector Quantization, SubVQ, as the name suggests, is a technique whereby the original database of Gaussian mean/variance pairs is quantized such that the entire database can

be represented by a subset of a few Gaussians. The original model is sometimes referred to as the *full continuous* model (since it is not quantized) and the quantized version is referred to as *sub-vector quantized* model.

The quantization of the data results in what are referred to as *Codewords*, quantized version of Components that make a Senone. Since Codewords are obtained by quantizing Components, each Component can be represented by a cruder form, the Codeword. This way, several Components can be mapped into one Codeword. This mapping is referred to as the *MAP* table. Hence, using the MAP table, a Senone can now be represented by Codewords.

The major benefit of quantization is that fewer unique quantities are needed to represent the same data set. Therefore, in Sphinx upon quantization, 15,480 Componets for the RM1 Corpus can be represented by 4096 Codewords. A 3D representation is shown in Figure 4.10. It can be seen that each codeword is made of just *one* 39-dimensional Gaussian quantity.

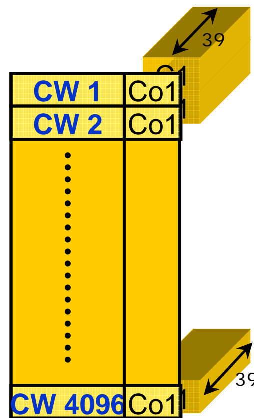


Figure 4.11: A 3-Dimensional Representation of the SubVQ Database

On evaluating these Codewords, Components that are likely to correspond to the best/top Components in a Senone can be obtained. Based on this, Components in the full-continuous model are evaluated towards the computation of the Senone score. Hence, assuming that the top 2 components are required to accurately pick the best Components, a 75% saving in computation can be achieved (since totally there are 8 components).

However, it has been found that Beam based pruning is more appropriated. Hence, similar to the discussion in Section 2.2.2.2, the best Codeword score for a Senone is obtained and to this best a beam is added giving the threshold. Components corresponding to Codeword scores that are greater than the threshold are treated as the best/top likely components and are set as

active. Similar to the feedback notation mentioned in Chapter 3, the list of all active Components is maintained as a Component Active List (CAL). This is essentially a bit-vector for each Senone and is set depending on whether the Component is active or not.

4.3.4 Implementation

A pseudo-code representation of the calculations is shown below.

```
1   for frame
2          for cw = 1 to N_CW
3                 for d = 1 to N_COEFF                                      ⎫
4                        if (d == 1)                                        ⎪
5                               part_sum = 0                                ⎪
6                        end if                                             ⎪
7                        diff = x_d – μ_s,c,d                               ⎪
8                        square = diff * diff                               ⎬ GAUS_DIST_BLK
9                        multiply = square * σ_s,c,d                        ⎪
10                       part_sum = part_sum + multiply                     ⎪
11                end for                                                   ⎪
12                summ = K – part_sum                                       ⎪
13                component_scr' = f * summ                                 ⎭
14         end for
15         for s = 1 to N_SENONE
16                for c = 1 to N_COMP
17                       TMP_MAX = codeword_scr[MAP_s,c]
18                end for
19                SUB-VQ_THRESOLD = TMP_MAX + SUB-VQ_BEAM
20                for c = 1 to N_COMP
21                       if (codeword_scr[MAP_s,c] >= SUB-VQ_THRESOLD)
22                              COMPONENT_ACTIVE_s,c = 1
23                       else
24                              COMPONENT_ACTIVE_s,c = 0
25                       end if
26                end for
27         end for
28  end frame
```

Figure 4.12: Pseudo-code for Sub-Vector Quantization Calculation

As can be seen from the above code, similar to brute-force AM, lines 3 thru 13 deal with the evaluation of Gaussian likelihoods with the only difference that Codeword scores are being computed. After the computation of the Codeword scores, the next step is to find the "active"

Components over all Senones. These active components are the ones that are likely to contribute towards the final Senone score the most.

4.4 TOP-LEVEL

All these blocks discussed above were integrated and the resulting top-level diagram is shown in Figure 4.7. Apart from showing the various blocks, this figure also shows the phases each block goes through.

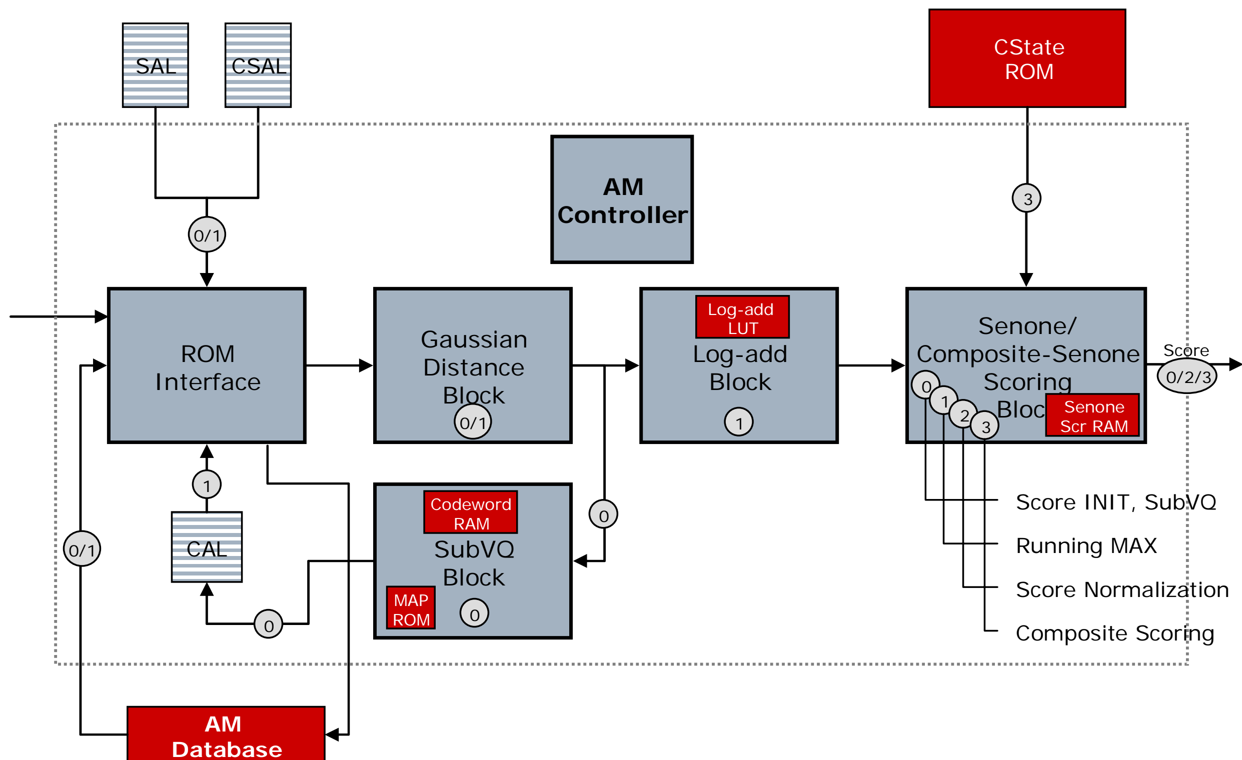


Figure 4.13: Top-level Block Diagram of Acoustic Modeling with SubVQ along with the phase breakup

The first phase, Phase 0, deals with the initialization of the Senone Scores to NEG_INFINITY in SSCR_CSSCR_BLK and the computation of the shortlist using SubVQ. For obtaining the shortlist of active Components, Codeword Scores for all Codewords are computed and the result is stored in the Codeword_RAM in the SubVQ_BLK. After this, computations shown in pseudo-code 4.x are performed giving a list of active Components. This is stored in the Component Active List FIFO (CAL_FIFO).

```
1   for frame
2         for cw = 1 to N_CW
3               GAUS_DIST()         /* Compute Codeword Scores */
4         end for
5         for s = 1 to N_SENONE
6               compute_CAL()       /* Compute list of active Components */
7         end for
8         for s = 1 to N_SENONE
9               for active components in CAL
10                    GAUS_DIST()
11                    LOG_ADD()
12              end for
13        end for
14        RELATIVE_SENONE_SCORE()
15        COMPOSITE_SENONE_SCRE()
16  end frame
```

Figure 4.14: High-level Pseudo-code representation of Acoustic Modeling Block (with SubVQ)

Phase 1 corresponds to computing of Senone Scores for active Senones (from SAL_FIFO) based on the active Components (from CAL_FIFO). The ROM Interface Block (ROM_INTF_BLK) merges data from these two FIFOs and generates appropriate addresses for accessing the AM_Database. The data output is then passed through the GAUS_DIST_BLK giving the Component Scores, which in-turn is passed through the LOG_ADD_BLK for the computation of Senone Scores. A running MAX of Senone Scores is computed and the scores themselves are stored in the Senone_Scr_RAM in the SSCR_CSSCR_BLK.

Once the SAL_FIFO is empty, it indicates that all active Senones have been computed implying that the next phase can begin processing. Phase 2 deals with the normalization of Senone Scores. The Senone Scores are accessed from the Senone_Scr_RAM, normalized and written back into the same memory.

Upon the completion of computing the final, normalized, Senone Scores, Composite Senones can be evaluated. For this, entries from the CSAL_FIFO are popped one at a time and passed onto the CState_ROM. The CState_ROM in turn generates addresses based on the Senones that combine to form the Composite State and passes that over to the SSCR_CSSCR_BLK. Corresponding Senone Scores are accessed from the Senone_Scr_RAM and a running MAX is maintained for each Composite. After processing all Senones, the MAX value is written back into the Senone_Scr_RAM where N_SENONE + CState_ID servers as the destination address into the Senone_Scr_RAM.

All scores that are written into the Senone_Scr_RAM are also output to the top-level, where they are stored in the SENONE_RAM. All these computations take place in the Acoustic Modeling Block.

5.0 THE "BESTN" COMPUTATION REDUCTION TECHNIQUE FOR AM

5.1 MOTIVATION

Although the Sub-Vector Quantization technique [34] does reduce the number of computations by almost 65-75%, these computations are based on Gaussian Likelihood computations that are both bandwidth and resource intensive, contributing to almost a 25% overhead (for RM1 Corpus). As a result, only 40-50% reduction in overall computations is possible due to this technique.

With these considerations in mind, the necessity of a technique that would address some of the limitations posed by SubVQ was felt. For this, a detailed analysis of the Gaussian Computation was made and, a new *"bestN"* technique is proposed. This technique is based on the same coarse-grain/fine-grain approach, but requires $1/8^{th}$ the bandwidth and 8-bit integer addition operations by using pre-computation and look-up, thereby addressing both the bandwidth and the overhead issues. Further, it is shown that the accuracy of the system is better than that of SubVQ for the same number of computations of the full model.

5.1.1 Ideal Characteristics of Computation Reduction Techniques

Any computation reduction technique can be deemed to be successful based on how well it is able to address the following three issues:

1. **Accuracy:** Since the application of computation reduction techniques are generally based on some approximations, the accuracy of the system is certain to be equal to or below that of the full-continuous model. The degradation in the resultant accuracy of the system is generally a very important benchmark that indicates how successful a computation reduction technique is.
2. **Overhead:** Another important aspect is the amount of processing required by the computation reduction technique so as to reduce the number of evaluations in the original

model. If the overhead is significant, then the overall benefit achieved by employing such a technique would be quite limited, as is the case of the SubVQ implementation.

3. **Implementation Issues:** While some techniques might seem to be theoretically attractive, practical implementation issues are also a very important consideration for successfully minimizing the computational load. These considerations must include additional chip-resources along with memory size and bandwidth requirements that such a technique might require. Special attention should also be paid to the overall system setup (especially the processor architecture) on which the technique is being targeted. Neglecting any of these aspects can lead to undesired inefficiencies, and in some cases even result in a severe degradation in the overall system performance.

It is shown that bestN addresses all the above issues outlined above. In the following sub-section, based on hindsight observation, a detailed flow of how the technique was derived is presented. A preliminary analysis on accuracy results based on this approach followed by a more detailed analysis allowing for implementation into practical systems is discussed. Finally, possible implementation techniques are presented followed by a summary of the results of Word Error Rate (WER) obtained for 9 setups based on this technique are discussed.

5.2 HINDSIGHT OBSERVATION

On observing that the computation of Gaussian Likelihoods was a computationally and bandwidth intensive task, a detailed analysis of the computations was performed. From Eq. 4.9, the evaluation of GAUS_DIST is given by,

$$K_{s,c} - \sum_{d=1}^{39}\left[\left(x_d - \mu_{s,c,d}\right)^2 * \sigma'^2_{s,c,d}\right] \qquad \text{Eq. 5.1}$$

On closer observation of this equation, it can seen that the portion inside the summation actually deals with the computation of the distance of the observed input, x_d, from the mean, $\mu_{s,c,d}$ scaled by the inverted variance, $\sigma'_{s,c,d}$. for a given dimension, d, within Component, c, for Senone, s. As discussed earlier, the only variable in this equation is x_d, the input feature frame from the Feature Extraction phase.

A further analysis of the probability evaluation was made. It was then felt that a likelihood graph superimposed onto the probability graph for a single feature dimension would provide some insights. A figure depicting this is shown in Figure 5.1.

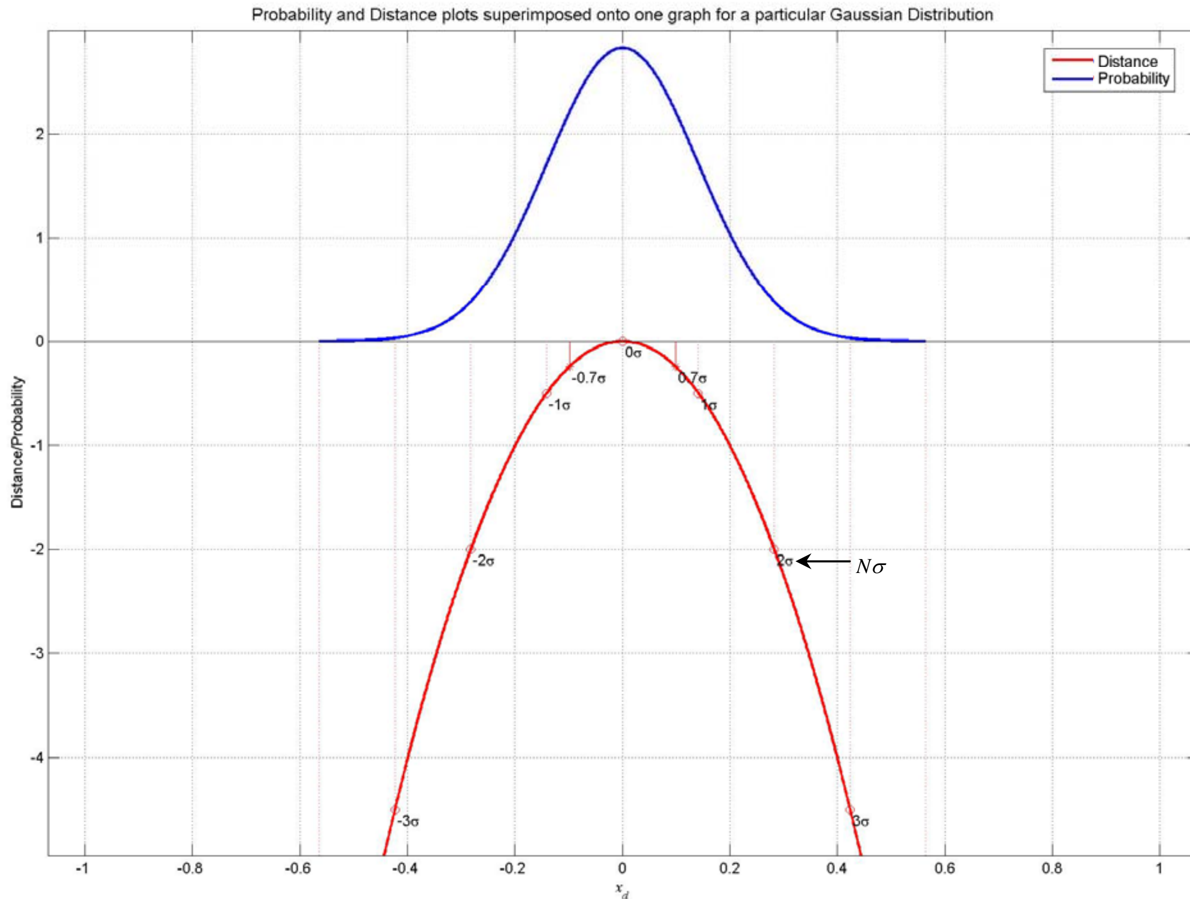


Figure 5.1: Gaussian Probability superimposed onto GAUS_DIST calculation for a single dimension

In this graph, for a given dimension, *d*, x-axis represents the inputs, *x*, the top half of the y-axis shows the probability of a Gaussian distribution for a given mean/variance pair, and the bottom half of the y-axis shows the corresponding Gaussian "distance" of Eq. 5.1. From the graph it can be seen that even though the probability for varying values of *x* varies more when closer to the mean, it is exactly the opposite when it comes to the evaluation of GAUS_DIST. The distance value is an exponentially increasing function of the distance from the mean. The farther the inputs form the mean, the greater is the distance. On observing this, more analysis was done with several dimensions superimposed on the probability-likelihood graph.

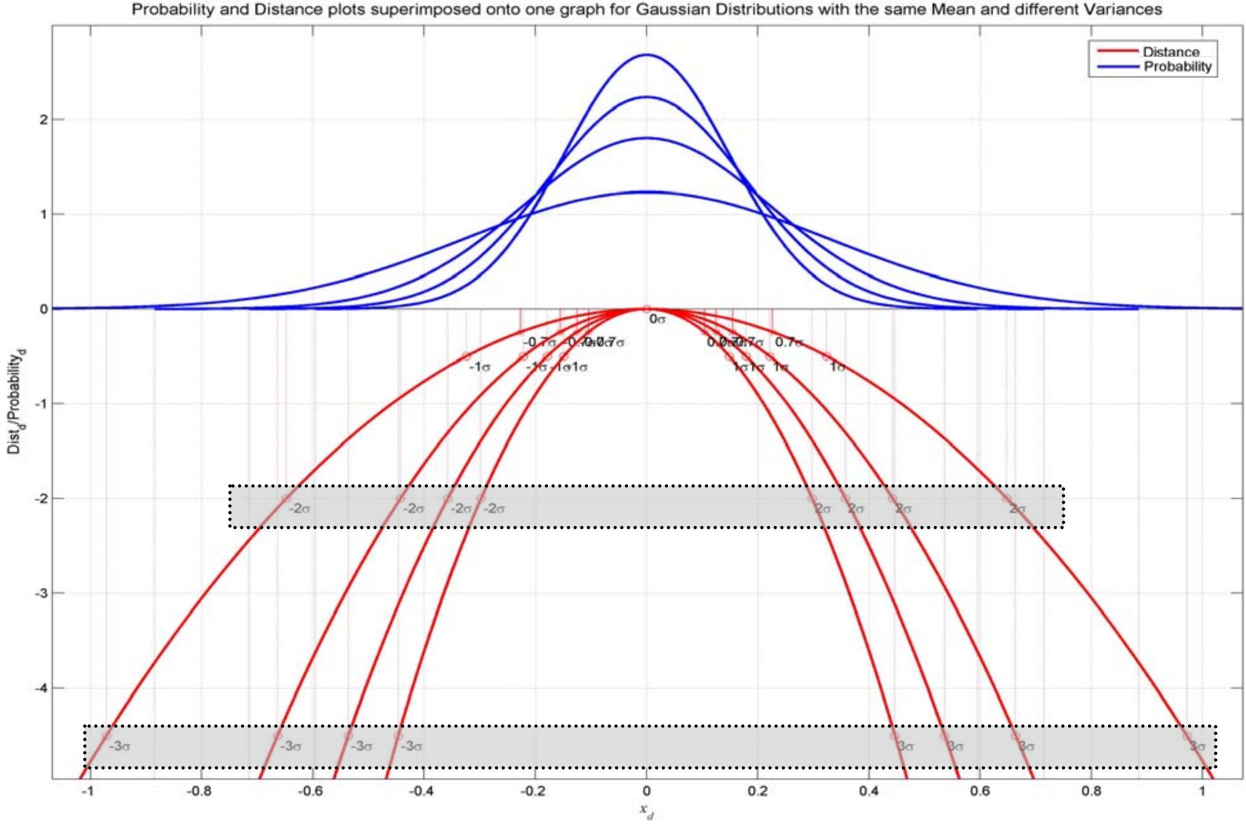


Figure 5.2: Distance Metric Superimposed on Probability graph for a single dimension

This graph provided one interesting observation. If the likelihood distance is represented as a function of the number of times the standard deviation away from the mean, then for a given multiple factor, *N*, the likelihood distance was the same. A derivation of how the *N* was obtained is shown below. Given that for a given dimension *d*, the likelihood distance is given by

$$dist_d = \frac{(x_d - m_d)^2}{2\sigma_d^2} \qquad \text{Eq. 5.2}$$

Representing the input, x_d, in terms of distance away from the mean in multiple factors of standard deviation, it can be represented by $x_d = m_d + N_d \sigma_d$.

Substituting it in the above equation, we get,

$$dist_d = \left[\frac{([m_d + N_d \sigma_d] - m_d)^2}{2{\sigma_d}^2} \right] \qquad \text{Eq. 5.3}$$

$$= \left[\frac{(N_d \sigma_d)^2}{2{\sigma_d}^2} \right] \qquad \text{Eq. 5.4}$$

$$= \left[\frac{(N_d)^2}{2} \right] \quad \text{Eq. 5.5}$$

Therefore, $N_d = \sqrt{2 * dist_d}$ Eq. 5.6

Based on this observation, a new metric was introduced whereby the closeness of the input from the mean for a given mean/variance pair is represented as a factor of multiples of standard deviation, $N\sigma$, away from the mean, rather than the traditional "distance" approach.

5.3 THE "BESTN" METHOD

Having represented the distance as a function of $N\sigma$, further analysis was done. The first step was to observe any trends that might be present. For this a 3D graph depicting the *N*'s of all 39 Dimensions for all 8 Components of a Senone was created and is shown in Figure 5.3.

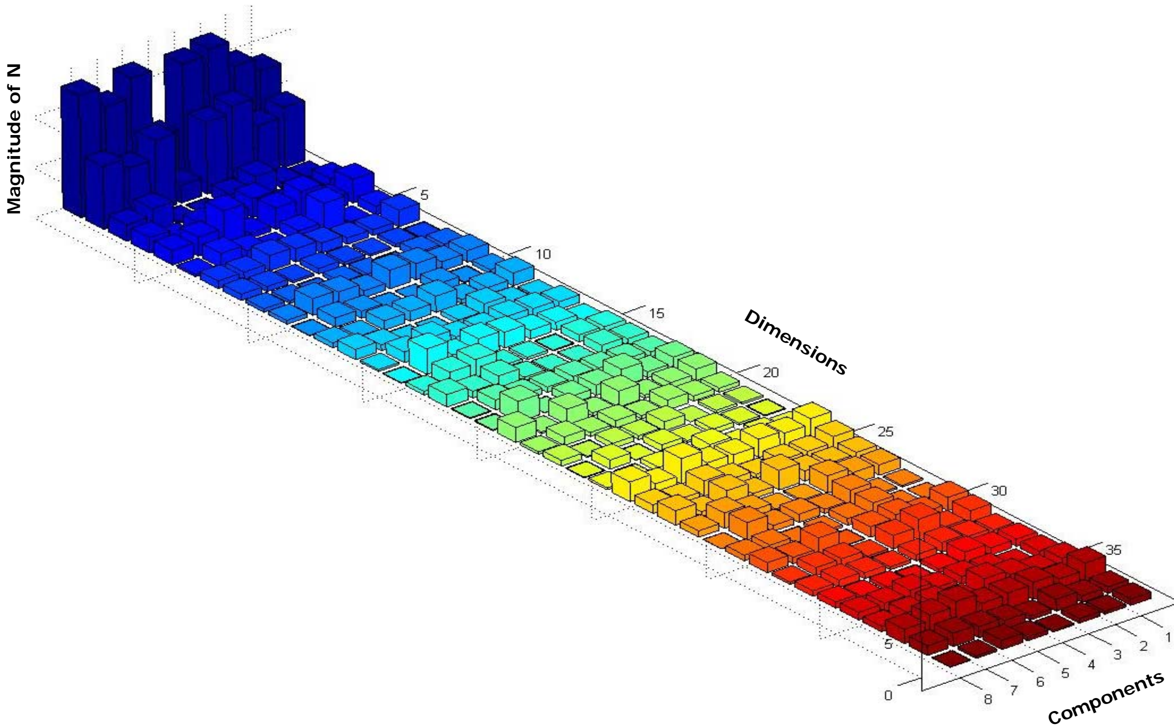


Figure 5.3: A 3D graph of N's for all Coefficients over all Components of 1 Senone

It was observed that there is a lot of variability over all dimensions for different Components. Some dimensions had a greater *N* value while others had a smaller value. To see if there was some trend across multiple Senones, the number of observations was further increased and a graph depicting the *N's* for 5 Senones was created (Figure 5.4). This proved to be an eye-bawler and it was difficult to gather any meaningful information.

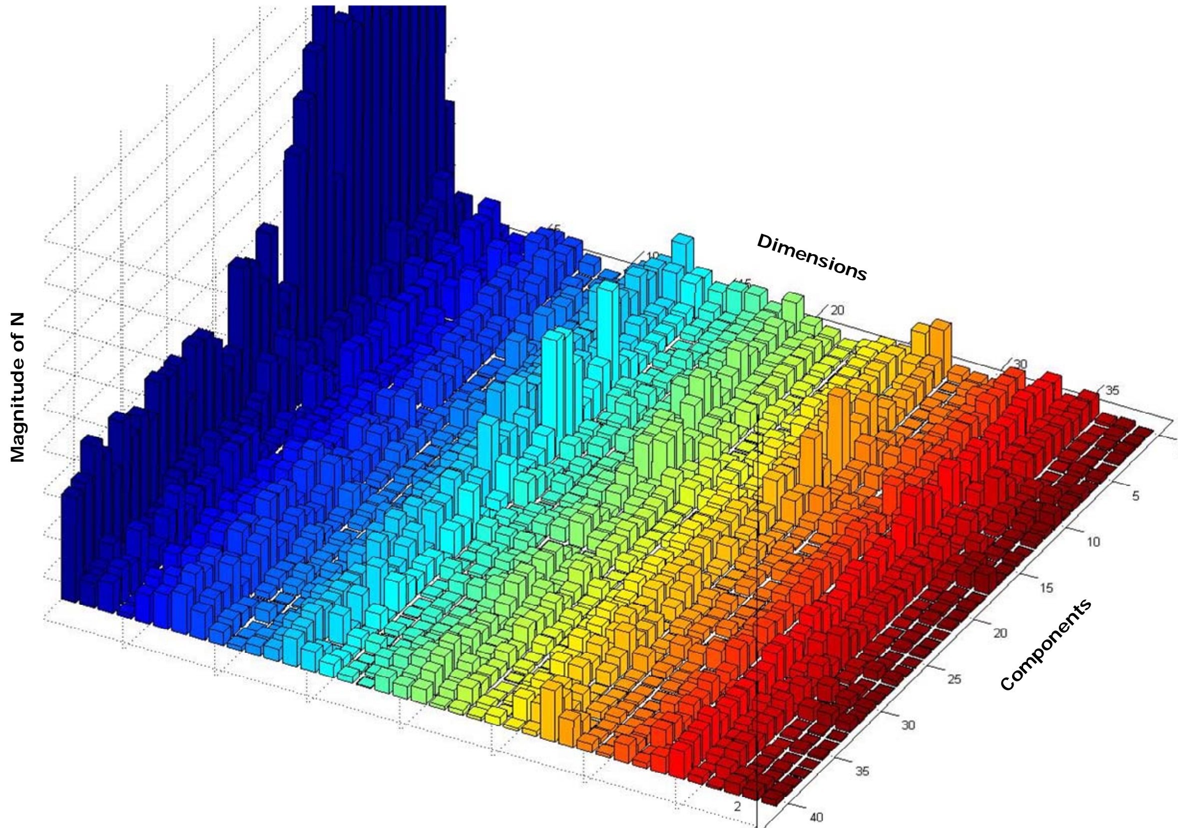


Figure 5.4: A 3D graph of N's for all Coefficients over all Components of 5 Senones

Not having found a clear trend, it became necessary to decrease the information at hand so as to help see if there was indeed some trend or correlation between distance when represented as a function of $N\sigma$. The information from each of the dimensions needed to be combined in a way so as to provide meaningful information at the Component level.

Amongst several possible ways of combining the various dimensions, one way stood out. Similar to the addition of the likelihood distances over all dimensions, the N_d from each dimension could be summed to give a single *N* value over all dimensions for the Component. A graphical representation of this process is shown below.

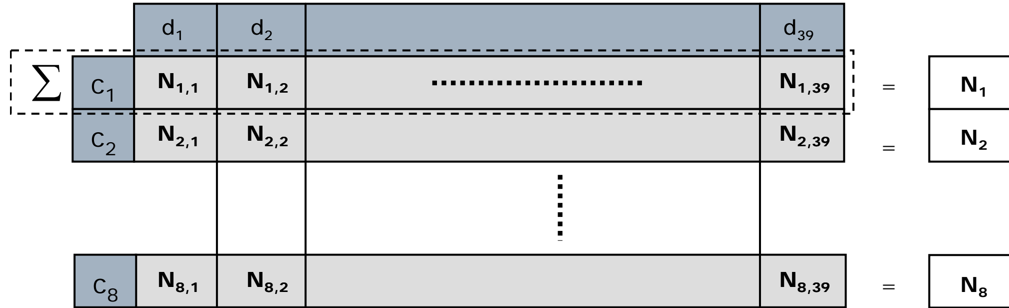


Figure 5.5: Summation of N's of individual dimensions of a Component

On obtaining the *N* for each component, it was necessary to look at it from a Senone perspective which is a combination of the Components. It can be seen from Figure 5.1 that greater the distance from the mean, the greater would be the *N* value. Further, from Figure 4.9 Components with smaller distances dominate the final Senone score implying that the Component with the *least N* needs to be found from among the various Components of the Senone. Hence, because the "*best" N* needs to found, the technique is referred to as *"bestN"*.

However, it is not necessary that the Component obtained as the "best" is indeed the dominating Component in the Senone. Similar to SubVQ, there is some room for error. To account for this the "top" instead of "best" Components are chosen. Hence, a rank-ordering based on ascending values of *N's* over all components in the Senone is done and the "top few" are chosen. A graphical representation of picking the "top" Components is show in Figure 5.6.

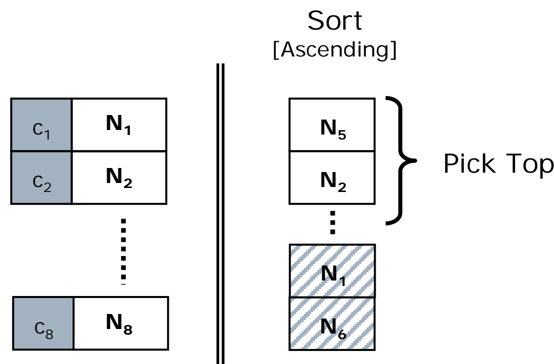


Figure 5.6: Selection of "top" Components based on ascending order of N's in a Senone

Based on this methodology, preliminary tests were run to ascertain the validity of the assumptions made. The preliminary analysis conducted consisted of 1 utterance from the RM1 corpus consisting of 398 frames of speech. These 398 frames of speech correspond to the

evaluation of 770,130 Senones. For the analysis, percentage error in various segments of Senone Scores was compiled. Three different approaches differing only in the kind of coarse-grain approach were taken.

The first approach was based on the Sub-vector Quantized models of Sphinx while the second one was based on the bestN method using SubVQ Models. The result from both these approaches is a *shortlist* (best few) of the Components which are closest to the Senone. The third approach was to find the effect of using the bestN approach over the full-continuous model whereby the top Components based on their *N* values were selected as the *shortlist*. The *shortlisted* Components are then computed using the full-continuous model and the Senone Scores were obtained. Each of these approaches has been labeled as Sphinx_SubVQ, bestN_SubVQ and bestN_FULL respectively. The results of the analysis are shown in the Table 5.1.

Table 5.1: Percentage Error in Senone Scores of the three setups w.r.t. the original scores

	TOTAL Senones = 770,130 [RM1] [1utt = 398 Frames]					
	Actual Numbers			Percentages		
% Error	Sphinx_SubVQ	bestN_subVQ	bestN_FULL	Sphinx_SubVQ	bestN_subVQ	bestN_FULL
< 1000	770,101	770,036	770,086	100.00	99.99	99.99
< 500	770,068	769,936	770,032	99.99	99.97	99.99
< 100	766,033	768,410	769,569	99.47	99.78	99.93
< 50	746,549	762,299	768,659	96.94	98.98	99.81
< 25	700,673	742,915	765,864	90.98	96.47	99.45
< 15	660,994	721,752	761,953	85.83	93.72	98.94
< 10	632,932	705,070	757,894	82.19	91.55	98.41
< 5	567,165	681,836	750,384	73.65	88.54	97.44
< 2	566,830	661,129	741,442	73.60	85.85	96.27
< 1	550,372	649,204	735,312	71.46	84.30	95.48

The table shows a detailed data analysis of the ballpark of the percentage errors of Senone Scores obtained by computing all Components versus those obtained by the techniques outlined above. The percentage errors are shown for various error ranges. The left half of the table shows the actual number of Senones that lie within a specified percentage range and the right side shows the percentage of Senones that lie within the specified error percentage.

From the table the very first observation made was that over 95% percent of the scores obtained based on the bestN_FULL approach had less than 1% error. This is a very important statistic since the evaluation of the *N's* is based on the true, un-quantized model. Hence, because of the phenomenal result of a majority of the Senone scores being computed within 1% error,

this statistic shows that the component with the least distance can be predicted very accurately based on the *"N"* method. From this it was concluded that the "bestN" method was a sound technique.

Further, it was observed that results from the bestN_SubVQ approach were not as good as those obtained from bestN_FULL. This loss in accuracy is a result of computing the *N's* for SubVQ model, quantized version of the full-continuous model. This shows that quantization leads to errors and therefore more Components need to be evaluated as part of the *shortlist* to ensure that the Senone Scores are being computed accurately. In effect this implies the selection of the "top few" Components instead of the "top" Component.

Lastly, it was seen that the results obtained by using Sphinx_SubVQ approach, only 72% of the data was below the 1% error when compared to bestN_SubVQ's 85%. Similar results were obtained while calculating average percentage error for all Senones per frame. A graph showing the results for the same sample utterance consisting of 398 frames is shown in Figure 5.7.

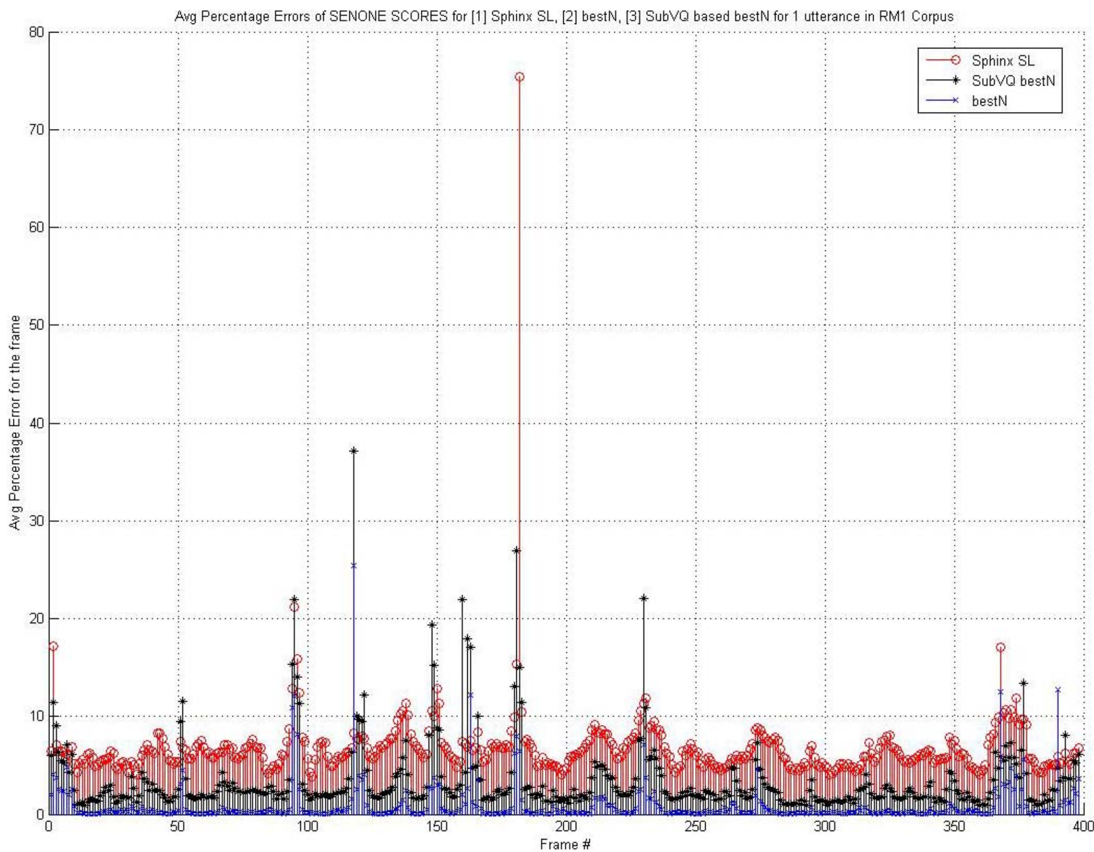


Figure 5.7: Average Percentage Error of Senone Scores per frame for all frames of sample utterance

This graph further illustrates the soundness of the bestN technique. The average percentage error for all Senones in a frame are consistently better for bestN_FULL followed by SubVQ_bestN which are in turn much better than that for Sphinx_SL. Based on these results, it was concluded that the bestN technique is a good approach and further analysis was carried out.

5.4 IMPLEMENTATION ASPECTS

The discussion so far has been theoretical. For the analysis performed to get the above results, computations were performed on-the-fly. This on-the-fly computation is fairly computationally intensive since it not only requires the computation of the "distance" value, but as shown in Eq. 5.6, the computation of the N is based on the square-root of the distance. A square-root operation is several times more involved than several multiply and add operations put together. Hence, the observation so far is possible in hindsight and a more efficient approach is needed that would make the implementation of this technique in systems more practical.

For decreasing the number of computations, a look-up based approach is ideal since pre-computed values need to be looked-up from memory as opposed to on-the-fly computations. The first step therefore was to analyze the possibility of using this look-up based approach for the bestN technique. One of the limitations of a look-up based approach is that the data set needs to be quantized. The remainder of this section deals with analyzing the effects of quantization of the only variables, N and the x.

The first step was to study the effect of quantizing N when computed on-the-fly. Several quantization levels were analyzed. Based on these results, the most significant ones, corresponding to quantization levels of full precision, $1/10^{th}$ (0.1), $1/4^{th}$ (0.25), ½ (0.5) and 1 (integer) are shown in Table 5.2. Similar to Table 5.1, the percentage of Senones lying in various percentage error ranges for Senone scores computed using full Sphinx versus bestN_SubVQ is shown.

Table 5.2: Ballpark of Senone Scores on quantizing for N for different levels

TOTAL Senones = 770,130 [RM1] [1utt = 398 Frames]					
	Quantization Level				
bestN_subVQ	FULL	0.1	0.25	0.5	1
< 1000	99.99	99.99	99.99	99.99	99.99
< 500	99.97	99.97	99.97	99.97	99.97
< 100	99.78	99.78	99.77	99.76	99.72
< 50	98.98	98.98	98.97	98.91	98.71
< 25	96.47	96.45	96.42	96.27	95.73
< 15	93.72	93.71	93.66	93.43	92.66
< 10	91.55	91.54	91.48	91.22	90.32
< 5	88.54	88.52	88.44	88.14	87.10
< 2	85.85	85.83	85.75	85.42	84.28
< 1	84.30	84.28	84.19	83.87	82.67

From the table it can be seen that there was a minimal effect of quantization. The percent of values that lie within 1% error were same for no quantization (full precision), to quantizing based on $1/10^{th}$ quantization. Further, when the values are quantized as integers (extreme right column), only a 2% drop in the number of values within 1% error is observed. A similar trend is repeated throughout the table.

From this observation it was concluded that *integer* based *N* values would be sufficient to provide the necessary accuracy. Therefore, integer based addition could replace fixed-precision based multiply and add/subtract operations.

The last step in the analysis was to quantize the only variable in Eq. 5.1, x_d. The quantization of x_d would enable for a true pre-computation/look-up based implementation. This quantization results in the creation of “bins” where each bin corresponds to pre-defined x range. A graphical representation of how such an approach would be implemented is shown in Figure 5.8.

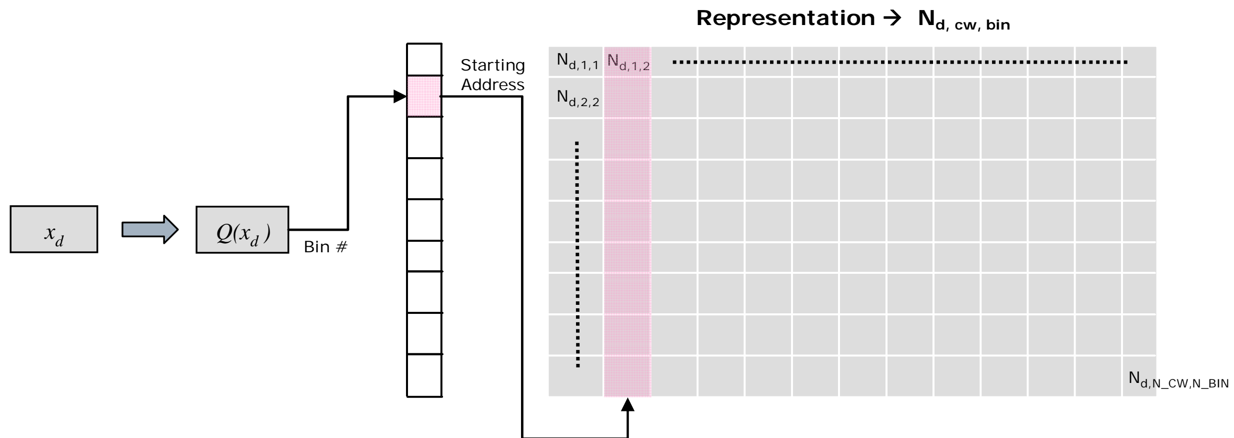


Figure 5.8: Implementation technique for bestN in a practical system

The input feature, x_d is first processed through a quantizing function which helps identify the bin corresponding to the input. On obtaining the bin number, the starting address corresponding to this bin is found and the *N's* corresponding to the quantized input for all Codewords can be accessed.

The quantization function could be based on either a uniform or non-uniform quantization scheme. To keep the implementation of the quantization function simple, uniform quantization was first considered. For this, then entire range corresponding to x_d was broken-up into bins with equal data ranges. Experiments proved that satisfactory results were obtained on using uniform quantization and hence non-uniform quantization was not considered. From an implementation perspective, uniformly quantized bins gave the ability to have a single quantization function for all dimensions and thereby help reduce implementation complexity.

A pseudo-code representation for the SubVQ_bestN implementation is shown below.

```
1   for frame
2          for d = 1 to N_COEFF
3                 bin_d = quantize{x_d}
4          end for
5          for cw = 1 to N_CW
6                 for d = 1 to N_COEFF
7                        N_cw = N_cw + N_d,cw,bin_d
8                 end for
9          end for
10         for s = 1 to N_SENONE
11                N_cw_SORT = ASCENDING_SORT {N_cw[MAP_s,1],..., N_cw[MAP_s,8]}
12                for f = 1 to N_SHORTLIST
13                       COMPONENT_ACTIVE_s,Ncw_SORT[f] = 1
14                end for
15         end for
16  end for
```

Figure 5.9: Pseudo-code representation of the bestN implementation

This code is written in a manner such that it is closer to the SubVQ implementation. It can be seen that lines 5 thru 9 consist of only addition operations which is in stark contrast to SubVQ's Gaussian Likelihood evaluation requiring 32-bit 2 multiply and 2 add/subtract operations. Line 11 deals with sorting the N's for each component amongst which the "top" few (given by N_SHORTLIST) are considered to be active (lines 12 thru 14). This is one way of implementing the bestN method based on SubVQ.

However, for better performance, the stated code can be modified by interchanging the *cw* and *d* loops of lines 5 and 6. This change would enable in accessing successive N's for a given bin instead of shuffling between the dimensions and thereby enable for a burst-mode DDR memory access implementation resulting in higher throughput and fewer burnt cycles.

5.5 TEST RESULTS

Based on the analysis presented above, two quantities were fixed:

1. The quantization of *N* was fixed to being an integer. The maximum range of *N* over the entire *x*-rage for all dimensions is 256 and therefore can be represented by 8-bits per entry.
2. Based on the dynamic range of *x*, 120 bins were fixed and the ranges uniformly quantized.

Upon fixing these quantities, detailed analysis were run on the entire RM1 Corpus. Three sets of experiments were run for 3 different setups. The three setups correspond to the bestN based on the full continuous model (bestN_FULL) with quantized versions of *N* and the quantized version of bestN_SubVQ without bins and with bins. The sets of experiments themselves only differ based on the number of "top" Components that were part of the shortlist.

Since Senone scores are intermediate results, the final results are quoted as a function of the Word Error Rate (WER) which show the accuracy of the entire system as a whole. The results were obtained by running the entire set of test sentences of the RM1 Corpus. The RM1 Test Corpus consists of 40 Speakers covering 8 different dialects. There are 42 utterances per speaker, 1680 in total corresponding to almost 96 minutes of speech. A tabular column depicting the results obtained is shown below.

Table 5.3: the Word Error Rate Results for 3 setups of bestN

TEST-RUN #	BASELINE	1	2	5	3	4	6	7	8	9
Shortlist size	-	3			2			1		
Setup	Original Sphinx	bestN_FULL	SVQ_bestN	SVQ_bestN_bins	bestN_FULL	SVQ_bestN	SVQ_bestN_bins	bestN_FULL	SVQ_bestN	SVQ_bestN_bins
Bins	-	-	-	Yes	-	-	Yes	-	-	Yes
% Savings	0	62.5			75			87.5		
WER	3	3.1	3.2	3.1	3.7	4.6	4.3	6.2	8.1	8

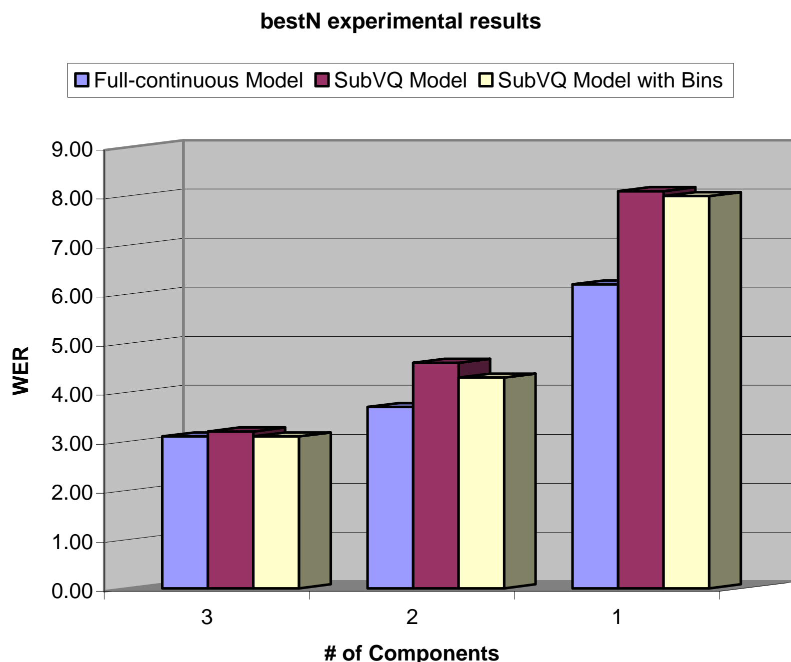


Figure 5.10: Word Error Rates for 3 sets of experiements with varying use of the number of Components

From the graph above, it can be seen that a shortlist of 3 Components gave the best performance with only 0.1% increase in the WER over baseline Sphinx's WER of 3.0. The bestN_FULL and bestN_SubVQ with bins have the same error rate. This is a very good indication that a greater shortlist can hide some of the effects due to quantization of the original models. As the shortlist size decreases, the WER increases. This can be thought of a direct result of the ability to pick the most contributing Components. bestN_FULL however performs a lot better than the other two methods. This can be attributed to the effect of quantization of the original models. The same trend extends to selecting the "top" component in the final experimental set.

5.6 CONCLUSION

Based on these results it is clear that the bestN method can help reduce the number of computations significantly. While SubVQ, based on 4096 Components requiring 160k Gaussians to be evaluated each comprising of 32-bit 2 multiplies and 2 add/subtract operations, the bestN

technique requires only 160k addition of 8-bit quantities. Hence, simple 8-bit adders can be implemented in the system to achieve the shortlist of Components.

Further, the evaluation of 160k Gaussians requires access to 2 32-bit values (mean, variance pairs) implying a bandwidth requirement of 130.21 Mbytes/sec. The bestN technique requires only 16.27 Mbytes/sec, a reduction by a factor of 8. However, because of using bins, the memory size requirement is greater for bestN. Considering 4096 Components, 120-bins and 39 dimensions, 19.16 MB of memory is required. (as opposed to 1.3MB for SubVQ)

From an implementation perspective, the increase in the required memory size is not a problem since the capacities of DRAM are increasing. Further, since only limited amount of data needs to be accessed at any given point in time, 160 KB, coupled with the ability to access data from consecutive memory locations allowing for burst-mode transfer, the computations can help in a low-power implementation.

In summary, a 8x reduction in the number of compute cycles, data bandwidth, and the required on-chip resources can be achieved by using the bestN computation reduction technique thereby addressing accuracy, overhead and implementation considerations simultaneously.

6.0 PHONEME & WORD MODELING

6.1 INTRODUCTION

This chapter is fully devoted to the discussion of the Phone and Word Blocks. Continuing the description of the system in decreasing order of computational complexity, the Phone Block followed by the Word Block is discussed in the subsequent sections. Details on the feedback generated by each of these blocks are also discussed.

The discussion of the Phone block starts with the discussion of the top-level phone block and its associated data-structures. This is followed by the description of the Senone Active List (SAL) feedback generated by the SAL_GEN Block using phone data-structures. After this, implementation details for the various blocks is provided along with a data flow description.

The Word Block is discussed in Section 6.3. A top-level working description of the block is first provided. This is followed by the discussion of the mapping of data structures from software to hardware. Finally, a dynamic memory allocation scheme is discussed that allows for maintaining active data.

6.2 PHONE BLOCK

6.2.1 Introduction

As discussed in Section 2.2.2, phonetic sounds are modeled by a 3-state Hidden Markov Model (HMM) statistical model (Figure 6.1). Each state, H_n, has a Gaussian distribution, Senone S_n, and a pair of transition probabilities pertaining to either staying in the same state, $t_{(n)(n)}$, or transitioning into the next, $t_{(n)(n+1)}$,. ssociated with it.

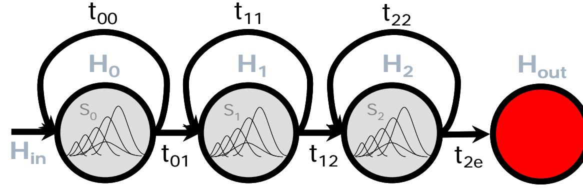


Figure 6.1: 3-state HMM Structure

As stated in Section 2.2.2, there are two main computations at this stage. The first one deals with calculating the scores of the individual states themselves, and the second one uses the computed score to determine pruning and propagation of the phone. Based on the set threshold for that frame, if the score of the phone falls below the threshold, then the phone is de-activated and if H_{out}, output score of the phone lies within the threshold, then the phone can be propagated to the next one(s) according to the word tree structure.

Restating the equations used towards phone computations,

$$\mathrm{H_0(t) = MAX[H_{in}(t), H_0(t-1) + t_{00}]} + S_0(t) \qquad \text{Eq. 6.1}$$

$$\mathrm{H_1(t) = MAX[H_0(t-1) + t_{01}, H_1(t-1) + t_{11}]} + S_1(t) \qquad \text{Eq. 6.2}$$

$$\mathrm{H_2(t) = MAX[H_1(t-1) + t_{12}, H_2(t-1) + t_{22}]} + S_2(t) \qquad \text{Eq. 6.3}$$

$$\mathrm{H_{out}(t) = H_2(t) + t_{2e}} \qquad \text{Eq. 6.4}$$

$$\mathrm{H_{best}(t) = MAX[H_0(t), H_1(t), H_2(t)]} \qquad \text{Eq. 6.5}$$

The computations relating to obtaining the threshold is shown below.

$$HMM_TH = B_HMM + HMM_BEAM \qquad \text{Eq. 6.6}$$

$$PHN_TH = B_HMM + PHN_BEAM \qquad \text{Eq. 6.7}$$

$$WRD_TH = B_HMM_wrd + WRD_BEAM \qquad \text{Eq. 6.8}$$

where; $\mathrm{B}_HMM = \mathrm{MAX[H_{best}(t)]}$

All these computations are performed in the PHN_CALC Block.

Having obtained the 3 thresholds, $H_{best}(t)$ and $H_{out}(t)$ of every phone are checked with the HMM_TH and PHN_TH to determine if the phone can be de-activated and if the phone can be propagated. These operations require comparators for checking the scores w.r.t the thresholds.

6.2.2 Phone Block Top-Level

Considering the basic operations, the design of the phone block was fairly straight forward. As a result of the design, there are two main blocks relating to the Phone Calculation (PHN_CALC) and Phone Pruning (PHN_PRN). Since there is data dependency between current *(t)* and previous frame *(t-1)*, all the scores corresponding to H_{in}, H_0, H_1, H_2, H_{out}, and H_{best} need to be stored in memory. This storage has been implemented as part of the PHN_WRD_RAM. Apart from the scores, the PHN_Id which has state and transition information for each individual phone is also required and therefore stored along with the scores in this RAM. In effect, each entry in the PHN_WRD_RAM consists of the following entries: PHN_ID | Hin | H0 | H1 | H2 | Hout | Hbest | WRD_END | WRD_ID.

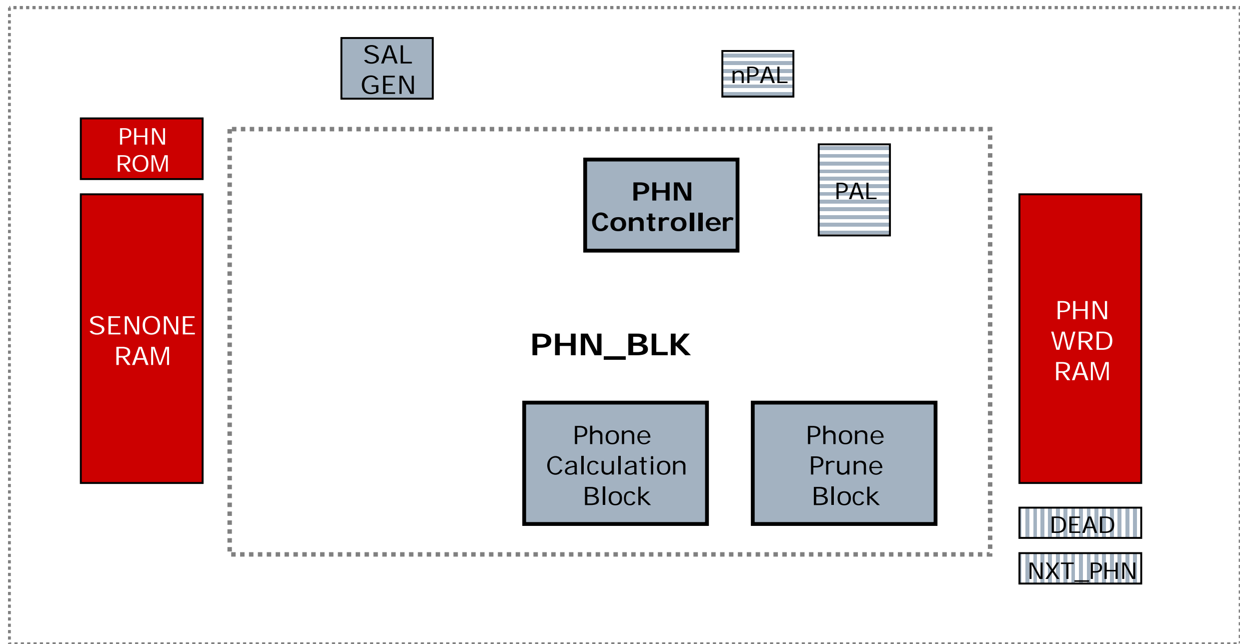


Figure 6.2: Top-level Block Diagram representing the various blocks associated with Phone Block Calculations

As mentioned earlier, all data movement between the various blocks takes place using FIFOs. The new-Phone Active List FIFO (nPAL_FIFO) and the Phone Active List FIFOs (PAL_FIFO) consist of pointers, TKNs, of active phones. The nPAL_FIFO consists of those phones that have been newly activated in the current frame, while the PAL_FIFO consists of the phones that remain to be active from the previous frame.

The phones that need to be de-activated are stored in the DEAD_FIFO while the NXT_PHN_FIFO on the other hand stores those phone TKNs that can be propagated in the word tree-structure. Both these FIFOs are writen into by PHN Block the during the PHN_PRN phase.

6.2.3 Phone Feedback: Generation of Senone Active List (SAL)

Since the computation of phone scores is based on Senone scores, Senones need to be computed first. This step assumes importance because of the enormous computational savings it provides by minimizing the Gaussian computations. On average, it helps reduce the number of Senones required to be evaluated by 50%. Hence, the first step is to generate the list of active Senones that the Acoustic Modeling Block needs to compute.

For this purpose, since every phone comprises of its own sequence of Senones, the PHN_ID of every active phone is required to obtain the list of Senones. Therefore the first processing step is to obtain the PHN_ID of corresponding to active phones stored as TKNs from both the nPAL and PAL FIFOs. After obtaining the PHN_ID, the PHN_ROM which has information about the phone to senone mapping is accessed and the entries corresponding to active PHN_IDs are passed to the SAL_GEN Block.

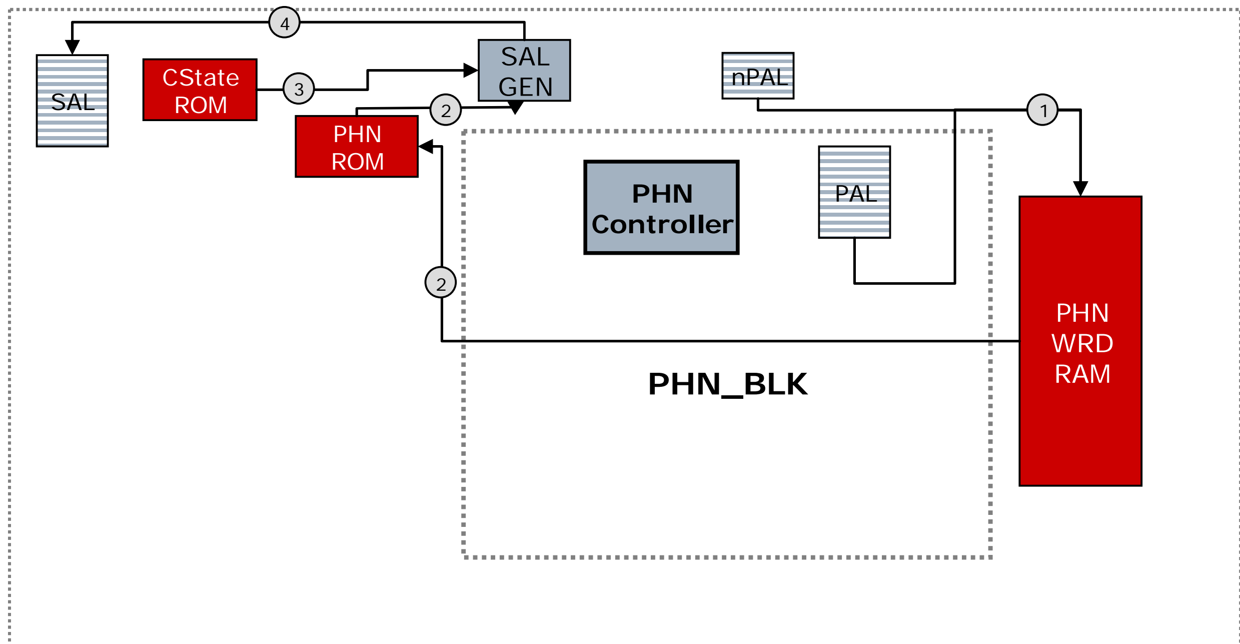


Figure 6.3: Top-level Block Diagram with Phase information for SAL Generation

The SAL_GEN Block stores the senones corresponding to active PHN_IDs sperately for each state. Doing so allows for the processing to be pipelined. Once all TKNs from the nPAL and PAL FIFOs have been processed, if composite senones are present, then they need to be processed first. For all active composites, the senones making-up the composite need to be obtained and activated. Once this is done, the Senone information from each of the states is assimilated to obtain a single list of active senones. This list is passed to the SAL FIFO which is then used by AM. A block diagram showing the blocks used for generating the SAL feedback is shown in Figure 6.2. It also contains some the data flow information of the sequence of operation. A pseudo-code representation of generation of this feedback technique is shown below.

```
/* Process nPAL FIFO */
while (nPAL not empty)
       TKN_ID = POP[nPAL]
       PHN_ID = PHN_WRD_RAM[TKN_ID]
       [SID_0 SID_1 SID_2] = PHN_ROM[PHN_ID]
       SEN_ACTIVE_ROM0[SID_0] = 1;
       SEN_ACTIVE_ROM1[SID_1] = 1;
       SEN_ACTIVE_ROM2[SID_2] = 1;
end while

/* Similar processing is done for the PAL FIFO */
while (PAL not empty)
       TKN_ID = POP[PAL]
       PHN_ID = PHN_WRD_RAM[TKN_ID]
       [SID_0 SID_1 SID_2] = PHN_ROM[PHN_ID]
       SEN_ACTIVE_ROM0[SID_0] = 1;
       SEN_ACTIVE_ROM1[SID_1] = 1;
       SEN_ACTIVE_ROM2[SID_2] = 1;
end while

/* Process Composite States (CState), if active */
if Composite states active
       for active CStates
              CSAL = PUSH CStates
              {SID_LIST} = Obtain list of SIDs from CState_ROM
              SEN_ACTIVE_ROM0[{SID_LIST}] = 1
       end for
end if

/* Assimilate the Active Senones */
for s = 1 to N_SENONE
       if (SEN_ACTIVE_ROM0[s] or SEN_ACTIVE_ROM1[s] or SEN_ACTIVE_ROM2[s])
              SAL = PUSH s into SAL
       else
              /* Do nothing */
       end if
end for
```

Figure 6.4: Pseudo-code for the Generation of the Senone Active List

6.2.4 Phone Calculation Block

The Phone Calculation Block (PHN_CALC) as the name suggests deals purely with the computation of the HMM state scores. The computation of Eq.. 6.1-6.5 is shown in the form of a block diagram (Figure. 6.3).

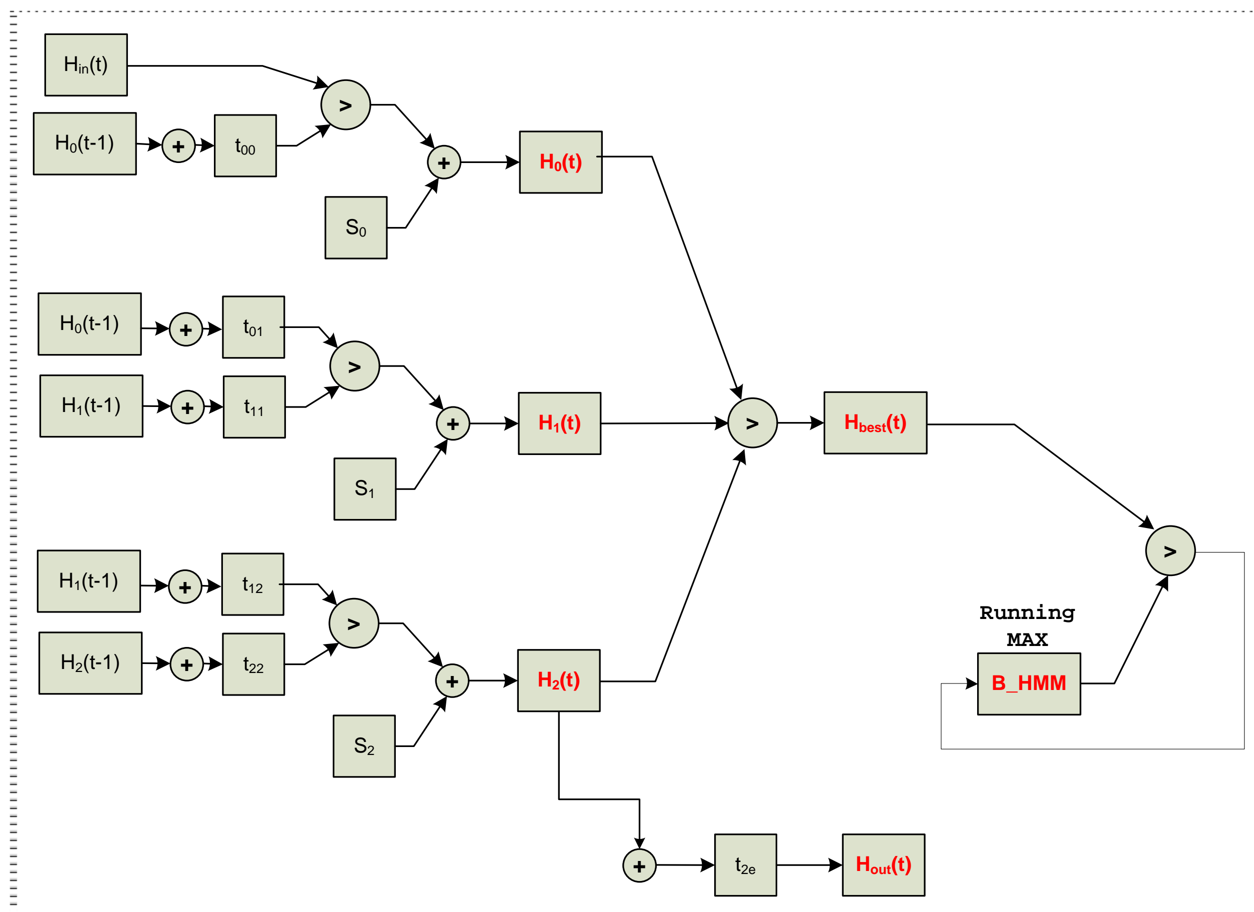


Figure 6.5: Block Diagram representation of computations in PHN_CALC

From an implementation perspective, it can be seen from this diagram that the computation of $H_0(t)$, $H_1(t)$ and $H_2(t)$ can be performed in parallel. Such an implementation however would require the availability of a huge amount of data simultaneously, 3 senones scores, 6 transitions, previous HMM-state scores $H_0(t\text{-}1)$, $H_1(t\text{-}1)$, $H_2(t\text{-}1)$ and current in-score, $H_{in}(t)$. Since 3 senone scores corresponding to different Senones are required every cycle, the only way to achieve this is to provide 3 separate Senone Score_RAMs, each of which are an exact mirror copy of the other.

This implementation is motivated from the aim of reducing the number of compute cycles, in this case, by a factor of 3. However, depending on the chip architecture and the resource constraints, this implementation can be modified so to perform 1 HMM-state score evaluation per cycle thereby requiring only $1/3^{rd}$ the number of math resources and a single Senone_RAM.

The data flow for PHN_CALC is shown in Figure 6.6. The first step is to pop active TKNs from the nPAL and PAL FIFOs. For every popped TKN, PHN_ID, H_0(t-1), H_1(t-1), H_2(t-1) and H_{in}(t) are obtained from the PHN_WRD_RAM. Corresponding to the PHN_ID, Senone Scores are obtained from the SENONE_RAM after a look-up into the PHN_ROM. Once all the information is ready, phone calculations are performed in the PHN_CALC Block, the output of which is stored back into the location it was accessed from in the PHN_WRD_RAM.

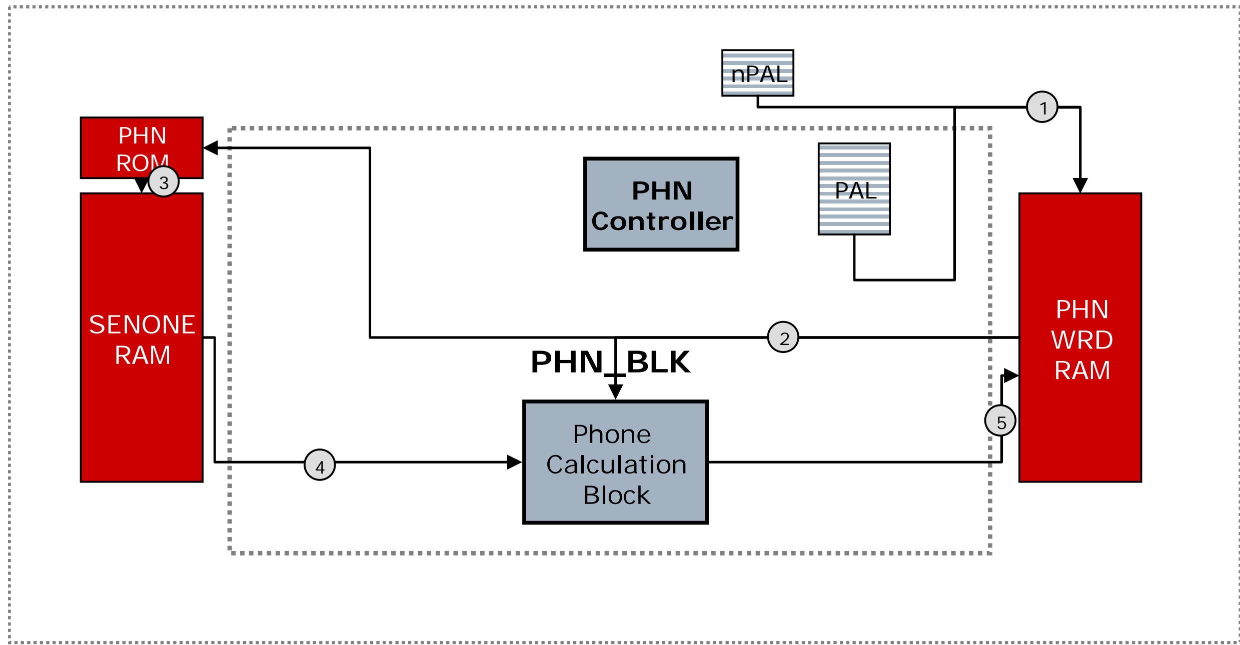


Figure 6.6: Top-level Block Diagram with Phase information for PHN_CALC

6.2.5 Prune Block

As mentioned earlier, once the HMM-state scores have been computed, the pruning and propagation of phones can be done. This operation is performed in the Phone Prune Block (PHN_PRN). The computations in this block are a simple compare operation and can be summarized by the pseudo-code shown below.

```
while (PAL not empty)
       TKN_ID = POP[PAL]
       [OUT_SCR BEST_SCR] = PHN_WRD_RAM[TKN_ID]
       if (BEST_SCR > HMM_TH)
              PUSH TKN into PAL          /* keep as active */
       else
              PUSH TKN into DEAD         /* de-activate */
       end if
       if (OUT_SCR > PHN_TH)
              PUSH TKN into NXT_PHN      /* propagate to next phone(s) */
       end if
 end while
```

Figure 6.7: Pseudo-code for the Phone Prune Phase

A graphical representation of the data-flow is shown in Figure 6.6.

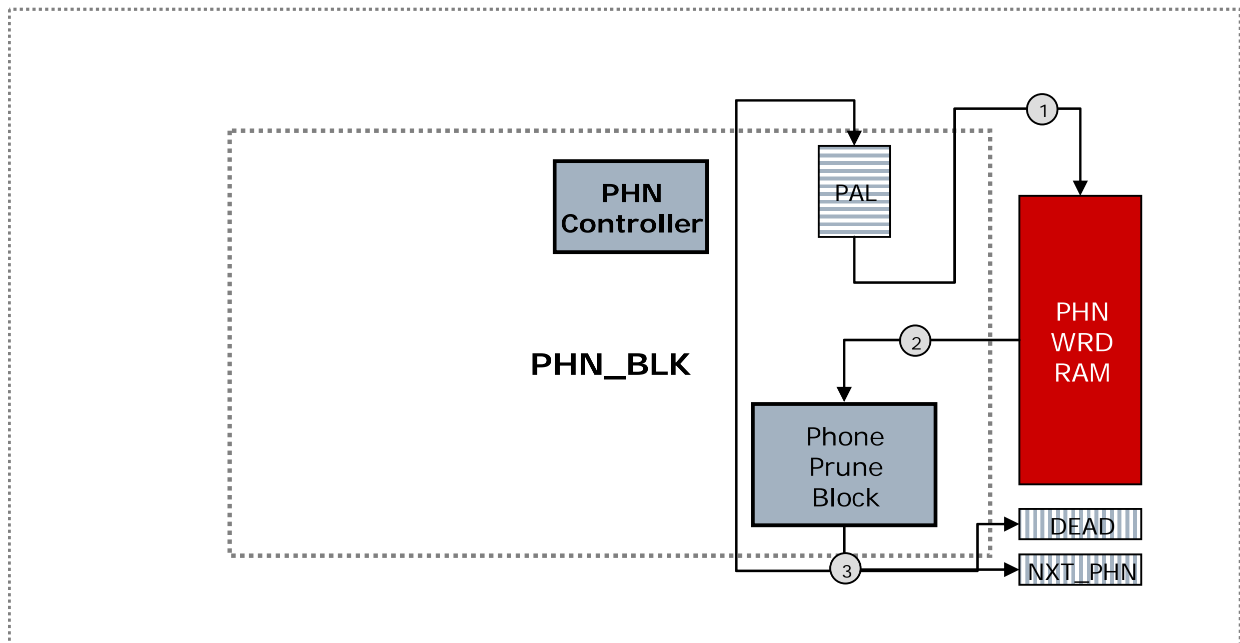


Figure 6.8: Top-level Block Diagram with Phase information for PHN_PRN

The phone pruning phase is the final phone processing phase.

6.3 WORD BLOCK

6.3.1 Introduction

The Word Block, as stated in Section 2.2.1, essentially consists of a word-to-phone mapping. As stated in Section 2.2.1, in order to decrease the number of computations, it is standard practice to implement this mapping by using a tree structure (Figure 6.9). Each circle in this figure represents a phone.

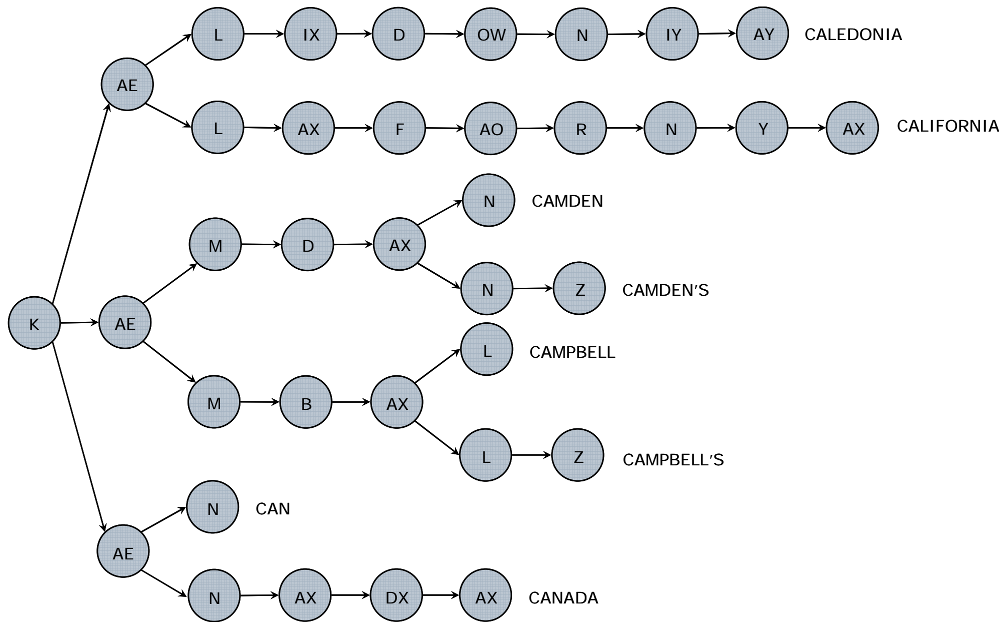


Figure 6.9: 8-word sample dictionary tree structure

In tree structure terminology, each circle is known as a *node*. Each node consists of both phone and word data. The phone data corresponds to the PHN_ID while the word data corresponds to the Probability of observing the word (LM_SCR). It also comprises of whether the phone corresponds to the last phone of a word (WRD_END). This information helps process the cross-word effects using Composite Senones and the propagation of the exited word into other words with the use of LC_ROOT_ROM (for the recognition of sequences of words). All this WRD_END | PHN_ID | LM_SCR information corresponding to each node is stored in the WRD_Database.

One of the first challenges during the design and implementation process was the mapping of the tree structure into hardware. In software it is implemented as a linked-list of linked-lists. A similar approach was taken for hardware design. Similar to a software linked-list structure, which as a starting address and a bunch of next addresses along with data which is traced until the NULL pointer indicating the end of list, hardware uses a starting address and a count of the number of entries in the list.

The starting address acts as the base address and is incremented by one every cycle until the count of number of values read is equal to the count of entries in this list. The resulting address acts as the read address into another ROM structure which contains the necessary data. In short, there are two memory units required to represent the tree structure – a starting address | count ROM and a data ROM. The word lexical tree is mapped using this methodology. The resulting data structure (consisting of two ROMs) is referred to as Lextree_ROM.

A top-level block diagram showing the various blocks in the Word block and all the various blocks it interacts with during processing are shown in Figure 6.10.

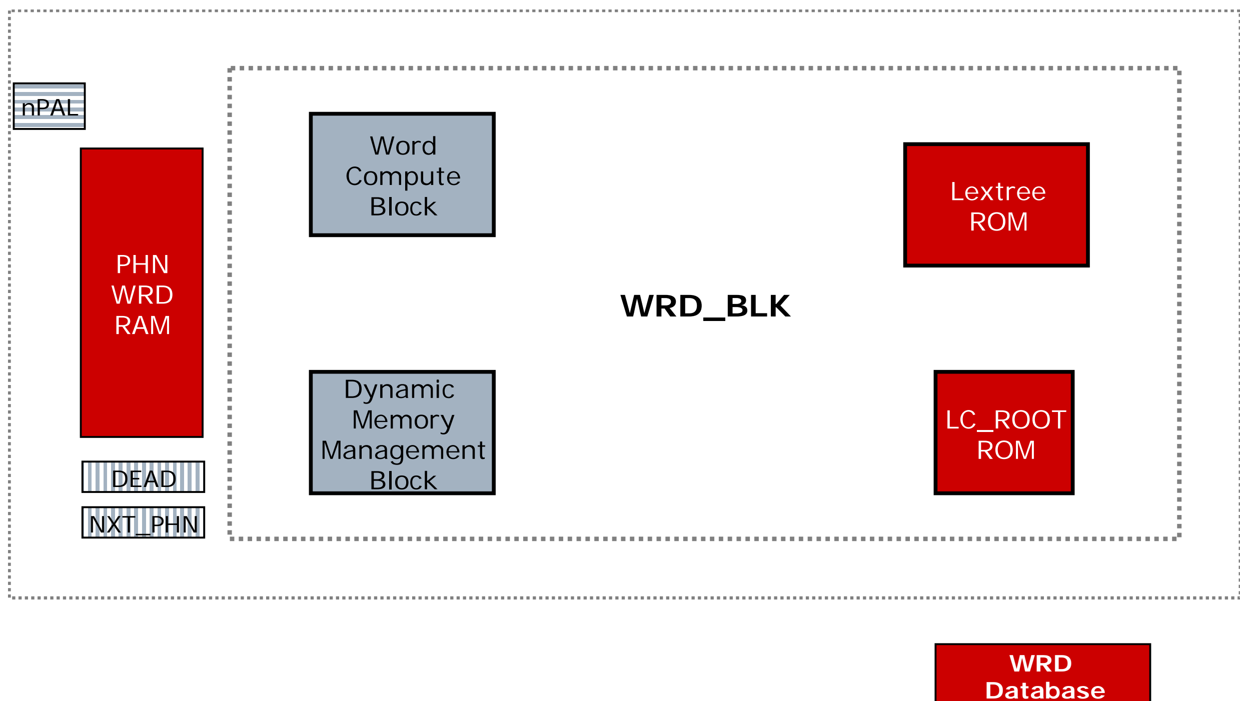


Figure 6.10: Top-level Block Diagram representing the various blocks associated with Word Block Calculations

Apart from the Lextree_ROM, LC_ROOT_ROM and WRD_Database, there are a couple of other blocks. These are the Word Compute Block (WRD_CMPT) and Dynamic Memory Management Block (MEM_MGMT). As the name suggests, the WRD_CMPT Block is responsible for all the computation that takes place in the entire WRD Block. The computation relates to deciding whether a word can be deemed to be successfully exited and towards what the H_{in} of the active phones is by making performing simple addition and compare operations. Details are provided later.

Since there are thousands of nodes in any given dictionary, having a static memory allocation scheme would imply a one-to-one mapping whereby for every node in the WRD_Database a corresponding entry in the PHN_WRD_RAM would be required to store the computed phone data. This however is not practical since the width of the PHN_WRD_RAM is of the order of a couple of hundred bits per entry. Further, the power consumption of such a big RAM also requires that the PHN_WRD_RAM be as small as practically possible.

Realizing this, it is essential to incorporate a mechanism whereby only the active data needed to be stored. The MEM_MGMT Block serves this purpose. It is responsible for managing the memory space of active phones. The memory is managed by using pointers, referred to as tokens (TKN) for a hardware implementation. Since memory locations in the PHN_WRD_RAM correspond with TKNs (active phones), and the WRD_Database correspond to nodes (NODE_IDs), it was necessary to have a two way mapping between TKNs and nodes. This two way mapping allows proper access of both the PHN_WRD_RAM and the WRD_Database in a reliable way eliminating the need for static memory allocation.

The availability of free memory locations is maintained within a FIFO (TKN_FIFO) whereby free TKN locations are stored in the FIFO. This way, a dynamic memory allocation scheme was incorporated in the system, the presence of which allows for the implementation of a truly scalable system.

With the knowledge of the top-level blocks, phase and data flow details are provided in the following sections. First the dead phase is described followed by the more complicated propagation of phones phase.

6.3.2 Word Phone De-activation Phase

The first phase of the Word Block relates to de-activating phones that have been found not be so promising. The operation is quite simple. TKNs from the DEAD_FIFO are popped one at a time and the memory contents pointed by these TKNs are reset. The PHN_ID is replaced by a large quantity, greater than the total number of PHN_IDs present in the system. All scores, H_{in}, H_0, H_1, H_2, H_{out}, and H_{best} are reset to NEG_INFINITY (a very large negative value). A basic block diagram showing this is shown in Figure 6.11.

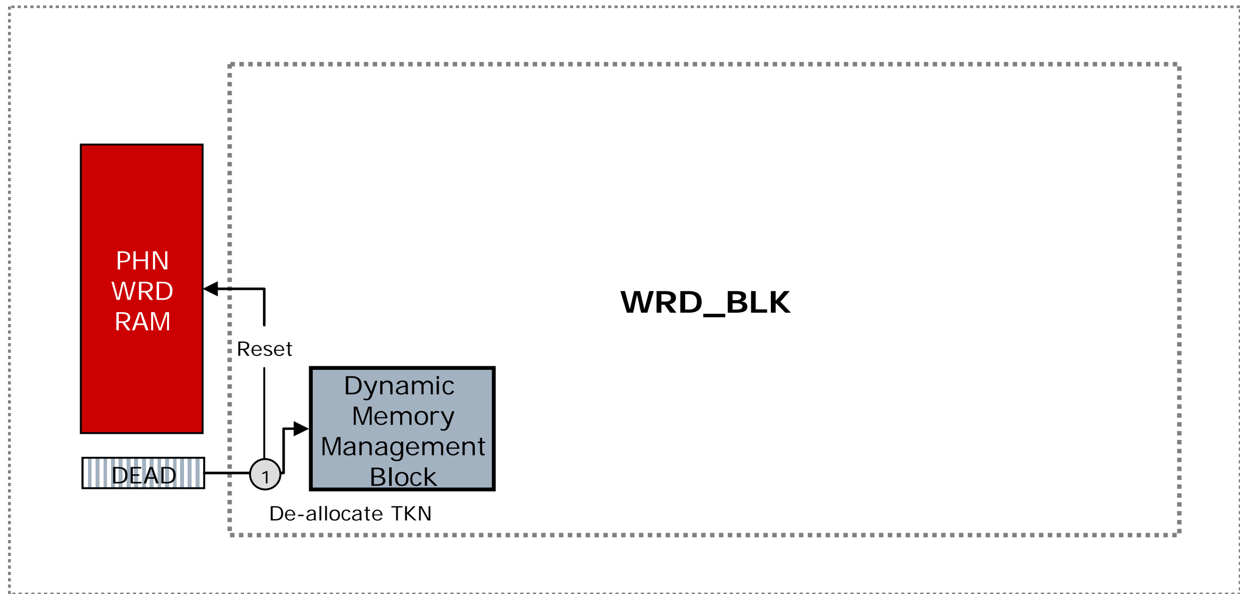


Figure 6.11: Top-level Block Diagram depicting the Dead Phase

From the Figure it can be seen that the de-activation phase is very straight forward. Only two actions need to be performed. As mentioned in the previous paragraph, all the contents of the PHN_WRD_RAM are reset. Also, the memory management unit needs to keep track of de-activated memory locations. Therefore TKNs from the DEAD_FIFO are sent to the MEM_MGMT block which stores in-active TKNs.

6.3.3 Word Phone Propagation Phase

Propagation of phone from the current to the next one(s) is a fairly complicated process. Processing of this phase accounts for the majority of the project number of cycles required for

processing data in the Word Block. At a very basic level, the processing consists of obtaining next-phone(s) for the current exiting phones from the Lextree_ROM and updating the H_{in} after using the information for each of the phones from the WRD_ROM. If this phone was previously in-active, then the TKN corresponding to such phones are pushed into the nPAL_FIFO.

From a word perspective, phones can be divided into word-end (WRD_END) phones and non-word end phones. As far as data-structures corresponding to the phone is concerned, there is no difference. The only difference is how such phones are processed when it has been determined that any one of them can be propagated. Since after exiting a word, there are several possible next phones, the beginning of the word tree for each of these need to be entered. This information is maintained is by another data-structure, LC_ROOT_ROM. Hence, depending on whether the exiting phone is a WRD_END phone, the Lextree_ROM is accessed either directly or though the LC_ROOT_ROM.

This is how phones are propagated. A more detailed description including implementation aspects is described below. For the purpose of showing the data flow, the entire phone propagation has been shown using two Figures (Figure. 6.12 and 6.13). Figure 6.12 shows operations 1 thru 4 while Figure 6.13 shows the remainder operations.

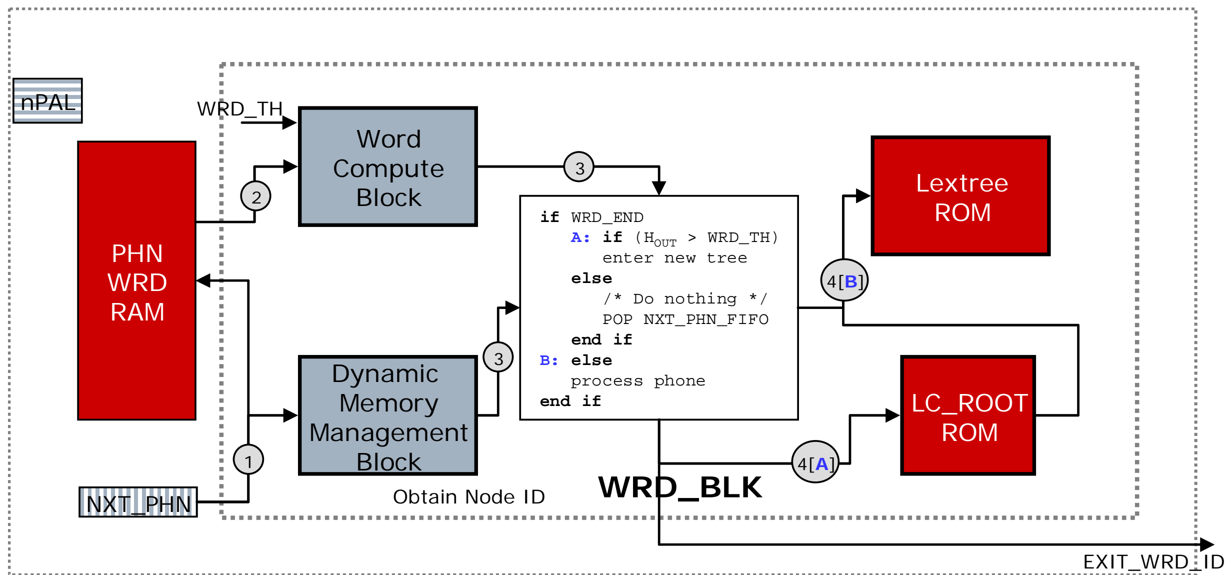


Figure 6.12: Top-level Block Diagram with Phase information for WRD_NXT_PHN phase (1)

The first step is to pop the NXT_PHN_FIFO and obtain the TKN that needs to be processed. The TKN is passed to both the PHN_WRD_RAM and the MEM_MGMT Blocks. The output from the PHN_WRD_RAM is the H_{out} score and the indication of whether this is

phone corresponds to WRD_END and is fed into the WRD_COMPUTE Block. The first step that occurs at this stage is to check if the current phone is a WRD_END phone. If this is true and if H_{out} is greater than the WRD_TH (passed on from the PHN Block) then it can be said that the word has been successfully exited. The EXIT_WRD_ID is output to both the application as well as LC_ROOT_ROM (as described above). If on the other hand if H_{out} is less than then threshold, then the phone is not deemed to have exited and therefore cannot be processed further. Hence, the next entry from the NXT_PHN_FIFO can be processed.

On the other hand, if the phone does not correspond to a WRD_END, then a TKN-to-NODE mapping needs to be done so as to be able to process it further in the Lextree_ROM. This mapping operation is done in the MEM_MGMT Block. This way, the 4th step in the processing is to read from the Lextree_ROM based on the current phone.

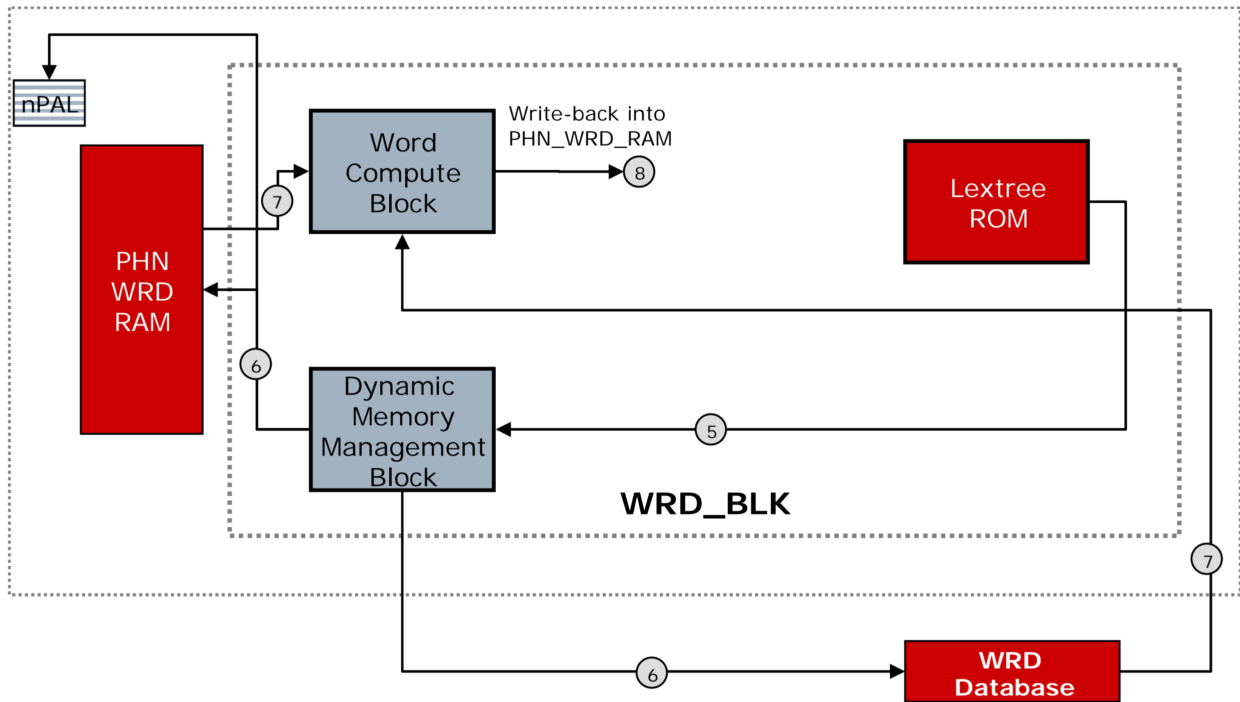


Figure 6.13: Top-level Block Diagram with Phase information for WRD_NXT_PHN phase (2)

The next phone(s) output from the Lextree_ROM is sent to the MEM_MGMT Block which keeps track of all active nodes. If the node is currently in-active then a new TKN is allocated for this node and the TKN_ID is sent to the nPAL_FIFO for the phone to be processed by the PHN Block. All TKNs (already active or newly active) are then forwarded to the PHN_WRD_RAM and the WRD Database. The data from both memories is sent to the

WRD_COMPUTE_BLK where $H_{out}(t-1)$ + LM_SCORE (of each next node) is compared against the $H_{in}(t-1)$ of the TKNs allocated to the next phone(s). The larger of the two is treated as $H_{in}(t)$ and the PHN_ID and WRD_ID data corresponding to the larger value are written into the PHN_WRD_RAM (indicated by operation 8 in Figure 6.13). This is the basic functioning of the processing that takes place when propagating phones.

The implementation of all these block has been done in a manner so as to allow for processing all next phone(s) in successive cycles (fully pipelined implementation). Since the data corresponding to the NXT_PHN TKN is used to compare values from the next phone(s), the NXT_PHN_FIFO is not popped until the processing of the current branch is completed. While this helps in simplifying the read and write accesses to the PHN_WRD_RAM, they also imply a pipeline stall. Unfortunately, this pipeline stall cannot be overcome.

7.0 SYSTEM INTEGRATION, TESTING, AREA AND PERFORMANCE

7.1 INTRODUCTION

Having described the individual blocks and how they work, this chapter discusses implementation details of the system. The first step in the implementation process was to define the target technology. This is described in the following section. Then, a brief description of the Design Environment is presented in Section 7.3. Section 7.4 discusses some of the System set-up, Testing, and Area and Performance results.

It is shown that with a fully pipelined design, a 1,000 word Command & Control based Speech Recognition application can be run faster than real-time when operating at 100 MHz. Further, it is shown that a highly efficient, fully pipelined implementation has been done on a Xilinx Virtex4 SX-35 with the entire design running at 154 MHz. Finally, the overall system resources required by this design can be implemented using 93,000 “equivalent gates”. This makes it ideally suited for incorporation as a co-processor to existing processors.

7.2 IMPLEMENTATION

7.2.1 Introduction

The first step in the implementation process was to define the target technology which would enable in achieving the two system requirements, real-time operation at 100 MHz and a system which is demonstrable. Looking at these goals, Field Programmable Gate Array (FPGAs) seem to be an ideal choice. They not only provide for extensive flexibility in the implementation process with the availability of hundreds of thousands of programmable logic slices, dedicated

math units and on-chip memory, but they also provide for a inexpensive prototyping environment.

7.2.2 FPGA Prototyping Environment

After extensive survey of the various FPGAs and the resources provided by them, Xilinx's Virtex4 FPGA [36], based on the latest 90nm fabrication technology, was chosen. The 90nm process allows for higher density chips performing at higher frequencies. The biggest Virtex4 SX chip (SX-55) contains as many as 320 memory blocks and 512 dedicated Multiply-Accumulate units (MACs), each with the ability to perform 18x18 multiplication and 48-bit accumulation on chip.

Amongst the three Virtex4 families available, LX, SX, and FX, the SX device is targeted towards heavy signal processing based applications. For this reason, it contains the maximum number of memory blocks and multiply units when compared to other Virtex4 families. Given the intensive computational nature of Speech Recognition algorithms, the SX family is an ideal choice. From a demonstration perspective, Xilinx offers a ML-402 development kit with an SX-35 device.

The ML-402 platform not only includes the SX-family chip, but also has the necessary off-chip resources making it an ideal demonstration platform. It consists of 64 MBytes of DDR memory (ideal for bandwidth intensive task of AM), a 9 Mbit SRAM (ideal for storing of the WRD Database), an LCD screen (for displaying the recognized words) and header expansion pins (enabling the interface of FPGA with the DSP development kit performing Feature Extraction). Based on these factors, the target chip decided on was the Xilinx Virtex4 SX-35 speed grade 10. A Block Diagram of the Prototyping environment is shown in Figure 7.2.

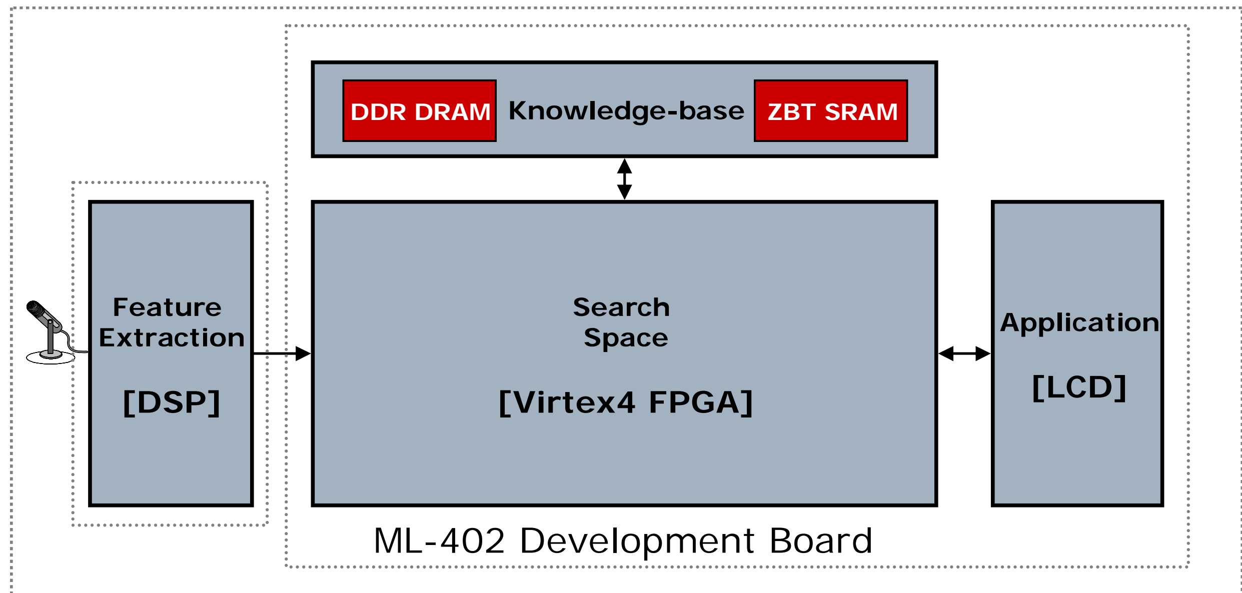


Figure 7.1: Prototyping Environment Top-Level Diagram

7.2.3 Highly efficient, fully pipelined design

Due to the sheer number of calculations involved in speech recognition, a system with tight power constraints should process data as fast as possible. Minimizing the number of computation cycles greatly helps in reducing dynamic power consumption and is critical for the design to be incorporated in embedded systems. As discussed in previous chapters, one way of achieveing this from an algorithmic perspective was to keep track of active “data” at every stage and compute only these quantities. For this feedback from one stage to the next was implemented (active Senones and Phones).

But the number of computations still required is fairly substantial. For this reason, it was necessary to decrease the number of compute cycles further. One of the best possible ways of achieving this is to pipeline the computations as much as possible. This way, even though the computation of a single quantity in reality requires multiple cycles, as in the case of Gaussian distance calculation, a pipelined design enables a higher throughput at an increased latency. If latency is not a cause of concern and if data can be streamed to ensure that the computation pipes are constantly full, the number of compute cycles can be reduced significantly.

The Gaussian Distance (GAUS_DIST) calculation of Eq. 4.9 is a very good example for this. The computation of GAUS_DIST for each dimension comprises of 1 subtract, 2 multiply and 1 add operation, thereby requiring 4 compute cycles. Given the necessary resources however, a pipelined design can reduce the number of cycles by a factor of 4 with a 4-stage pipe. Therefore, the total number of cycles required to perform brute-force AM is 603k cycles, rather than 2.4M cycles (corresponding to total number of operations).

Further, a pipelined design strategy also enables in increasing the overall performance of the system. From an implementation perspective, the prototyping set-up should have enough resources to meet these requirements. Specifically, special care was taken to ensure the availability of adequate DSP (math computation) and on-chip memory resources. Since a pipelined system would consist of many registers, an FPGA with several hundreds of thousands of logic cells is a good fit.

For this reason, the entire design is fully pipelined. The design can be thought of as being made of meta-pipes with each of the blocks with AM being a 25 stage pipe, PHN being a 15 stage pipe and WRD being a 12-stage pipe.

7.3 DESIGN ENVIRONMENT

7.3.1 Xilinx System Generator

Initially, for shortening the design cycle, a SysGen based implementation of the Gaussian Distance calculation of Acoustic Modeling was made in the MATLAB/Simulink environment. SysGen is a MATLAB Toolbox from Xilinx [37] which enables the creation of hardware blocks in Simulink. Pre-created Simulink blocks from the SysGen Library can be simply dragged and dropped into Simulink. They can then be configured according to the specific needs. After completing the design, the user can convert the Simulink design into RTL with the mere push of a button.

Although such a design environment enables in quickly implementating the design, it was observed that the F_{MAX} for Gaussian Distance evaluation was only 125 MHz, which is far below the quoted capability of the Virtex4's DSP-slices. The performance is also about 40% of that

obtained by hand-coding this block. One reason for this inferior performance can be attributed to SysGen being designed for a generic implementation.

While the generic implementation helps in easier GUI management where several details can be hidden from a naïve user with little knowledge of hardware design, it comes at the cost of significantly increased overhead for any implementation, which, in turn, results in the inferior performance observed.

To account for this loss in performance, there are certain constraints built into the tool thereby limiting the amount of flexibility a designer has when implementing certain special cases the system being implemented might require. This can be thought of as a direct result of limiting the performance degradation below a certain threshold. Because of the above two factors, it was necessary to hand-code the entire system. For this, a RTL design environment was chosen.

7.3.2 Xilinx ISE

The entire system was designed using Xiilinx's design environment, ISE 7.1. All hardware design files were coded in VHDL using Xilinx's text editor within ISE. Since synthesis of a design is based on "estimates", it does not provide a very accurate representation of the performance of the circuit after place and route. But it was observed that the Xilinx Synthesis Tool (XST) provided a fairly accurate estimate of the design's performance, being off by no more than 20 MHz when compared to the results obtained after Place & Route. This led to significantly shortening the design cycle, since critical paths could be obtained from the synthesis results. To ensure the best performance, Xilinx libraries were used for configuring the memory and math blocks. This was done using Xilinx's Core Generator front-end, a GUI-based tool that allows for configuring the memory and math units.

The entire design consists of almost 100 VHDL files! This shows the complexity of the design and the various blocks that go along with it. While some of these files correspond to the various FIFOs, memory and math blocks, there are several design files corresponding to the top-level blocks and sub-blocks. In total, there are 4 top-level entities corresponding to the System, AM, PHN, and WRD Blocks. Each of these blocks is associated with a Controller. The VHDL implementation corresponding to these blocks has been attached for reference in Appendix A.

7.4 SYSTEM SET-UP, TESTING & RESULTS

7.4.1 System Set-up

A table detailing the number of computational resources in terms of math and comparator units, along with the number of cycles required for the computations in each block based on the assumption of a fully pipelined design was discussed in detail in Section 3.2.2. To this table additional information regarding the memory size for each block shown in Figure 7.2 has been added and is shown in Table 7.1.

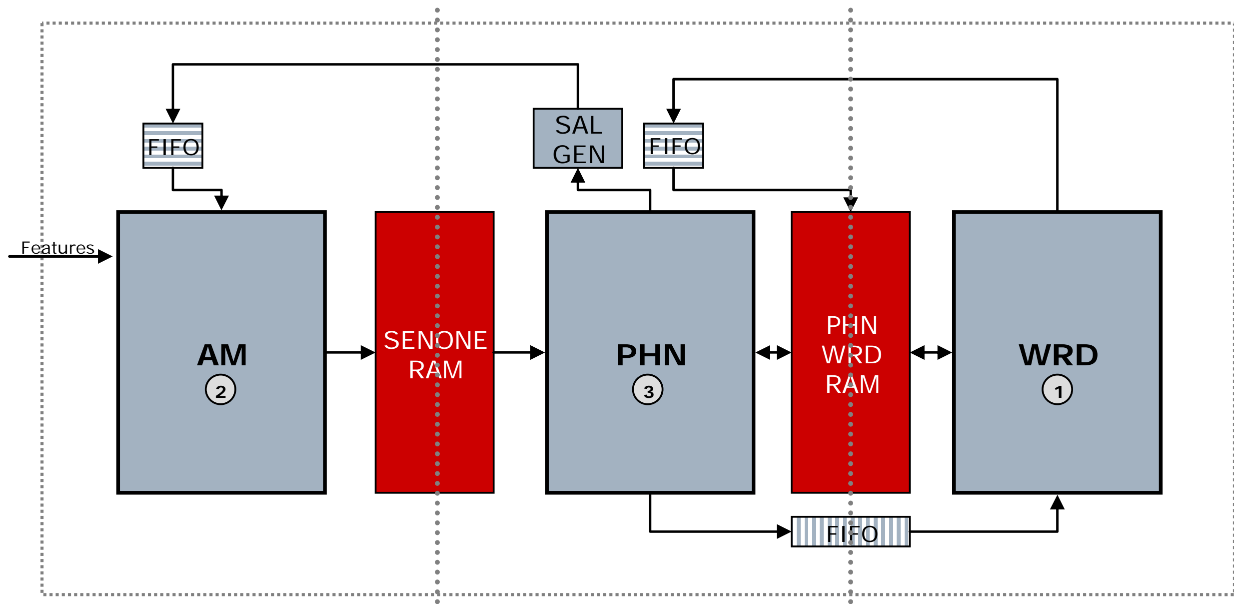


Figure 7.2: Top-Level System Block Diagram

Table 7.1: Detailed Resource and Timing Requirements of the System

		AM	AM_PHN	PHN	PHN_WRD	WRD	TOTAL
Math Units	Add	6	-	9	-	1	**16**
	Multiply	3	-	-	-	-	**3**
Comparator Units		3	-	6	-	2	**11**
Memory [Kbits]	On-chip	456	384	264	881	792	**2,777**
	Off-chip	38,638	-	-	-	2,079	**40,717**
# of cyles [per 10ms frame]		603,720	-	8,192	-	102,400	**714,312**
Memory Bandwidth [MB/sec]		495	-	-	-	5	**-**

7.4.1.1 Compute Cycles

As discussed in Section 3.2.2, while the total number of operations required by the system is significantly more than that allowed for by a 1 million cycle-budget, by pipelining the design, the number of compute cycles required to perform the same number of operations is significantly reduced. Based on this assumption, the number of compute cycles for a 10ms frame is presented in Table 7.1.

The number of cycles presented in the table is based on the assumption that the computations in AM are brute-force, with no use of feedback from PHN Block. The compute cycles for the PHN Block are based on the worst-case estimate obtained by observing Sphinx, where it was found that at any give time only 4,000 phones are simultaneously active.

Finally, the number of compute cycles in the WRD Block is based on the assumption that at any given point in time not more than half of the 4,000 active phones will be propagated. Since the WRD data-structure is based on a tree-structure, even though the design is pipelined, in order to decrease implementation complexity, phones that need to be propagated are not processed until the predecessor has been fully processed. This results in a pipeline stall. The loss in compute cycles because of this pipeline stall has been accounted in the projected number of compute cycles for the WRD Block.

Based on these estimates, assuming a 1-cycle memory access latency, it can be seen that even assuming the brute-force AM approach, all data for the RM1 1,000-word Command & Control dictionary can be processed in 710k cycles, with 290k cycles to spare (for a 1 million cycle budget). Therefore, a system based on a fully-pipelined architecture can perform at faster than real-time.

However, to decrease the number of computations further, as described in Chapter 3, feedback from the PHN to the AM Block has been implemented. It is shown in Section 7.5.3 that the incorporation of this feedback into the architecture can reduce the number of computations significantly thereby decreasing the number of overall compute cycles further from those presented in Table 7.1.

7.4.1.2 Memory

From Table 7.1 it can be seen that a decision of on-chip and off-chip memory was made. This decision was based entirely on the size of the data-structures. Since the data-structures of

both the AM and WRD Blocks are very large for being incorporated on-chip, they need to be implemented as external off-chip memories. The memories corresponding to the data-structure of these blocks contain Gaussian mean/variance pairs and the word tree structure, respectively. All other memories are fairly small, and hence can be implemented on-chip.

However, the larger the set of words that need to be recognized, the more complex the word tree structure gets. Hence, a more test-friendly environment, with as little complexity as possible, was needed to help focus on the core computations that are performed in each of the blocks. This would aid in efficiently debugging the system and identifying any bugs either in the computations themselves, or in the dataflow. For this reason, a small sub-set of 15-words was chosen from the RM1 Speech Corpus. This shortlist of words is referred to as the *test dictionary*.

The words in the test dictionary were carefully chosen so as to mimic the overall tree-structure for the entire RM1 corpus and account for any special cases that the tree-structure might include so as to test the system as thoroughly as possible by accounting for all possible cases. Hence, once the system is tested satisfactorily for the test dictionary, the entire 1,000-word dictionary can be implemented with minimal or no further de-bugging.

By using a 15-word test dictionary, the only data-structures that required change was related to the WRD Block. Specifically, the size of the WRD database is significantly smaller than that presented in Table 7.1 for the test dictionary. Hence, the WRD database in the *test system* has been implemented as on-chip. All other data-structures relating to the AM and PHN blocks need not be modified and have been implemented as shown in Table 7.1.

7.4.2 Testing

All the blocks shown in Figure 7.2 were individually tested and integrated. For the purpose of verification, first, a functionally accurate model of the system (presented in Figure 3.3) was developed in MATLAB. For the same inputs, the outputs of the AM and PHN Blocks were obtained and the necessary de-bugging was done. This way, once the core-computations were ensured to be correct, the MATLAB implementation was used as the reference for testing and de-bugging the computations in hardware.

A few simulation waveforms corresponding to the core-computations in the AM and PHN Blocks with a one-to-one comparison with MATLAB are presented in the following sub-

section. For the purpose of testing the system based on the sample 15-word test dictionary, 3-test utterances were run and found to output the correct words.

7.4.3 Simulation Results

Simulation waveforms depicting the operation of the various blocks are shown in the following figures. Figure 7.3 shows the signals of the System Controller. The waveform shows the controller going through the phases in the order outlined in Figure 3.4. The computations start from the Dead phase within the WRD Block, followed by the propagation of phones. Then the generation of SAL is performed, whereby a list of active Senones corresponding to active Phones is computed. Following this, Senone scores in the AM Block are calculated. Finally, the Phone scores are computed, followed by pruning/propagation of the phones based on whether the scores pass the computed thresholds. This indicates the completion of the processing for the current frame.

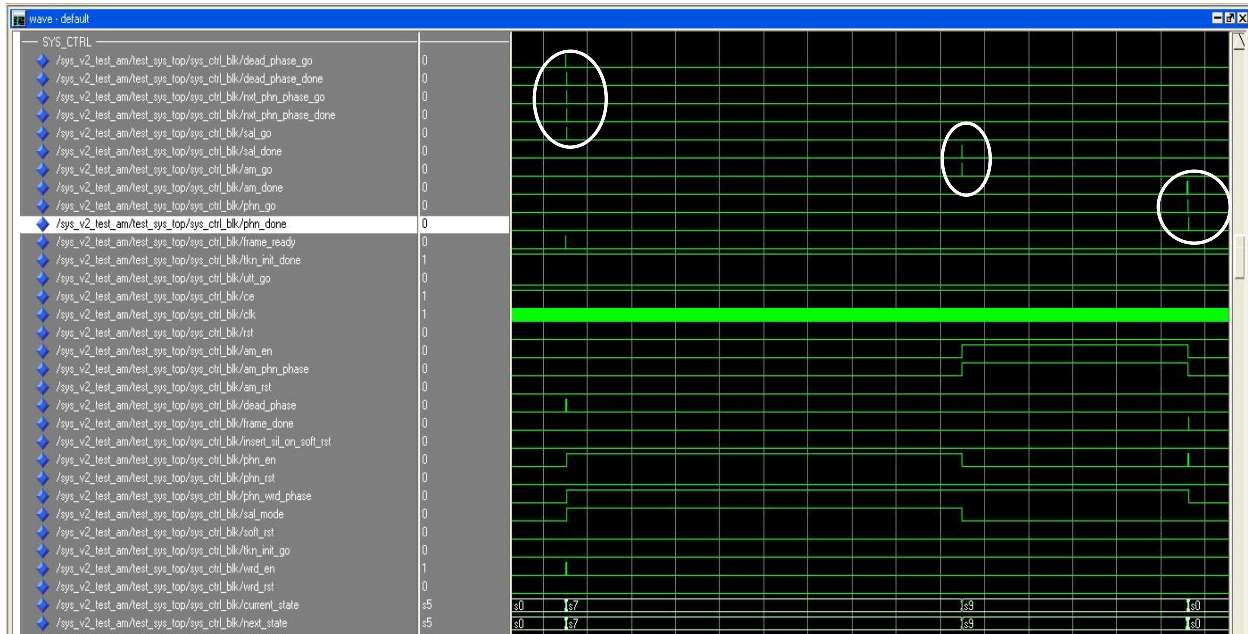


Figure 7.3: Simulation Result for System Controller going through the various phases

Figure 7.4 shows the generation of SAL corresponding to the SILENCE phone (phones 102, 103, and 104).

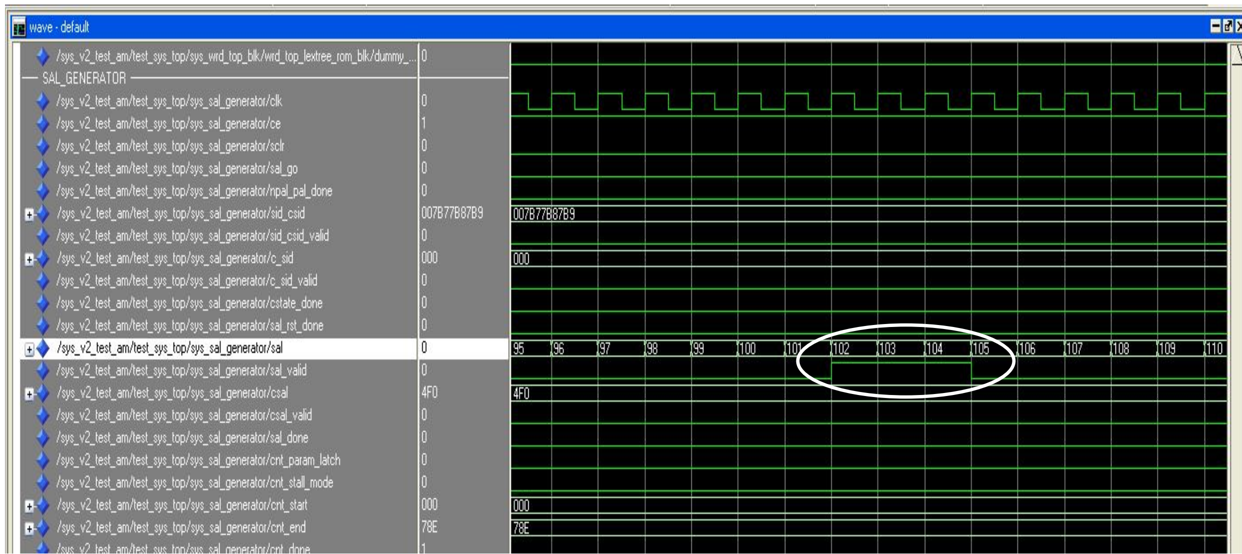


Figure 7.4: Simulation Result for SAL Generator (the active phones correspond to the SIL phone)

Figure 7.5 shows the AM Controller going through its 4 phases.

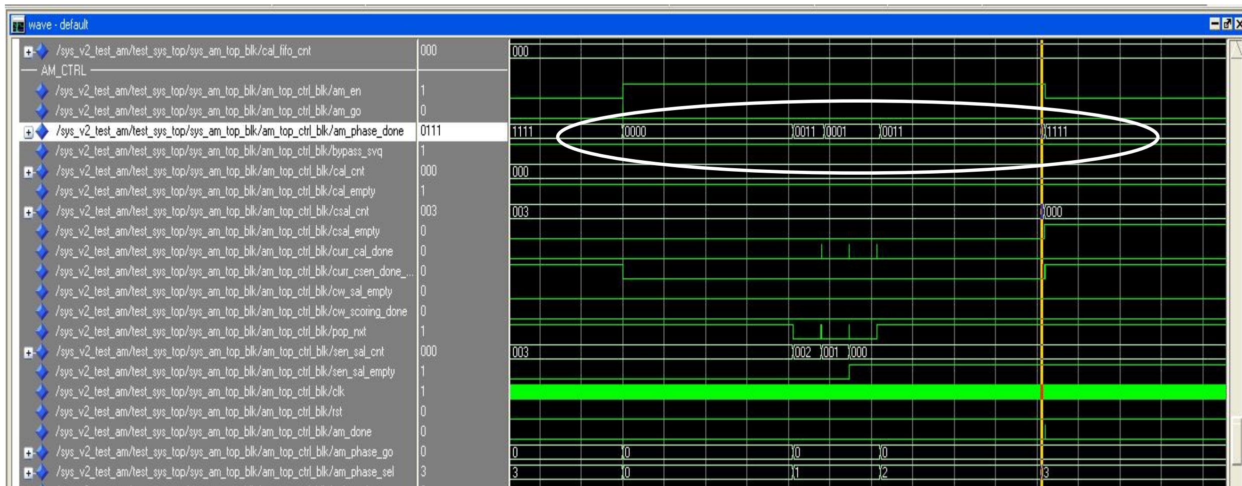


Figure 7.5: Simulation Result showing AM Controller going through the 4 phases

Finally, Figure 7.6 shows a write-back operation into the PHN_WRD_RAM after the computation of the Phone scores. This can be compared with the scores obtained from MATLAB as shown in the figure.

Figure 7.6: Simulation and MATLAB results for Phone Scores

7.5 AREA & PERFORMANCE

A summary of the resource utilization and performance of the individual blocks and the system as a whole is shown in Table 7.1. The resources of interest include the number of dedicated on-chip multiply and add blocks (DSP Slices), on-chip memory (Block RAMs) and Logic Slices required by each of the blocks. Further, from the perspective of implementation on an ASIC, the "equivalent gate count", for each of the blocks is also presented. The last column in the table details the overall performance of each of these blocks, which are based on Post Place & Route results. Finally, the last row presents the percentage of chip resources used by the entire system when implemented on a Virtex4 SX-35 chip.

Table 7.2: Area and Performance of the entire System as implemented on the Virtex4 SX-35 FPGA

Resources	DSP Slices	RAM Blocks	Logic Slices	Eq. Gate Count	Performance [MHz]
AM	19	27	1,404	27,806	210
PHN	22	14	1,201	32,076	161
WRD	3	8	730	13,143	192
SYSTEM [#]	**44**	**106**	**4,522**	**93,967**	**154**
SYSTEM [%]	22%	55%	29%	-	-

A detailed description of the utilization of individual resources is presented in the following sub-section. Performance results are discussed in Section 7.5.2. A floorplan of the final system as implemented on the Virtex4 SX-35 FPGA is shown in Figure 7.7.

7.5.1 Area

7.5.1.1 DSP Blocks

In order to ensure that the system works at peak performance, extensive use of DSP Slices was made, wherever possible. As expected, AM, the computationally intensive block, requires a fair number of DSP Slices. Since the Virtex4 architecture provides for 18x18-bit multipliers in each DSP Block, 32x32-bit multiplication is achieved by making use of 4 DSP Blocks. There are 3 multipliers in the AM Block (all used for GAUS_DIST calculation), thereby requiring 12 DSP Blocks. The rest of the DSP Slices in this block are used towards addition and subtraction operations spread over the various blocks.

It might come as a surprise that the PHN Block requires as many as 22 DSP Blocks. This is a direct result of the parallel processing approach taken for this block as pointed out in Section 6.2, where each HMM-state requires 3-additions and 1-compare operation (all 32-bits). To help achieve maximum performance, comparators have also been implemented using DSP Blocks, the result of the sign obtained by subtraction of the two quantities being compared, giving the comparison result.

Finally, as expected, since the WRD Block operation is dominated by maintaining the tree structure for the word to phone mapping, it requires only 3 DSP Blocks. It can also be seen

from the table that all DSP resources are accounted for inside the individual blocks, indicating that data-management takes place only at the top-level of the system.

7.5.1.2 RAM Blocks

The RAM Blocks in AM are mostly consumed by the LOG_ADD Look-up table. Further, a few RAM Blocks are used for storing the Senone scores. The AM_Database is considered to be external to the system and hence has not been included in the above numbers. The PHN Block of the PHN_ROM takes 14 RAM Blocks. The WRD Block consists of mainly the Lextree_ROM and the WRD_Database (implemented as on-chip for the simple 15-word test dictionary).

However, not all Block RAMs have been accounted for thus far. This is because of the two major memory data-structures at the top-level of the system, the SENONE_RAM and PHN_WRD_RAM. It could be argued that these are shared memories and hence can be treated as part of either of any of the blocks (AM or PHN or WRD). Therefore, they have been accounted for as part of the system top-level consuming over 50% of the overall memory requirement of the system.

7.5.1.3 Logic Slices

Considering that the entire design has been completely pipelined, it can be seen that a fair number of logic slices are used up by both the AM and PHN Blocks. A majority of the slices in the PHN Block are devoted towards synchronization of the Senone Scores and the Phone Scores from the previous frame. The WRD Block again requires relatively fewer logic slices.

From the synthesis reports in Xilinx, it was observed that the implementation of the entire system would require around *93,000 "equivalent gates"*. This includes all on-chip resources utilized, including DSP, memory and logic slices utilized. This is a very interesting statistic since it shows that the silicon footprint required by the design is quite small, thereby making it a practical option for implementation in commercial products available today.

7.5.1.4 Design Floorplan on Xilinx's Virtex4 SX-35

The Floorplan for the entire system as implemented on Xilinx's Virtex4 SX-35 FPGA is shown in Figure 7.7.

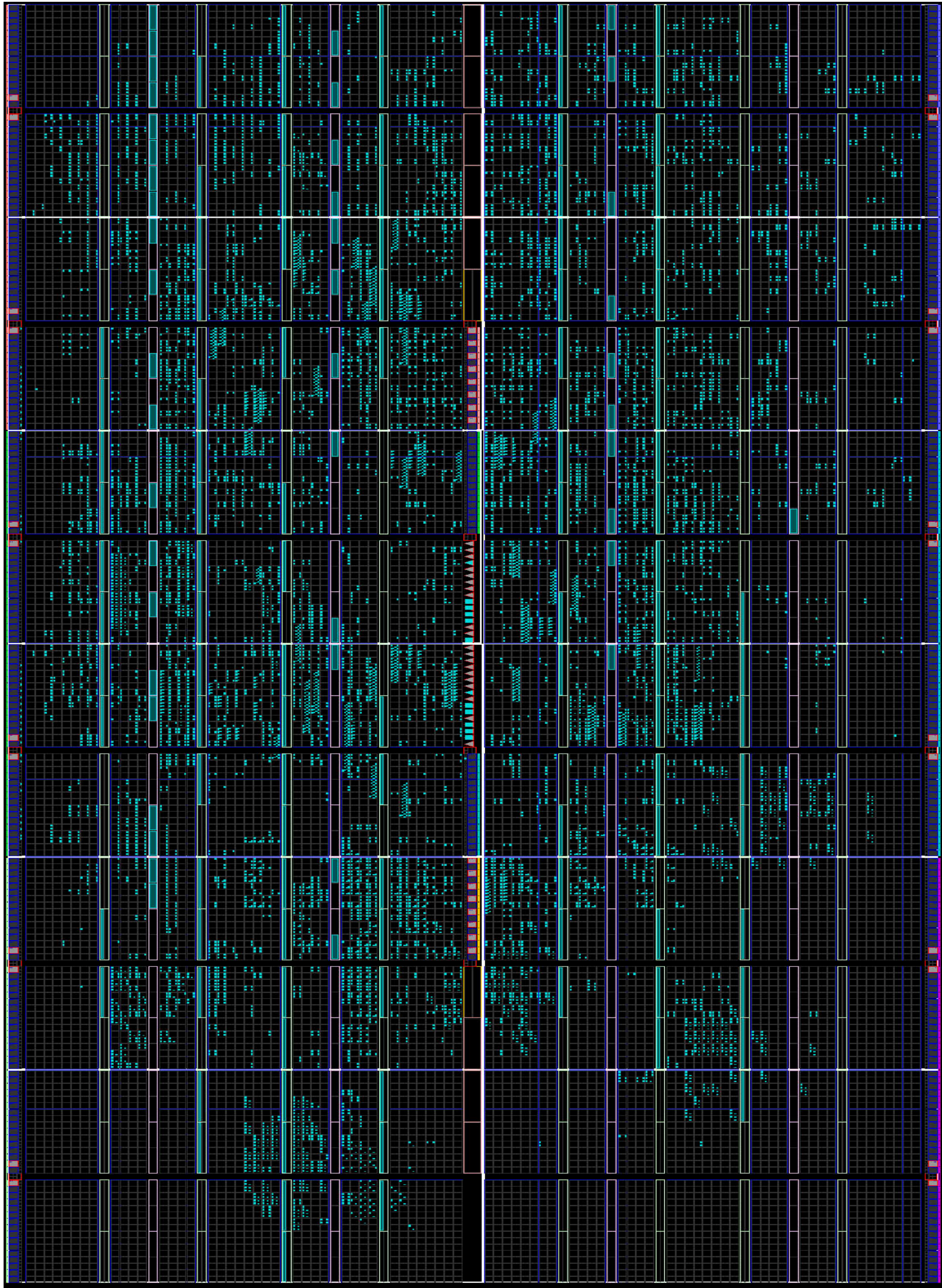


Figure 7.7: Floorplan for the entire Speech Recognition System on a Virtex4 SX-35

7.5.2 Performance

The performance of the system is shown in Table 7.2. It can be seen that the computational bottleneck in the AM Block has been successfully implemented with a performance of 210 MHz. The overall performance is limited by the LOG_ADD Block, considering that the computationally intensive GAUS_DIST Block runs at 310 MHz.

The performance of the PHN Block is limited by the requirement of accessing several memory structures, each consisting of wide data busses. Hence, despite using DSP-slice resources for the PHN_CALC Block, a peak performance of only 165 MHz could be obtained. The WRD Block, on the other hand, is not limited by this aspect and hence a performance in excess of 190 MHz was obtained.

The overall system performance slightly degrades when compared to the performance of the individual blocks. Despite several post-place and route re-runs, the critical path was consistently observed to be in the routing, with the worst critical path consisting of 25% logic delay and 75% path delay. Therefore, the overall performance of the system can be improved by better floorplanning.

7.5.3 Reduction in Computation using Feedback

Because of the incorporation of feedback at every stage, the amount of active data determines the number of computations that are performed at each stage. Hence, in order to quantify the gains from feedback, the number of active quantities from simulation would be needed. However, since this process is tedious and time-consuming, the MATLAB version of the system which is functionally equivalent to hardware was used for this purpose. Some results obtained for the 3 test utterances for the 15-word test dictionary are presented in the table below (Table 7.3).

Table 7.3: Table showing the number of Phones and Senones active for 3 test-utterances

Test Utterance #	Average PAL Size	Maximum PAL Size	Average SAL Size	Maximum SAL Size
1	7.1	21	29.4	47
2	5.11	17	25.3	72
3	5.5	23	19.7	41

The average and worst case number of active Phones and Senones is shown in this table. It can be seen that on an average, only 7 Phones are active simultaneously with the maximum being 23 for the 3rd utterance. This shows that out of 450 possible phones for the test dictionary, only 23 are ever active simultaneously thereby indicating the benefit of the feedback in decreasing the number of overall computations. Further, since at-most 5% of the data is active, instead of having one-to-one mapping for every phone in the PHN_WRD_RAM, the on-chip resources can be optimized to provide for only the active data. This is where the dynamic memory management in the WRD Block comes in handy. The incorporation of this scheme allows for the implementation of a scalable system.

The savings in the number of computations is considerably more in AM. It can be seen from the table that at any time instant, only 72 Senones at most need to be computed for the 3 test utterances, with the average being 25 Senones. This is in comparison to a total of 1935 Senones in the AM_Database.

7.6 SUMMARY

From the results presented in this chapter, it can be concluded that a highly efficient, fully pipelined system has been implemented that would enable the processing of data for a 1,000-word Command & Control dictionary with a 1 million cycle budget in real-time. With only 93,000 equivalent gates required for the implementation of the system, this design can be incorporated as a co-processor to commercially available General Purpose Processors. Finally, with the incorporation of feedback into the architecture, it can be seen that the number of overall computations can be reduced significantly.

8.0 CONCLUSIONS

In conclusion, an architecture that enables the implementation of a continuous speech recognition system has been designed and simulated successfully. It was shown that by using a highly pipelined implementation, data for every 10ms speech frame could be processed in less than 1 million cycles thereby implying that the system is real-time at 100 MHz for a Command & Control dictionary. From an implementation perspective, the system was implemented efficiently with an F_{max} of 154 MHz (post place & route) on a Virtex4 SX-35 device.

It was shown that the design can be implemented using 93,000 equivalent gates indicating a small silicon footprint. With the results obtained, it was concluded that this design can be readily incorporated as a dedicated speech recognition co-processor into existing processors with minimal increase in overall cost of implementation.

It was shown that the inclusion of feedback can help in reducing the number of computations significantly. Further, since only a small fraction of the overall data is active simultaneously, the inclusion of a dynamic memory management scheme helps manage the active data space, thereby enabling the implementation of a scalable design.

It was shown that the Gaussian Distance computations in Acoustic Modeling can be implemented using 32-bit fixed-point computations, with only 10^{-3} % error on an average when compared to 32-bit floating-point computations. Finally, a new computation reduction technique, *bestN*, was proposed. It was shown that with 0.1% degradation in word recognition accuracy when selecting the top 3 components, the computations for obtaining the shortlist of Components can be reduced to 8-bit addition operations instead of the traditional 32-bit multiply/add operations. This implies a bandwidth reduction by a factor of 8 for a slight increase in storage memory.

8.1 MAJOR CONTRIBUTIONS

The major contributions of this thesis are as follows:

- Designed and implemented a scalable, fully pipelined custom hardware architecture for real-time, speaker-independent continuous speech recognition.
- Using dedicated FIFOs, incorporated feedback into the architecture in an efficient way from every stage of the design so as to enable the computation of only the active data, thereby minimizing the number of computations to be performed.
- Incorporated dynamic memory management into the architecture for maintaining all active data at the Word and Phone level, thereby allowing for the implementation of a scalable system.
- Converted single-precision floating-point computations into 32-bit custom fixed-point computations for Gaussian probability evaluation in Acoustic Modeling with an average of 10^{-3} % loss in accuracy.
- Explored a computation reduction technique in Acoustic Modeling from a fully hardware implementation perspective. This lead to proposing the "*bestN*" technique that allows for a bandwidth reduction by a factor of 8 and reduces the computation operations into 8-bit integer addition as opposed to several 32-bit multiply & add/subtract operations with 0.1% degradation in word recognition accuracy when compared to the baseline Sphinx 3 system on a 1,000 word Command & Control task.

8.2 CONCLUSIONS

The conclusions drawn from this research are as follows:

- Speech Recognition does not require multi-GHz processors for real-time performance. A system based on dedicated hardware running at 100 MHz is sufficient for this purpose.
- A fully pipelined approach greatly helps in increasing the throughput of the system. Such an approach is especially useful for applications like speech recognition, where because

of the regular nature of the computations and limited data-dependency within a block, a highly pipelined system can be implemented for attaining maximum throughput.

- Despite being a highly pipelined design, the system can be implemented using 93,000 equivalent gates which shows that the silicon footprint of such a design is reasonably small for it to be incorporated as a dedicated speech recognition co-processor.
- Incorporating feedback can significantly decrease the number of computations in the system without impact on the overall recognition accuracy.
- Floating-point operations for the computation of Gaussian probabilities in Acoustic Modeling need not be performed. Instead, 32-bit custom fixed-point computations can be used for little loss in accuracy.
- Using the bestN technique for computation reduction in Acoustic Modeling, 32-bit multiply/add operations can give way to 8-bit integer addition operations. For the same bus bandwidth, a speedup of 8x can be achieved.

8.3 FUTURE DIRECTIONS

Having shown that the architecture developed requires few gates for being implemented, the design can be readily moved into a ASIC flow. With availability of sufficient on-chip memory, the current implementation can be used for recognizing in excess of 1,000 words.

APPENDIX A

VHDL CODE FOR TOP-LEVEL ENTITIES OF THE ENTIRE SYSTEM

This Appendix contains the VHDL code of the implemented system. Top-level entities for the AM, PHN, WRD Blocks, and the System have been included here for reference.

A.1 SYSTEM TOP-LEVEL

```
--------------------------------------------------------------------------------
-- Organization:   University of Pittsburgh
-- Author:         Kshitij Gupta
--
-- Design Name:    SYSTEM TOP-lEVEL
-- Module Name:    SYS_TOP_v2 - struct
-- Project Name:   University of Pittsburgh's Speech Recognition System-on-a-Chip
-- Target Device:  Xilinx Virtex4 SX-35
-- Tool versions:  ISE 7.1i
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity SYS_TOP_v2 is
        port (
                clk : IN std_logic;
                ce : IN std_logic;
                HARD_RST : IN std_logic;
                UTT_GO : IN std_logic;
                FRAME_READY : IN std_logic;
                X : IN std_logic_vector(31 downto 0);
                X_valid : IN std_logic;
                MK : IN std_logic_vector(31 downto 0);
                MK_valid : IN std_logic;
                VW : IN std_logic_vector(31 downto 0);
                VW_valid : IN std_logic;
                ROM_BUSY_STALL : IN std_logic;
                X_ADDR : OUT std_logic_vector(5 downto 0);
                X_ADDR_valid : OUT std_logic;
                MV_ADDR : OUT std_logic_vector(20 downto 0);
                MV_ADDR_valid : OUT std_logic;
                EXIT_WRD_ID : OUT std_logic_vector(4 downto 0);
                EXIT_WRD_ID_VALID : OUT std_logic;
                EXIT_WRD_SCR : OUT std_logic_vector(31 downto 0);
                FRAME_DONE : OUT std_logic
        );
end SYS_TOP_v2;

architecture struct of SYS_TOP_v2 is

component SYS_CTRL_v2
   PORT(
      AM_DONE                 : IN     std_logic;
      DEAD_PHASE_DONE         : IN     std_logic;
      FRAME_READY             : IN     std_logic;
      NXT_PHN_PHASE_DONE      : IN     std_logic;
      PHN_DONE                : IN     std_logic;
      SAL_DONE                : IN     std_logic;
      TKN_INIT_DONE           : IN     std_logic;
      UTT_GO                  : IN     std_logic;
      ce                      : IN     std_logic;
      clk                     : IN     std_logic;
      rst                     : IN     std_logic;
      AM_EN                   : OUT    std_logic;
      AM_GO                   : OUT    std_logic;
      AM_PHN_PHASE            : OUT    std_logic;
      AM_RST                  : OUT    std_logic;
      DEAD_PHASE              : OUT    std_logic;
      DEAD_PHASE_GO           : OUT    std_logic;
      FRAME_DONE              : OUT    std_logic;
      INSERT_SIL_on_SOFT_RST  : OUT    std_logic;
      NXT_PHN_PHASE_GO        : OUT    std_logic;
      PHN_EN                  : OUT    std_logic;
      PHN_GO                  : OUT    std_logic;
```

```
      PHN_RST                : OUT    std_logic;
      PHN_WRD_PHASE          : OUT    std_logic;
      SAL_GO                 : OUT    std_logic;
      SAL_MODE               : OUT    std_logic;
      SOFT_RST               : OUT    std_logic;
      TKN_INIT_GO            : OUT    std_logic;
      WRD_EN                 : OUT    std_logic;
      WRD_RST                : OUT    std_logic
   );
end component;

component AM_CState_ROM_v2
        port(
                clk : IN std_logic;
                ce : IN std_logic;
                sclr : IN std_logic;
                CSID_in : IN std_logic_vector(10 downto 0);
                CSID_in_valid : IN std_logic;
                CSID_out : OUT std_logic_vector(11 downto 0);
                CSID_out_valid : OUT std_logic;
                CURR_CSEN_done : OUT std_logic;
                SID_out : OUT std_logic_vector(10 downto 0);
                SID_out_valid : OUT std_logic
        );
end component;

component sal_fifo
    port (
    clk: IN std_logic;
    sinit: IN std_logic;
    din: IN std_logic_VECTOR(10 downto 0);
    wr_en: IN std_logic;
    rd_en: IN std_logic;
    dout: OUT std_logic_VECTOR(10 downto 0);
    full: OUT std_logic;
    empty: OUT std_logic;
    rd_ack: OUT std_logic;
    data_count: OUT std_logic_VECTOR(10 downto 0));
end component;

component csal_fifo
    port (
    clk: IN std_logic;
    sinit: IN std_logic;
    din: IN std_logic_VECTOR(10 downto 0);
    wr_en: IN std_logic;
    rd_en: IN std_logic;
    dout: OUT std_logic_VECTOR(10 downto 0);
    full: OUT std_logic;
    empty: OUT std_logic;
    rd_ack: OUT std_logic;
    data_count: OUT std_logic_VECTOR(10 downto 0));
end component;

component AM_TOP_LEVEL_v2
        PORT (
                clk : IN std_logic;
                ce : IN std_logic;
                sclr : IN std_logic;
                AM_GO : IN std_logic;
                SVQ_BYPASS_GLOBAL : IN std_logic;
                CW_SAL_SID : IN std_logic_vector(10 downto 0);
                CW_SAL_SID_valid : IN std_logic;
                SEN_SAL_SID : IN std_logic_vector(10 downto 0);
                SEN_SAL_SID_valid : IN std_logic;
                CW_SAL_EMPTY : IN std_logic;
                SEN_SAL_EMPTY : IN std_logic;
                CSAL_EMPTY : IN std_logic;
                SEN_SAL_SID_CNT : IN std_logic_vector(10 downto 0);
                CSAL_CSID_CNT : IN std_logic_vector(10 downto 0);
                ROM_BUSY_STALL : IN std_logic;
                X : IN std_logic_vector(31 downto 0);
                X_valid : IN std_logic;
                MK : IN std_logic_vector(31 downto 0);
                VW : IN std_logic_vector(31 downto 0);
                MKVW_valid : IN std_logic;
                CSID : IN std_logic_vector(11 downto 0);
                CSID_valid : IN std_logic;
```

```
                C_SID : IN std_logic_vector(10 downto 0);
                C_SID_valid : IN std_logic;
                CURR_CSEN_DONE : IN std_logic;
                CW_SAL_POP : OUT std_logic;
                SEN_SAL_POP : OUT std_logic;
                CSAL_POP : OUT std_logic;
                FEAT_ADDR : OUT std_logic_vector(5 downto 0);
                FEAT_ADDR_VALID : OUT std_logic;
                AM_ROM_ADDR : OUT std_logic_vector(20 downto 0);
                AM_ROM_ADDR_VALID : OUT std_logic;
                AM_SID_CSID : OUT std_logic_vector(11 downto 0);
                AM_sscr_out : OUT std_logic_vector(31 downto 0);
                AM_sscr_out_valid : OUT std_logic;
                AM_DONE : OUT std_logic;
                ERR_AM_ACTIVE_SENONE_CNT_IS_ZERO : OUT std_logic;
                ERR_AM_N_ACTIVE_COMPONENTS_IS_ZERO : OUT std_logic;
                ERR_AM_FEAT_AM_ROM_ADDR_NOT_VALID_SIMULTANEOUSLY : OUT std_logic;
                ERR_AM_SENSAL_CAL_cnt_EMPTY_UNEQUAL : OUT std_logic
        );
end component;

component sal_top_level
        port (
                clk, ce, sclr : std_logic;
                SAL_GO : IN std_logic;
                nPAL_PAL_DONE : IN std_logic;
                sid_csid : IN std_logic_vector(41 downto 0);
                sid_csid_valid : IN std_logic;
                C_SID : IN std_logic_vector(10 downto 0);
                C_SID_valid : IN std_logic;
                CState_done : IN std_logic;
                SAL_RST_DONE : OUT std_logic;
                SAL : OUT std_logic_vector(10 downto 0);
                SAL_valid : OUT std_logic;
                CSAL : OUT std_logic_vector(10 downto 0);
                CSAL_valid : OUT std_logic;
                SAL_DONE : OUT std_logic
        );
end component;

component SSEQ_CSSEQ_ROM_WRAPPER
        port(
                clk : IN std_logic;
                ce : IN std_logic;
                sclr : IN std_logic;
                SSID : IN std_logic_vector(13 downto 0);
                SSID_valid : IN std_logic;
                addr_seq : OUT std_logic_vector(41 downto 0);
                addr_seq_valid : OUT std_logic
        );
end component;

COMPONENT am_phn_ram
        PORT(
                clk : IN std_logic;
                ce : IN std_logic;
                sclr : IN std_logic;
                phase_sel : IN std_logic;
                am_sscr_din : IN std_logic_vector(31 downto 0);
                am_sid : IN std_logic_vector(11 downto 0);
                am_sscr_din_valid : IN std_logic;
                phn_sid_t012 : IN std_logic_vector(41 downto 0);
                phn_sid_t012_valid : IN std_logic;
                phn_sscr_t012 : OUT std_logic_vector(179 downto 0);
                phn_sscr_t012_valid : OUT std_logic
        );
END COMPONENT;

component phn_top
        port (
                clk, ce, sclr : IN std_logic;
                rst : IN std_logic;
                SAL_MODE : IN std_logic;
                SAL_GO : IN std_logic;
                SAL_RST_DONE : IN std_logic;
                PHN_GO : IN std_logic;
                tmat_sscr_in : IN std_logic_vector(179 downto 0);
                tmat_sscr_in_valid : IN std_logic;
```

```
                 nPAL_TKN : IN std_logic_vector(10 downto 0);
                 nPAL_TKN_valid : IN std_logic;
                 nPAL_empty : IN std_logic;
                 ram_rd_dout : IN std_logic_vector(215 downto 0);
                 ram_rd_dout_valid : IN std_logic_vector(3 downto 0);
                 nPAL_din_s : OUT std_logic_vector(10 downto 0);
                 nPAL_PUSH_s : OUT std_logic;
                 nPAL_POP : OUT std_logic;
                 ram_wr_addr : OUT std_logic_vector(9 downto 0);
                 ram_wr_din : OUT std_logic_vector(215 downto 0);
                 ram_wr_en : OUT std_logic_vector(3 downto 0);
                 ram_rd_addr : OUT std_logic_vector(9 downto 0);
                 ram_rd_en : OUT std_logic_vector(3 downto 0);
                 nPAL_PAL_DONE : OUT std_logic;
                 dead_PUSH_s : OUT std_logic;
                 nxt_phn_PUSH_s : OUT std_logic;
                 TKN_prn : OUT std_logic_vector(10 downto 0);
                 WRD_TH : OUT std_logic_vector(31 downto 0);
                 th_valid : OUT std_logic;
                 PHN_DONE : OUT std_logic
        );
end component;

COMPONENT phn_wrd_ram
        PORT(
                 clk : IN std_logic;
                 ce : IN std_logic;
                 sclr : IN std_logic;
                 wr_addr : IN std_logic_vector(9 downto 0);
                 wr_din : IN std_logic_vector(215 downto 0);
                 wr_en : IN std_logic_vector(3 downto 0);
                 rd_addr : IN std_logic_vector(9 downto 0);
                 rd_en : IN std_logic_vector(3 downto 0);
                 rd_dout : OUT std_logic_vector(215 downto 0);
                 rd_dout_valid : OUT std_logic_vector(3 downto 0)
        );
END COMPONENT;

component phn_wrd_fifo
    port (
    clk: IN std_logic;
    sinit: IN std_logic;
    din: IN std_logic_VECTOR(10 downto 0);
    wr_en: IN std_logic;
    rd_en: IN std_logic;
    dout: OUT std_logic_VECTOR(10 downto 0);
    full: OUT std_logic;
    empty: OUT std_logic;
    rd_ack: OUT std_logic;
    rd_err: OUT std_logic;
    wr_err: OUT std_logic;
    data_count: OUT std_logic_VECTOR(9 downto 0));
end component;

component wrd_top
        PORT(
                 clk : IN std_logic;
                 ce : IN std_logic;
                 sclr : IN std_logic;
                 DEAD_PHASE : IN std_logic;
                 TKN_INIT_GO : IN std_logic;
                 WRD_TH : IN std_logic_vector(31 downto 0);
                 RAM_rd_dout : IN std_logic_vector(215 downto 0);
                 RAM_rd_dout_valid : IN std_logic_vector(3 downto 0);
                 DEAD_GO : IN std_logic;
                 DEAD_empty : IN std_logic;
                 DEAD_TKN : IN std_logic_vector(10 downto 0);
                 DEAD_TKN_valid : IN std_logic;
                 NXT_PHN_GO : IN std_logic;
                 NXT_PHN_empty : IN std_logic;
                 NXT_PHN_TKN : IN std_logic_vector(10 downto 0);
                 NXT_PHN_TKN_valid : IN std_logic;
                 TKN_INIT_DONE : OUT std_logic;
                 RAM_rd_addr : OUT std_logic_vector(9 downto 0);
                 RAM_rd_addr_valid : OUT std_logic_vector(3 downto 0);
                 RAM_wr_addr : OUT std_logic_vector(9 downto 0);
                 RAM_wr_din : OUT std_logic_vector(215 downto 0);
                 RAM_wr_addr_valid : OUT std_logic_vector(3 downto 0);
```

```
                 nPAL_TKN : OUT std_logic_vector(10 downto 0);
                 nPAL_TKN_valid : OUT std_logic;
                 DEAD_pop : OUT std_logic;
                 DEAD_phase_done : OUT std_logic;
                 NXT_PHN_pop : OUT std_logic;
                 NXT_PHN_phase_done : OUT std_logic;
                 WRD_EXIT_SCR : OUT std_logic_vector(31 downto 0);
                 WRD_EXIT_ID : OUT std_logic_vector(4 downto 0);
                 WRD_EXIT_ID_valid : OUT std_logic;
                 WRD_ERR_TKN_FIFO_EMPTY : OUT std_logic;
                 WRD_ERR_TKN_FIFO_WR_ERROR : OUT std_logic;
                 WRD_ERR_LAST_PHN_ACCESSED : OUT std_logic;
                 WRD_DATA_MINE_TKN_FIFO_CNT : OUT std_logic_vector(9 downto 0)
        );
end component;

signal sclr : std_logic;
signal SOFT_RST : std_logic;
signal INS_SIL_on_SOFT_RST : std_logic;
signal TKN_INIT_GO : std_logic;
signal TKN_INIT_DONE : std_logic;
signal AM_EN : std_logic;
signal AM_GO : std_logic;
signal AM_RST : std_logic;
signal AM_DONE : std_logic;
signal PHN_EN : std_logic;
signal PHN_GO : std_logic;
signal PHN_RST : std_logic;
signal PHN_DONE : std_logic;
signal WRD_EN : std_logic;
signal WRD_RST : std_logic;
signal DEAD_GO : std_logic;
signal NXT_PHN_GO : std_logic;
signal DEAD_PHASE_DONE : std_logic;
signal NXT_PHN_PHASE_DONE : std_logic;
signal DEAD_PHASE : std_logic;
signal AM_PHN_PHASE : std_logic;
signal PHN_WRD_PHASE : std_logic;
signal SAL_MODE : std_logic;
signal CSID_in : std_logic_vector(10 downto 0);
signal CSID_in_valid : std_logic;
signal CSID_out : std_logic_vector(11 downto 0);
signal CSID_out_valid : std_logic;
signal CURR_CSEN_DONE : std_logic;
signal C_SID : std_logic_vector(10 downto 0);
signal C_SID_valid : std_logic;
signal am_CSID : std_logic_vector(11 downto 0);
signal am_CSID_valid : std_logic;
signal am_CURR_CSEN_DONE : std_logic;
signal am_C_SID : std_logic_vector(10 downto 0);
signal am_C_SID_valid : std_logic;
signal phn_C_SID : std_logic_vector(10 downto 0);
signal phn_C_SID_valid : std_logic;
signal phn_CURR_CSEN_DONE : std_logic;
signal phn_SAL_sid : std_logic_vector(10 downto 0);
signal phn_SAL_PUSH : std_logic;
signal SAL_CW_FULL : std_logic;
signal SAL_CW_count : std_logic_vector(10 downto 0);
signal am_SAL_CW_POP : std_logic;
signal SAL_CW_EMPTY : std_logic;
signal am_SAL_CW_sid : std_logic_vector(10 downto 0);
signal am_SAL_CW_valid : std_logic;
signal SAL_SEN_FULL : std_logic;
signal SAL_SEN_count : std_logic_vector(10 downto 0);
signal am_SAL_SEN_POP : std_logic;
signal SAL_SEN_EMPTY : std_logic;
signal am_SAL_SEN_sid : std_logic_vector(10 downto 0);
signal am_SAL_SEN_valid : std_logic;
signal CSAL_FULL : std_logic;
signal CSAL_count : std_logic_vector(10 downto 0);
signal phn_CSAL_csid : std_logic_vector(10 downto 0);
signal phn_CSAL_PUSH : std_logic;
signal am_CSAL_POP : std_logic;
signal CSAL_EMPTY : std_logic;
signal am_CSAL_csid : std_logic_vector(10 downto 0);
signal am_CSAL_valid : std_logic;
signal SVQ_BYPASS_GLOBAL : std_logic;
signal MKVW_valid : std_logic;
```

```
signal ERR_AM : std_logic_vector(3 downto 0);
signal phn_SSID : std_logic_vector(13 downto 0);
signal phn_SSID_valid : std_logic;
signal SSEQ_ADDR : std_logic_vector(41 downto 0);
signal SSEQ_ADDR_valid : std_logic;
signal sal_SSEQ_ADDR_valid : std_logic;
signal sscr_SSEQ_ADDR_valid : std_logic;
signal SAL_GO : std_logic;
signal SAL_RST_DONE : std_logic;
signal SAL_DONE : std_logic;
signal am_SID_CSID : std_logic_vector(11 downto 0);
signal am_SSCR_dout : std_logic_vector(31 downto 0);
signal am_SSCR_dout_valid : std_logic;
signal phn_sscr_t012 : std_logic_vector(179 downto 0);
signal phn_sscr_t012_valid : std_logic;
signal nPAL_PAL_DONE : std_logic;
signal TKN_prn : std_logic_vector(10 downto 0);
signal RAM_wr_addr : std_logic_vector(9 downto 0);
signal RAM_wr_din : std_logic_vector(215 downto 0);
signal RAM_wr_en : std_logic_vector(3 downto 0);
signal RAM_rd_addr : std_logic_vector(9 downto 0);
signal RAM_rd_en : std_logic_vector(3 downto 0);
signal RAM_rd_dout : std_logic_vector(215 downto 0);
signal RAM_rd_dout_valid : std_logic_vector(3 downto 0);
signal phn_RAM_wr_addr : std_logic_vector(9 downto 0);
signal phn_RAM_wr_din : std_logic_vector(215 downto 0);
signal phn_RAM_wr_en : std_logic_vector(3 downto 0);
signal phn_RAM_rd_addr : std_logic_vector(9 downto 0);
signal phn_RAM_rd_en : std_logic_vector(3 downto 0);
signal phn_RAM_rd_dout : std_logic_vector(215 downto 0);
signal phn_RAM_rd_dout_valid : std_logic_vector(3 downto 0);
signal wrd_RAM_wr_addr : std_logic_vector(9 downto 0);
signal wrd_RAM_wr_din : std_logic_vector(215 downto 0);
signal wrd_RAM_wr_en : std_logic_vector(3 downto 0);
signal wrd_RAM_rd_addr : std_logic_vector(9 downto 0);
signal wrd_RAM_rd_en : std_logic_vector(3 downto 0);
signal wrd_RAM_rd_dout : std_logic_vector(215 downto 0);
signal wrd_RAM_rd_dout_valid : std_logic_vector(3 downto 0);
signal phn_WRD_TH, wrd_WRD_TH : std_logic_vector(31 downto 0);
signal phn_WRD_TH_valid, wrd_WRD_TH_valid : std_logic;
signal ERR_WRD : std_logic_vector(2 downto 0);
signal WRD_TKN_CNT : std_logic_vector(9 downto 0);
signal phn_nPAL_din : std_logic_vector(10 downto 0);
signal phn_nPAL_PUSH : std_logic;
signal wrd_nPAL_din : std_logic_vector(10 downto 0);
signal wrd_nPAL_PUSH : std_logic;
signal wrd_nPAL_din_s : std_logic_vector(10 downto 0);
signal wrd_nPAL_PUSH_s : std_logic;
signal nPAL_din : std_logic_vector(10 downto 0);
signal nPAL_PUSH : std_logic;
signal nPAL_wr_err : std_logic;
signal nPAL_FULL : std_logic;
signal nPAL_dout : std_logic_vector(10 downto 0);
signal nPAL_dout_valid : std_logic;
signal nPAL_POP : std_logic;
signal nPAL_rd_err : std_logic;
signal nPAL_EMPTY : std_logic;
signal nPAL_CNT : std_logic_vector(9 downto 0);
signal NXT_PHN_din : std_logic_vector(10 downto 0);
signal NXT_PHN_PUSH : std_logic;
signal NXT_PHN_wr_err : std_logic;
signal NXT_PHN_FULL : std_logic;
signal NXT_PHN_dout : std_logic_vector(10 downto 0);
signal NXT_PHN_dout_valid : std_logic;
signal NXT_PHN_POP : std_logic;
signal NXT_PHN_rd_err : std_logic;
signal NXT_PHN_EMPTY : std_logic;
signal NXT_PHN_CNT : std_logic_vector(9 downto 0);
signal DEAD_din : std_logic_vector(10 downto 0);
signal DEAD_PUSH_s : std_logic;
signal DEAD_PUSH : std_logic;
signal DEAD_wr_err : std_logic;
signal DEAD_FULL : std_logic;
signal DEAD_dout : std_logic_vector(10 downto 0);
signal DEAD_dout_valid : std_logic;
signal DEAD_POP : std_logic;
signal DEAD_rd_err : std_logic;
signal DEAD_EMPTY : std_logic;
```

```
signal DEAD_CNT : std_logic_vector(9 downto 0);
signal SAL_SEN_CNT_reg : std_logic_vector(10 downto 0);
signal SAL_CW_CNT_reg : std_logic_vector(10 downto 0);
signal CSAL_CSEN_CNT_reg : std_logic_vector(10 downto 0);
signal nPAL_CNT_reg : std_logic_vector(9 downto 0);
signal WRD_TKN_CNT_reg : std_logic_vector(9 downto 0);

constant NEG_INF : std_logic_vector(31 downto 0) := X"C8000000";

begin

sclr <= SOFT_RST or HARD_RST;
SVQ_BYPASS_GLOBAL <= '1';
MKVW_valid <= MK_valid or VW_valid;

SYS_CTRL_BLK : SYS_CTRL_v2
   PORT MAP(
      AM_DONE            => AM_DONE,
      DEAD_PHASE_DONE    => DEAD_PHASE_DONE,
      FRAME_READY        => FRAME_READY,
      NXT_PHN_PHASE_DONE => NXT_PHN_PHASE_DONE,
      PHN_DONE           => PHN_DONE,
      SAL_DONE           => SAL_DONE,
      TKN_INIT_DONE      => TKN_INIT_DONE,
      UTT_GO             => UTT_GO,
      ce                 => ce,
      clk                => clk,
      rst                => HARD_RST,
      AM_EN              => AM_EN,
      AM_GO              => AM_GO,
      AM_PHN_PHASE       => AM_PHN_PHASE,
      AM_RST             => AM_RST,
      DEAD_PHASE         => DEAD_PHASE,
      DEAD_PHASE_GO      => DEAD_GO,
      FRAME_DONE         => FRAME_DONE,
      INSERT_SIL_on_SOFT_RST => INS_SIL_on_SOFT_RST,
      NXT_PHN_PHASE_GO   => NXT_PHN_GO,
      PHN_EN             => PHN_EN,
      PHN_GO             => PHN_GO,
      PHN_RST            => PHN_RST,
      PHN_WRD_PHASE      => PHN_WRD_PHASE,
      SAL_GO             => SAL_GO,
      SAL_MODE           => SAL_MODE,
      SOFT_RST           => SOFT_RST,
      TKN_INIT_GO        => TKN_INIT_GO,
      WRD_EN             => WRD_EN,
      WRD_RST            => WRD_RST
   );

SYS_AM_CState_ROM_BLK : AM_CState_ROM_v2
        port map(
                clk => clk,
                ce => ce,
                sclr => sclr,
                CSID_in => CSID_in,
                CSID_in_valid => CSID_in_valid,
                CSID_out => CSID_out,
                CSID_out_valid => CSID_out_valid,
                CURR_CSEN_done => CURR_CSEN_DONE,
                SID_out => C_SID,
                SID_out_valid => C_SID_valid
        );

process(SAL_MODE, C_SID, C_SID_valid, CURR_CSEN_DONE, phn_CSAL_csid, phn_CSAL_PUSH, am_CSAL_csid,
am_CSAL_valid, CSID_out, CSID_out_valid)
begin
        if (SAL_MODE = '1') then
                CSID_in <= phn_CSAL_csid;
                CSID_in_valid <= phn_CSAL_PUSH;
                am_CSID <= (OTHERS => '0');
                am_CSID_valid <= '0';
                am_C_SID_valid <= '0';
                am_curr_CSEN_done <= '0';
                phn_C_SID <= C_SID;
                phn_C_SID_valid <= C_SID_valid;
                phn_curr_CSEN_done <= CURR_CSEN_DONE;
        else
                CSID_in <= am_CSAL_csid;
                CSID_in_valid <= am_CSAL_valid;
```

```
                am_CSID <= CSID_out;
                am_CSID_valid <= CSID_out_valid;
                am_C_SID <= C_SID;
                am_C_SID_valid <= C_SID_valid;
                am_curr_CSEN_done <= CURR_CSEN_DONE;
                phn_C_SID_valid <= '0';
                phn_curr_CSEN_done <= '0';
        end if;
end process;

SYS_FIFO_SAL_CW : sal_fifo
          port map (
              clk => clk,
              sinit => sclr,
              din => phn_SAL_sid,
              wr_en => phn_SAL_PUSH,
              rd_en => am_SAL_CW_POP,
              dout => am_SAL_CW_sid,
              full => SAL_CW_FULL,
              empty => SAL_CW_EMPTY,
              rd_ack => am_SAL_CW_valid,
              data_count => SAL_CW_count
                 );

SYS_FIFO_SAL_SEN : sal_fifo
          port map (
              clk => clk,
              sinit => sclr,
              din => phn_SAL_sid,
              wr_en => phn_SAL_PUSH,
              rd_en => am_SAL_SEN_POP,
              dout => am_SAL_SEN_sid,
              full => SAL_SEN_FULL,
              empty => SAL_SEN_EMPTY,
              rd_ack => am_SAL_SEN_valid,
              data_count => SAL_SEN_count
                 );

SYS_FIFO_CSAL : csal_fifo
          port map (
              clk => clk,
              sinit => sclr,
              din => phn_CSAL_csid,
              wr_en => phn_CSAL_PUSH,
              rd_en => am_CSAL_POP,
              dout => am_CSAL_csid,
              full => CSAL_FULL,
              empty => CSAL_EMPTY,
              rd_ack => am_CSAL_valid,
              data_count => CSAL_count
                 );

SYS_AM_TOP_BLK : AM_TOP_LEVEL_v2
        PORT MAP(
                clk => clk,
                ce => AM_EN,
                sclr => AM_RST,
                AM_GO => AM_GO,
                SVQ_BYPASS_GLOBAL => SVQ_BYPASS_GLOBAL,
                CW_SAL_SID => am_SAL_CW_sid,
                CW_SAL_SID_valid => am_SAL_CW_valid,
                SEN_SAL_SID => am_SAL_SEN_sid,
                SEN_SAL_SID_valid => am_SAL_SEN_valid,
                CW_SAL_EMPTY => SAL_CW_EMPTY,
                SEN_SAL_EMPTY => SAL_SEN_EMPTY,
                CSAL_EMPTY => CSAL_EMPTY,
                SEN_SAL_SID_CNT => SAL_SEN_count,
                CSAL_CSID_CNT => CSAL_count,
                ROM_BUSY_STALL => ROM_BUSY_STALL,
                X => X,
                X_valid => X_valid,
                MK => MK,
                VW => VW,
                MKVW_valid => MKVW_valid,
                CSID => am_CSID,
                CSID_valid => am_CSID_valid,
                C_SID => am_C_SID,
                C_SID_valid => am_C_SID_valid,
```

```
                CURR_CSEN_DONE => am_curr_CSEN_done,
                CW_SAL_POP => am_SAL_CW_POP,
                SEN_SAL_POP => am_SAL_SEN_POP,
                CSAL_POP => am_CSAL_POP,
                FEAT_ADDR => X_ADDR,
                FEAT_ADDR_VALID => X_ADDR_valid,
                AM_ROM_ADDR => MV_ADDR,
                AM_ROM_ADDR_VALID => MV_ADDR_valid,
                AM_SID_CSID => am_SID_CSID,
                AM_sscr_out => am_SSCR_dout,
                AM_sscr_out_valid => am_SSCR_dout_valid,
                AM_DONE => AM_DONE,
                ERR_AM_ACTIVE_SENONE_CNT_IS_ZERO => ERR_AM(0),
                ERR_AM_N_ACTIVE_COMPONENTS_IS_ZERO => ERR_AM(1),
                ERR_AM_FEAT_AM_ROM_ADDR_NOT_VALID_SIMULTANEOUSLY => ERR_AM(2),
                ERR_AM_SENSAL_CAL_cnt_EMPTY_UNEQUAL => ERR_AM(3)
        );

SYS_SAL_GENERATOR : sal_top_level
        port map (
                clk => clk,
                ce => ce,
                sclr => sclr,
                SAL_GO => SAL_GO,
                nPAL_PAL_DONE => nPAL_PAL_DONE,
                sid_csid => SSEQ_ADDR,
                sid_csid_valid => sal_SSEQ_ADDR_valid,
                C_SID => phn_C_SID,
                C_SID_valid => phn_C_SID_valid,
                CState_done => phn_curr_CSEN_done,
                SAL_RST_DONE => SAL_RST_DONE,
                SAL => phn_SAL_sid,
                SAL_valid => phn_SAL_PUSH,
                CSAL => phn_CSAL_csid,
                CSAL_valid => phn_CSAL_PUSH,
                SAL_DONE => SAL_DONE
        );

phn_SSID <= phn_RAM_rd_dout(215 downto 202);
phn_SSID_valid <= phn_RAM_rd_dout_valid(3);

SYS_SSEQ_CSSEQ_ROM_BLK : SSEQ_CSSEQ_ROM_WRAPPER
        port map(
                clk => clk,
                ce => ce,
                sclr => sclr,
                SSID => phn_SSID,
                SSID_valid => phn_SSID_valid,
                addr_seq => SSEQ_ADDR,
                addr_seq_valid => SSEQ_ADDR_valid
        );

sal_SSEQ_ADDR_valid <= SSEQ_ADDR_valid and SAL_MODE;
sscr_SSEQ_ADDR_valid <= SSEQ_ADDR_valid and NOT(SAL_MODE);

SYS_AM_PHN_RAM_BLK : am_phn_ram
        PORT MAP(
                clk => clk,
                ce => ce,
                sclr => sclr,
                phase_sel => AM_PHN_PHASE,
                am_sscr_din => am_SSCR_dout,
                am_sid => am_SID_CSID,
                am_sscr_din_valid => am_SSCR_dout_valid,
                phn_sid_t012 => SSEQ_ADDR,
                phn_sid_t012_valid => sscr_SSEQ_ADDR_valid,
                phn_sscr_t012 => phn_sscr_t012,
                phn_sscr_t012_valid => phn_sscr_t012_valid
        );

SYS_PHN_TOP_BLK : phn_top
        port map(
                clk => clk,
                ce => PHN_EN,
                sclr => PHN_RST,
                rst => sclr,
                SAL_MODE => SAL_MODE,
                SAL_GO => SAL_GO,
```

```
                SAL_RST_DONE => SAL_RST_DONE,
                PHN_GO => PHN_GO,
                tmat_sscr_in => phn_sscr_t012,
                tmat_sscr_in_valid => phn_sscr_t012_valid,
                nPAL_TKN => nPAL_dout,
                nPAL_TKN_valid => nPAL_dout_valid,
                nPAL_empty => nPAL_EMPTY,
                ram_rd_dout => RAM_rd_dout,
                ram_rd_dout_valid => RAM_rd_dout_valid,
                nPAL_din_s => phn_nPAL_din,
                nPAL_PUSH_s => phn_nPAL_PUSH,
                nPAL_POP => nPAL_POP,
                ram_wr_addr => phn_RAM_wr_addr,
                ram_wr_din => phn_RAM_wr_din,
                ram_wr_en => phn_RAM_wr_en,
                ram_rd_addr => phn_RAM_rd_addr,
                ram_rd_en => phn_RAM_rd_en,
                nPAL_PAL_DONE => nPAL_PAL_DONE,
                dead_PUSH_s => DEAD_PUSH_s,
                nxt_phn_PUSH_s => NXT_PHN_PUSH,
                TKN_prn => TKN_prn,
                WRD_TH => phn_WRD_TH,
                th_valid => phn_WRD_TH_valid,
                PHN_DONE => PHN_DONE
        );

SYS_PHN_WRD_RAM_BLK : phn_wrd_ram
        PORT MAP(
                clk => clk,
                ce => ce,
                sclr => sclr,
                wr_addr => RAM_wr_addr,
                wr_din => RAM_wr_din,
                wr_en => RAM_wr_en,
                rd_addr => RAM_rd_addr,
                rd_en => RAM_rd_en,
                rd_dout => RAM_rd_dout,
                rd_dout_valid => RAM_rd_dout_valid
        );

SYS_PHN_WRD_RAM_WRAPPER_BLK : process(PHN_WRD_PHASE, phn_RAM_wr_addr, phn_RAM_wr_din, phn_RAM_wr_en,
phn_RAM_rd_addr, phn_RAM_rd_en, wrd_RAM_wr_addr, wrd_RAM_wr_din, wrd_RAM_wr_en, wrd_RAM_rd_addr,
wrd_RAM_rd_en, RAM_rd_dout, RAM_rd_dout_valid)
begin
        case PHN_WRD_PHASE is
                when '1' =>
                        RAM_wr_addr <= phn_RAM_wr_addr;
                        RAM_wr_din <= phn_RAM_wr_din;
                        RAM_wr_en <= phn_RAM_wr_en;
                        RAM_rd_addr <= phn_RAM_rd_addr;
                        RAM_rd_en <= phn_RAM_rd_en;
                        phn_RAM_rd_dout <= RAM_rd_dout;
                        phn_RAM_rd_dout_valid <= RAM_rd_dout_valid;
                when '0' =>
                        RAM_wr_addr <= wrd_RAM_wr_addr;
                        RAM_wr_din <= wrd_RAM_wr_din;
                        RAM_wr_en <= wrd_RAM_wr_en;
                        RAM_rd_addr <= wrd_RAM_rd_addr;
                        RAM_rd_en <= wrd_RAM_rd_en;
                        wrd_RAM_rd_dout <= RAM_rd_dout;
                        wrd_RAM_rd_dout_valid <= RAM_rd_dout_valid;
                when OTHERS =>
                        RAM_wr_addr <= (OTHERS => '0');
                        RAM_wr_en <= "0000";
                        RAM_rd_addr <= (OTHERS => '0');
                        RAM_rd_en <= "0000";
        end case;
end process;

process(clk, ce, sclr, PHN_GO, phn_WRD_TH, phn_WRD_TH_valid)
begin
        if (clk'event and clk = '1') then
                if (sclr = '1') then
                        wrd_WRD_TH <= (OTHERS => '0');
                        wrd_WRD_TH_valid <= '1';
                elsif (PHN_GO = '1') then
                        wrd_WRD_TH <= NEG_INF;
                        wrd_WRD_TH_valid <= '0';
```

```
                elsif (ce = '1') then
                        if (phn_WRD_TH_valid = '1') then
                                wrd_WRD_TH <= phn_WRD_TH;
                                wrd_WRD_TH_valid <= phn_WRD_TH_valid;
                        end if;
                end if;
        end if;
end process;

process(PHN_WRD_PHASE, phn_nPAL_PUSH, phn_nPAL_din, wrd_nPAL_PUSH, wrd_nPAL_din)
begin
        case PHN_WRD_PHASE is
                when '1' =>
                        nPAL_PUSH <= phn_nPAL_PUSH;
                        nPAL_din <= phn_nPAL_din;
                when '0' =>
                        nPAL_PUSH <= wrd_nPAL_PUSH;
                        nPAL_din <= wrd_nPAL_din;
                when OTHERS =>
                        npal_PUSH <= '0';
        end case;
end process;

process(NXT_PHN_PHASE_DONE, wrd_nPAL_din_s, wrd_nPAL_PUSH_s)
begin
        if (NXT_PHN_PHASE_DONE = '1') then
                wrd_nPAL_din <= "10000000000";
                wrd_nPAL_PUSH <= '1';
        else
                wrd_nPAL_din <= wrd_nPAL_din_s;
                wrd_nPAL_PUSH <= wrd_nPAL_PUSH_s;
        end if;
end process;

process(INS_SIL_on_SOFT_RST, TKN_prn, DEAD_PUSH_s)
begin
        if (INS_SIL_on_SOFT_RST = '1') then
                DEAD_din <= '0' & "1111111111";
                DEAD_PUSH <= '1';
        else
                DEAD_din <= TKN_prn;
                DEAD_PUSH <= DEAD_PUSH_s;
        end if;
end process;

NXT_PHN_din <= TKN_prn;

SYS_nPAL_FIFO : phn_wrd_fifo
  port map (clk, sclr, nPAL_din, nPAL_PUSH, nPAL_POP, nPAL_dout, nPAL_FULL, nPAL_EMPTY, nPAL_dout_valid,
nPAL_rd_err, nPAL_wr_err, nPAL_CNT);

SYS_DEAD_FIFO : phn_wrd_fifo
  port map (clk, sclr, DEAD_din, DEAD_PUSH, DEAD_POP, DEAD_dout, DEAD_FULL, DEAD_EMPTY, DEAD_dout_valid,
DEAD_rd_err, DEAD_wr_err, DEAD_CNT);

SYS_NXT_PHN_FIFO : phn_wrd_fifo
  port map (clk, sclr, NXT_PHN_din, NXT_PHN_PUSH, NXT_PHN_POP, NXT_PHN_dout, NXT_PHN_FULL, NXT_PHN_EMPTY,
NXT_PHN_dout_valid, NXT_PHN_rd_err, NXT_PHN_wr_err, NXT_PHN_CNT);

SYS_WRD_TOP_BLK : wrd_top
        PORT MAP(
                clk => clk,
                ce => WRD_EN,
                sclr => WRD_RST,
                DEAD_PHASE => DEAD_PHASE,
                TKN_INIT_GO => TKN_INIT_GO,
                WRD_TH => wrd_WRD_TH,
                RAM_rd_dout => wrd_RAM_rd_dout,
                RAM_rd_dout_valid => wrd_RAM_rd_dout_valid,
                DEAD_GO => DEAD_GO,
                DEAD_empty => DEAD_EMPTY,
                DEAD_TKN => DEAD_dout,
                DEAD_TKN_valid => DEAD_dout_valid,
                NXT_PHN_GO => NXT_PHN_GO,
                NXT_PHN_empty => NXT_PHN_EMPTY,
                NXT_PHN_TKN => NXT_PHN_dout,
                NXT_PHN_TKN_valid => NXT_PHN_dout_valid,
                TKN_INIT_DONE => TKN_INIT_DONE,
```

```
                RAM_rd_addr => wrd_RAM_rd_addr,
                RAM_rd_addr_valid => wrd_RAM_rd_en,
                RAM_wr_addr => wrd_RAM_wr_addr,
                RAM_wr_din => wrd_RAM_wr_din,
                RAM_wr_addr_valid => wrd_RAM_wr_en,
                nPAL_TKN => wrd_nPAL_din_s,
                nPAL_TKN_valid => wrd_nPAL_PUSH_s,
                DEAD_pop => DEAD_POP,
                DEAD_phase_done => DEAD_PHASE_DONE,
                NXT_PHN_pop => NXT_PHN_POP,
                NXT_PHN_phase_done => NXT_PHN_PHASE_DONE,
                WRD_EXIT_SCR => EXIT_WRD_SCR,
                WRD_EXIT_ID => EXIT_WRD_ID,
                WRD_EXIT_ID_valid => EXIT_WRD_ID_VALID,
                WRD_ERR_TKN_FIFO_EMPTY => ERR_WRD(0),
                WRD_ERR_TKN_FIFO_WR_ERROR => ERR_WRD(1),
                WRD_ERR_LAST_PHN_ACCESSED => ERR_WRD(2),
                WRD_DATA_MINE_TKN_FIFO_CNT => WRD_TKN_CNT
        );

process(clk, ce, sclr, SAL_SEN_count, SAL_CW_count, CSAL_count, nPAL_CNT, WRD_TKN_CNT)
begin
        if (clk'event and clk = '1') then
                if (sclr = '1') then
                        SAL_SEN_CNT_reg <= (OTHERS => '0');
                        SAL_CW_CNT_reg <= (OTHERS => '0');
                        CSAL_CSEN_CNT_reg <= (OTHERS => '0');
                        nPAL_CNT_reg <= (OTHERS => '0');
                        WRD_TKN_CNT_reg <= (OTHERS => '0');
                elsif (ce = '1') then
                        SAL_SEN_CNT_reg <= SAL_SEN_count;
                        SAL_CW_CNT_reg <= SAL_CW_count;
                        CSAL_CSEN_CNT_reg <= CSAL_count;
                        nPAL_CNT_reg <= nPAL_CNT;
                        WRD_TKN_CNT_reg <= WRD_TKN_CNT;
                end if;
        end if;
end process;

end struct;
```

A.2 SYSTEM CONTROLLER

```vhdl
--------------------------------------------------------------------------------
-- Organization:   University of Pittsburgh
-- Author:         Kshitij Gupta
--
-- Design Name:    SYSTEM CONTROLLER
-- Module Name:    SYS_CTRL_v2 - fsm
-- Project Name:   University of Pittsburgh's Speech Recognition System-on-a-Chip
-- Target Device:  Xilinx Virtex4 SX-35
-- Tool versions:  ISE 7.1i
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity SYS_CTRL_v2 is
   PORT(
      AM_DONE                 : IN     std_logic;
      DEAD_PHASE_DONE         : IN     std_logic;
      FRAME_READY             : IN     std_logic;
      NXT_PHN_PHASE_DONE      : IN     std_logic;
      PHN_DONE                : IN     std_logic;
      SAL_DONE                : IN     std_logic;
      TKN_INIT_DONE           : IN     std_logic;
      UTT_GO                  : IN     std_logic;
      ce                      : IN     std_logic;
      clk                     : IN     std_logic;
      rst                     : IN     std_logic;
      AM_EN                   : OUT    std_logic;
      AM_GO                   : OUT    std_logic;
      AM_PHN_PHASE            : OUT    std_logic;
      AM_RST                  : OUT    std_logic;
      DEAD_PHASE              : OUT    std_logic;
      DEAD_PHASE_GO           : OUT    std_logic;
      FRAME_DONE              : OUT    std_logic;
      INSERT_SIL_on_SOFT_RST  : OUT    std_logic;
      NXT_PHN_PHASE_GO        : OUT    std_logic;
      PHN_EN                  : OUT    std_logic;
      PHN_GO                  : OUT    std_logic;
      PHN_RST                 : OUT    std_logic;
      PHN_WRD_PHASE           : OUT    std_logic;
      SAL_GO                  : OUT    std_logic;
      SAL_MODE                : OUT    std_logic;
      SOFT_RST                : OUT    std_logic;
      TKN_INIT_GO             : OUT    std_logic;
      WRD_EN                  : OUT    std_logic;
      WRD_RST                 : OUT    std_logic
   );
end SYS_CTRL_v2;

architecture fsm of SYS_CTRL_v2 is

   TYPE STATE_TYPE IS (
      s0,
      s1,
      s2,
      s3,
      s4,
      s5,
      s6,
      s7,
      s8,
      s9,
      s10,
```

```
      s11,
      s12,
      s13,
      s14,
      s15,
      s16,
      s17,
      s18,
      s19,
      s20
   );

   -- State vector declaration
   ATTRIBUTE state_vector : string;
   ATTRIBUTE state_vector OF fsm : ARCHITECTURE IS "current_state";

   -- Declare current and next state signals
   SIGNAL current_state : STATE_TYPE;
   SIGNAL next_state : STATE_TYPE;

BEGIN

   -----------------------------------------------------------------
   clocked_proc : PROCESS (
      clk
   )
   -----------------------------------------------------------------
   BEGIN
      IF (clk'EVENT AND clk = '1') THEN
         IF (rst = '1') THEN
            current_state <= s0;
         ELSIF (ce = '1') THEN
            current_state <= next_state;
         END IF;
      END IF;
   END PROCESS clocked_proc;

   -----------------------------------------------------------------
   nextstate_proc : PROCESS (
      AM_DONE,
      DEAD_PHASE_DONE,
      FRAME_READY,
      NXT_PHN_PHASE_DONE,
      PHN_DONE,
      SAL_DONE,
      TKN_INIT_DONE,
      UTT_GO,
      current_state
   )
   -----------------------------------------------------------------
   BEGIN
      -- Default Assignment
      AM_GO <= '0';
      AM_RST <= '0';
      DEAD_PHASE_GO <= '0';
      FRAME_DONE <= '0';
      INSERT_SIL_on_SOFT_RST <= '0';
      NXT_PHN_PHASE_GO <= '0';
      PHN_GO <= '0';
      PHN_RST <= '0';
      SAL_GO <= '0';
      SOFT_RST <= '0';
      TKN_INIT_GO <= '0';
      WRD_RST <= '0';

      -- Combined Actions
      CASE current_state IS
         WHEN s0 =>
            IF (UTT_GO = '1') THEN
               SOFT_RST <= '1';
               AM_RST <= '1';
               PHN_RST <= '1';
               WRD_RST <= '1';
               AM_EN <= '0';
               PHN_EN <= '0';
               WRD_EN <= '0';
               next_state <= s1;
            ELSIF (FRAME_READY = '1') THEN
```

```
         next_state <= s14;
      ELSIF (UTT_GO = '0'
            OR
            FRAME_READY = '0') THEN
         AM_EN <= '0';
         PHN_EN <= '0';
         WRD_EN <= '0';
         AM_PHN_PHASE <= '0';
         PHN_WRD_PHASE <= '0';
         SAL_MODE <= '0';
         DEAD_PHASE <= '0';
         TKN_INIT_GO <= '0';
         INSERT_SIL_on_SOFT_RST <= '0';
         next_state <= s0;
      ELSE
         next_state <= s0;
      END IF;
   WHEN s1 =>
      TKN_INIT_GO <= '1';
      DEAD_PHASE <= '0';
      WRD_EN <= '1';
      INSERT_SIL_on_SOFT_RST <= '1';
      next_state <= s2;
   WHEN s2 =>
      IF (TKN_INIT_DONE = '1') THEN
         next_state <= s15;
      ELSE
         next_state <= s2;
      END IF;
   WHEN s3 =>
      IF (DEAD_PHASE_DONE = '1') THEN
         next_state <= s4;
      ELSE
         next_state <= s3;
      END IF;
   WHEN s4 =>
      NXT_PHN_PHASE_GO <= '1';
      DEAD_PHASE <= '0';
      next_state <= s5;
   WHEN s5 =>
      IF (NXT_PHN_PHASE_DONE = '1') THEN
         next_state <= s6;
      ELSE
         next_state <= s5;
      END IF;
   WHEN s6 =>
      WRD_EN <= '0';
      SAL_GO <= '1';
      SAL_MODE <= '1';
      PHN_EN <= '1';
      PHN_WRD_PHASE <= '1';
      next_state <= s7;
   WHEN s7 =>
      IF (SAL_DONE = '1') THEN
         SAL_MODE <= '0';
         PHN_EN <= '0';
         next_state <= s8;
      ELSE
         next_state <= s7;
      END IF;
   WHEN s8 =>
      AM_GO <= '1';
      AM_PHN_PHASE <= '1';
      AM_EN <= '1';
      next_state <= s9;
   WHEN s9 =>
      IF (AM_DONE = '1') THEN
         next_state <= s18;
      ELSE
         next_state <= s9;
      END IF;
   WHEN s10 =>
      PHN_GO <= '1';
      PHN_WRD_PHASE <= '1';
      PHN_EN <= '1';
      next_state <= s20;
   WHEN s11 =>
      IF (PHN_DONE = '1') THEN
```

```
            next_state <= s19;
         ELSE
            next_state <= s11;
         END IF;
      WHEN s12 =>
         FRAME_DONE <= '1';
         next_state <= s13;
      WHEN s13 =>
         next_state <= s0;
      WHEN s14 =>
         DEAD_PHASE <= '1';
         DEAD_PHASE_GO <= '1';
         PHN_WRD_PHASE <= '0';
         WRD_EN <= '1';
         next_state <= s3;
      WHEN s15 =>
         next_state <= s16;
      WHEN s16 =>
         WRD_EN <= '0';
         next_state <= s17;
      WHEN s17 =>
         IF (FRAME_READY = '1') THEN
            next_state <= s14;
         ELSE
            next_state <= s17;
         END IF;
      WHEN s18 =>
         AM_PHN_PHASE <= '0';
         AM_EN <= '0';
         next_state <= s10;
      WHEN s19 =>
         PHN_WRD_PHASE <= '0';
         PHN_EN <= '0';
         next_state <= s12;
      WHEN s20 =>
         next_state <= s11;
      WHEN OTHERS =>
         next_state <= s0;
   END CASE;
  END PROCESS nextstate_proc;
end fsm;
```

A.3 ACOUSTIC MODELING BLOCK: TOP-LEVEL

```
--------------------------------------------------------------------------------
-- Organization:   University of Pittsburgh
-- Author:         Kshitij Gupta
--
-- Design Name:    AM TOP-LEVEL
-- Module Name:    AM_TOP_LEVEL_v2 - struct
-- Project Name:   University of Pittsburgh's Speech Recognition System-on-a-Chip
-- Target Device:  Xilinx Virtex4 SX-35
-- Tool versions:  ISE 7.1i
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity AM_TOP_LEVEL_v2 is
        PORT (
                clk : IN std_logic;
                ce : IN std_logic;
                sclr : IN std_logic;
                AM_GO : IN std_logic;
                SVQ_BYPASS_GLOBAL : IN std_logic;
                CW_SAL_SID : IN std_logic_vector(10 downto 0);
                CW_SAL_SID_valid : IN std_logic;
                SEN_SAL_SID : IN std_logic_vector(10 downto 0);
                SEN_SAL_SID_valid : IN std_logic;
                CW_SAL_EMPTY : IN std_logic;
                SEN_SAL_EMPTY : IN std_logic;
                CSAL_EMPTY : IN std_logic;
                SEN_SAL_SID_CNT : IN std_logic_vector(10 downto 0);
                CSAL_CSID_CNT : IN std_logic_vector(10 downto 0);
                ROM_BUSY_STALL : IN std_logic;
                X : IN std_logic_vector(31 downto 0);
                X_valid : IN std_logic;
                MK : IN std_logic_vector(31 downto 0);
                VW : IN std_logic_vector(31 downto 0);
                MKVW_valid : IN std_logic;
                CSID : IN std_logic_vector(11 downto 0);
                CSID_valid : IN std_logic;
                C_SID : IN std_logic_vector(10 downto 0);
                C_SID_valid : IN std_logic;
                CURR_CSEN_DONE : IN std_logic;
                CW_SAL_POP : OUT std_logic;
                SEN_SAL_POP : OUT std_logic;
                CSAL_POP : OUT std_logic;
                FEAT_ADDR : OUT std_logic_vector(5 downto 0);
                FEAT_ADDR_VALID : OUT std_logic;
                AM_ROM_ADDR : OUT std_logic_vector(20 downto 0);
                AM_ROM_ADDR_VALID : OUT std_logic;
                AM_SID_CSID : OUT std_logic_vector(11 downto 0);
                AM_sscr_out : OUT std_logic_vector(31 downto 0);
                AM_sscr_out_valid : OUT std_logic;
                AM_DONE : OUT std_logic;
                ERR_AM_ACTIVE_SENONE_CNT_IS_ZERO : OUT std_logic;
                ERR_AM_N_ACTIVE_COMPONENTS_IS_ZERO : OUT std_logic;
                ERR_AM_FEAT_AM_ROM_ADDR_NOT_VALID_SIMULTANEOUSLY : OUT std_logic;
                ERR_AM_SENSAL_CAL_cnt_EMPTY_UNEQUAL : OUT std_logic
        );
end AM_TOP_LEVEL_v2;

architecture struct of AM_TOP_LEVEL_v2 is

COMPONENT am_ctrl_fsm_v2
        PORT (
      AM_EN               : IN     std_logic;
      AM_GO               : IN     std_logic;
```

```
      AM_PHASE_DONE       : IN     std_logic_vector ( 3 DOWNTO 0 );
      BYPASS_SVQ          : IN     std_logic;
      CAL_CNT             : IN     std_logic_vector ( 11 DOWNTO 0 );
      CAL_EMPTY           : IN     std_logic;
      CSAL_CNT            : IN     std_logic_vector ( 10 DOWNTO 0 );
      CSAL_EMPTY          : IN     std_logic;
      CURR_CAL_DONE       : IN     std_logic;
      CURR_CSEN_DONE_ctrl : IN     std_logic;
      CW_SAL_EMPTY        : IN     std_logic;
      CW_SCORING_DONE     : IN     std_logic;
      POP_NXT             : IN     std_logic;
      SEN_SAL_CNT         : IN     std_logic_vector ( 10 DOWNTO 0 );
      SEN_SAL_EMPTY       : IN     std_logic;
      clk                 : IN     std_logic;
      rst                 : IN     std_logic;
      AM_DONE             : OUT    std_logic;
      AM_PHASE_GO         : OUT    std_logic_vector ( 3 DOWNTO 0 );
      AM_PHASE_SEL        : OUT    std_logic_vector ( 1 DOWNTO 0 );
      CAL_MODE            : OUT    std_logic;
      CAL_POP             : OUT    std_logic;
      CSAL_POP            : OUT    std_logic;
      CW_SAL_POP          : OUT    std_logic;
      CW_SCORING_GO       : OUT    std_logic;
      SEN_SAL_POP         : OUT    std_logic;
      SVQ_MODE            : OUT    std_logic
         );
END COMPONENT;

COMPONENT am_rom_interface_v2
         PORT(
                  clk : IN std_logic;
                  ce : IN std_logic;
                  sclr : IN std_logic;
                  SVQ_MODE : IN std_logic;
                  CW_SCORING_GO : IN std_logic;
                  SAL_SID : IN std_logic_vector(11 downto 0);
                  SAL_SID_valid : IN std_logic;
                  CAL : IN std_logic_vector(7 downto 0);
                  CAL_valid : IN std_logic;
                  ROM_BUSY_STALL : IN std_logic;
                  SVQ_cnt_done : OUT std_logic;
                  FEAT_ADDR : OUT std_logic_vector(5 downto 0);
                  FEAT_ADDR_VALID : OUT std_logic;
                  ROM_ADDR : OUT std_logic_vector(20 downto 0);
                  ROM_ADDR_VALID : OUT std_logic;
                  UNIT_N : OUT std_logic_vector(11 downto 0);
                  UNIT_NEW : OUT std_logic;
                  UNIT_DONE : OUT std_logic;
                  POP_NXT_SID : OUT std_logic;
                  FEAT_cnt39_done : OUT std_logic;
                  ERR_AM_N_ACTIVE_COMPONENTS_IS_ZERO : OUT std_logic
         );
END COMPONENT;

component am_gaus_dist_v2
         port (
                  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;
                  MV_KW_SEL : IN std_logic;
                  MK : IN std_logic_vector (31 downto 0);
                  VW : IN std_logic_vector (31 downto 0);
                  MKVW_valid : IN std_logic;
                  UNIT_N_IN : IN std_logic_vector (11 downto 0);
                  CURR_UNIT_NEW_IN : IN std_logic;
                  CURR_UNIT_DONE_IN : IN std_logic;
                  ALL_UNIT_DONE_IN : IN std_logic;
                  scr : OUT std_logic_vector (31 downto 0);
                  scr_done : OUT std_logic;
                  W_reg : OUT std_logic_vector (31 downto 0);
                  W_valid_reg : OUT std_logic;
                  UNIT_N_OUT : OUT std_logic_vector (11 downto 0);
                  CURR_UNIT_NEW_OUT : OUT std_logic;
                  CURR_UNIT_DONE_OUT : OUT std_logic;
                  ALL_UNIT_DONE_OUT : OUT std_logic
         );
```

```
end component;

COMPONENT am_gaus_mixwt_cgen
	PORT(
		AB_IN : IN std_logic_vector(31 downto 0);
		CE_IN : IN std_logic;
		CLK_IN : IN std_logic;
		C_IN : IN std_logic_vector(31 downto 0);
		RST_IN : IN std_logic;
		P_OUT : OUT std_logic_vector(31 downto 0)
	);
END COMPONENT;

component AM_LogAdd_v2
	port (
		clk : IN std_logic;
		ce : IN std_logic;
		sclr : IN std_logic;
		cscr : IN std_logic_vector (31 downto 0);
		cscr_valid : IN std_logic;
		SID_in : IN std_logic_vector (11 downto 0);
		CURR_SEN_NEW_in : IN std_logic;
		CURR_SEN_DONE_in : IN std_logic;
		ALL_SEN_DONE_in : IN std_logic;
		SID_out : OUT std_logic_vector (11 downto 0);
		tmp_sscr_out : OUT std_logic_vector (31 downto 0);
		tmp_sscr_out_valid : OUT std_logic;
		ALL_SEN_DONE_out : OUT std_logic
	);
end component;

component AM_SSCR_CSSCR_v2
	port(
		clk : IN std_logic;
		ce : IN std_logic;
		sclr : IN std_logic;
		AM_PHASE_SEL : IN std_logic_vector(1 downto 0);
		AM_PHASE_GO : IN std_logic_vector(3 downto 0);
		SID_in : IN std_logic_vector(11 downto 0);
		tmp_sscr_in : IN std_logic_vector(31 downto 0);
		tmp_sscr_in_valid : IN std_logic;
		ALL_SEN_DONE : IN std_logic;
		CSID : IN std_logic_vector(11 downto 0);
		CSID_valid : IN std_logic;
		C_SID_in : IN std_logic_vector(10 downto 0);
		C_SID_in_valid : IN std_logic;
		CURR_CSEN_DONE_in : IN std_logic;
		ALL_CSEN_DONE_in : IN std_logic;
		AM_PHASE_DONE : OUT std_logic_vector(3 downto 0);
		SID_out : OUT std_logic_vector(11 downto 0);
		sscr_out : OUT std_logic_vector(31 downto 0);
		sscr_out_valid : OUT std_logic;
		CURR_CSEN_DONE_out : OUT std_logic
	);
end component;

component SubVQ_TOP_LEVEL_v2
	port (
		clk : IN std_logic;
		ce : IN std_logic;
		sclr : IN std_logic;
		CAL_MODE : IN std_logic;
		CWID_in : IN std_logic_vector(11 downto 0);
		CW_scr_in : IN std_logic_vector(31 downto 0);
		CW_scr_valid_in : IN std_logic;
		ALL_CW_DONE_in : IN std_logic;
		SAL_SID : IN std_logic_vector(10 downto 0);
		SAL_SID_valid : IN std_logic;
		CW_SCORING_DONE : OUT std_logic;
		CAL_out : OUT std_logic_vector(7 downto 0);
		CAL_PUSH_out : OUT std_logic
	);
end component;

component cal_fifo_v2
    port (
    clk: IN std_logic;
    sinit: IN std_logic;
```

```
    din: IN std_logic_VECTOR(7 downto 0);
    wr_en: IN std_logic;
    rd_en: IN std_logic;
    dout: OUT std_logic_VECTOR(7 downto 0);
    full: OUT std_logic;
    empty: OUT std_logic;
    rd_ack: OUT std_logic;
    data_count: OUT std_logic_VECTOR(11 downto 0));
end component;

signal AM_PHASE_SEL : std_logic_vector(1 downto 0);
signal AM_PHASE_GO : std_logic_vector(3 downto 0);
signal AM_PHASE_DONE : std_logic_vector(3 downto 0);
signal SVQ_BYPASS : std_logic;
signal SVQ_MODE : std_logic;
signal CAL_MODE : std_logic;
signal CW_SCORING_GO : std_logic;
signal CW_SCORING_DONE : std_logic;
signal SVQ_cnt_done : std_logic;
signal CURR_CAL_DONE : std_logic;
signal CURR_CSEN_DONE_ctrl : std_logic;
signal cw_used : std_logic;
signal POP_NXT_SEN_SAL_SID : std_logic;
signal FEAT_cnt39_done : std_logic;
signal FEAT_cnt39_done_reg : std_logic_vector(3 downto 0);
signal SEN_SAL_SID_s : std_logic_vector(11 downto 0);
signal UNIT_N : std_logic_vector(11 downto 0);
signal UNIT_NEW : std_logic;
signal UNIT_DONE : std_logic;
signal ALL_UNIT_DONE : std_logic;
signal N_UNIT_BUF : std_logic_vector(11 downto 0);
signal N_UNIT_NEW_BUF_s : std_logic_vector(2 downto 0);
signal N_UNIT_NEW_BUF : std_logic;
signal N_UNIT_DONE_BUF : std_logic;
signal N_UNIT_DONE_BUF_s : std_logic_vector(2 downto 0);
signal ALL_UNIT_DONE_BUF : std_logic;
signal ALL_UNIT_DONE_BUF_s : std_logic_vector(2 downto 0);
signal n_unit_no : std_logic_vector(11 downto 0);
signal curr_new : std_logic;
signal curr_done : std_logic;
signal scr : std_logic_vector(31 downto 0);
signal scr_valid : std_logic;
signal all_done : std_logic;
signal n_cw : std_logic_vector(11 downto 0);
signal cw_scr : std_logic_vector(31 downto 0);
signal cw_scr_valid : std_logic;
signal all_done_cw : std_logic;
signal n_cw_reg : std_logic_vector(11 downto 0);
signal cw_scr_reg : std_logic_vector(31 downto 0);
signal cw_scr_valid_reg : std_logic;
signal all_done_cw_reg : std_logic;
signal n_sen : std_logic_vector(11 downto 0);
signal curr_new_sen : std_logic;
signal curr_done_sen : std_logic;
signal sen_scr : std_logic_vector(31 downto 0);
signal sen_scr_valid : std_logic;
signal all_done_sen : std_logic;
signal MixWt : std_logic_vector(31 downto 0);
signal MixWt_valid : std_logic;
signal n_sen_ladd : std_logic_vector(11 downto 0);
signal cscr_valid : std_logic;
signal cscr : std_logic_vector(31 downto 0);
signal curr_new_sen_ladd : std_logic;
signal curr_done_sen_ladd : std_logic;
signal all_done_sen_ladd : std_logic;
signal n_sen_sscr : std_logic_vector(11 downto 0);
signal tmp_sscr : std_logic_vector(31 downto 0);
signal tmp_sscr_valid : std_logic;
signal all_done_sen_sscr :std_logic;
signal CAL_FIFO_dout_s : std_logic_vector(7 downto 0);
signal CAL_FIFO_dout_valid_s : std_logic;
signal CAL_FIFO_din : std_logic_vector(7 downto 0);
signal CAL_PUSH : std_logic;
signal CAL_FULL : std_logic;
signal CAL_POP : std_logic;
signal CAL_FIFO_dout : std_logic_vector(7 downto 0);
signal CAL_FIFO_dout_valid : std_logic;
signal CAL_EMPTY : std_logic;
```

```
signal CAL_FIFO_CNT : std_logic_vector(11 downto 0);

constant SEN_CNT_TH : std_logic_vector(11 downto 0) := X"320";
constant NEG_INF : std_logic_vector(31 downto 0) := X"C8000000";

begin

CURR_CAL_DONE <= UNIT_DONE and FEAT_cnt39_done_reg(0);
SEN_SAL_SID_s <= '0' & SEN_SAL_SID;
ERR_AM_FEAT_AM_ROM_ADDR_NOT_VALID_SIMULTANEOUSLY <= (X_valid xor MKVW_valid) and
(NOT(FEAT_cnt39_done_reg(3)));

process(AM_GO, SVQ_BYPASS_GLOBAL, SEN_SAL_SID_CNT)
begin
        if (AM_GO = '1') then
                if ( (SVQ_BYPASS_GLOBAL = '1') or (SEN_SAL_SID_CNT < SEN_CNT_TH) ) then
                        SVQ_BYPASS <= '1';
                else
                        SVQ_BYPASS <= '0';
                end if;
        end if;
end process;

process(AM_GO, SEN_SAL_SID_CNT)
begin
        if (AM_GO = '1') then
                if (SEN_SAL_SID_CNT = X"000") then
                        ERR_AM_ACTIVE_SENONE_CNT_IS_ZERO <= '1';
                else
                        ERR_AM_ACTIVE_SENONE_CNT_IS_ZERO <= '0';
                end if;
        end if;
end process;

process(AM_GO, CW_SAL_EMPTY, CURR_CAL_DONE, SEN_SAL_EMPTY, CAL_EMPTY)
begin
        if (AM_GO <= '1') then
                ERR_AM_SENSAL_CAL_cnt_EMPTY_UNEQUAL <= '0';
                cw_used <= '0';
        elsif ( (CW_SAL_EMPTY AND CURR_CAL_DONE) = '1') then
                cw_used <= '1';
        elsif (cw_used = '1') then
                ERR_AM_SENSAL_CAL_cnt_EMPTY_UNEQUAL        <= SEN_SAL_EMPTY xor CAL_EMPTY;
        end if;
end process;

process(SVQ_BYPASS, SEN_SAL_SID_valid, CAL_FIFO_dout, CAL_FIFO_dout_valid)
begin
        if (SVQ_BYPASS = '1') then
                CAL_FIFO_dout_s <= "11111111";
                CAL_FIFO_dout_valid_s <= SEN_SAL_SID_valid;
        else
                CAL_FIFO_dout_s <= CAL_FIFO_dout;
                CAL_FIFO_dout_valid_s <= CAL_FIFO_dout_valid;
        end if;
end process;

AM_TOP_CTRL_BLK : am_ctrl_fsm_v2
        PORT MAP(
                AM_EN => ce,
                AM_GO => AM_GO,
                rst => sclr,
                AM_PHASE_DONE => AM_PHASE_DONE,
                BYPASS_SVQ => SVQ_BYPASS,
                CAL_CNT => CAL_FIFO_CNT,
                CAL_EMPTY => CAL_EMPTY,
                CSAL_CNT => CSAL_CSID_CNT,
                CSAL_EMPTY => CSAL_EMPTY,
                CURR_CAL_DONE => CURR_CAL_DONE,
                CURR_CSEN_DONE_ctrl => CURR_CSEN_DONE_ctrl,
                CW_SAL_EMPTY => CW_SAL_EMPTY,
                CW_SCORING_DONE => CW_SCORING_DONE,
                POP_NXT => POP_NXT_SEN_SAL_SID,
                SEN_SAL_CNT => SEN_SAL_SID_CNT,
                SEN_SAL_EMPTY => SEN_SAL_EMPTY,
                clk => clk,
                AM_DONE => AM_DONE,
                AM_PHASE_GO => AM_PHASE_GO,
```

```
                AM_PHASE_SEL => AM_PHASE_SEL,
                CAL_MODE => CAL_MODE,
                CAL_POP => CAL_POP,
                CSAL_POP => CSAL_POP,
                CW_SAL_POP => CW_SAL_POP,
                CW_SCORING_GO => CW_SCORING_GO,
                SEN_SAL_POP => SEN_SAL_POP,
                SVQ_MODE => SVQ_MODE
        );

AM_TOP_ROM_INTERFACE_BLK : am_rom_interface_v2
        PORT MAP(
                clk => clk,
                ce => ce,
                sclr => sclr,
                SVQ_MODE => SVQ_MODE,
                CW_SCORING_GO => CW_SCORING_GO,
                SAL_SID => SEN_SAL_SID_s,
                SAL_SID_valid => SEN_SAL_SID_valid,
                CAL => CAL_FIFO_dout_s,
                CAL_valid => CAL_FIFO_dout_valid_s,
                ROM_BUSY_STALL => ROM_BUSY_STALL,
                SVQ_cnt_done => SVQ_cnt_done,
                FEAT_ADDR => FEAT_ADDR,
                FEAT_ADDR_VALID => FEAT_ADDR_VALID,
                ROM_ADDR => AM_ROM_ADDR,
                ROM_ADDR_VALID => AM_ROM_ADDR_VALID,
                UNIT_N => UNIT_N,
                UNIT_NEW => UNIT_NEW,
                UNIT_DONE => UNIT_DONE,
                POP_NXT_SID => POP_NXT_SEN_SAL_SID,
                FEAT_cnt39_done => FEAT_cnt39_done,
                ERR_AM_N_ACTIVE_COMPONENTS_IS_ZERO => ERR_AM_N_ACTIVE_COMPONENTS_IS_ZERO
        );

process(clk, ce, sclr, MKVW_valid, FEAT_cnt39_done, UNIT_N, UNIT_NEW, UNIT_DONE, ALL_UNIT_DONE)
begin
        if (clk'event and clk = '1') then
                if (sclr = '1') then
                        N_UNIT_BUF <= (OTHERS => '0');
                        N_UNIT_NEW_BUF_s <= (OTHERS => '0');
                        N_UNIT_DONE_BUF_s <= (OTHERS => '0');
                        ALL_UNIT_DONE_BUF_s <= (OTHERS => '0');
                        FEAT_cnt39_done_reg <= (OTHERS => '0');
                elsif (ce = '1') then
                        if (MKVW_valid = '1') then
                                N_UNIT_BUF <= UNIT_N;
                                N_UNIT_NEW_BUF_s <= N_UNIT_NEW_BUF_s(1 downto 0) & UNIT_NEW;
                                N_UNIT_DONE_BUF_s <= N_UNIT_DONE_BUF_s(1 downto 0) & UNIT_DONE;
                                ALL_UNIT_DONE_BUF_s <= ALL_UNIT_DONE_BUF_s(1 downto 0) & ALL_UNIT_DONE;
                                FEAT_cnt39_done_reg <= FEAT_cnt39_done_reg(2 downto 0) & FEAT_cnt39_done;
                        end if;
                end if;
        end if;
end process;

N_UNIT_NEW_BUF <= N_UNIT_NEW_BUF_s(2);
N_UNIT_DONE_BUF <= N_UNIT_DONE_BUF_s(2);
ALL_UNIT_DONE_BUF <= ALL_UNIT_DONE_BUF_s(2);

AM_TOP_GAUS_DIST_BLK : am_gaus_dist_v2
        port map(
                clk => clk,
                ce => ce,
                sclr => sclr,
                X => X,
                X_valid => X_valid,
                MV_KW_SEL => FEAT_cnt39_done_reg(1),
                MK => MK,
                VW => VW,
                MKVW_valid => MKVW_valid,
                UNIT_N_IN => N_UNIT_BUF,
                CURR_UNIT_NEW_IN => N_UNIT_NEW_BUF,
                CURR_UNIT_DONE_IN => N_UNIT_DONE_BUF,
                ALL_UNIT_DONE_IN => ALL_UNIT_DONE_BUF,
                scr => scr,
                scr_done => scr_valid,
                W_reg => MixWt,
```

```
                UNIT_N_OUT => n_unit_no,
                CURR_UNIT_NEW_OUT => curr_new,
                CURR_UNIT_DONE_OUT => curr_done,
                ALL_UNIT_DONE_OUT => all_done
        );

process(SVQ_cnt_done, SEN_SAL_EMPTY, CAL_EMPTY, UNIT_DONE)
begin
        if (SVQ_MODE = '1') then
                ALL_UNIT_DONE <= SVQ_cnt_done;
        else
                ALL_UNIT_DONE <= SEN_SAL_EMPTY and CAL_EMPTY and UNIT_DONE;
        end if;
end process;

process (SVQ_MODE, scr, scr_valid, n_unit_no, curr_new, curr_done, all_done)
begin
        case SVQ_MODE is
                when '1' =>
                        cw_scr <= scr;
                        cw_scr_valid <= scr_valid;
                        n_cw <= n_unit_no;
                        all_done_cw <= all_done;
                        sen_scr <= NEG_INF;
                        sen_scr_valid <= '0';
                        n_sen <= (OTHERS => '0');
                        curr_new_sen <= '0';
                        curr_done_sen <= '0';
                        all_done_sen <= '0';
                when '0' =>
                        cw_scr <= NEG_INF;
                        cw_scr_valid <= '0';
                        n_cw <= (OTHERS => '0');
                        all_done_cw <= '0';
                        sen_scr <= scr;
                        sen_scr_valid <= scr_valid;
                        n_sen <= n_unit_no;
                        curr_new_sen <= curr_new;
                        curr_done_sen <= curr_done;
                        all_done_sen <= all_done;
                when OTHERS =>
                        cw_scr <= NEG_INF;
                        cw_scr_valid <= '0';
                        n_cw <= (OTHERS => '0');
                        all_done_cw <= '0';
                        sen_scr <= NEG_INF;
                        sen_scr_valid <= '0';
                        n_sen <= (OTHERS => '0');
                        curr_new_sen <= '0';
                        curr_done_sen <= '0';
                        all_done_sen <= '0';
        end case;
end process;

process(clk, ce, sclr, MixWt_valid, sen_scr_valid, n_sen, curr_new_sen, curr_done_sen, all_done_sen)
begin
        if (clk'event and clk = '1') then
                if (sclr = '1') then
                        cscr_valid <= '0';
                elsif (sclr = '0' and ce = '1') then
                        if ( (sen_scr_valid and MixWt_valid) = '1') then
                                cscr_valid <= '1';
                                n_sen_ladd <= n_sen;
                                curr_new_sen_ladd <= curr_new_sen;
                                curr_done_sen_ladd <= curr_done_sen;
                                all_done_sen_ladd <= all_done_sen;
                        else
                                cscr_valid <= '0';
                                n_sen_ladd <= (OTHERS => '0');
                                curr_new_sen_ladd <= '0';
                                curr_done_sen_ladd <= '0';
                                all_done_sen_ladd <= '0';
                        end if;
                end if;
        end if;
end process;

AM_TOP_MIXWT_BLK : am_gaus_mixwt_cgen
```

```
        PORT MAP(
                AB_IN => sen_scr,
                CE_IN => ce,
                CLK_IN => clk,
                C_IN => MixWt,
                RST_IN => sclr,
                P_OUT => cscr
        );

AM_TOP_LOGADD_BLK : AM_LogAdd_v2
        port map(
                clk => clk,
                ce => ce,
                sclr => sclr,
                cscr => cscr,
                cscr_valid => cscr_valid,
                SID_in => n_sen_ladd,
                CURR_SEN_NEW_in => curr_new_sen_ladd,
                CURR_SEN_DONE_in => curr_done_sen_ladd,
                ALL_SEN_DONE_in => all_done_sen_ladd,
                SID_out => n_sen_sscr,
                tmp_sscr_out => tmp_sscr,
                tmp_sscr_out_valid => tmp_sscr_valid,
                ALL_SEN_DONE_out => all_done_sen_sscr
        );

AM_TOP_SSCR_CSSCR_BLK : AM_SSCR_CSSCR_v2
        port map(
                clk => clk,
                ce => ce,
                sclr => sclr,
                AM_PHASE_SEL => AM_PHASE_SEL,
                AM_PHASE_GO => AM_PHASE_GO,
                SID_in => n_sen_sscr,
                tmp_sscr_in => tmp_sscr,
                tmp_sscr_in_valid => tmp_sscr_valid,
                ALL_SEN_DONE => all_done_sen_sscr,
                CSID => CSID,
                CSID_valid => CSID_valid,
                C_SID_in => C_SID,
                C_SID_in_valid => C_SID_valid,
                CURR_CSEN_DONE_in => CURR_CSEN_DONE,
                ALL_CSEN_DONE_in => CSAL_EMPTY,
                AM_PHASE_DONE => AM_PHASE_DONE,
                SID_out => AM_SID_CSID,
                sscr_out => AM_sscr_out,
                sscr_out_valid => AM_sscr_out_valid,
                CURR_CSEN_DONE_out => CURR_CSEN_DONE_ctrl
        );

--AM_TOP_SubVQ_TOP_BLK : SubVQ_TOP_LEVEL_v2
--      port map(
--              clk => clk,
--              ce => ce,
--              sclr => sclr,
--              CAL_MODE => CAL_MODE,
--              CWID_in => n_cw,
--              CW_scr_in => cw_scr,
--              CW_scr_valid_in => cw_scr_valid,
--              ALL_CW_DONE_in => all_done_cw,
--              SAL_SID => CW_SAL_SID,
--              SAL_SID_valid => CW_SAL_SID_valid,
--              CW_SCORING_DONE => CW_SCORING_DONE,
--              CAL_out => CAL_FIFO_din,
--              CAL_PUSH_out => CAL_PUSH
--      );

process(clk, ce, sclr)
begin
        if (clk'event and clk = '1') then
                if (sclr = '1') then
                        n_cw_reg <= (OTHERS => '0');
                        cw_scr_reg <= NEG_INF;
                        cw_scr_valid_reg <= '0';
                        all_done_cw_reg <= '0';
                        CW_SCORING_DONE <= '0';
                        CAL_FIFO_din <= (OTHERS => '0');
```

```
                        CAL_PUSH <= '0';
                elsif (ce = '1') then
                        n_cw_reg <= n_cw;
                        cw_scr_reg <= cw_scr;
                        cw_scr_valid_reg <= cw_scr_valid;
                        all_done_cw_reg <= all_done_cw;
                        CW_SCORING_DONE <= CW_SCORING_GO;
                        CAL_FIFO_din <= (OTHERS => '1');
                        CAL_PUSH <= '0';
                end if;
        end if;
end process;

AM_TOP_CAL_FIFO : cal_fifo_v2
  port map (
      clk => clk,
      sinit => sclr,
      din => CAL_FIFO_din,
      wr_en => CAL_PUSH,
      rd_en => CAL_POP,
      dout => CAL_FIFO_dout,
      full => CAL_FULL,
      empty => CAL_EMPTY,
      rd_ack => CAL_FIFO_dout_valid,
      data_count => CAL_FIFO_CNT
        );

end struct;
```

A.4 ACOUSTIC MODELING BLOCK: CONTROLLER

```
--------------------------------------------------------------------------------
-- Organization:   University of Pittsburgh
-- Author:         Kshitij Gupta
--
-- Design Name:    AM CONTROLLER
-- Module Name:    AM_CTRL_FSM_v2 - fsm
-- Project Name:   University of Pittsburgh's Speech Recognition System-on-a-Chip
-- Target Device:  Xilinx Virtex4 SX-35
-- Tool versions:  ISE 7.1i
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity AM_CTRL_FSM_v2 is
   PORT(
      AM_EN               : IN     std_logic;
      AM_GO               : IN     std_logic;
      AM_PHASE_DONE       : IN     std_logic_vector ( 3 DOWNTO 0 );
      BYPASS_SVQ          : IN     std_logic;
      CAL_CNT             : IN     std_logic_vector ( 11 DOWNTO 0 );
      CAL_EMPTY           : IN     std_logic;
      CSAL_CNT            : IN     std_logic_vector ( 10 DOWNTO 0 );
      CSAL_EMPTY          : IN     std_logic;
      CURR_CAL_DONE       : IN     std_logic;
      CURR_CSEN_DONE_ctrl : IN     std_logic;
      CW_SAL_EMPTY        : IN     std_logic;
      CW_SCORING_DONE     : IN     std_logic;
      POP_NXT             : IN     std_logic;
      SEN_SAL_CNT         : IN     std_logic_vector ( 10 DOWNTO 0 );
      SEN_SAL_EMPTY       : IN     std_logic;
      clk                 : IN     std_logic;
      rst                 : IN     std_logic;
      AM_DONE             : OUT    std_logic;
      AM_PHASE_GO         : OUT    std_logic_vector ( 3 DOWNTO 0 );
      AM_PHASE_SEL        : OUT    std_logic_vector ( 1 DOWNTO 0 );
      CAL_MODE            : OUT    std_logic;
      CAL_POP             : OUT    std_logic;
      CSAL_POP            : OUT    std_logic;
      CW_SAL_POP          : OUT    std_logic;
      CW_SCORING_GO       : OUT    std_logic;
      SEN_SAL_POP         : OUT    std_logic;
      SVQ_MODE            : OUT    std_logic
   );
end AM_CTRL_FSM_v2;

architecture fsm of AM_CTRL_FSM_v2 is

   TYPE STATE_TYPE IS (
      s8,
      s9,
      s10,
      s7,
      s11,
      s12,
      AM_PH_0,
      s13,
      SVQ_DONE,
      AM_IDLE,
      s6,
      s2,
      s5,
      s3,
      s4
   );
```

```
   -- State vector declaration
   ATTRIBUTE state_vector : string;
   ATTRIBUTE state_vector OF fsm : ARCHITECTURE IS "current_state";

   -- Declare current and next state signals
   SIGNAL current_state : STATE_TYPE;
   SIGNAL next_state : STATE_TYPE;

BEGIN

   -----------------------------------------------------------------
   clocked_proc : PROCESS (
      clk,
      rst
   )
   -----------------------------------------------------------------
   BEGIN
      IF (rst = '1') THEN
         current_state <= AM_IDLE;
      ELSIF (clk'EVENT AND clk = '1') THEN
         IF (AM_EN = '1') THEN
            current_state <= next_state;
         END IF;
      END IF;
   END PROCESS clocked_proc;

   -----------------------------------------------------------------
   nextstate_proc : PROCESS (
      AM_GO,
      AM_PHASE_DONE,
      BYPASS_SVQ,
      CAL_EMPTY,
      CSAL_EMPTY,
      CURR_CAL_DONE,
      CURR_CSEN_DONE_ctrl,
      CW_SAL_EMPTY,
      CW_SCORING_DONE,
      POP_NXT,
      SEN_SAL_EMPTY,
      current_state
   )
   -----------------------------------------------------------------
   BEGIN
      CASE current_state IS
         WHEN s8 =>
            IF (SEN_SAL_EMPTY = '0'
                AND
                POP_NXT = '1') THEN
               next_state <= s7;
            ELSIF (SEN_SAL_EMPTY  = '1'
                   AND
                   CAL_EMPTY = '1') THEN
               next_state <= s9;
            ELSE
               next_state <= s8;
            END IF;
         WHEN s9 =>
            IF (AM_PHASE_DONE(1) = '1') THEN
               next_state <= s10;
            ELSE
               next_state <= s9;
            END IF;
         WHEN s10 =>
            IF (AM_PHASE_DONE(2) = '1') THEN
               next_state <= s11;
            ELSE
               next_state <= s10;
            END IF;
         WHEN s7 =>
            next_state <= s8;
         WHEN s11 =>
            IF (CSAL_EMPTY = '1') THEN
               next_state <= s13;
            ELSE
               next_state <= s12;
            END IF;
         WHEN s12 =>
```

```
        IF (AM_PHASE_DONE(3) = '1') THEN
           next_state <= s13;
         ELSIF (CSAL_EMPTY = '0'
                AND
                CURR_CSEN_DONE_ctrl = '1') THEN
           next_state <= s11;
         ELSE
           next_state <= s12;
         END IF;
      WHEN AM_PH_0 =>
         IF (AM_PHASE_DONE(0) = '1') THEN
            next_state <= s7;
         ELSE
            next_state <= AM_PH_0;
         END IF;
      WHEN s13 =>
         next_state <= AM_IDLE;
      WHEN SVQ_DONE =>
         next_state <= AM_PH_0;
      WHEN AM_IDLE =>
         IF (AM_GO = '1') THEN
            next_state <= s2;
         ELSIF (AM_GO = '0') THEN
            next_state <= AM_IDLE;
         ELSE
            next_state <= AM_IDLE;
         END IF;
      WHEN s6 =>
         IF (CW_SAL_EMPTY = '0' AND CURR_CAL_DONE = '1') THEN
            next_state <= s5;
         ELSIF (CW_SAL_EMPTY = '1'
                AND
                CURR_CAL_DONE = '1') THEN
            next_state <= SVQ_DONE;
         ELSE
            next_state <= s6;
         END IF;
      WHEN s2 =>
         IF (BYPASS_SVQ = '1') THEN
            next_state <= SVQ_DONE;
         ELSIF (BYPASS_SVQ = '0') THEN
            next_state <= s3;
         ELSE
            next_state <= s2;
         END IF;
      WHEN s5 =>
         next_state <= s6;
      WHEN s3 =>
         next_state <= s4;
      WHEN s4 =>
         IF (CW_SCORING_DONE = '1') THEN
            next_state <= s5;
         ELSE
            next_state <= s4;
         END IF;
      WHEN OTHERS =>
         next_state <= AM_IDLE;
   END CASE;
END PROCESS nextstate_proc;

-----------------------------------------------------------------
output_proc : PROCESS (
   AM_GO,
   AM_PHASE_DONE,
   BYPASS_SVQ,
   CSAL_EMPTY,
   CW_SCORING_DONE,
   current_state
)
-----------------------------------------------------------------
BEGIN
   -- Default Assignment
   AM_DONE <= '0';
   AM_PHASE_GO <= (others => '0');
   CAL_POP <= '0';
   CSAL_POP <= '0';
   CW_SAL_POP <= '0';
   CW_SCORING_GO <= '0';
```

```
      SEN_SAL_POP <= '0';

      -- Combined Actions
      CASE current_state IS
         WHEN s9 =>
            IF (AM_PHASE_DONE(1) = '1') THEN
               AM_PHASE_SEL <= "10";
               AM_PHASE_GO <= "0100";
            END IF;
         WHEN s10 =>
            IF (AM_PHASE_DONE(2) = '1') THEN
               AM_PHASE_SEL <= "11";
               --AM_PHASE_GO <= "1000";
            END IF;
         WHEN s7 =>
            SEN_SAL_POP <= '1';
            CAL_POP <= '0';
         WHEN s11 =>
            IF (NOT(CSAL_EMPTY = '1')) THEN
               CSAL_POP <= '1';
            END IF;
         WHEN AM_PH_0 =>
            IF (AM_PHASE_DONE(0) = '1') THEN
               AM_PHASE_SEL <= "01";
               AM_PHASE_GO <= "0010";
            END IF;
         WHEN s13 =>
            AM_DONE <= '1';
         WHEN SVQ_DONE =>
            AM_PHASE_SEL <= "00";
            AM_PHASE_GO <= "0001";
            SVQ_MODE <= '0';
            CAL_MODE <= '0';
         WHEN AM_IDLE =>
            IF (AM_GO = '1') THEN
               CAL_MODE <= '0';
               SVQ_MODE <= '0';
            END IF;
         WHEN s2 =>
            IF (BYPASS_SVQ = '1') THEN
            ELSIF (BYPASS_SVQ = '0') THEN
               SVQ_MODE <= '1';
               CW_SCORING_GO <= '1';
            END IF;
         WHEN s5 =>
            CW_SAL_POP <= '1';
            CAL_MODE <= '1';
         WHEN s4 =>
            IF (CW_SCORING_DONE = '1') THEN
               SVQ_MODE <= '0';
            END IF;
         WHEN OTHERS =>
            NULL;
      END CASE;
   END PROCESS output_proc;
end fsm;
```

A.5 PHONE BLOCK: TOP-LEVEL

```
--------------------------------------------------------------------------------
-- Organization:   University of Pittsburgh
-- Author:         Kshitij Gupta
--
-- Design Name:    PHN TOP-LEVEL
-- Module Name:    phn_top - struct
-- Project Name:   University of Pittsburgh's Speech Recognition System-on-a-Chip
-- Target Device:  Xilinx Virtex4 SX-35
-- Tool versions:  ISE 7.1i
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity phn_top is
        port (
                clk, ce, sclr : IN std_logic;
                rst : IN std_logic;
                SAL_MODE : IN std_logic;
                SAL_GO : IN std_logic;
                SAL_RST_DONE : IN std_logic;
                PHN_GO : IN std_logic;
                tmat_sscr_in : IN std_logic_vector(179 downto 0);
                tmat_sscr_in_valid : IN std_logic;
                nPAL_TKN : IN std_logic_vector(10 downto 0);
                nPAL_TKN_valid : IN std_logic;
                nPAL_empty : IN std_logic;
                ram_rd_dout : IN std_logic_vector(215 downto 0);
                ram_rd_dout_valid : IN std_logic_vector(3 downto 0);
                nPAL_din_s : OUT std_logic_vector(10 downto 0);
                nPAL_PUSH_s : OUT std_logic;
                nPAL_POP : OUT std_logic;
                ram_wr_addr : OUT std_logic_vector(9 downto 0);
                ram_wr_din : OUT std_logic_vector(215 downto 0);
                ram_wr_en : OUT std_logic_vector(3 downto 0);
                ram_rd_addr : OUT std_logic_vector(9 downto 0);
                ram_rd_en : OUT std_logic_vector(3 downto 0);
                nPAL_PAL_DONE : OUT std_logic;
                dead_PUSH_s : OUT std_logic;
                nxt_phn_PUSH_s : OUT std_logic;
                TKN_prn : OUT std_logic_vector(10 downto 0);
                WRD_TH : OUT std_logic_vector(31 downto 0);
                th_valid : OUT std_logic;
                PHN_DONE : OUT std_logic
        );
end phn_top;

architecture struct of phn_top is

COMPONENT phn_ctrl
        PORT(
      PAL_DONE            : IN     std_logic;
      PAL_EMPTY           : IN     std_logic;
      PHN_GO              : IN     std_logic;
      SAL_GO              : IN     std_logic;
      SAL_RST_DONE        : IN     std_logic;
      ce                  : IN     std_logic;
      clk                 : IN     std_logic;
      nPAL_DONE           : IN     std_logic;
      nPAL_EMPTY          : IN     std_logic;
      prune_done          : IN     std_logic;
      sclr                : IN     std_logic;
      scr_done            : IN     std_logic;
      CURR_PHASE_DONE     : OUT    std_logic;
      PAL_POP             : OUT    std_logic;
```

```
      PHN_DONE            : OUT    std_logic;
      PHN_PHASE_SEL       : OUT    std_logic_vector ( 1 DOWNTO 0 );
      STALL_POP                     : OUT    std_logic;
      err_nPAL_empty      : OUT    std_logic;
      err_PAL_empty       : OUT    std_logic;
      nPAL_POP            : OUT    std_logic;
      phn_ptr_ram_disable : OUT    std_logic
         );
END COMPONENT;

component phn_wrd_fifo
    port (
    clk: IN std_logic;
    sinit: IN std_logic;
    din: IN std_logic_VECTOR(10 downto 0);
    wr_en: IN std_logic;
    rd_en: IN std_logic;
    dout: OUT std_logic_VECTOR(10 downto 0);
    full: OUT std_logic;
    empty: OUT std_logic;
    rd_ack: OUT std_logic;
    rd_err: OUT std_logic;
    wr_err: OUT std_logic;
    data_count: OUT std_logic_VECTOR(9 downto 0));
end component;

COMPONENT phn_scr_info_buf
        PORT(
                clk : IN std_logic;
                ce : IN std_logic;
                TKN_in : IN std_logic_vector(9 downto 0);
                TKN_in_valid : IN std_logic;
                curr_hi012 : IN std_logic_vector(127 downto 0);
                curr_hi012_valid : IN std_logic;
                scr_wend : IN std_logic;
                scr_wend_valid : IN std_logic;
                scr_done_in : IN std_logic;
                TKN_in_reg : OUT std_logic_vector(9 downto 0);
                TKN_in_valid_reg : OUT std_logic;
                curr_hi012_reg : OUT std_logic_vector(127 downto 0);
                curr_hi012_reg_valid : OUT std_logic;
                scr_wend_reg : OUT std_logic;
                scr_wend_valid_reg : OUT std_logic;
                scr_done_in_reg : OUT std_logic
                );
END COMPONENT;

COMPONENT phn_phone_calc
        PORT(
                clk : IN std_logic;
                ce : IN std_logic;
                sclr : IN std_logic;
                PHN_GO : IN std_logic;
                TKN_in : IN std_logic_vector(9 downto 0);
                TKN_in_valid : IN std_logic;
                W_END_in : IN std_logic;
                W_END_valid : IN std_logic;
                ALL_PHN_SCR_DONE : IN std_logic;
                ALL_PHN_SCR_DONE_valid : IN std_logic;
                sscr_h012 : IN std_logic_vector(95 downto 0);
                sscr_h012_valid : IN std_logic;
                curr_hi012 : IN std_logic_vector(127 downto 0);
                curr_hi012_valid : IN std_logic;
                tmat : IN std_logic_vector(83 downto 0);
                tmat_valid : IN std_logic;
                TKN_out : OUT std_logic_vector(9 downto 0);
                TKN_out_valid : OUT std_logic;
                nxt_hiob : OUT std_logic_vector(95 downto 0);
                nxt_h012 : OUT std_logic_vector(95 downto 0);
                nxt_valid : OUT std_logic;
                hmm_th : OUT std_logic_vector(31 downto 0);
                nxt_phn_th : OUT std_logic_vector(31 downto 0);
                wrd_th : OUT std_logic_vector(31 downto 0);
                th_valid : OUT std_logic
        );
END COMPONENT;
```

```
COMPONENT phn_prune
        PORT(
                clk : IN std_logic;
                ce : IN std_logic;
                sclr : IN std_logic;
                PHN_GO : IN std_logic;
                TKN_in : IN std_logic_vector(9 downto 0);
                TKN_in_valid : IN std_logic;
                HMM_ACTIVE_TH : IN std_logic_vector(31 downto 0);
                NXT_PHN_TH : IN std_logic_vector(31 downto 0);
                TH_VALID : IN std_logic;
                scr_hob : IN std_logic_vector(63 downto 0);
                scr_valid : IN std_logic;
                TKN_out : OUT std_logic_vector(10 downto 0);
                TKN_out_valid : OUT std_logic;
                dead_PUSH : OUT std_logic;
                PAL_PUSH : OUT std_logic;
                nxt_phn_PUSH : OUT std_logic
        );
END COMPONENT;

component shift_reg_generic
        generic(
                width : integer;
                stages : integer
        );
        port(
                clk, ce : IN std_logic;
                din : IN std_logic_vector(width-1 downto 0);
                en_in : IN std_logic;
                dout : OUT std_logic_vector(width-1 downto 0);
                en_out : OUT std_logic
        );
end component;

component shift_reg_1bit_generic
        generic(
                stages : integer
        );
        port(
                clk, ce : IN std_logic;
                din : IN std_logic;
                en_in : IN std_logic;
                dout : OUT std_logic;
                en_out : OUT std_logic
        );
end component;

signal PHN_PHASE_SEL : std_logic_vector(1 downto 0);
signal PHN_PHASE : std_logic_vector(3 downto 0);
signal CURR_PHASE_DONE : std_logic;
signal phn_ptr_ram_disable, phn_ptr_ram_disable_reg : std_logic;
signal sscr_h012, sscr_h012_reg  : std_logic_vector(95 downto 0);
signal sscr_h012_valid, sscr_h012_valid_reg : std_logic;
signal tmat_scr, tmat_scr_reg : std_logic_vector(83 downto 0);
signal tmat_scr_valid, tmat_scr_valid_reg : std_logic;
signal curr_hi012, curr_hi012_reg : std_logic_vector(127 downto 0);
signal curr_hi012_valid, curr_hi012_valid_reg : std_logic;
signal nxt_h012, nxt_hiob : std_logic_vector(95 downto 0);
signal nxt_valid : std_logic;
signal TKN_in_phn, TKN_out_phn : std_logic_vector(9 downto 0);
signal TKN_in_phn_valid, TKN_out_phn_valid : std_logic;
signal TKN_in_phncalc, TKN_in_phncalc_reg : std_logic_vector(9 downto 0);
signal TKN_in_phncalc_valid, TKN_in_phncalc_valid_reg : std_logic;
signal TKN_in_prn : std_logic_vector(9 downto 0);  signal TKN_in_prn_valid : std_logic;
signal TKN_prn_active : std_logic_vector(10 downto 0);  signal TKN_prn_active_valid : std_logic;
signal nPAL_TKN_reg : std_logic_vector(10 downto 0);
signal nPAL_TKN_valid_reg : std_logic;
signal nPAL_POP_s : std_logic;
signal nPAL_done : std_logic;
signal PAL_POP_s : std_logic;
signal PAL_POP : std_logic;
signal PAL_TKN : std_logic_vector(10 downto 0);
signal PAL_TKN_valid : std_logic;
signal PAL_TKN_reg : std_logic_vector(10 downto 0);
signal PAL_TKN_valid_reg : std_logic;
signal PAL_din : std_logic_vector(10 downto 0);
signal PAL_PUSH : std_logic;
```

```
signal PAL_PUSH_s : std_logic;
signal PAL_PUSH_t : std_logic;
signal PAL_full, PAL_empty, PAL_rd_err, PAL_wr_err : std_logic;
signal PAL_data_cnt : std_logic_vector(9 downto 0);
signal PAL_done: std_logic;
signal STALL_POP : std_logic;
signal nSTALL_POP : std_logic;
signal sal_done_s, scr_done_s, prn_done_s : std_logic;
signal sal_done_s_reg, scr_done_s_reg, prn_done_s_reg : std_logic;
signal scr_done_s_buf : std_logic;
signal scr_wend, scr_wend_reg: std_logic;
signal scr_wend_valid, scr_wend_valid_reg : std_logic;
signal HMM_ACTIVE_TH, NXT_PHN_TH : std_logic_vector(31 downto 0);
signal th_valid_s : std_logic;
signal prn_hob : std_logic_vector(63 downto 0);
signal prn_hob_valid : std_logic;
signal done_in_dummy, done_out_dummy : std_logic_vector(3 downto 0);
signal err_nPAL_empty, err_PAL_empty : std_logic;

begin

nPAL_PAL_DONE <= sal_done_s_reg;
nSTALL_POP <= NOT(STALL_POP);

th_valid <= th_valid_s;
sscr_h012 <= tmat_sscr_in(95 downto 0);
sscr_h012_valid <= tmat_sscr_in_valid;
tmat_scr <= tmat_sscr_in(179 downto 96);
tmat_scr_valid <= tmat_sscr_in_valid;
curr_hi012 <= ram_rd_dout(105 downto 74) & ram_rd_dout(201 downto 106);
curr_hi012_valid <= ram_rd_dout_valid(1) and ram_rd_dout_valid(2) and (PHN_PHASE(1) or PHN_PHASE(2));
scr_wend <= ram_rd_dout(215);
scr_wend_valid <= ram_rd_dout_valid(0) and (PHN_PHASE(1) or PHN_PHASE(2));
prn_hob <= ram_rd_dout(73 downto 10);
prn_hob_valid <= ram_rd_dout_valid(1) and PHN_PHASE(3);
TKN_prn <= TKN_prn_active;

process(PHN_PHASE_SEL)
begin
        case PHN_PHASE_SEL is
                when "00" => PHN_PHASE <= "0001";
                when "01" => PHN_PHASE <= "0010";
                when "10" => PHN_PHASE <= "0100";
                when "11" => PHN_PHASE <= "1000";
                when OTHERS => PHN_PHASE <= "0000";
        end case;
end process;

process(PHN_PHASE_SEL, TKN_out_phn, TKN_out_phn_valid, nxt_valid, nxt_h012, nxt_hiob)
begin
        ram_wr_addr <= TKN_out_phn;
        if ((TKN_out_phn_valid and nxt_valid) = '1') then
                case PHN_PHASE_SEL is
                        when "00" =>
                                ram_wr_en <= "0000";
                                ram_wr_din <= (OTHERS => '0');
                        when "01" =>
                                ram_wr_en <= "0110";
                                ram_wr_din <= "00000000000000" & nxt_h012 & nxt_hiob & "0000000000";
                        when "10" =>
                                ram_wr_en <= "0110";
                                ram_wr_din <= "00000000000000" & nxt_h012 & nxt_hiob & "0000000000";
                        when "11" =>
                                ram_wr_en <= "0000";
                                ram_wr_din <= (OTHERS => '0');
                        when OTHERS =>
                                ram_wr_en <= "0000";
                                ram_wr_din <= (OTHERS => '0');
                end case;
        else
                ram_wr_en <= "0000";
                ram_wr_din <= (OTHERS => '0');
        end if;
end process;

process(PHN_PHASE_SEL, TKN_in_phn, TKN_in_phn_valid)
begin
```

```
        ram_rd_addr <= TKN_in_phn;
        if (TKN_in_phn_valid = '1') then
                case PHN_PHASE_SEL is
                        when "00" =>
                                ram_rd_en <= "1000";
                        when "01" =>
                                ram_rd_en <= "1110";
                        when "10" =>
                                ram_rd_en <= "1110";
                        when "11" =>
                                ram_rd_en <= "0010";
                        when OTHERS =>
                                ram_rd_en <= "0000";
                end case;
        else
                ram_rd_en <= "0000";
        end if;
end process;

process(nPAL_TKN(10), nPAL_TKN_valid)
begin
        if (nPAL_TKN_valid = '1') then
                case nPAL_TKN(10) is
                        when '0' => nPAL_done <= '0';
                        when '1' =>      nPAL_done <= '1';
                        when OTHERS => nPAL_done <= '0';
                end case;
        else
                nPAL_done <= '0';
        end if;
end process;

REG_FB_nPAL_DATA_BLK : shift_reg_generic
        generic map(11, 1)
        port map(clk, ce, nPAL_TKN, nPAL_TKN_valid, nPAL_TKN_reg, nPAL_TKN_valid_reg);

process(SAL_MODE, PHN_PHASE, nPAL_TKN_reg, nPAL_TKN_valid_reg)
begin
        if ((nPAL_TKN_valid_reg and SAL_MODE) = '1') then
                nPAL_PUSH_s <= '1';
                nPAL_din_s <= nPAL_TKN_reg;
        else
                nPAL_PUSH_s <= '0';
        end if;
end process;

process(PHN_PHASE_SEL, SAL_MODE, nPAL_TKN_reg, nPAL_TKN_valid_reg, PAL_TKN_reg, PAL_TKN_valid_reg,
TKN_prn_active, TKN_prn_active_valid, PAL_PUSH_t, phn_ptr_ram_disable_reg)
begin
        case PHN_PHASE_SEL is
                when "00" =>
                        PAL_PUSH_s <= PAL_TKN_valid_reg and SAL_MODE;
                        PAL_din_s <= PAL_TKN_reg;
                when "01" =>
                        PAL_PUSH_s <= nPAL_TKN_valid_reg and (not(phn_ptr_ram_disable_reg));
                        PAL_din_s <= nPAL_TKN_reg;
                when "10" =>
                        PAL_PUSH_s <= PAL_TKN_valid_reg;
                        PAL_din_s <= PAL_TKN_reg;
                when "11" =>
                        PAL_PUSH_s <= PAL_PUSH_t and TKN_prn_active_valid;
                        PAL_din_s <= TKN_prn_active;
                when OTHERS =>
                        PAL_PUSH_s <= '0';
        end case;
end process;

REG_PHN_RAM_DISABLE_BLK : shift_reg_1bit_generic
        generic map(1)
        port map(clk, ce, phn_ptr_ram_disable, done_in_dummy(3), phn_ptr_ram_disable_reg,
done_out_dummy(3));

REG_FB_PAL_DATA_BLK : shift_reg_generic
        generic map(11, 1)
        port map(clk, ce, PAL_TKN, PAL_TKN_valid, PAL_TKN_reg, PAL_TKN_valid_reg);

process(PAL_TKN(10), PAL_TKN_valid)
begin
```

```
        if (PAL_TKN_valid = '1') then
                case PAL_TKN(10) is
                        when '0' => PAL_done <= '0';
                        when '1' =>      PAL_done <= '1';
                        when OTHERS => PAL_done <= '0';
                end case;
        else
                PAL_done <= '0';
        end if;
end process;

process(SAL_GO, PAL_din_s, PAL_PUSH_s)
begin
        if (SAL_GO = '1') then
                PAL_PUSH <= '1';
                PAL_din <= "10000000001";
        else
                PAL_PUSH <= PAL_PUSH_s;
                PAL_din <= PAL_din_s;
        end if;
end process;

process(clk, ce, PHN_PHASE_SEL, phn_ptr_ram_disable, nPAL_TKN, nPAL_TKN_valid, PAL_TKN, PAL_TKN_valid)
begin
        if (clk'event and clk = '1') then
                if (ce = '1') then
                        case PHN_PHASE_SEL is
                                when "00" =>
                                        if (nPAL_TKN_valid = '1' and PAL_TKN_valid = '0') then
                                                TKN_in_phn_valid <= nPAL_TKN_valid and
(not(phn_ptr_ram_disable));
                                                TKN_in_phn <= nPAL_TKN(9 downto 0);
                                        elsif (nPAL_TKN_valid = '0' and PAL_TKN_valid = '1') then
                                                TKN_in_phn_valid <= PAL_TKN_valid and
(not(phn_ptr_ram_disable));
                                                TKN_in_phn <= PAL_TKN(9 downto 0);
                                        else
                                                TKN_in_phn_valid <= '0';
                                                TKN_in_phn <= (OTHERS => '0');
                                        end if;
                                when "01" =>
                                        TKN_in_phn_valid <= nPAL_TKN_valid and
(not(phn_ptr_ram_disable));
                                        TKN_in_phn <= nPAL_TKN(9 downto 0);
                                when "10" =>
                                        TKN_in_phn_valid <= PAL_TKN_valid and
(not(phn_ptr_ram_disable));
                                        TKN_in_phn <= PAL_TKN(9 downto 0);
                                when "11" =>
                                        TKN_in_phn_valid <= PAL_TKN_valid and
(not(phn_ptr_ram_disable));
                                        TKN_in_phn <= PAL_TKN(9 downto 0);
                                when OTHERS =>
                                        TKN_in_phn_valid <= '0';
                                        TKN_in_phn <= (OTHERS => '0');
                        end case;
                end if;
        end if;
end process;

process(PHN_PHASE_SEL, TKN_in_phn, TKN_in_phn_valid)
begin
        case PHN_PHASE_SEL is
                when "01" =>
                        TKN_in_phncalc <= TKN_in_phn;
                        TKN_in_phncalc_valid <= TKN_in_phn_valid;
                when "10" =>
                        TKN_in_phncalc <= TKN_in_phn;
                        TKN_in_phncalc_valid <= TKN_in_phn_valid;
                when "11" =>
                        TKN_in_prn <= TKN_in_phn;
                        TKN_in_prn_valid <= TKN_in_phn_valid;
                when OTHERS =>
                        TKN_in_phncalc <= (OTHERS => '0');
                        TKN_in_prn <= (OTHERS => '0');
                        TKN_in_phncalc_valid <= '0';
                        TKN_in_prn_valid <= '0';
        end case;
```

```
end process;

process(PHN_PHASE_SEL, CURR_PHASE_DONE)
begin
        case PHN_PHASE_SEL is
                when "00" =>
                        sal_done_s <= CURR_PHASE_DONE;
                        scr_done_s <= '0';
                        prn_done_s <= '0';
                when "10" =>
                        sal_done_s <= '0';
                        scr_done_s <= CURR_PHASE_DONE;
                        prn_done_s <= '0';
                when "11" =>
                        sal_done_s <= '0';
                        scr_done_s <= '0';
                        prn_done_s <= CURR_PHASE_DONE;
                when OTHERS =>
                        sal_done_s <= '0';
                        scr_done_s <= '0';
                        prn_done_s <= '0';
        end case;
end process;

PHN_CTRL_BLK : phn_ctrl
        PORT MAP(
                clk => clk,
                ce => ce,
                sclr => sclr,
                SAL_GO => SAL_GO,
                SAL_RST_DONE => SAL_RST_DONE,
                PHN_GO => PHN_GO,
                nPAL_DONE => nPAL_done,
                nPAL_EMPTY => nPAL_empty,
                PAL_DONE => PAL_done,
                PAL_EMPTY => PAL_empty,
                prune_done => prn_done_s_reg,
                scr_done => scr_done_s_reg,
                PHN_PHASE_SEL => PHN_PHASE_SEL,
                nPAL_POP => nPAL_POP_s,
                PAL_POP => PAL_POP_s,
                STALL_POP => STALL_POP,
                CURR_PHASE_DONE => CURR_PHASE_DONE,
                err_nPAL_empty => err_nPAL_empty,
                err_PAL_empty => err_PAL_empty,
                phn_ptr_ram_disable => phn_ptr_ram_disable,
                PHN_DONE => PHN_DONE
        );

nPAL_POP <= (nPAL_POP_s and nSTALL_POP and PHN_PHASE(1)) or (nPAL_POP_s and NOT(PHN_PHASE(1)));
PAL_POP <= (PAL_POP_s and nSTALL_POP and PHN_PHASE(2)) or (PAL_POP_s and NOT(PHN_PHASE(2)));

SHIFT_REG_SAL_DONE_BLK : shift_reg_1bit_generic
        generic map(12)
        port map(clk, ce, sal_done_s, done_in_dummy(0), sal_done_s_reg, done_out_dummy(0));

SHIFT_REG_SCR_DONE_BLK : shift_reg_1bit_generic
        generic map(8)
        port map(clk, ce, scr_done_s_buf, done_in_dummy(1), scr_done_s_reg, done_out_dummy(1));

SHIFT_REG_PRN_DONE_BLK : shift_reg_1bit_generic
        generic map(6)
        port map(clk, ce, prn_done_s, done_in_dummy(2), prn_done_s_reg, done_out_dummy(2));

PAL_FIFO : phn_wrd_fifo
  port map (clk, rst, PAL_din, PAL_PUSH, PAL_POP, PAL_TKN, PAL_full, PAL_empty, PAL_TKN_valid, PAL_rd_err,
PAL_wr_err, PAL_data_cnt);

BUF_PHN_SCR_IN_INFO_BLK : phn_scr_info_buf
        PORT MAP(
                clk => clk,
                ce => ce,
                TKN_in => TKN_in_phncalc,
                TKN_in_valid => TKN_in_phncalc_valid,
                curr_hi012 => curr_hi012,
                curr_hi012_valid => curr_hi012_valid,
                scr_wend => scr_wend,
                scr_wend_valid => scr_wend_valid,
```

```
                scr_done_in => scr_done_s,
                TKN_in_reg => TKN_in_phncalc_reg,
                TKN_in_valid_reg => TKN_in_phncalc_valid_reg,
                curr_hi012_reg => curr_hi012_reg,
                curr_hi012_reg_valid => curr_hi012_valid_reg,
                scr_wend_reg => scr_wend_reg,
                scr_wend_valid_reg => scr_wend_valid_reg,
                scr_done_in_reg => scr_done_s_buf
        );

PHN_CALC_BLK : phn_phone_calc
        PORT MAP(
                clk => clk,
                ce => ce,
                sclr => sclr,
                PHN_GO => PHN_GO,
                TKN_in => TKN_in_phncalc_reg,
                TKN_in_valid => TKN_in_phncalc_valid_reg,
                W_END_in => scr_wend_reg,
                W_END_valid => scr_wend_valid_reg,
                ALL_PHN_SCR_DONE => scr_done_s_buf,
                ALL_PHN_SCR_DONE_valid => scr_done_s_buf,
                sscr_h012 => sscr_h012,
                sscr_h012_valid => sscr_h012_valid,
                curr_hi012 => curr_hi012_reg,
                curr_hi012_valid => curr_hi012_valid_reg,
                tmat => tmat_scr,
                tmat_valid => tmat_scr_valid,
                TKN_out => TKN_out_phn,
                TKN_out_valid => TKN_out_phn_valid,
                nxt_hiob => nxt_hiob,
                nxt_h012 => nxt_h012,
                nxt_valid => nxt_valid,
                hmm_th => HMM_ACTIVE_TH,
                nxt_phn_th => NXT_PHN_TH,
                wrd_th => WRD_TH,
                th_valid => th_valid_s
        );

PHN_PRUNE_BLK : phn_prune
        PORT MAP(
                clk => clk,
                ce => ce,
                sclr => sclr,
                PHN_GO => PHN_GO,
                TKN_in => TKN_in_prn,
                TKN_in_valid => TKN_in_prn_valid,
                HMM_ACTIVE_TH => HMM_ACTIVE_TH,
                NXT_PHN_TH => NXT_PHN_TH,
                TH_VALID => th_valid_s,
                scr_hob => prn_hob,
                scr_valid => prn_hob_valid,
                TKN_out => TKN_prn_active,
                TKN_out_valid => TKN_prn_active_valid,
                dead_PUSH => dead_PUSH_s,
                PAL_PUSH => PAL_PUSH_t,
                nxt_phn_PUSH => nxt_phn_PUSH_s
        );

end struct;
```

A.6 PHONE BLOCK: CONTROLLER

```
--------------------------------------------------------------------------------
-- Organization:   University of Pittsburgh
-- Author:         Kshitij Gupta
--
-- Design Name:    PHN CONTROLLER
-- Module Name:    phn_ctrl - fsm
-- Project Name:   University of Pittsburgh's Speech Recognition System-on-a-Chip
-- Target Device:  Xilinx Virtex4 SX-35
-- Tool versions:  ISE 7.1i
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity phn_ctrl is
   PORT(
      PAL_DONE            : IN     std_logic;
      PAL_EMPTY           : IN     std_logic;
      PHN_GO              : IN     std_logic;
      SAL_GO              : IN     std_logic;
      SAL_RST_DONE        : IN     std_logic;
      ce                  : IN     std_logic;
      clk                 : IN     std_logic;
      nPAL_DONE           : IN     std_logic;
      nPAL_EMPTY          : IN     std_logic;
      prune_done          : IN     std_logic;
      sclr                : IN     std_logic;
      scr_done            : IN     std_logic;
      CURR_PHASE_DONE     : OUT    std_logic;
      PAL_POP             : OUT    std_logic;
      PHN_DONE            : OUT    std_logic;
      PHN_PHASE_SEL       : OUT    std_logic_vector ( 1 DOWNTO 0 );
      STALL_POP           : OUT    std_logic;
      err_npal_empty      : OUT    std_logic;
      err_pal_empty       : OUT    std_logic;
      nPAL_POP            : OUT    std_logic;
      phn_ptr_ram_disable : OUT    std_logic
   );
end phn_ctrl;

ARCHITECTURE fsm OF phn_ctrl IS

   TYPE STATE_TYPE IS (
      s0,
      s1,
      s7,
      s13,
      s16,
      s3,
      s4,
      s5,
      s6,
      s9,
      s10,
      s11,
      s12,
      s15,
      s9_stall,
      s11_stall
   );

   -- State vector declaration
   ATTRIBUTE state_vector : string;
```

```
   ATTRIBUTE state_vector OF fsm : ARCHITECTURE IS "current_state";

   -- Declare current and next state signals
   SIGNAL current_state : STATE_TYPE;
   SIGNAL next_state : STATE_TYPE;

BEGIN

   -----------------------------------------------------------------
   clocked_proc : PROCESS (
      clk
   )
   -----------------------------------------------------------------
   BEGIN
      IF (clk'EVENT AND clk = '1') THEN
         IF (sclr = '1') THEN
            current_state <= s0;
         ELSIF (ce = '1') THEN
            current_state <= next_state;
         END IF;
      END IF;
   END PROCESS clocked_proc;

   -----------------------------------------------------------------
   nextstate_proc : PROCESS (
      PAL_DONE,
      PAL_EMPTY,
      PHN_GO,
      SAL_GO,
      SAL_RST_DONE,
      current_state,
      nPAL_DONE,
      nPAL_EMPTY,
      prune_done,
      sclr,
      scr_done
   )
   -----------------------------------------------------------------
   BEGIN
      -- Default Assignment
      CURR_PHASE_DONE <= '0';
      PAL_POP <= '0';
      PHN_DONE <= '0';
      err_npal_empty <= '0';
      err_pal_empty <= '0';
      nPAL_POP <= '0';
      phn_ptr_ram_disable <= '0';

      -- Combined Actions
      CASE current_state IS
         WHEN s0 =>
            IF (SAL_GO = '1') THEN
               next_state <= s1;
            ELSIF (sclr = '1' or SAL_GO = '0') THEN
               PHN_PHASE_SEL <= (OTHERS => '0');
               STALL_POP <= '0';
               next_state <= s0;
            ELSE
               next_state <= s0;
            END IF;
         WHEN s1 =>
            IF (SAL_RST_DONE = '1') THEN
               PHN_PHASE_SEL <= "00";
               STALL_POP <= '0';
               nPAL_POP <= '1';
               next_state <= s3;
            ELSE
               next_state <= s1;
            END IF;
         WHEN s7 =>
            IF ((PHN_GO = '1') AND (nPAL_EMPTY = '1')) THEN
               STALL_POP <= '0';
               phn_ptr_ram_disable <= '1';
               err_npal_empty <= '1';
               next_state <= s10;
            ELSIF (PHN_GO = '1') THEN
               STALL_POP <= '0';
               nPAL_POP <= '1';
```

```
            PHN_PHASE_SEL <= "01";
            next_state <= s9;
         ELSE
            next_state <= s7;
         END IF;
      WHEN s13 =>
         IF (scr_done = '1') THEN
            PAL_POP <= '1';
            PHN_PHASE_SEL <= "11";
            next_state <= s15;
         ELSIF (scr_done = '0') THEN
            next_state <= s13;
         ELSE
            next_state <= s13;
         END IF;
      WHEN s16 =>
         IF (prune_done = '1') THEN
            PHN_DONE <= '1';
            next_state <= s0;
         ELSIF (prune_done = '0') THEN
            next_state <= s16;
         ELSE
            next_state <= s16;
         END IF;
      WHEN s3 =>
         IF (nPAL_DONE = '1') THEN
            phn_ptr_ram_disable <= '1';
            next_state <= s4;
         ELSIF (nPAL_DONE = '0') THEN
            nPAL_POP <= '1';
            next_state <= s3;
         ELSE
            next_state <= s3;
         END IF;
      WHEN s4 =>
         PAL_POP <= '1';
         next_state <= s5;
      WHEN s5 =>
         IF (PAL_DONE = '1') THEN
            phn_ptr_ram_disable <= '1';
            next_state <= s6;
         ELSIF (PAL_DONE = '0') THEN
            PAL_POP <= '1';
            next_state <= s5;
         ELSE
            next_state <= s5;
         END IF;
      WHEN s6 =>
         CURR_PHASE_DONE <= '1';
         next_state <= s7;
      WHEN s9 =>
         IF (nPAL_DONE = '1') THEN
            phn_ptr_ram_disable <= '1';
            next_state <= s10;
         ELSE
            STALL_POP <= '1';
            next_state <= s9_stall;
         END IF;
      WHEN s10 =>
         IF (PAL_EMPTY = '1') THEN
            phn_ptr_ram_disable <= '1';
            err_pal_empty <= '1';
            next_state <= s12;
         ELSE
            PAL_POP <= '1';
            PHN_PHASE_SEL <= "10";
            next_state <= s11;
         END IF;
      WHEN s11 =>
         IF (PAL_DONE = '1') THEN
            phn_ptr_ram_disable <= '1';
            next_state <= s12;
         ELSE
            STALL_POP <= '1';
            next_state <= s11_stall;
         END IF;
      WHEN s12 =>
         CURR_PHASE_DONE <= '1';
```

```
            next_state <= s13;
         WHEN s15 =>
            IF (PAL_DONE = '1') THEN
               phn_ptr_ram_disable <= '1';
               CURR_PHASE_DONE <= '1';
               next_state <= s16;
            ELSIF (PAL_EMPTY = '0') THEN
               PAL_POP <= '1';
               next_state <= s15;
            ELSE
               next_state <= s15;
            END IF;
         WHEN s9_stall =>
            IF (nPAL_DONE = '0') THEN
               nPAL_POP <= '1';
               STALL_POP <= '0';
               next_state <= s9;
            ELSE
               next_state <= s9_stall;
            END IF;
         WHEN s11_stall =>
            IF (PAL_DONE = '0') THEN
               PAL_POP <= '1';
               STALL_POP <= '0';
               next_state <= s11;
            ELSE
               next_state <= s11_stall;
            END IF;
         WHEN OTHERS =>
            next_state <= s0;
      END CASE;
   END PROCESS nextstate_proc;
end fsm;
```

A.7 WORD BLOCK: TOP-LEVEL

```
--------------------------------------------------------------------------------
-- Organization:   University of Pittsburgh
-- Author:         Kshitij Gupta
--
-- Design Name:    WRD TOP-LEVEL
-- Module Name:    wrd_top - struct
-- Project Name:   University of Pittsburgh's Speech Recognition System-on-a-Chip
-- Target Device:  Xilinx Virtex4 SX-35
-- Tool versions:  ISE 7.1i
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity wrd_top is
        PORT(
                clk : IN std_logic;
                ce : IN std_logic;
                sclr : IN std_logic;
                DEAD_PHASE : IN std_logic;
                TKN_INIT_GO : IN std_logic;
                WRD_TH : IN std_logic_vector(31 downto 0);
                RAM_rd_dout : IN std_logic_vector(215 downto 0);
                RAM_rd_dout_valid : IN std_logic_vector(3 downto 0);
                DEAD_GO : IN std_logic;
                DEAD_empty : IN std_logic;
                DEAD_TKN : IN std_logic_vector(10 downto 0);
                DEAD_TKN_valid : IN std_logic;
                NXT_PHN_GO : IN std_logic;
                NXT_PHN_empty : IN std_logic;
                NXT_PHN_TKN : IN std_logic_vector(10 downto 0);
                NXT_PHN_TKN_valid : IN std_logic;
                TKN_INIT_DONE : OUT std_logic;
                RAM_rd_addr : OUT std_logic_vector(9 downto 0);
                RAM_rd_addr_valid : OUT std_logic_vector(3 downto 0);
                RAM_wr_addr : OUT std_logic_vector(9 downto 0);
                RAM_wr_din : OUT std_logic_vector(215 downto 0);
                RAM_wr_addr_valid : OUT std_logic_vector(3 downto 0);
                nPAL_TKN : OUT std_logic_vector(10 downto 0);
                nPAL_TKN_valid : OUT std_logic;
                DEAD_pop : OUT std_logic;
                DEAD_phase_done : OUT std_logic;
                NXT_PHN_pop : OUT std_logic;
                NXT_PHN_phase_done : OUT std_logic;
                WRD_EXIT_SCR : OUT std_logic_vector(31 downto 0);
                WRD_EXIT_ID : OUT std_logic_vector(4 downto 0);
                WRD_EXIT_ID_valid : OUT std_logic;
                WRD_ERR_TKN_FIFO_EMPTY : OUT std_logic;
                WRD_ERR_TKN_FIFO_WR_ERROR : OUT std_logic;
                WRD_ERR_LAST_PHN_ACCESSED : OUT std_logic;
                WRD_DATA_MINE_TKN_FIFO_CNT : OUT std_logic_vector(9 downto 0)
        );
end wrd_top;

architecture struct of wrd_top is

COMPONENT fsm_wrd_cntrl
        PORT(
                SIL_processed : IN std_logic;
                W_END : IN std_logic;
                W_EXIT_VALID : IN std_logic;
                ce : IN std_logic;
                clk : IN std_logic;
                curr_node_processed : IN std_logic;
                DEAD_empty : IN std_logic;
```

```
                NXT_PHN_empty : IN std_logic;
                NXT_PHN_GO : IN std_logic;
                rst : IN std_logic;
                wrd_exit_info_valid : IN std_logic;
                wrd_info_valid : IN std_logic;
                DEAD_phase_done : OUT std_logic;
                DEAD_pop : OUT std_logic;
                insert_SIL : OUT std_logic;
                lex_lcroot_tkn_sel : OUT std_logic;
                lex_tkn_in_en : OUT std_logic;
                NXT_PHN_phase_done : OUT std_logic;
                NXT_PHN_pop : OUT std_logic
        );
END COMPONENT;

component wrd_tkn_alloc_dealloc
        port (
                clk, ce, sclr : IN std_logic;
                TKN_INIT_GO : IN std_logic;
                DEAD_PHASE : IN std_logic;
                NID_in_LEX : IN std_logic_vector(9 downto 0);
                NID_in_LEX_valid : IN std_logic;
                TKN_RAM_rd_addr : IN std_logic_vector(9 downto 0);
                TKN_RAM_rd_addr_valid : IN std_logic;
                TKN_DEAD_dout : IN std_logic_vector(9 downto 0);
                TKN_DEAD_dout_valid : IN std_logic;
                TKN_INIT_PHASE : OUT std_logic;
                TKN_INIT_ID : OUT std_logic_vector(9 downto 0);
                TKN_INIT_ID_VALID : OUT std_logic;
                nTKN_ACTIVE : OUT std_logic;
                ACTIVE_TKN : OUT std_logic_vector(9 downto 0);
                ACTIVE_TKN_valid : OUT std_logic;
                NID_out_LEX : OUT std_logic_vector(9 downto 0);
                NID_out_LEX_valid : OUT std_logic;
                NID_out_NID_ROM : OUT std_logic_vector(9 downto 0);
                NID_out_NID_ROM_valid : OUT std_logic;
                TKN_INIT_DONE : OUT std_logic;
                TKN_FIFO_EMPTY : OUT std_logic;
                TKN_FIFO_FULL : OUT std_logic;
                TKN_FIFO_WR_ERROR : OUT std_logic;
                TKN_FIFO_CNT : OUT std_logic_vector(9 downto 0);
                SIL_DEAD : OUT std_logic
        );
end component;

component wrd_lex_rom
        port (
                clk : IN std_logic;
                ce : IN std_logic;
                sclr : IN std_logic;
                EXIT_NID : IN std_logic_vector (9 downto 0);
                EXIT_NID_valid : IN std_logic;
                NID : OUT std_logic_vector (9 downto 0);
                NID_valid : OUT std_logic;
                LEX_BRANCH_CNT : OUT std_logic_vector(3 downto 0);
                LEX_BRANCH_CNT_valid : OUT std_logic;
                LEX_CURR_BRANCH_PROCESSED : OUT std_logic;
                ERR_LAST_PHONE_ACCESSED : OUT std_logic
        );
end component;

component wrd_node_rom
   port (
    addr: IN std_logic_VECTOR(8 downto 0);
    clk: IN std_logic;
    dout: OUT std_logic_VECTOR(45 downto 0);
    en: IN std_logic;
    nd: IN std_logic;
    rfd: OUT std_logic;
    rdy: OUT std_logic;
    sinit: IN std_logic
        );
end component;

COMPONENT wrd_compute_data
        PORT(
                clk, ce, sclr : IN std_logic;
                WRD_TH : IN std_logic_vector(31 downto 0);
```

```
                 TKN_INIT_PHASE : IN std_logic;
                 TKN_INIT_ID : IN std_logic_vector(9 downto 0);
                 TKN_INIT_ID_VALID : IN std_logic;
                 DEAD_PHASE : IN std_logic;
                 DEAD_TKN_out : IN std_logic_vector(9 downto 0);
                 DEAD_TKN_out_valid : IN std_logic;
                 NXT_PHN_TKN_out : IN std_logic_vector(9 downto 0);
                 NXT_PHN_TKN_out_valid : IN std_logic;
                 lex_branch_cnt : IN std_logic_vector(3 downto 0);
                 ram_rd_dout : IN std_logic_vector(215 downto 0);
                 ram_rd_dout_valid : IN std_logic_vector(3 downto 0);
                 NODE_DATA : IN std_logic_vector(45 downto 0);
                 NODE_DATA_valid : IN std_logic;
                 ACTIVE_TKN : IN std_logic_vector(9 downto 0);
                 ACTIVE_TKN_valid : IN std_logic;
                 INSERT_SIL : IN std_logic;
                 WID : OUT std_logic_vector(9 downto 0);
                 WRD_END : OUT std_logic;
                 WRD_info_valid : OUT std_logic;
                 WRD_EXIT_VALID : OUT std_logic;
                 WRD_EXIT_info_valid : OUT std_logic;
                 WRD_EXIT_SCR : OUT std_logic_vector(31 downto 0);
                 ram_wr_data : OUT std_logic_vector(215 downto 0);
                 ram_wr_addr : OUT std_logic_vector(9 downto 0);
                 ram_wr_en : OUT std_logic_vector(3 downto 0);
                 ram_rd_addr : OUT std_logic_Vector(9 downto 0);
                 ram_rd_en : OUT std_logic_vector(3 downto 0);
                 CURR_NODE_PROCESSED : OUT std_logic
         );
END COMPONENT;

component wrd_lcroot_rom
   port (
    addr: IN std_logic_VECTOR(4 downto 0);
    clk: IN std_logic;
    dout: OUT std_logic_VECTOR(9 downto 0);
    en: IN std_logic;
    nd: IN std_logic;
    rfd: OUT std_logic;
    rdy: OUT std_logic;
    sinit: IN std_logic
         );
end component;

signal SIL_DEAD : std_logic;
signal SIL_DEAD_latch : std_logic;
signal INSERT_SIL : std_logic;
signal lex_lcroot_tkn_sel : std_logic;
signal lex_tkn_in_en : std_logic;
signal CURR_NODE_PROCESSED : std_logic;
signal ACTIVE_TKN : std_logic_vector(9 downto 0);
signal ACTIVE_TKN_valid : std_logic;
signal WID : std_logic_vector(9 downto 0);
signal WRD_END : std_logic;
signal WRD_info_valid : std_logic;
signal WRD_EXIT_VALID : std_logic;
signal WRD_EXIT_info_valid : std_logic;
signal WRD_EXIT_VALID_reg : std_logic;
signal WRD_EXIT_info_valid_reg : std_logic;
signal WID_s : std_logic_vector(4 downto 0);
signal WRD_exit_valid_s : std_logic;
signal NID_LEX_to_NODE : std_logic_vector(9 downto 0);
signal NID_LEX_to_NODE_valid : std_logic;
signal NID_LCROOT_to_LEX : std_logic_vector(9 downto 0);
signal NID_LCROOT_to_LEX_valid : std_logic;
signal NID_LCROOT_to_LEX_valid_reg : std_logic;
signal LEX_ROM_addr : std_logic_vector(9 downto 0);
signal LEX_ROM_addr_valid : std_logic;
signal NID_NODE_to_LEX : std_logic_vector(9 downto 0);
signal NID_NODE_to_LEX_valid : std_logic;
signal NID_NODE_to_LEX_valid_reg : std_logic_vector(3 downto 0);
signal NID_to_NODE_ROM : std_logic_vector(9 downto 0);
signal NID_to_NODE_ROM_valid : std_logic;
signal NID_to_NODE_ROM_s : std_logic_vector(8 downto 0);
signal NODE_DATA : std_logic_vector(45 downto 0);
signal NODE_DATA_valid : std_logic;
signal TKN_INIT_PHASE : std_logic;
signal TKN_INIT_ID : std_logic_vector(9 downto 0);
```

```
signal TKN_INIT_ID_VALID : std_logic;
signal TKN_RAM_rd_addr : std_logic_vector(9 downto 0);
signal TKN_RAM_rd_addr_valid : std_logic;
signal LEX_BRANCH_CNT : std_logic_vector(3 downto 0);
signal LEX_BRANCH_CNT_valid : std_logic;
signal LEX_CURR_BRANCH_PROCESSED : std_logic;
signal TKN_FIFO_FULL : std_logic;
signal rfd_dummy0, rfd_dummy1 : std_logic;

constant LEX_ROM_SIL_PHN_ADDR : std_logic_vector(9 downto 0) := "0111001101";

begin

nPAL_TKN <= '0' & ACTIVE_TKN;

process(clk, ce, sclr, NID_LCROOT_to_LEX_valid, NID_NODE_to_LEX_valid)
begin
	if (clk'event and clk = '1') then
		if (sclr = '1') then
			NID_LCROOT_to_LEX_valid_reg <= '0';
			NID_NODE_to_LEX_valid_reg <= (OTHERS => '0');
		elsif (ce = '1') then
			NID_LCROOT_to_LEX_valid_reg <= NID_LCROOT_to_LEX_valid;
			NID_NODE_to_LEX_valid_reg <= NID_NODE_to_LEX_valid_reg(2 downto 0) &
NID_NODE_to_LEX_valid;
		end if;
	end if;
end process;

process(INSERT_SIL, lex_tkn_in_en, lex_lcroot_tkn_sel, NID_NODE_to_LEX, NID_NODE_to_LEX_valid_reg,
NID_LCROOT_to_LEX, NID_LCROOT_to_LEX_valid_reg)
	variable LEX_ROM_addr_s : std_logic_vector(9 downto 0);
	variable LEX_ROM_addr_valid_s : std_logic;
begin
	case INSERT_SIL is
		when '0' =>
			case lex_lcroot_tkn_sel is
				when '0' =>
					LEX_ROM_addr_s := NID_NODE_to_LEX;
					LEX_ROM_addr_valid_s := NID_NODE_to_LEX_valid_reg(3) and
lex_tkn_in_en;
				when '1' =>
					LEX_ROM_addr_s := NID_LCROOT_to_LEX;
					LEX_ROM_addr_valid_s := NID_LCROOT_to_LEX_valid_reg and
lex_tkn_in_en;
				when OTHERS =>
					LEX_ROM_addr_s := (OTHERS => '0');
					LEX_ROM_addr_valid_s := '0';
			end case;
			LEX_ROM_addr <= LEX_ROM_addr_s;
			LEX_ROM_addr_valid <= LEX_ROM_addr_valid_s;
		when '1' =>
			LEX_ROM_addr <= LEX_ROM_SIL_PHN_ADDR;
			LEX_ROM_addr_valid <= '1';
		when OTHERS =>
			LEX_ROM_addr <= (OTHERS => '0');
			LEX_ROM_addr_valid <= '0';
	end case;
end process;

process(DEAD_PHASE, NXT_PHN_TKN, NXT_PHN_TKN_valid, DEAD_TKN, DEAD_TKN_valid)
begin
	case DEAD_PHASE is
		when '0' =>
			TKN_RAM_rd_addr <= NXT_PHN_TKN(9 downto 0);
			TKN_RAM_rd_addr_valid <= NXT_PHN_TKN_valid;
		when '1' =>
			TKN_RAM_rd_addr <= DEAD_TKN(9 downto 0);
			TKN_RAM_rd_addr_valid <= DEAD_TKN_valid;
		when OTHERS =>
			TKN_RAM_rd_addr <= (OTHERS => '0');
			TKN_RAM_rd_addr_valid <= '0';
	end case;
end process;

process(clk, ce, sclr, WRD_EXIT_VALID, WRD_EXIT_info_valid)
begin
	if (clk'event and clk = '1') thena
```

```
                if (sclr = '1') then
                        WRD_EXIT_VALID_reg <= '0';
                        WRD_EXIT_info_valid_reg <= '0';
                elsif (ce = '1') then
                        WRD_EXIT_VALID_reg <= WRD_EXIT_VALID;
                        WRD_EXIT_info_valid_reg <= WRD_EXIT_info_valid;
                end if;
        end if;
end process;

WRD_TOP_CTRL_BLK : fsm_wrd_cntrl
        PORT MAP(
                SIL_processed => SIL_DEAD_latch,
                W_END => WRD_END,
                W_EXIT_VALID => WRD_EXIT_VALID_reg,
                ce => ce,
                clk => clk,
                curr_node_processed => CURR_NODE_PROCESSED,
                DEAD_empty => DEAD_empty,
                DEAD_go => DEAD_GO,
                NXT_PHN_empty => NXT_PHN_empty,
                NXT_PHN_GO => NXT_PHN_GO,
                rst => sclr,
                wrd_exit_info_valid => WRD_EXIT_info_valid_reg,
                wrd_info_valid => WRD_info_valid,
                DEAD_phase_done => DEAD_phase_done,
                DEAD_pop => DEAD_pop,
                insert_SIL => INSERT_SIL,
                lex_lcroot_tkn_sel => lex_lcroot_tkn_sel,
                lex_tkn_in_en => lex_tkn_in_en,
                NXT_PHN_phase_done => NXT_PHN_phase_done,
                NXT_PHN_pop => NXT_PHN_pop
        );

WRD_TOP_TKN_ALLOC_DEALLOC_BLK : wrd_tkn_alloc_dealloc
        port map (
                clk => clk,
                ce => ce,
                sclr => sclr,
                TKN_INIT_GO => TKN_INIT_GO,
                DEAD_PHASE => DEAD_PHASE,
                NID_in_LEX => NID_LEX_to_NODE,
                NID_in_LEX_valid => NID_LEX_to_NODE_valid,
                TKN_RAM_rd_addr => TKN_RAM_rd_addr,
                TKN_RAM_rd_addr_valid => TKN_RAM_rd_addr_valid,
                TKN_DEAD_dout => DEAD_TKN(9 downto 0),
                TKN_DEAD_dout_valid => DEAD_TKN_valid,
                TKN_INIT_PHASE => TKN_INIT_PHASE,
                TKN_INIT_ID => TKN_INIT_ID,
                TKN_INIT_ID_VALID => TKN_INIT_ID_VALID,
                nTKN_ACTIVE => nPAL_TKN_valid,
                ACTIVE_TKN => ACTIVE_TKN,
                ACTIVE_TKN_valid => ACTIVE_TKN_valid,
                NID_out_LEX => NID_NODE_to_LEX,
                NID_out_LEX_valid => NID_NODE_to_LEX_valid,
                NID_out_NID_ROM => NID_to_NODE_ROM,
                NID_out_NID_ROM_valid => NID_to_NODE_ROM_valid,
                TKN_INIT_DONE => TKN_INIT_DONE,
                TKN_FIFO_EMPTY => WRD_ERR_TKN_FIFO_EMPTY,
                TKN_FIFO_FULL => TKN_FIFO_FULL,
                TKN_FIFO_WR_ERROR => WRD_ERR_TKN_FIFO_WR_ERROR,
                TKN_FIFO_CNT => WRD_DATA_MINE_TKN_FIFO_CNT,
                SIL_DEAD => SIL_DEAD
        );

process (DEAD_GO, SIL_DEAD)
begin
        if (DEAD_GO = '1') then
                SIL_DEAD_latch <= '0';
        else
                if (SIL_DEAD = '1') then
                        SIL_DEAD_latch <= SIL_DEAD;
                end if;
        end if;
end process;

WRD_TOP_LEXTREE_ROM_BLK : wrd_lex_rom
        port map (
```

```
                 clk => clk,
                 ce => ce,
                 sclr => sclr,
                 EXIT_NID => LEX_ROM_addr,
                 EXIT_NID_valid => LEX_ROM_addr_valid,
                 NID => NID_LEX_to_NODE,
                 NID_valid => NID_LEX_to_NODE_valid,
                 LEX_BRANCH_CNT => LEX_BRANCH_CNT,
                 LEX_BRANCH_CNT_valid => LEX_BRANCH_CNT_valid,
                 LEX_CURR_BRANCH_PROCESSED => LEX_CURR_BRANCH_PROCESSED,
                 ERR_LAST_PHONE_ACCESSED => WRD_ERR_LAST_PHN_ACCESSED
        );

NID_to_NODE_ROM_s <= NID_to_NODE_ROM(8 downto 0);
WRD_TOP_NODE_ROM_BLK : wrd_node_rom
  port map (
      addr => NID_to_NODE_ROM_s,
      clk => clk,
      dout => NODE_DATA,
      en => ce,
      nd => NID_to_NODE_ROM_valid,
      rfd => rfd_dummy0,
      rdy => NODE_DATA_valid,
      sinit => sclr);

WRD_TOP_COMPUTE_DATA_BLK : wrd_compute_data
        PORT MAP(
                 clk => clk,
                 ce => ce,
                 sclr => sclr,
                 WRD_TH => WRD_TH,
                 TKN_INIT_PHASE => TKN_INIT_PHASE,
                 TKN_INIT_ID => TKN_INIT_ID,
                 TKN_INIT_ID_VALID => TKN_INIT_ID_VALID,
                 DEAD_PHASE => DEAD_PHASE,
                 DEAD_TKN_out => DEAD_TKN(9 downto 0),
                 DEAD_TKN_out_valid => DEAD_TKN_valid,
                 NXT_PHN_TKN_out => NXT_PHN_TKN(9 downto 0),
                 NXT_PHN_TKN_out_valid => NXT_PHN_TKN_valid,
                 lex_branch_cnt => LEX_BRANCH_CNT,
                 ram_rd_dout => RAM_rd_dout,
                 ram_rd_dout_valid => RAM_rd_dout_valid,
                 NODE_DATA => NODE_DATA,
                 NODE_DATA_valid => NODE_DATA_valid,
                 ACTIVE_TKN => ACTIVE_TKN,
                 ACTIVE_TKN_valid => ACTIVE_TKN_valid,
                 INSERT_SIL => INSERT_SIL,
                 WID => WID,
                 WRD_END => WRD_END,
                 WRD_info_valid => WRD_info_valid,
                 WRD_EXIT_VALID => WRD_EXIT_VALID,
                 WRD_EXIT_info_valid => WRD_EXIT_info_valid,
                 WRD_EXIT_SCR => WRD_EXIT_SCR,
                 ram_wr_data => RAM_wr_din,
                 ram_wr_addr => RAM_wr_addr,
                 ram_wr_en => RAM_wr_addr_valid,
                 ram_rd_addr => RAM_rd_addr,
                 ram_rd_en => RAM_rd_addr_valid,
                 CURR_NODE_PROCESSED => CURR_NODE_PROCESSED
        );

WID_s <= WID(4 downto 0);
WRD_exit_valid_s <= WRD_EXIT_VALID and WRD_END and WRD_info_valid;

WRD_TOP_LCROOT_ROM_BLK : wrd_lcroot_rom
  port map (
      addr => WID_s,
      clk => clk,
      dout => NID_LCROOT_to_LEX,
      en => ce,
      nd => WRD_exit_valid_s,
      rfd => rfd_dummy1,
      rdy => NID_LCROOT_to_LEX_valid,
      sinit => sclr);
WRD_EXIT_ID <= WID_s;
WRD_EXIT_ID_valid <= WRD_exit_valid_s;

end struct;
```

A.8 WORD BLOCK: CONTROLLER

```
--------------------------------------------------------------------------------
-- Organization:   University of Pittsburgh
-- Author:         Kshitij Gupta
--
-- Design Name:    WRD CONTROLLER
-- Module Name:    fsm_wrd_cntrl - struct
-- Project Name:   University of Pittsburgh's Speech Recognition System-on-a-Chip
-- Target Device:  Xilinx Virtex4 SX-35
-- Tool versions:  ISE 7.1i
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity fsm_wrd_cntrl is
   PORT(
      SIL_processed       : IN     std_logic;
      W_END               : IN     std_logic;
      W_EXIT_VALID        : IN     std_logic;
      ce                  : IN     std_logic;
      clk                 : IN     std_logic;
      curr_node_processed : IN     std_logic;
      DEAD_empty          : IN     std_logic;
      DEAD_go             : IN     std_logic;
      NXT_PHN_empty       : IN     std_logic;
      NXT_PHN_go          : IN     std_logic;
      rst                 : IN     std_logic;
      wrd_exit_info_valid : IN     std_logic;
      wrd_info_valid      : IN     std_logic;
      DEAD_phase_done     : OUT    std_logic;
      DEAD_pop            : OUT    std_logic;
      insert_SIL          : OUT    std_logic;
      lex_lcroot_tkn_sel  : OUT    std_logic;
      lex_tkn_in_en       : OUT    std_logic;
      NXT_PHN_phase_done  : OUT    std_logic;
      NXT_PHN_pop         : OUT    std_logic
   );
end fsm_wrd_cntrl;

architecture fsm of fsm_wrd_cntrl is


   TYPE STATE_TYPE IS (
      s0,
      s1,
      s2,
      s3,
      s4,
      s5,
      s6,
      s7,
      s8,
      s9,
      s10,
      s11,
      s12,
      s4_wait
   );

   -- State vector declaration
   ATTRIBUTE state_vector : string;
   ATTRIBUTE state_vector OF fsm : ARCHITECTURE IS "current_state";

   -- Declare current and next state signals
   SIGNAL current_state : STATE_TYPE;
```

```
   SIGNAL next_state : STATE_TYPE;

BEGIN

   ----------------------------------------------------------------
   clocked_proc : PROCESS (
      clk
   )
   ----------------------------------------------------------------
   BEGIN
      IF (clk'EVENT AND clk = '1') THEN
         IF (rst = '1') THEN
            current_state <= s0;
         ELSIF (ce = '1') THEN
            current_state <= next_state;
         END IF;
      END IF;
   END PROCESS clocked_proc;

   ----------------------------------------------------------------
   nextstate_proc : PROCESS (
      SIL_processed,
      W_END,
      W_EXIT_VALID,
      curr_node_processed,
      current_state,
      dead_empty,
      dead_go,
      nxt_phn_empty,
      nxt_phn_go,
      wrd_exit_info_valid,
      wrd_info_valid
   )
   ----------------------------------------------------------------
   BEGIN
      -- Default Assignment
      dead_phase_done <= '0';
      dead_pop <= '0';
      insert_SIL <= '0';
      lex_lcroot_tkn_sel <= '0';
      lex_tkn_in_en <= '0';
      nxt_phn_phase_done <= '0';
      nxt_phn_pop <= '0';

      -- Combined Actions
      CASE current_state IS
         WHEN s0 =>
            IF (dead_go = '1') THEN
               next_state <= s9;
            ELSIF (dead_go = '0') THEN
               next_state <= s0;
            ELSE
               next_state <= s0;
            END IF;
         WHEN s1 =>
            IF (nxt_phn_empty = '0') THEN -- NOT EMPTY
               nxt_phn_pop <= '1';
               next_state <= s2;
            ELSIF (nxt_phn_empty = '1') THEN -- EMPTY
               next_state <= s5;
            ELSE
               next_state <= s1;
            END IF;
         WHEN s2 =>
            IF (wrd_info_valid = '1' and W_END = '0') THEN
               lex_tkn_in_en <= '1';
               lex_lcroot_tkn_sel <= '0';
               next_state <= s3;
            ELSIF (wrd_info_valid = '1'
                   and
                   W_END = '1') THEN
               next_state <= s4;
            ELSE
               next_state <= s2;
            END IF;
         WHEN s3 =>
            IF (curr_node_processed = '1') THEN
               next_state <= s1;
```

```
            ELSE
               next_state <= s3;
            END IF;
         WHEN s4 =>
            IF (wrd_exit_info_valid = '1'
                and
                W_EXIT_VALID = '0') THEN
               next_state <= s1;
            ELSIF (wrd_exit_info_valid = '1'
                   and
                   W_EXIT_VALID = '1') THEN
               next_state <= s4_wait;
            ELSE
               next_state <= s4;
            END IF;
         WHEN s5 =>
            IF (SIL_processed = '1') THEN
               insert_SIL <= '1';
               next_state <= s6;
            ELSIF (SIL_processed = '0') THEN
               next_state <= s7;
            ELSE
               next_state <= s5;
            END IF;
         WHEN s6 =>
            IF (curr_node_processed = '1') THEN
               next_state <= s7;
            ELSE
               next_state <= s6;
            END IF;
         WHEN s7 =>
            nxt_phn_phase_done <= '1';
            next_state <= s0;
         WHEN s8 =>
            IF (nxt_phn_go ='1') THEN
               next_state <= s1;
            ELSE
               next_state <= s8;
            END IF;
         WHEN s9 =>
            IF (dead_empty = '1') THEN
               next_state <= s10;
            ELSIF (dead_empty = '0') THEN
               dead_pop <= '1';
               next_state <= s9;
            ELSE
               next_state <= s9;
            END IF;
         WHEN s10 =>
            next_state <= s11;
         WHEN s11 =>
            next_state <= s12;
         WHEN s12 =>
            dead_phase_done <= '1';
            next_state <= s8;
         WHEN s4_wait =>
            lex_lcroot_tkn_sel <= '1';
            lex_tkn_in_en <= '1';
            next_state <= s3;
         WHEN OTHERS =>
            next_state <= s0;
      END CASE;
   END PROCESS nextstate_proc;

end fsm;
```

BIBLIOGRAPHY

[1] Huang, X., Acero, A. and Hon, H., Spoken Language Processing, Prentice Hall PTR, NJ, 2001.

[2] Results or a medium vocabulary test, CMU Sphinx, http://cmusphinx.sourceforge.net/MediumVocabResults.html

[3] ARM922T (Rev 0) Technical Reference Manual, ARM Inc.

[4] K.Agaram, S.W.Keckler, and D.C.Burger. "A Characterization of Speech Recognition on Modern Computer Systems", *4th IEEE Workshop on Workload Characterization*, at MICRO-34, December 2001.

[5] C.Lai, S.Su, and Q.Zhao, "Performance Analysis of Speech Recognition Software," *Fifth Workshop on Computer Architecture Evaluation using Commercial Workloads,* February 2002.

[6] Ravishankar, M., Singh, R., Raj, B. and Stern, R., The 1999 CMU 10x Real Time Broadcast News Transcription System, *Proc. DARPA Workshop on Automatic Transcription of Broadcast News*, Washington DC, May 2000.

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

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

[9] Resource Management 1, Linguistic Data Consortium, University of Pennsylvania, http://www.ldc.upenn.edu/Catalog/CatalogEntry.jsp?catalogId=LDC93S3B.

[10] Binu Mathew, Al Davis and Zhen 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.

[11] X. Li and J. Bilmes, "Feature pruning in likelihood evaluation of HMM-based speech recognition," in *Proc. IEEE ASRU Workshop*, 2003.

[12] Dragon Naturally Speaking, ScanSoft Inc., http://www.scansoft.co.uk/naturallyspeaking/.

[13] ViaVoice, IBM Inc., http://www-306.ibm.com/software/voice/viavoice/.

[14] T Hain, P.C. Woodland, G Evermann, D. Povey, "The CU-HTK March 2000 Hub5e Transcription System", *Proceedings of the 2000 DARPA Speech Recognition Workshop*.

[15] Sensory Inc., http://www.sensoryinc.com/.

[16] RSC-4x family, Sensory Inc., http://www.sensoryinc.com/html/products/rsc4x.html.

[17] SC-6x family, Sensory Inc., http://www.sensoryinc.com/html/products/scseries.html.

[18] P.S. Gopalakrisnan, L.R. Bahl, R.L. Mercer. "A Tree Search Strategy for Large Vocabulary Continuous Speech Recognition." *Proc ICASSP,* Vol 1, pp. 572-575, Detroit, 1995.

[19] Rabiner, L.R., "A tutorial on Hidden Markov Models and selected applications in speech recognition," *Proc. IEEE*, 77, No.2, 1989, pp.257-286.

[20] Viterbi, A. J., "Error bounds for Convolutional Codes and an asymptotically optimum decoding algorithm," *IEEE Trans. On Information Theory*, 13(2):260-269, 1967.

[21] Bocchieri, E., "A Study of Beam Search Algorithm for Large Vocabulary Continuous Speech Recognition and Methods of Improved Efficiency.," *Proc. of Eurospeech*, p1521-1524, Berlin, 1993.

[22] M. K. Ravishankar, "Parallel implementation of fast beam search for speaker-independent continuous speech recognition", Computer Science & Automation, Indian Institute of Science, Bangalore, India, 1993.

[23] L.R. Bahl, P.V. de Souza, P.S. Gopalakrishnan, D. Nahamoo, M.A. Picheny. "Context Dependent Modeling of Phones in Continuous Speech Using Decision Trees." *Proc DARPA Speech and Natural Language Processing Workshop,* pp. 264-270, Feb 1991.

[24] T. S. Anantharaman and R. Bisiani. A Hardware Accelerator for Speech Recognition Algorithms. ISCA, pages 216-223, 1986.

[25] S.J. Melnikoff, S.F. Quigley, M.J. Russell, "Performing Speech Recognition on Multiple Parallel Files Using Continous Hidden Markov Models on an FPGA," *International Conference on Field-Programmable Technology and its Applications,* 2002.

[26] Sergiu Nedevschi, Rabin K. Patra, Eric A. Brewer, "Hardware Speech Recognition for User Interfaces in Low Cost, Low Power Devices", *Design Automation Conference*, Anaheim, June 2005.

[27] Taylor Serires, Wolform, http://mathworld.wolfram.com/MercatorSeries.html.

[28] Logarithms, Wolfrom, http://mathworld.wolfram.com/Logarithm.html.

[29] MATLAB Fixed-point Toolbox, Mathworks Inc., http://www.mathworks.com/products/fixed/.

[30] MATLAB 7, Release 14, Mathworks Inc., http://www.mathworks.com/products/new_products/latest_features.html.

[31] Bocchieri, E., Wilpon, J., "Discriminative Feature Selection for Speech Recognition", *Computer Speech and Language7*, p229-246. AT&T, Bell Laboratories, 1993.

[32] Li, X., Blimes, J. "Feature Pruning in Likelihood Evaluation of HMM-Based Speech Recognition", Unversity of Washington, 2003.

[33] 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.

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

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

[36] Xilinx Virtex4 FPGA, Xilinx Inc., http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/index.htm

[37] Xilinx System Generator, Xilinx Inc., http://www.xilinx.com/ise/optional_prod/system_generator.htm

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

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